1 Contents 3
  1.1 Installation ................................................................. 3
  1.2 BATCHIE Data Format ....................................................... 3
  1.3 Running BATCHIE ............................................................ 5
  1.4 Configuring BATCHIE ....................................................... 7
  1.5 Extending BATCHIE .......................................................... 9
  1.6 Command Line Interface .................................................. 9
  1.7 batchie package ............................................................. 11

2 Indices and tables 41

Python Module Index 43

Index 45
Welcome to the documentation for the Python implementation of BATCHIE

If you use BATCHIE please cite our preprint:

A Bayesian active learning platform for scalable combination drug screens
Christopher Tosh, Mauricio Tec, Jessica White, Jeffrey F Quinn, Glorymar Ibanez Sanchez,
Paul Calder, Andrew L Kung, Filemon S Dela Cruz, Wesley Tansey
1.1 Installation

This tutorial will walk you through the process of setting up an environment to run BATCHIE.

1.1.1 Option 1 (Recommended): Using Nextflow + Docker

BATCHIE uses several python packages with C extensions, so the easiest way to get started is using the up to date docker image we maintain on docker hub.

docker pull jeffquinmsk/batchie:latest

To install Nextflow see the instructions here: https://www.nextflow.io/docs/latest/getstarted.html

1.1.2 Option 2: Install Using pip

For advanced usage, BATCHIE can be installed directly as a python package using pip.

pip install git+https://github.com/tansey-lab/batchie

1.2 BATCHIE Data Format

The main data format for BATCHIE is the batchie.data.Screen object.

1.2.1 Example

See below for an example of creating a screen object and saving it to disk:

```python
from batchie.data import Screen

screen = Screen(
    observations=np.array([0.1, 0.2, 0.0, 0.0]),
    observation_mask=np.array([True, True, False, False]),
    sample_names=np.array(["sample_A", "sample_B",
```

(continues on next page)


```

    "sample_C",
    "sample_D"
  ], dtype=str),
plate_names=np.array([
    "plate_A",
    "plate_A",
    "plate_B",
    "plate_B"
  ], dtype=str),
treatment_names=np.array(
    [
      ["drug_A", "drug_B"],
      ["drug_A", "drug_B"],
      ["drug_A", "drug_B"],
      ["drug_A", "control"]
    ], dtype=str
  ),
treatment_doses=np.array(
    [
      [2.0, 2.0],
      [1.0, 2.0],
      [2.0, 1.0],
      [2.0, 0]
    ]
  ),
control_treatment_name="control"

screen.save_h5("my_screen.screen.h5")
```

1.2.2 Explanation of Terms

A Screen contains one or more experiments. Each experiment has a set of conditions, the conditions are the treatment_names (drug names for instance) and the treatment_doses (drug concentration for instance). If some treatments are considered “controls” or to have no effect, they should be equal to the indicated control_treatment_name.

Each experiment is on a particular sample, indicated by the sample_names.

Each experiment has an optional observation, passed via the observations parameter.

The observation is the outcome of observing the experiment, a single floating point number. In the case of high throughput drug screening, this is the viability of the cells. If the experiment has no observation yet, then you can indicate that by passing an observation_mask with False values for the experiments that have no observation yet.

Each experiment belongs to a particular plate, passed via the plate_names parameter.

You create a batchie screen object in python by passing it numpy arrays for all of these various parameters. All the numpy arrays you pass in are expected to be the same length. The numpy arrays are also expected to be aligned, aka index N of each array corresponds to the same experiment for all N in range(screen.size)

Furthermore treatment_names and treatment_doses are expected to be the same width (just set the dose to 0 for controls). All of these things are sanity checked so if you make a mistake, initializing the screen will raise an exception.
Assuming you are comfortable working with numpy, it should be very straightforward to create a screen object from your dataset.

1.2.3 Explanation of Example

In this example in the code block above we have a screen that is looking at 4 experiments, with arity of 2 (looking at combinations of 2 treatments per experiment). There are two unique plates, with two experiments each. On plate_B the second experiment has a control, so we are only measuring the effect of drug_A in that experiment.

The observation_mask indicates that the experiments in plate_A have been observed, but the experiments in plate_B have not (the value of the observations array for experiments where observation_mask is False is ignored, you can just set it to zero or any value).

1.2.4 On-Disk Format

The on-disk format for the screen is a compressed HDF5 file. We use the extension .screen.h5 for these files by convention. Once you have your .screen.h5 file prepared, you are ready to start running BATCHIE!

1.3 Running BATCHIE

After you have created your BATCHIE batchie.data.Screen file (see here: BATCHIE Data Format), you can run BATCHIE using our configurable pipeline.

There are two main modes for running BATCHIE, Prospective and Retrospective.

Our pipeline adheres to nf-core standards, and accepts all the nf-core configuration options, see here: https://nf-co.re/docs/usage/configuration

1.3.1 Prerequisites

You must have Nextflow and docker installed to run the examples shown here. See Option 1 (Recommended): Using Nextflow + Docker for more details.

Because our pipeline is recursive, we are not able to run it completely in nextflow. We have a small python wrapper around nextflow in nextflow/scripts/batchie.py for running this pipeline. All that is required to run this script is a recent version of python3 and to have nextflow installed. The batchie python package does not need to be installed locally, since nextflow can launch all jobs in our container if the -profile docker, -profile singularity, -profile podman, etc. flags are used.

1.3.2 Prospective

Assume you have a batchie.data.Screen containing one or more batchie.data.Plate s that you have not yet observed.

In prospective mode, BATCHIE will tell you which of those unobserved plates should be run next. The prospective pipeline has the following steps:

1. Train the specified model given the observed plates
2. Use the trained model to predict on all unobserved plates
3. Use the model predictions on unobserved data to score all the unobserved plates
4. Chose one or more plates with the best scores

5. If `--batch-size` is specified, repeat from step 3 until `<batch_size>` plates have been revealed, if `<batch_size>` plates have been revealed, repeat from step 1.

From the root of the repository, you would run a command like this:

```bash
python nextflow/scripts/batchie.py \
   --mode prospective \
   --outdir /tmp/batchie \
   --screen nextflow/tests/data/masked_screen.h5 \
   --batch-size 2 \
   -c nextflow/config/example_prospective.config \
   --max_cpus 1 \
   --max_memory 2G \
   -profile docker
```

The output directory will be organized by iteration and plate (if `batch_size` > 1, then there is multiple plates per iteration).

In the output directory there will be a simple text file with the id of the best plate to observe next:

```
$ cat /tmp/batchie/iter_0/plate_0/masked_screen/selected_plate
1
$ cat /tmp/batchie/iter_0/plate_1/masked_screen/selected_plate
2
```

### 1.3.3 Retrospective

Assume you have a `batchie.data.Screen` where all `batchie.data.Plate`s are observed, representing data from a high throughput screen that was run in the past.

In retrospective simulation mode we will run these set up steps:

1. Mask all of the observations.
2. Sample a fraction of the unobserved experiments and set it aside as “holdout” for evaluating model accuracy.
3. Pick one initial plate to reveal. For the initial plate reveal, one has the option of using a configured heuristic algorithm or picking an existing plate at random.

After these initial steps, we repeat the following series of steps until all plates have been revealed:

1. If all plates have been revealed, exit.
2. Train the specified model given the revealed plate(s)
3. Evaluate the model on the holdout data and save
4. Use the trained model to predict on all unobserved plates
5. Use the model predictions on unobserved data to score all the unobserved plates
6. Chose one or more plates with the best scores to reveal, reveal them
7. If `--batch-size` is specified, repeat from step 5 until `<batch_size>` plates have been revealed, if `<batch_size>` plates have been revealed, repeat from step 1.

From the root of the repository, you would run a command like this:
batchie, Release 0.1

```bash
python3 nextflow/scripts/batchie.py \
   --mode retrospective \
   -c nextflow/config/example_retrospective.config \
   --screen nextflow/tests/data/unmasked_screen.h5 \
   --batch-size 2 \
   --outdir /tmp/batchie \
   --max_cpus 1 \
   --max_memory 2G \
   -profile docker
```

The output directory will be similar to prospective mode.

```bash
$ cat /tmp/batchie/iter_0/plate_0/unmasked_screen/selected_plate
1
$ cat /tmp/batchie/iter_0/plate_1/unmasked_screen/selected_plate
2
```

However there will also be `model_evaluation.h5` files for each iteration which save how the model performed on the holdout set given training on the plates revealed up until that point. These can be collected for analysis after the simulation completes.

### 1.3.4 Computational Cost

The BATCHIE pipeline is generally more CPU/time bound than memory bound. MCMC sampling can take a significant amount of time to complete. The BATCHIE pipeline does not use GPU or hardware acceleration at this time.

The nf-core standard options `--max_cpus`, `--max_memory`, and `--max_time` can be used to limit the resources of individual jobs. `--max_cpus` should probably always be set to 1 since no individual jobs utilize multiprocessing at this time. Parallelizable steps, which include pairwise distance calculation and plate scoring, are parallelized at the job level. The number of concurrent jobs for these parallelizable steps is controlled by the `--n_chunks` parameter.

MCMC sampling is not parallelizable but we allow running multiple MCMC chains to ensure approximation of the posterior. The number of chains can be controlled with the `--n_chains` parameter.

Jobs which fail will be reattempted with higher limits on resources (bounded by the specified maximums). By default 3 retries will be attempted for each job.

### 1.4 Configuring BATCHIE

#### 1.4.1 Pipeline Wide Parameters

Pipeline-wide parameters affect the execution of the pipeline as a whole. These parameters can be specified on the command line when running `batchie.py`.

`--n_chunks` is the number of chunks to split distance matrix/plate score calculation into. `--n_chains` is the number of parallel MCMC chains to run. `--batch-size` is the number of plates to select per iteration `--outdir` is the output directory `--screen` is the screen to use for running the pipeline.
1.4.2 Process Specific Parameters

BATCHIE as a pipeline is made up of several component Python command line interface applications. Documentation for each individual command line application is available here: Command Line Interface. Command line parameters can be passed into each application using a single nextflow .config file.

Several pre-defined configuration files are available in nextflow/config. We recommend using these as templates for your own configuration files.

Let's take a look at an example configuration file to explain its contents:

```plaintext
process {
    publishDir = { "${params.outdir}" }

    withName: TRAIN_MODEL {
        ext.args = [
            '--model', 'SparseDrugCombo',
            '--model-param', 'n_embedding_dimensions=10',
            '--n-burnin', '10',
            '--n-samples', '10',
            '--thin', '1'
        ].join(' ')
    }

    withName: CALCULATE_DISTANCE_MATRIX_CHUNK {
        ext.args = [
            '--model', 'SparseDrugCombo',
            '--model-param', 'n_embedding_dimensions=10',
            '--distance-metric', 'MSEDistance'
        ].join(' ')
    }

    withName: CALCULATE_SCORE_CHUNK {
        ext.args = [
            '--model', 'SparseDrugCombo',
            '--model-param', 'n_embedding_dimensions=10',
            '--scorer', 'GaussianDBALScorer',
            '--seed', '12'
        ].join(' ')
    }
}
```

This is boilerplate that tells nextflow that the output for all processes should be published to the directory specified in the pipeline-wide --outdir parameter:

```plaintext
process {
    publishDir = { "${params.outdir}" }
}
```

This section:

```plaintext
withName: TRAIN_MODEL {
    ext.args = [
        '--model', 'SparseDrugCombo',
        '--model-param', 'n_embedding_dimensions=10',
```
Specifies what command line options will be passed to the call to `train_model`. You'll define a similar section for each other application which you want to pass command line options to. Some applications don't have any required command line options, so you can leave those sections out entirely.

You can always refer to the Command Line Interface documentation for more information on what options are available for each process.

## 1.5 Extending BATCHIE

BATCHIE is designed to be extended with new models and methods.

### 1.5.1 Adding a new model

To add a new model for prediction of outcomes, you will need to create an implementation of the `batchie.core.BayesianModel` class. You will also need to implement the `batchie.core.Theta` and `batchie.core.ThetaHolder` class for working with and serializing the model parameters.

The command line applications where the model is used (for instance `train_model`) accept `--model` and `--model-params` arguments. `--model` is the name of the class implementing the model, and `--model-params` are arbitrary key-value pairs that will be passed to the model constructor (`__init__` method).

### 1.5.2 Other Extensible Classes

Other than the model, the `batchie.core.Scorer`, `batchie.core.DistanceMetric`, and `batchie.core.PlatePolicy` can also be implemented and specified in the command line applications in the same way.

For retrospective simulations `batchie.core.RetrospectivePlateGenerator`, `batchie.core.InitialRetrospectivePlateGenerator`, and `batchie.core.RetrospectivePlateSmother`, can also be implemented and specified for the `prepare_retrospective_simulation` command line application.

## 1.6 Command Line Interface

Batchie provides a suite of command line utilities that allow users to script running the pipeline end to end.
1.6.1 train_model

1.6.2 evaluate_model

1.6.3 calculate_distance_matrix

1.6.4 calculate_scores

1.6.5 select_next_plate

1.6.6 reveal_plate

This is a utility for revealing plates in a retrospective simulation.


Named Arguments

-v, --verbose Enable verbose logging
Default: False
-P, --progress Enable progress bar
Default: False
--screen A batchie Screen in hdf5 format with some plates observed.
--output Where to save screen with the next plate revealed.
--plate-id The plate(s) to reveal.

1.6.7 extract_screen_metadata

extract_screen_metadata.py


Named Arguments

-v, --verbose Enable verbose logging
Default: False
-P, --progress Enable progress bar
Default: False
--screen A batchie Screen in hdf5 format.
--output Output json file to save metadata to.
1.6.8 prepare_retrospective_simulation

1.7 batchie package

1.7.1 Submodules

1.7.2 batchie.common module

batchie.common.copy_array_with_control_treatments_set_to_zero(arr: numpy.ndarray, treatment_array: numpy.ndarray)

batchie.common.select_unique_zipped_numpy_arrays(arrs)

Returns a boolean array that selects unique combinations of several same length numpy arrays.

Parameters
arrs – Arrays of the same length.

Returns
Boolean array indicating unique combinations.

1.7.3 batchie.core module

class batchie.core.BayesianModel(n_unique_treatments: int, n_unique_samples: int)

Bases: ABC

This class represents a Bayesian model.

A Bayesian model has internal state. Each batchie.core.BayesianModel should have a companion batchie.core.Theta and batchie.core.ThetaHolder class which represents the models internal state in a serializable way.

The internal state of the model can be set explicitly via batchie.core.BayesianModel.set_model_state(). or it can be advanced via batchie.core.BayesianModel.step().

A batchie.core.BayesianModel can have data added to it via batchie.core.BayesianModel.add_observations(). If data is present, the model should use that data somehow when BayesianModel#step is called. batchie.core.BayesianModel.n_obs() should report the number of datapoints that have been added to the model.

A batchie.core.BayesianModel can be used to predict the outcome of an Experiment via batchie.core.BayesianModel.predict().

A batchie.core.BayesianModel must report its variance via batchie.core.BayesianModel.variance().

add_observations(data: ScreenBase)

Add observations to the model.

Parameters
    data – The data to add.

abstract n_obs() → int

Return the number of observations that have been added to the model.

Returns
    Integer number of observations
abstract reset_model()  
Reset the internal state of the model to its initial state.

abstract property rng: numpy.random.Generator  
Return the PRNG for this model instance.

    Returns  
The PRNG for this model instance.

abstract set_rng(rng: numpy.random.Generator)  
Set the PRNG for this model instance.

    Parameters  
rng – The PRNG to use.

class batchie.core.DistanceMatrix  
Bases: ABC

abstract add_value(i, j, value)  
Add a value to the distance matrix.

    Parameters  
  • i – The row index.
  • j – The column index.
  • value – The value to add.

abstract classmethod load(filename)  
Load a distance matrix from a file.

    Parameters  
filename – The filename to load from.

abstract save(filename)  
Save the distance matrix to a file.

    Parameters  
filename – The filename to save to.

abstract to_dense()  
Return a dense representation of the distance matrix.

    Returns  
A dense representation of the distance matrix.

class batchie.core.DistanceMetric  
Bases: object

This class represents a symmetric distance metric between two arrays of model predictions.

distance(a: numpy.ndarray, b: numpy.ndarray) → float  
Calculate the distance between two arrays of model predictions.

    Parameters  
  • a – The first array of model predictions.
  • b – The second array of model predictions.

    Returns  
The distance between the two arrays.
class batchie.core.InitialRetrospectivePlateGenerator
Bases: ABC

When running a retrospective active learning simulation, results are sensitive to the initial plate which is revealed. For this reason users might want to implement a special routine for revealing the initial plate separate from the subsequent plates.

generate_and_unmask_initial_plate(screen: Screen, rng: numpy.random.BitGenerator) → Screen
Generate and unmask the initial plate.

Parameters

- screen – A fully observed batchie.data.Screen
- rng – The PRNG to use.

Returns

The same batchie.data.Screen with the initial plate observed, and all other plates masked.

class batchie.core.MCMCModel
Bases: object

This class subclasses BayesianModel and implements batchie.core.MCMCModel.step()

abstract get_model_state() → Theta
Get the internal state of the model.

abstract step()
Advance the internal state of the model by one step.

In the case of an MCMC model, this would mean taking one more MCMC step. Other types of models should implement accordingly.

class batchie.core.Metric(model: BayesianModel)
Bases: object

evaluate(sample: Theta) → float
Evaluate the metric on a single parameter set.

Parameters

- sample – The parameter set to evaluate.

Returns

The value of the metric.

evaluate_all(results_holder: ThetaHolder) → numpy.ndarray
Evaluate the metric on all parameter sets in the results_holder.

Parameters

- results_holder – The parameter sets to evaluate.

Returns

An array of metric values.

class batchie.core.PlatePolicy
Bases: object

Given a batchie.data.Screen, which is a set of potential `batchie.data.Plate`'s, implementations of this class will determine which set of `batchie.data.Plate`'s is eligible for the next round.
filter_eligible_plates(batch_plates: list[Plate], unobserved_plates: list[Plate], rng: numpy.random.Generator) → list[Plate]

class batchie.core.RetrospectivePlateGenerator
Bases: ABC
When running a retrospective active learning simulation, the user might want to reorganize the dataset into different plates than were originally run. This class will generate these plate groupings from the individual observations in the retrospective dataset.
generate_plates(screen: Screen, rng: numpy.random.BitGenerator) → Screen
Generate plates from the remaining unobserved experiments in the input screen.

Parameters
- screen – A partially observed batchie.data.Screen
- rng – The PRNG to use.

class batchie.core.RetrospectivePlateSmoother
Bases: ABC
After plates have been generated for a retrospective simulation using a batchie.core.RetrospectivePlateGenerator, those plates may be of very uneven sizes, which is not desirable. Implementations of this class should aim to merge plates together and/or drop experiments until plate sizes are more even. We call this process “plate smoothing”.
smooth_plates(screen: Screen, rng: numpy.random.BitGenerator) → Screen
Smooth the plates in the screen.

Parameters
- screen – A partially observed batchie.data.Screen
- rng – The PRNG to use.

class batchie.core.Scorer
Bases: object
This class represents a scoring function for batchie.data.Plate instances.
The score should represent how desirable it is to observe the given plate, with a lower score being more desirable.
score(model: BayesianModel, plates: dict[int, ScreenSubset], distance_matrix: DistanceMatrix, samples: ThetaHolder, rng: numpy.random.Generator, progress_bar: bool) → dict[int, float]

class batchie.core.ScoresHolder
Bases: ABC
This class represents a set of scores for a set of plates.
add_score(plate_id: int, score: float)
Add a score for a given plate.

Parameters
- plate_id – The plate id to add the score for.
- score – The score to add.
**get_score**(*plate_id: int*) → float
Get the score for a given plate.

**Parameters**
- **plate_id** – The plate id to get the score for.

**Returns**
- The score for the given plate.

**plate_id_with_minimum_score**(*eligible_plate_ids: list[int] = None*) → int
Get the plate id with the minimum score.

**Parameters**
- **eligible_plate_ids** – The set of plates to consider.

**Returns**
- The plate id with the minimum score.

class **batchie.core.SimulationTracker**(*plate_ids_selected: list[list[int]], losses: list[float], seed: int*)
Bases: object
This class tracks the state of a retrospective active learning simulation. It will record the plates that were revealed at each step and the total loss of the predictor trained on the plates revealed up until that point.

classmethod **load**(*fn*)
Load this instance from a JSON file.

**Parameters**
- **fn** – The filename to load from.

**save**(*fn*)
Save this instance to a JSON file.

**Parameters**
- **fn** – The filename to save to.

class **batchie.core.Theta**
Bases: object
This class represents the set of parameters for a BayesianModel. Should be implemented by a dataclass or similarly serializable class.

**equals**(*other*)

**abstract classmethod from_dicts**(*private_params: dict, shared_params: dict*)
Instantiate **batchie.core.Theta** from dictionary

**Returns**
- a dictionary mapping class variables to arrays/numerical values.

**abstract predict_conditional_mean**(*data: ScreenBase*) → numpy.ndarray
Predict the conditional mean of an **batchie.data.ExperimentBase** in modeling space.

**Returns**
- An array of means for each item in the Experiment.

**abstract predict_conditional_variance**(*data: ScreenBase*) → numpy.ndarray
Predict the conditional variance of an **batchie.data.ExperimentBase**.

**Returns**
- An array of variances for each item in the Experiment.
abstract predict_viability(data: ScreenBase) \rightarrow \text{numpy.ndarray}

Predict the conditional mean of an \texttt{batchie.data.ExperimentBase} in viability space.

Returns
An array of means for each item in the Experiment.

abstract private_parameters_dict() \rightarrow \text{dict[str, numpy.ndarray]}

The private parameters of a \texttt{batchie.core.Theta}.

Returns
a dictionary mapping class variables to arrays/numerical values.

shared_parameters_dict() \rightarrow \text{dict[str, numpy.ndarray]}

The shared parameters of a \texttt{batchie.core.Theta}.

Returns
a dictionary mapping class variables to arrays.

class batchie.core.ThetaHolder(n_thetas: int, *args, **kwargs)

Bases: \texttt{ABC}

This class represents a container for multiple parameter sets for a BayesianModel. This class provides methods to save these parameter sets to an H5 file.

add_theta(theta: Theta)

Add a new parameter set to the container.

Parameters
theta -- The parameter set to add.

combine(other)

Combine these parameters sets with another container of parameter sets.

Parameters
other -- Another ThetaHolder instance.

classmethod concat(instances: list)

Combine multiple instances of ThetaHolder into one.

Parameters
instances -- A list of ThetaHolder instances.

get_theta(step_index: int) \rightarrow \text{Theta}

Returns the parameter set at the given index.

Parameters
step_index -- The index of the parameter set to return.

property is_complete

Returns
True if the container is full, False otherwise.

static load_h5(path: str)

Load a ThetaHolder from an H5 file.

Parameters
path -- The path to the H5 file.
property n_thetas

Returns
The number of parameter sets in the container.

save_h5(fn: str)
Save the parameter sets to an H5 file.

Parameters
fn – The filename to save to.

class batchie.core.VIModel

Bases: object
This class subclasses BayesianModel and implements batchie.core.VIModel.sample()

abstract sample(num_samples: int) → list[Theta]
Returns a list of Theta samples. Length of the list should be num_samples.

1.7.4 batchie.data module

class batchie.data.Plate(screen: Screen, selection_vector: numpy.ndarray)

Bases: ScreenSubset
A subset of an batchie.data.Screen defined by a boolean selection vector
This class is not meant to be instantiated directly, but rather is returned by the batchie.data.Screen.get_plate method.
The difference between a batchie.data.Plate and an batchie.data.ScreenSubset is that a batchie.data.Plate is guaranteed to contain only one unique plate id.

merge(other)
Merge this plate with another plate, mutate the parent batchie.data.Screen in place.

Parameters
other – batchie.data.Plate

property plate_id
Return the plate id of this plate.

Returns
int, plate id

property plate_name
Return the original plate name of this plate.

Returns
str, plate name

class batchie.data.Screen(treatment_names: numpy.ndarray, treatment_doses: numpy.ndarray,
sample_names: numpy.ndarray, plate_names: numpy.ndarray, observations: numpy.ndarray | None = None, observation_mask: numpy.ndarray | None = None, control_treatment_name=":", treatment_mapping: Tuple[numpy.ndarray, numpy.ndarray | None = None], sample_mapping: Tuple[numpy.ndarray, numpy.ndarray | None = None])

Bases: ScreenBase
The principal data structure in batchie.
An `batchie.data.Screen` is a collection of experiments. Some of the experiments may be observed and some may not be observed. Anything not enumerated as an experimental condition in this top level class will be “invisible” to batchie.

An `batchie.data.Screen` can be subset into `batchie.data.Plate`'s or `batchie.data.ScreenSubset` of multiple plates. `batchie.data.Screen` is the only data class that can be subdivided.

**combine**(other)
Union this screen with another screen.

**Warning:** treatment, sample, and plate ids are not guaranteed to be the same in the resulting new screen instance.

**Parameters**
other – `batchie.data.Screen`

**Returns**
Unioned `batchie.data.Screen`

**classmethod concat**(screens: list[Screen])
Concatenate a list of `batchie.data.Screen`'s into a single `batchie.data.Screen`.

**Parameters**
screens – list of `batchie.data.Screen`

**Returns**
Unioned `batchie.data.Screen`

**get_plate**(plate_id: int) → Plate
Return a `batchie.data.Plate` defined by a plate id.

**Parameters**
plate_id – int, plate id

**Returns**
A `batchie.data.Plate`

**static load_h5**(path)
Load screen from h5 archive.

**Parameters**
path – str, path to h5 archive

**property observation_mask**
Return the array of observation masks in the screen. If the array is true, it means the condition is observed, if false it is unobserved.

**Returns**
1d array of observation masks

**property observations**
Return the array of observations in the screen.

We do not use any NaN values in our arrays, the observation value for a condition set where `batchie.data.Screen.observation_mask` is False is undefined. Its up to the user to decide how to handle this.

**Returns**
1d array of observations
property plate_ids
   Return the array of plate ids in the screen.
   Plate ids are always 0 indexed integers from 0 to `batchie.data.ScreenBase.n_unique_plates` - 1 with no gaps.
   Returns
   1d array of plate ids

property plate_mapping: Tuple[numpy.ndarray, numpy.ndarray]
   Returns
   a tuple of two 1d arrays that map plate name to id.

property plates
   Return a list of all :py:class:`batchie.data.Plate` s in the screen.
   Returns
   list of :py:class:`batchie.data.Plate` s

property sample_ids
   Return the array of sample ids in the screen.
   Sample ids are always 0 indexed integers from 0 to `batchie.data.ScreenBase.n_unique_samples` - 1 with no gaps.
   Returns
   1d array of sample ids

property sample_mapping: Tuple[numpy.ndarray, numpy.ndarray]
   Returns
   a tuple of two 1d arrays that map sample name to id.

property sample_names
   Return the array of sample names (provided string names)
   Returns
   1d array of sample names

save_h5(fn)
   Save screen to h5 archive.

   Parameters
   fn – str, path to h5 archive

set_observed(selection_mask: numpy.ndarray, observations: numpy.ndarray)

property single_treatment_effects: numpy.ndarray | None
   Return the array of single treatment effects in the screen.
   Returns
   2d array of single treatment effects

subset(selection_vector: numpy.ndarray) → ScreenSubset
   Return a `batchie.data.ScreenSubset` defined by a boolean selection vector.

   Parameters
   selection_vector – 1d array of bools

   Returns
   `batchie.data.ScreenSubset`
**subset_observed() → ScreenSubset | None**

Return a `batchie.data.ScreenSubset` containing all conditions that are observed. Returns none if all conditions are unobserved.

**Returns**

`batchie.data.ScreenSubset`

**subset_unobserved() → ScreenSubset | None**

Return a `batchie.data.ScreenSubset` containing all conditions that are not observed. Returns none if `batchie.data.Screen.is_observed` is True.

**Returns**

`batchie.data.ScreenSubset`

**property treatment_doses**

Return the array of treatment doses (floating point drug concentrations)

**Returns**

N-dimension array of treatment doses

**property treatment_ids**

Return the array of treatment ids in the screen.

Treatment ids are always 0 indexed integers from 0 to `batchie.data.ScreenBase.n_unique_treatments` - 1 with no gaps.

**Returns**

2d array of treatment ids

**property treatment_mapping: Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]**

**Returns**

a tuple of three 1d arrays that map tuples of (name, dose) to id.

**property treatment_names**

Return the array of treatment names (provided drug names)

**Returns**

N-dimension array of treatment names

**class batchie.data.ScreenBase**

**Bases:** ABC

Base class for the principal data structure in batchie.

An `batchie.data.Screen` is a collection of experimental conditions, and optionally observations for those of those conditions. The conditions are defined by a set of treatment names and doses, and a set of sample names. Observations are scalar floating point numbers, with one scalar per condition.

`batchie.data.Screen` class also defines the concept of a plate, which is a grouping of experimental conditions. The terminology plate comes from the world of high throughput biological screening, where plastic plates with 96, 384, or 1536 individual wells are used to hold distinct biochemical reactions. In batchie, this concept is abstracted to the concept of a plate being the discrete unit of experimental conditions that can be observed at one time. We also abstract away the concept of the plate having to be a fixed size each time.

**abstract combine(other)**

**property is_observed:** bool

Return True if all observations are available, False otherwise

**Returns**

bool
property n_plates
    Return the number of plates in the screen.

    Returns
    int, number of plates

property n_unique_samples
    Return the number of unique samples in the screen.

    Returns
    int, number of unique samples

property n_unique_treatments
    Return the number of unique treatments in the screen.

    Returns
    int, number of unique treatments

abstract property observation_mask

abstract property observations

abstract property plate_ids

abstract property plate_mapping

abstract property sample_ids

abstract property sample_mapping

abstract property sample_names

property sample_space_size
    Return the size of the universe of possible samples.

    Returns
    int

abstract property single_treatment_effects: numpy.ndarray | None

property size
    Return the number of experimental conditions contained in the experiment.

    Returns
    int, number of experimental conditions

property treatment_arity
    Return the number of treatments per experiment.

    Returns
    int, number of treatments per experiment

abstract property treatment_doses

abstract property treatment_ids

abstract property treatment_mapping

abstract property treatment_names
property treatment_space_size
Return the size of the universe of possible treatments.

Returns
int

property unique_plate_ids
Return the unique plate ids in the screen.

Returns
1d array of unique plate ids

property unique_sample_ids
Return the unique sample ids in the screen.

Returns
1d array of unique sample ids

property unique_treatments
Return the unique treatments in the screen (excludes “control” treatments).

Returns
2d array of unique treatments

class batchie.data.ScreenSubset(screen: Screen, selection_vector: numpy.ndarray)
Bases: ScreenBase
A subset of an batchie.data.Screen defined by a boolean selection vector.
This class is not meant to be instantiated directly, but rather is returned by the batchie.data.Screen subset() method.

combine(other)
Union this subset with another subset of the same screen.

Parameters
other – batchie.data.ScreenSubset

Returns
Unioned batchie.data.ScreenSubset

classmethod concat(screen_subsets: list)
Concatenate a list of batchie.data.ScreenSubset`s into a single :py:class:`batchie.data.ScreenSubset.

Parameters
screen_subsets – list of batchie.data.ScreenSubset

Returns
Unioned batchie.data.ScreenSubset

property control_treatment_name
invert()
Return the inverse of this subset, i.e. the subset of the screen that is not contained in this subset.

Returns
batchie.data.ScreenSubset

property observation_mask
property observations
property plate_ids
property plate_mapping
property sample_ids
property sample_mapping
property sample_names
property single_treatment_effects: numpy.ndarray | None

subset(selection_vector)
Return a new batchie.data.ScreenSubset defined by a boolean selection vector.

Parameters
selection_vector – 1d array of bools

Returns
batchie.data.ScreenSubset
to_screen()  
Promote this subset to an batchie.data.Screen.

Returns
batchie.data.Screen

property treatment_doses
property treatment_ids
property treatment_mapping
property treatment_names

batchie.data.create_single_treatment_effect_array(sample_ids: numpy.ndarray, treatment_ids: numpy.ndarray, observation: numpy.ndarray)
Create a n_observation x n_treatment array where each entry is the single treatment effect for the corresponding sample and treatment ids in the input arrays.

Parameters
• sample_ids – 1d array of sample ids
• treatment_ids – 2d array of treatment ids
• observation – 1d array of observations

batchie.data.create_single_treatment_effect_map(sample_ids: numpy.ndarray, treatment_ids: numpy.ndarray, observation: numpy.ndarray)
Create a map from (sample_id, treatment_id) to single observation (a scalar).

Parameters
• sample_ids – 1d array of sample ids
• treatment_ids – 1d array of treatment ids
• observation – 1d array of observations
batchie.data.encode_1d_array_to_0_indexed_ids(arr: numpy.ndarray, existing_mapping: Tuple[numpy.ndarray, numpy.ndarray] | None = None)

Encode a 1d array of strings to 0-indexed integers.

Parameters
- **arr** – 1d array of strings
- **existing_mapping** – Prior mapping

Returns
integer array containing only values between 0 and n-1, where n is the number of unique values in arr

batchie.data.encode_treatment_arrays_to_0_indexed_ids(treatment_name_arr: numpy.ndarray, treatment_dose_arr: numpy.ndarray, control_treatment_name: str = batchie.common.CONTROL_SENTINEL_VALUE, existing_mapping: Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray] | None = None)

Encode treatment names and doses (which are arrays of string) to 0-indexed integers, where the control treatment is always mapped to batchie.common.CONTROL_SENTINEL_VALUE

Parameters
- **treatment_name_arr** – array of treatment names
- **treatment_dose_arr** – array of treatment doses
- **control_treatment_name** – The string value of the control treatment
- **existing_mapping** – Prior mapping

batchie.data.filter_dataset_to_treatments_that_appear_in_at_least_one_combo(screen: Screen)

Utility function to filter down an batchie.data.Screen to only the treatments that appear in at least one combo.

Parameters
- **screen** – an batchie.data.Screen

Returns
A filtered batchie.data.Screen

batchie.data.filter_dataset_to_unique_treatments(screen: Screen | ScreenSubset)

Ensure that the dataset only has one experiment per treatment and sample condition by arbitrarily dropping duplicates.

Parameters
- **screen** – an batchie.data.ScreenSubset

Returns
A batchie.data.ScreenSubset with the same or smaller number of experiments compared to the input.

batchie.data.numpy_array_is_0_indexed_integers(arr: numpy.ndarray)

Test numpy array arr contains only integers between 0 and n-1 with no gaps, where n is the number of unique values in arr.

If the array contains batchie.common.CONTROL_SENTINEL_VALUE, then we test that the array contains only integers between 0 and n-2, and the sentinel value.
Parameters
- **arr** – numpy array

Returns
- bool

### 1.7.5 batchie.distance_calculation module

class batchie.distance_calculation.ChunkedDistanceMatrix(size, n_chunks=1, chunk_index=0, chunk_size=None)

Bases: `DistanceMatrix`

Class which can represent part or a whole pairwise distance matrix.

The distance matrix is stored in a sparse format, but can be converted to a dense format if all values are present.

Several partial ChunkedDistanceMatrix classes can be combined. This is useful for parallelization of the distance matrix computation.

**add_value(i, j, value)**

Add a value to the distance matrix.

**Parameters**
- **i** – The row index.
- **j** – The column index.
- **value** – The value to add.

**combine(other)**

**classmethod concat(matrices: list)**

**is_complete()**

**classmethod load(filename)**

Load a distance matrix from a file.

**Parameters**
- **filename** – The filename to load from.

**save(filename)**

Save the distance matrix to a file.

**Parameters**
- **filename** – The filename to save to.

**to_dense()**

Return a dense representation of the distance matrix.

**Returns**
- A dense representation of the distance matrix.
Calculate the pairwise distance matrix between predictions in viability space.

For all pairs of thetas in the given `ThetaHolder`, predictions will be made on the unobserved conditions in the given `Experiment` and the distance between the predictions produced by the two theta values will be calculated and populated into a `ChunkedDistanceMatrix` instance.

If \( n_{\text{chunks}} > 1 \), then the distance matrix is split into \( n_{\text{chunks}} \) roughly equal chunks, and only the chunk with index \( \text{chunk\_index} \) is calculated. This is useful for parallelization.

**Parameters**

- `model` – The model to use for prediction
- `thetas` – The set of model parameters to use for prediction
- `distance_metric` – The distance metric to use
- `data` – The data to predict
- `chunk_index` – The index of the chunk to calculate
- `n_chunks` – The number of chunks to split the distance matrix into
- `progress` – Whether to show a progress bar

**Returns**

A `ChunkedDistanceMatrix` containing the pairwise distances

```python
batchie.distance_calculation.consume(iterator, n)
```

Advance the iterator \( n \)-steps ahead. If \( n \) is none, consume entirely.

**Parameters**

- `iterator` – The iterator to consume
- `n` – The number of steps to advance the iterator

```python
batchie.distance_calculation.get_lower_triangular_indices_chunk(n: int, chunk_index: int, n_chunks: int)
```

Assuming we want to split the number of lower triangular indices of a square matrix with dimension \( n \) into roughly equal chunks, return the indices for the chunk with index \( \text{chunk\_index} \)

**Parameters**
- \text{n} – The dimension of the square matrix
- \text{chunk\_index} – The index of the chunk to return
- \text{n\_chunks} – The number of chunks to split the indices into

\textbf{Returns}
A list of indices

\text{batchie.distance\_calculation.get\_number\_of\_lower\_triangular\_indices}(n: \text{int})
Get the number of lower triangular indices of a square matrix with dimension \text{n}

\textbf{Parameters}
- \text{n} – The dimension of the square matrix

\textbf{Returns}
The number of lower triangular indices

\text{batchie.distance\_calculation.lower\_triangular\_indices}(n: \text{int})
Iterate all the lower triangular indices of a square matrix with dimension \text{n}

\textbf{Parameters}
- \text{n} – The dimension of the square matrix

\textbf{Returns}
A generator which yields the indices

\subsection*{1.7.6 \texttt{batchie.fast\_mvn} module}
Methods for sampling from multivariate normal distributions.

\text{batchie.fast\_mvn.sample\_mvn\_from\_precision}(Q, mu=None, mu\_part=None, chol\_factor=False, \text{rng}=None)
Fast sampling from a multivariate normal with precision parameterization.
Supports sparse arrays.

\textbf{Parameters}
- \text{Q} – The precision matrix
- \text{mu} – If provided, assumes the model is N(mu, Q^{-1})
- \text{mu\_part} – If provided, assumes the model is N(Q^{-1} mu\_part, Q^{-1})
- \text{chol\_factor} – If true, assumes Q is a (lower triangular) Cholesky decomposition of the precision matrix

\subsection*{1.7.7 \texttt{batchie.introspection} module}

\text{batchie.introspection.create\_instance}(package\_name: \text{str}, class\_name: \text{str}, base\_class: \text{type}, \text{kwargs}: \text{dict})
Create an instance of a class from a package by name.

\textbf{Parameters}
- \text{package\_name} – The name of the package to search.
- \text{class\_name} – The name of the class to search for.
• **base_class** – The base class that the class should inherit from.
• **kwargs** – Keyword arguments to pass to the class constructor.

`batchie.introspection.get_class(package_name: str, class_name: str, base_class: type) → type`

Get a class from a package by name.

**Parameters**

• **package_name** – The name of the package to search.
• **class_name** – The name of the class to search for.
• **base_class** – The base class that the class should inherit from.

`batchie.introspection.get_required_init_args_with_annotations(cls) → Dict[str, Any]`

Get a dictionary of required __init__ arguments and their type annotations for a given class.

**Parameters**

`cls` – The class to inspect.

**Returns**

A dictionary with argument names as keys and their type annotations as values.

### 1.7.8 batchie.log_config module

`batchie.log_config.add_logging_args(parser)`

`batchie.log_config.configure_logging(args)`

Configure logging based on the given arguments.

**Parameters**

`args` – Parsed command line arguments.

### 1.7.9 batchie.retrospective module

**class** `batchie.retrospective.BatchieEnsemblePlateSmoother` *(min_size: int, n_iterations: int, min_n_cell_line_plates: int)*

Bases: `RetrospectivePlateSmoother`

Apply the following smoothers in sequence to the input `batchie.data.Screen`:

- MergeMinPlateSmoother
- MergeTopBottomPlateSmoother
- OptimalSizeSmoother
- NPlatePerCellLineSmoother

**class** `batchie.retrospectiveFixedSizeSmoother` *(plate_size: int)*

Bases: `RetrospectivePlateSmoother`

Filter all plates smaller than the given size and randomly truncate all plates larger than a fixed size to the given size.

**class** `batchie.retrospective.MergeMinPlateSmoother` *(min_size: int)*

Bases: `RetrospectivePlateSmoother`

Iteratively combine the smallest two plates for each sample until all plates are above a user specified size.
class batchie.retrospектив.MergeTopBottomPlateSmooother(n_iterations: int)
    Bases: RetrospectivePlateSmoother
    Iteratively combine the largest and smallest plates for each sample. Runs for a user specified number of iterations.

class batchie.retrospектив.NPlatePerCellLineSmooother(min_n_cell_line_plates: int)
    Bases: RetrospectivePlateSmoother
    Remove all experiments involving cell lines which have less than the user specified min_n_cell_line_plates

class batchie.retrospектив.OptimalSizeSmooother
    Bases: RetrospectivePlateSmoother
    The cost function for any particular plate size is the sum of two terms, the first term is the number of experiments you have to completely throw out because they are in plates below the threshold, the second term is the number of experiments that need to be trimmed out of plates that are over the threshold. This smoother optimizes this cost function and then drops all plates smaller than the optimal size and sub-samples all plates larger than the optimal size until all plates are the same size.

class batchie.retrospектив.PairwisePlateGenerator(subset_size: int, anchor_size: int)
    Bases: RetrospectivePlateGenerator

class batchie.retrospектив.PlatePermutationPlateGenerator(force_include_plate_names: list[str] | None = None)
    Bases: RetrospectivePlateGenerator
    This generator will create new plates by permuting the plate labels. Plates can be excluded from permutation with the force_include_plate_names argument

class batchie.retrospектив.SampleSegregatingPermutationPlateGenerator(max_plate_size: int)
    Bases: RetrospectivePlateGenerator
    This generator will generate plates that only contain experiments for a single sample. If there are more than max_plate_size experiments for a single sample then the experiments will be split across multiple equal sized plates.

class batchie.retrospектив.SparseCoverPlateGenerator(reveal_single_treatment_experiments: bool)
    Bases: InitialRetrospectivePlateGenerator

batchie.retrospектив.calculate_mse(observed_screen: Screen, thetas: ThetaHolder) → float
    Calculate the mean squared error between the masked observations and the unmasked observations

    Parameters
    • observed_screen – A Screen that is fully observed
    • thetas – The set of model parameters to use for prediction

    Returns
    The average mean squared error between predicted and observed values

batchie.retrospектив.create_plate_balanced_holdout_set_among_masked_plates(screen: ~batchie.data.Screen, fraction: float, rng: numpy.random.BitGenerator) -> (%class batchie.data.Screen, %class batchie.data.Screen)
Create a holdout set from a retrospective screen (where all data is observed but some plates are artificially masked) by sampling a fraction of each unobserved plate.

**Parameters**
- `screen` – The screen to create a holdout set for
- `fraction` – The fraction of each unobserved plate to hold out

**Returns**
A tuple of (training_screen, holdout_screen)

```python
def create_random_holdout(screen: ~batchie.data.Screen, fraction: float, rng: numpy.random.BitGenerator) -> (<class 'batchie.data.Screen'>, <class 'batchie.data.Screen'>):
    # Create a random subset of a screen, of size fraction of the original screen.
    # Parameters
    # screen – The screen to create a holdout set for
    # fraction – The fraction of the screen to hold out
    # Returns
    # A tuple of (training_screen, holdout_screen)
```

**batchie.retrospective.mask_screen**

```python
def mask_screen(screen: Screen) -> Screen:
    # Utility function to reveal observations in the masked screen from the observed screen.
    # Parameters
    # screen – A `batchie.data.Screen` that is partially masked, but with real observations present in the internal observation array
    # plate_ids – The plate ids to reveal
    # Returns
    # `batchie.data.Screen`
```

**batchie.retrospective.reveal_plates**

```python
def reveal_plates(screen: Screen, plate_ids: list[int]) -> Screen:
    # Utility function to reveal observations in the masked screen from the observed screen.
    # Parameters
    # screen – A `batchie.data.Screen` that is partially masked, but with real observations present in the internal observation array
    # plate_ids – The plate ids to reveal
    # Returns
    # `batchie.data.Screen`
```

### 1.7.10 batchie.sampling module

**batchie.sampling.sample**

```python
def sample(model, results: ThetaHolder, seed: int = None, n_chains: int = None, chain_index: int = None, n_burnin: int = None, thin: int = None, progress_bar=False) -> ThetaHolder:
    # Sample from the model posterior using the given parameters.
    # Parameters
    # model – The model which will be sampled from.
    # results – The object which will store the results
    # seed – The seed to use for the random number generator
    # n_chains – The number of parallel chains to run
    # chain_index – The index of the current chain
    # n_burnin – The number of burnin steps to run
    # thin – The thinning factor
    # progress_bar – Whether to display a progress bar
```


Returns
a `ThetaHolder` containing the sampled parameters

1.7.11 batchie.synergy module

`batchie.synergy.calculate_synergy(sample_ids: numpy.ndarray, treatment_ids: numpy.ndarray, observation: numpy.ndarray, strict: bool = False)`

Calculate synergy for a given set of observations, sample ids, and treatment ids.
If single treatment observations for all of the treatments in a multi-treatment observation are not present, the observation is skipped. If strict is True, an error is raised instead.

1.7.12 batchie.distance.mse module

```python
class batchie.distance.mse.MSEDistance(sigmoid: bool = True)
    Bases: DistanceMetric
    Mean squared error distance metric
    distance(a: numpy.ndarray, b: numpy.ndarray)
    Calculate the distance between two arrays of model predictions.
    Parameters
    • a – The first array of model predictions.
    • b – The second array of model predictions.
    Returns
    The distance between the two arrays.
```

1.7.13 batchie.models.sparse_combo module

```python
class batchie.models.sparse_combo.LegacySparseDrugComboImpl(n_dims: int, n_drugdoses: int, n_clines: int, intercept: bool = True, fake_intercept: bool = True, individual_eff: bool = True, mult_gamma_proc: bool = True, local_shrinkage: bool = True, a0: float = 1.1, b0: float = 1.1, min_Mu: float = -10.0, max_Mu: float = 10.0, **kwargs)
    Bases: object
    Original implementation of Bayesian tensor factorization model for predicting combination drug response. Preserved here without changes to ensure reproducibility of results.
    bliss(cline: numpy.ndarray, dd1: numpy.ndarray, dd2: numpy.ndarray)
    encode_obs()
    ess_pars()
    get(attr, ix)```
mcmc_step() \rightarrow \text{None}

n_obs()

predict(cline: numpy.ndarray, dd1: numpy.ndarray, dd2: numpy.ndarray)

predict_single_drug(cline: numpy.ndarray, dd1: numpy.ndarray)

reset_model()

class batchie.models.sparse_combo.SparseDrugCombo(n_embedding_dimensions: int,
    n_unique_treatments: int, n_unique_samples: int,
    fake_intercept: bool = True, individual_eff: bool = True,
    mult_gamma_proc: bool = True,
    local_shrinkage: bool = True, a0: float = 1.1, b0: float = 1.1,
    min_Mu: float = -10.0, max_Mu: float = 10.0, rng: numpy.random.Generator | None = None,
    predict_interactions: bool = False,
    interaction_log_transform: bool = True, intercept: bool = True)

Bases: 

get_model_state() \rightarrow \text{SparseDrugComboMCMCSample}
    Get the internal state of the model.

n_obs() \rightarrow \text{int}
    Return the number of observations that have been added to the model.

    Returns
    Integer number of observations

reset_model()
    Reset the internal state of the model to its initial state.

property rng: numpy.random.Generator
    Return the PRNG for this model instance.

    Returns
    The PRNG for this model instance.

set_rng(rng: numpy.random.Generator)
    Set the PRNG for this model instance.

    Parameters
    rng -- The PRNG to use.

step()
    Advance the internal state of the model by one step.

    In the case of an MCMC model, this would mean taking one more MCMC step. Other types of models
    should implement accordingly.

class batchie.models.sparse_combo.SparseDrugComboMCMCSample(W: numpy.ndarray, W0:
    numpy.ndarray, V2: numpy.ndarray, V1: numpy.ndarray, V0:
    numpy.ndarray, alpha: float, precision: float)

Bases: \text{Theta}

A single sample from the MCMC chain for the sparse drug combo model
V0: numpy.ndarray
V1: numpy.ndarray
V2: numpy.ndarray
W: numpy.ndarray
W0: numpy.ndarray
alpha: float

classmethod from_dicts(private_params, shared_params)
Instantiates batchie.core.Theta from dictionary

Returns
a dictionary mapping class variables to arrays/numerical values.

precision: float

predict_conditional_mean(data: ScreenBase) → numpy.ndarray
Predict the conditional mean of an batchie.data.ExperimentBase in modeling space.

Returns
An array of means for each item in the Experiment.

predict_conditional_variance(data: ScreenBase) → numpy.ndarray
Predict the conditional variance of an batchie.data.ExperimentBase.

Returns
An array of variances for each item in the Experiment.

predict_viability(data: ScreenBase) → numpy.ndarray
Predict the conditional mean of an batchie.data.ExperimentBase in viability space.

Returns
An array of means for each item in the Experiment.

private_parameters_dict() → dict[str, numpy.ndarray]
The private parameters of a batchie.core.Theta.

Returns
a dictionary mapping class variables to arrays/numerical values.

batchie.models.sparse_combo.interactions_to_logits(interaction: numpy.ndarray, single_effects: numpy.ndarray, log_transform: bool)

batchie.models.sparse_combo.predict(mcmc_sample: SparseDrugComboMCMCSample, data: ScreenBase, viability: bool)

batchie.models.sparse_combo.predict_single_drug(mcmc_sample: SparseDrugComboMCMCSample, data: ScreenBase, viability: bool)

1.7. batchie package
1.7.14 batchie.models.sparse_combo_interaction module

```python
class batchie.models.sparse_combo_interaction.LegacySparseDrugComboInteractionImpl:
  n_dims: int,
  n_drugdoses: int,
  n_clines: int,
  mult_gamma_proc: bool = True,
  local_shrinkage: bool = True,
  a0: float = 1.1, b0: float = 1.1,
  min_Mu: float = -10.0, max_Mu: float = 10.0)
```

Bases: object

This is the original implementation of the sparse drug combo interaction model. Preserved here without changes to ensure reproducibility of results.

```python
class encode_obs()
```

```python
mcmc_step() → None
```

```python
class n_obs()
```

```python
reset_model()
```

```python
class batchie.models.sparse_combo_interaction.SparseDrugComboInteraction:
  n_embedding_dimensions: int,
  n_unique_treatments: int,
  n_unique_samples: int,
  mult_gamma_proc: bool = True,
  local_shrinkage: bool = True, a0: float = 1.1, b0: float = 1.1,
  min_Mu: float = -10.0, max_Mu: float = 10.0)
```

Bases: BayesianModel, MCMCModel
get_model_state() → SparseDrugComboInteractionMCMCSample
Get the internal state of the model.

n_obs() → int
Return the number of observations that have been added to the model.

    Returns
    Integer number of observations

reset_model()
Reset the internal state of the model to its initial state.

property rng: numpy.random.Generator
Return the PRNG for this model instance.

    Returns
    The PRNG for this model instance.

set_rng(rng: numpy.random.Generator)
Set the PRNG for this model instance.

    Parameters
    rng – The PRNG to use.

step()
Advance the internal state of the model by one step.
In the case of an MCMC model, this would mean taking one more MCMC step. Other types of models
should implement accordingly.

variance(data: ScreenBase) → float


Bases: Theta
A single sample from the MCMC chain for the sparse drug combo model

V2: numpy.ndarray
W: numpy.ndarray

classmethod from_dicts(private_params: dict, shared_params: dict)
Instantiate batchie.core.Theta from dictionary

    Returns
    a dictionary mapping class variables to arrays/numerical values.

precision: float
predictconditional_mean(data: ScreenBase) \rightarrow \text{nump.ndarray}
Predict the conditional mean of an `batchie.data.ExperimentBase` in modeling space.

Returns
An array of means for each item in the Experiment.

predictconditional_variance(data: ScreenBase) \rightarrow \text{nump.ndarray}
Predict the conditional variance of an `batchie.data.ExperimentBase`.

Returns
An array of variances for each item in the Experiment.

predictviability(data: ScreenBase) \rightarrow \text{nump.ndarray}
Predict the conditional mean of an `batchie.data.ExperimentBase` in viability space.

Returns
An array of means for each item in the Experiment.

privateparameters_dict() \rightarrow \text{dict[str, nump.ndarray]}
The private parameters of a `batchie.core.Theta`.

Returns
a dictionary mapping class variables to arrays/numerical values.

sharedparameters_dict() \rightarrow \text{dict[str, nump.ndarray]}
The shared parameters of a `batchie.core.Theta`.

Returns
a dictionary mapping class variables to arrays.

single_effect_lookup: dict

1.7.15 batchie.policies.k_per_sample module

class batchie.policies.k_per_sample.KPerSamplePlatePolicy(k: int)
Bases: PlatePolicy

filter_eligible_plates(batch_plates: list[Plate], unobserved_plates: list[Plate], rng: nump.random.Generator) \rightarrow list[Plate]

1.7.16 batchie.scoring.gaussian_dbal module

class batchie.scoring.gaussian_dbal.GaussianDBALScorer(max_chunk=50, max_triples=5000, **kwargs)
Bases: Scorer

score(model: BayesianModel, plates: dict[int, ScreenSubset], distance_matrix: ChunkedDistanceMatrix, samples: ThetaHolder, rng: nump.random.Generator, progress_bar: bool) \rightarrow dict[int, float]

Compute the Monte Carlo approximation of the DBAL ideal score $\hat{s}_n(P)$ in a vectorized way for each of the given plates.

$$\hat{s}_n(P) = \frac{1}{\binom{n}{3}} \sum_{i<j<k} d(\theta_i, \theta_j)L_{\theta_i}(\theta_j, \theta_k; P)e^{2H_{\theta_i}(P)}$$

**Parameters**

- **predictions** – model predictions over all plates of shape (n_plates, n_thetas, n_experiments)
- **variances** – an array of variances for model predictions over all plates, of size (n_plate, n_thetas, max_n_experiments). For plates smaller than the maximum size, the variances should be padded with NaNs up to the maximum size.
- **distance_matrix** – a square array of shape (n_thetas, n_thetas) of distances between model parameterizations
- **rng** – PRNG
- **max_combos** – the maximum number of theta triplets to sample
- **distance_factor** – a multiplicative factor for the distance matrix

**Returns**

an array of shape (n_plates,) of approximated scores for each plate in per_plate_predictions

`batchie.scoring.gaussian_dbal.dbal_fast_gaussian_scoring_heteroscedastic(per_plate_predictions: list[numpy.ndarray], variances: list[numpy.ndarray], distance_matrix: numpy.ndarray, rng: numpy.random.Generator, max_combos: int = 5000, distance_factor: float = 1.0)`

`batchie.scoring.gaussian_dbal.dbal_fast_gaussian_scoring_homoscedastic(per_plate_predictions: list[numpy.ndarray], variances: numpy.ndarray, distance_matrix: numpy.ndarray, rng: numpy.random.Generator, max_combos: int = 5000, distance_factor: float = 1.0)`
- **distance_matrix** – a square array of shape (n_thetas, n_thetas) of distances between model parameterizations
- **rng** – PRNG
- **max_combos** – the maximum number of theta triplets to sample
- **distance_factor** – a multiplicative factor for the distance matrix

**Returns**

an array of shape (n_plates,) of approximated scores for each plate in per_plate_predictions

batchie.scoring.gaussian_dbal.generate_combination_at_sorted_index(index, n, k)

Generate all range(n) choose k combinations.
Represent each combination as a descending sorted tuple.
Sort all the tuples is ascending order, and return the tuple that would be found at index.
Do this without materializing the actual list of combinations.

**Parameters**

- **index** – The index of the combination to return
- **n** – The number of items to choose from
- **k** – The number of items to choose

**Returns**

A tuple of length k representing the combination

batchie.scoring.gaussian_dbal.get_combination_at_sorted_index(index, n, k)

batchie.scoring.gaussian_dbal.pad_ragged_arrays_to_dense_array(arrays: list[numpy.ndarray], pad_value: float = 0.0)

Given a list of arrays, each with N dimensions, each of which have different sizes, return a dense array of N + 1 dimensions, of size (len(array), maximum_of_dimension_0, … maximum_of_dimension_N) where all the arrays are padded to the maximum size. Padding value defaults to 0.0.

**Parameters**

- **arrays** – A list of arrays

- **pad_value** – A floating point number (default is 0)

**Returns**

A dense array of the arrays

### 1.7.17 batchie.scoring.main module

**class** batchie.scoring.main.ChunkedScoresHolder(size: int)

**Bases:** ScoresHolder

**add_score**(plate_id: int, score: float)

Add a score for a given plate.

**Parameters**

- **plate_id** – The plate id to add the score for.
- **score** – The score to add.
.. classmethod:: concat(scores_list: list[ScoresHolder])

.. classmethod:: get_score(plate_id: int) \rightarrow float

    Get the score for a given plate.

    Parameters
    ----------
    plate_id : int
        The plate id to get the score for.

    Returns
    -------
    The score for the given plate.

.. classmethod:: load_h5(fn)

.. method:: plate_id_with_minimum_score(eligible_plate_ids: list[int] = None) \rightarrow int

    Get the plate id with the minimum score.

    Parameters
    ----------
    eligible_plate_ids : list[int], optional
        The set of plates to consider.

    Returns
    -------
    The plate id with the minimum score.

.. method:: save_h5(fn)

.. function:: batchie.scoring.main.score_chunk(model: BayesianModel, scorer: Scorer, thetas: ThetaHolder, screen: Screen, distance_matrix: ChunkedDistanceMatrix, rng: numpy.random.Generator | None = None, progress_bar: bool = False, n_chunks: int = 1, chunk_index: int = 0, batch_plate_ids: list[int] | None = None) \rightarrow ChunkedScoresHolder

    Score a subset of all unobserved plates in a screen.

    Parameters
    ----------
    model : BayesianModel
        The model to use for scoring
    scorer : Scorer
        The scorer to use for scoring
    thetas : ThetaHolder
        The samples to use for scoring
    screen : Screen
        The screen to score
    distance_matrix : ChunkedDistanceMatrix
        The distance matrix to use for scoring
    rng : PRNG
        PRNG to use for sampling
    progress_bar : bool
        Whether to show a progress bar
    n_chunks : int
        The number of chunks to split the unobserved plates into
    chunk_index : int
        The index of the chunk to score
    batch_plate_ids : list[int], optional
        A list of plate ids that have already been selected in the batch

    Returns
    -------
    ChunkedScoresHolder containing the scores for each plate in the current chunk

.. function:: batchie.scoring.main.select_next_plate(scores: ScoresHolder, screen: Screen, policy: PlatePolicy | None, batch_plate_ids: list[int] | None = None, rng: numpy.random.Generator | None = None) \rightarrow Plate | None

    Select the next :class:`batchie.data.Plate` to observe

    Parameters
batchie, Release 0.1

- **scores** – The scores for each plate
- **screen** – The screen which defines the set of plates to choose from
- **policy** – The policy to use for plate selection
- **batch_plate_ids** – The plates currently selected in the batch
- **rng** – PRNG to use for sampling

**Returns**
A list of plates to observe

### 1.7.18 batchie.scoring.size module

class batchie.scoring.size.SizeScorer
Bases: Scorer

A scorer that returns the number of conditions in the Plate as the score.

score(model: BayesianModel, plates: dict[int, Plate], distance_matrix: ChunkedDistanceMatrix, samples: ThetaHolder, rng: numpy.random.Generator, progress_bar: bool) → dict[int, float]

### 1.7.19 batchie.scoring.rand module

class batchie.scoring.rand.RandomScorer
Bases: Scorer

A scorer that returns a random score for each plate, used for baseline comparison

score(model: BayesianModel, plates: dict[int, Plate], distance_matrix: ChunkedDistanceMatrix, samples: ThetaHolder, rng: numpy.random.Generator, progress_bar: bool) → dict[int, float]
INDICES AND TABLES

- genindex
- modindex
- search
batchie.common, 11
batchie.core, 11
batchie.data, 17
batchie.distance.mse, 31
batchie.distance_calculation, 25
batchie.fast_mvn, 27
batchie.introspection, 27
batchie.log_config, 28
batchie.models.sparse_combo, 31
batchie.models.sparse_combo_interaction, 34
batchie.policies.k_per_sample, 36
batchie.retrospective, 28
batchie.sampling, 30
batchie.scoring.gaussian_dbal, 36
batchie.scoring.main, 38
batchie.scoring.rand, 40
batchie.scoring.size, 40
batchie.synergy, 31
A

add_logging_args() (in module batchie.log_config), 28
add_observations() (batchie.core.BayesianModel method), 11
add_score() (batchie.core.ScoresHolder method), 14
add_score() (batchie.scoring.main.ChunkedScoresHolder method), 38
add_theta() (batchie.core.ThetaHolder method), 16
add_value() (batchie.core.DistanceMatrix method), 12
add_value() (batchie.distance_calculation.ChunkedDistanceMatrix method), 25
alpha (batchie.models.sparse_combo.SparseDrugComboMCMCSample attribute), 33

batchie.scoring.gaussian_dbal module, 36
batchie.scoring.main module, 38
batchie.scoring.rand module, 40
batchie.scoring.size module, 40
batchie.synergy module, 31
BatchieEnsemblePlateSmoother (class in batchie.retrospective), 28
BayesianModel (class in batchie.core), 11
bliss() (batchie.models.sparse_combo.LegacySparseDrugComboImpl method), 31

B

batchie.common module, 11
batchie.core module, 11
batchie.data module, 17
batchie.distance.mse module, 31
batchie.distance_calculation module, 25
batchie.fast_mvn module, 27
batchie.introspection module, 27
batchie.log_config module, 28
batchie.models.sparse_combo module, 31
batchie.models.sparse_combo_interaction module, 34
batchie.policies.k_per_sample module, 36
batchie.retrospective module, 28
batchie.sampling module, 30

C

calculate_mse() (in module batchie.retrospective), 29
calculate_pairwise_distance_matrix_on_predictions() (in module batchie.distance_calculation), 25
calculate_synergy() (in module batchie.synergy), 31
ChunkedDistanceMatrix (class in batchie.distance_calculation), 25
ChunkedScoresHolder (class in batchie.scoring.main), 38
combine() (batchie.core.ThetaHolder method), 16
combine() (batchie.data.Screen method), 18
combine() (batchie.data.ScreenBase method), 20
combine() (batchie.data.ScreenSubset method), 22
combine() (batchie.distance_calculation.ChunkedDistanceMatrix method), 25
combine() (batchie.scoring.main.ChunkedScoresHolder method), 38
concat() (batchie.core.ThetaHolder class method), 16
concat() (batchie.data.Screen class method), 18
concat() (batchie.data.ScreenSubset class method), 22
concat() (batchie.distance_calculation.ChunkedDistanceMatrix class method), 25
concat() (batchie.scoring.main.ChunkedScoresHolder class method), 39
configure_logging() (in module batchie.log_config), 28
predict() (batchie.models.sparse_combo.SparseDrugComboMCMCSample, 34)
reset_model() (batchie.models.sparse_combo_interaction.LegacySparseDrugComboInteractionMCMCSample, 34)
predict() (batchie.models.sparse_combo_interaction.SparseDrugComboInteractionMCMCSample, 34)
reset_model() (batchie.models.sparse_combo_interaction.SparseDrugComboInteractionMCMCSample, 34)
predict() (batchie.models.sparse_combo.LegacySparseDrugComboInteractionMCMCSample, 34)
reset_model() (batchie.models.sparse_combo_interaction.LegacySparseDrugComboInteractionMCMCSample, 34)
predict() (in module batchie.models.sparse_combo, 33)
predict_conditional_mean() (batchie.core.Theta method, 15)
predict_conditional_variance() (batchie.models.sparse_combo.SparseDrugComboMCMCSample, 33)
predict_viability() (batchie.models.sparse_combo.SparseDrugComboInteractionMCMCSample, 35)
predict() (in module batchie.models.sparse_combo, 33)
predict_conditional_mean() (batchie.core.Theta method, 15)
predict_conditional_variance() (batchie.models.sparse_combo.SparseDrugComboMCMCSample, 33)
predict_viability() (batchie.models.sparse_combo.SparseDrugComboInteractionMCMCSample, 35)
predict() (in module batchie.models.sparse_combo, 33)
predict_conditional_mean() (batchie.core.Theta method, 15)
predict_conditional_variance() (batchie.models.sparse_combo.SparseDrugComboMCMCSample, 33)
predict_viability() (batchie.models.sparse_combo.SparseDrugComboInteractionMCMCSample, 35)
predict() (in module batchie.models.sparse_combo, 33)
predict_conditional_mean() (batchie.models.sparse_combo.SparseDrugComboMCMCSample, 33)
predict_viability() (batchie.models.sparse_combo.SparseDrugComboInteractionMCMCSample, 35)

**R**

RandomScorer (class in batchie.scoring.rand, 40)
reset_model() (batchie.core.BayesianModel method, 11)
reset_model() (batchie.models.sparse_combo.LegacySparseDrugComboInteractionMCMCSample, 32)
reset_model() (batchie.models.sparse_combo.SparseDrugComboMCMCSample, 32)
ScreenSubset (class in batchie.data), 22
select_next_plate() (in module batchie.scoring.main), 39
select_unique_zipped_numpy_arrays() (in module batchie.common), 11
set_rng() (batchie.data.Screen method), 19
set_rng() (batchie.core.BayesianModel method), 12
set_rng() (batchie.models.sparse_combo.SparseDrugCombo method), 32
set_rng() (batchie.models.sparse_combo_interaction.SparseDrugComboInteraction method), 35
shared_parameters_dict() (batchie.core.Theta method), 16
shared_parameters_dict() (batchie.models.sparse_combo_interaction.SparseDrugComboInteraction method), 36
SimulationTracker (class in batchie.core), 15
single_effect_lookup (batchie.models.sparse_combo_interaction.SparseDrugComboInteraction attribute), 36
single_treatment_effects (batchie.data.Screen property), 19
single_treatment_effects (batchie.data.ScreenBase property), 21
single_treatment_effects (batchie.data.ScreenBase property), 23
size (batchie.data.ScreenBase property), 21
SizeScorer (class in batchie.scoring.size), 40
smooth_plates() (batchie.core.RetrospectivePlateSmother method), 14
SparseCoverPlateGenerator (class in batchie.retropective), 29
SparseDrugCombo (class in batchie.models.sparse_combo), 32
SparseDrugComboInteraction (class in batchie.models.sparse_combo_interaction), 34
SparseDrugComboInteractionMCMCSample (class in batchie.models.sparse_combo_interaction), 35
SparseDrugComboMCMCSample (class in batchie.models.sparse_combo), 32
step() (batchie.core.MCMCModel method), 13
step() (batchie.models.sparse_combo.SparseDrugCombo method), 32
step() (batchie.models.sparse_combo_interaction.SparseDrugComboInteraction method), 35
subset() (batchie.data.Screen method), 19
subset() (batchie.data.ScreenSubset method), 23
subset_observed() (batchie.data.Screen method), 19
subset_unobserved() (batchie.data.Screen method), 20
Theta (class in batchie.core), 15
ThetaHolder (class in batchie.core), 16
to_dense() (batchie.core.DistanceMatrix method), 12
to_dense() (batchie.distance_calculation.ChunkedDistanceMatrix method), 25
to_screen() (batchie.data.ScreenSubset method), 23
treatment_arity (batchie.data.ScreenBase property), 21
treatment_doses (batchie.data.Screen property), 20
treatment_doses (batchie.data.Screen property), 20
treatment_doses (batchie.data.ScreenSubset property), 23
treatment_ids (batchie.data.Screen property), 20
treatment_ids (batchie.data.ScreenBase property), 21
treatment_ids (batchie.data.ScreenBase property), 21
treatment_mapping (batchie.data.ScreenBase property), 23
treatment_names (batchie.data.Screen property), 20
treatment_names (batchie.data.ScreenBase property), 21
treatment_names (batchie.data.ScreenBase property), 23
treatment_space_size (batchie.data.ScreenBase property), 21
unique_plate_ids (batchie.data.ScreenBase property), 22
unique_sample_ids (batchie.data.ScreenBase property), 22
unique_treatments (batchie.data.ScreenBase property), 22
unmask_screen() (in module batchie.retropective), 30
U
V
V0 (batchie.models.sparse_combo.SparseDrugComboMCMCSample attribute), 32
V1 (batchie.models.sparse_combo.SparseDrugComboMCMCSample attribute), 33
V2 (batchie.models.sparse_combo.SparseDrugComboMCMCSample attribute), 33
V2 (batchie.models.sparse_combo_interaction.SparseDrugComboInteraction attribute), 35
variance() (batchie.models.sparse_combo_interaction.SparseDrugComboInteraction method), 35
VIModel (class in batchie.core), 17
W
W (batchie.models.sparse_combo.SparseDrugComboMCMCSample attribute), 33
W (batchie.models.sparse_combo_interaction.SparseDrugComboInteractionMCMCSample attribute), 35
W₀ (batchie.models.sparse_combo.SparseDrugComboMCMCSample attribute), 33