Maccabee

Introduction

Maccabee provides a new mechanism for benchmarking causal inference methods. This method uses sampled Data Generating Processes defined over empirical covariates to provide more realistic, less biased evaluation of estimator performance across the distributional problem space of causal inference.

If you're ready to get started see Installing Maccabee and Quick Start Tutorials. If you'd like to learn more about the motivation and design of Maccabee, see the Design and Implementation page.

Package Design Principles

Fundamentally, this package only succeeds if it provides a useful and usable way to benchmark new methods for causal inference developed by its users. Maccabee's features are focused around four design principles to achieve this end:

- **Minimal imposition on method design:** attention has been paid to ensuring model developers can use their own empirical data and models with Maccabee painlessly. This includes support for benchmarking models written in both Python and R to avoid the need for language translation.
- **Quickstart but powerful customization:** The package includes high-quality data and pre-tuned parameters. This means that little boilerplate code is required to run a benchmark and receive results. This helps new users understand, and get value out of, the package quickly. At the same time, there is a large control surface to give advanced users the tools they need to support heavily-customized benchmarking processes.
- **Support for optimized, parallel execution:** valid Monte Carlo benchmarks require large sample sizes. In turn, this requires effecient, optimized code and the ability to access and utilize sufficient computational power. Maccabee provides code compilation for sampled DGPs - which greatly improves execution time - and parallelization tools that enable execution across multiple cores. Together, these tools make large-sample benchmarks feasible.
- **Smooth side-by-side support of old and new approaches:** Maccabee allows for user-specified DGPs to be used side by side with the sampled DGPs enabled by the package. This allows users to switch between/compare the new and old approaches while using a single benchmarking tool. It also allows users to exploit the advanced functionality outlined above even if they don't use the core sampling functionality.

Table of Contents

Installing Maccabee

Standard Python Installation

Maccabee is available as a python package. To install it, run the following command at the command line:

```
pip install maccabee
```

Installing optional R dependencies

If you plan to work with R models, install the Maccabee package along with optional R dependencies by running:

```
pip install maccabee[r]
```

Docker Image

There is a Docker image available with Maccabee and all dependencies pre-installed. This docker image is primarily designed to run a Jupyter notebook server.

The image is called *maccabeebench/jupyter_maccabee:latest*. To run a notebook server, use the command below. It will mount your current working directory at *~/work* and make the server accessible at *localhost:8888*.

```
docker run --rm -p 8888:8888 \
  -v $(PWD):/home/jovyan/work \
  maccabeebench/jupyter_maccabee:latest \
  start.sh jupyter notebook \
  --NotebookApp.token='' \
  --NotebookApp.notebook_dir='~/work' \
  --NotebookApp.ip='0.0.0.0'
```

Quick Start Tutorials

Benchmarking with Sampled DGPs

This walkthrough previews Maccabee's core functionality - benchmarking causal inference estimators using sampled, synthetic treatment and outcome functions combined with empirical covariate data. You'll find detail on each of the methods and objects mentioned below in the maccabee package reference.

The Destination: Sampled DGP Benchmarks

The code below benchmarks a standard linear regression estimate of the Average Treatement Effect (ATE) under 3 distributional settings - low/medium/high outcome mechanism nonlinearity - with treatment assignment mechanism nonlinearity kept low in all three settings.

The logical process for this is as follows - we (repeatedly) sample parameter-conformant treatment assignment and outcome functions defined over some covariate data (in this case a standard normal dataset). Each pair of sampled functions is used to generate treatment assignment and potential outcome data based on the empirical covariate data. The estimator being benchmarked is fit to this data, estimates are collected, an estimator performance metrics are calculated based on the ground-truth (which is found using the sampled functions). The performance metrics are averaged over many sampled data generating processes, and many times for each sampled DGP, to account for sources of performance variation in the functions and generated data.

Maccabee makes this process extremely easy. The code below executes all of the above:

[22]:

```
from maccabee.constants import Constants
from maccabee.data_sources.data_source_builders import build_random_normal_datasource
from maccabee.benchmarking import benchmark_model_using_sampled_dgp_grid
from maccabee.modeling.models import LinearRegressionCausalModel

import pandas as pd

LOW, MEDIUM, HIGH = Constants.AxisLevels.LEVELS

param_grid = {
Constants.AxisNames.TREATMENT_NONLINEARITY: [LOW],
Constants.AxisNames.OUTCOME_NONLINEARITY: [HIGH, MEDIUM, LOW]
}

normal_data_source = build_random_normal_datasource(
    n_covars=5,
    n_observations=1000)

results = benchmark_model_using_sampled_dgp_grid(
    model_class=LinearRegressionCausalModel,
    estimand=Constants.Model.ATE_ESTIMAND,
    data_source=normal_data_source,
    dgp_param_grid=param_grid,
    num_dgp_samples=10,
    num_sampling_runs_per_dgp=5,
    num_samples_from_dgp=96)
```

[45]:

```
pd.DataFrame(results)
```

[45]:

	param_outcome_nonlinearity	param_treatment_nonlinearity	RMSE	RMSE (std)	AMBP	AMBP (std)	MABP	MABP (std)
0	HIGH	LOW	0.081	0.030	17.892	25.949	21.771	25.519
1	MEDIUM	LOW	0.043	0.020	2.644	3.316	3.654	3.986
2	LOW	LOW	0.017	0.001	0.100	0.067	0.995	0.445

The results are consistent with the well known fact that the performance of a linear regression estimator degrades as the non-linearity of the outcome function increases. This is evident from all three performance metrics, although the variance of the AMBP and MABP appears too large for a reliable conclusion based on these values.

In the code above, the user has supplied:

- A model which will be 'fit' to the data to estimate causal effects - in this case a simple `LinearRegressionCausalModel` which is supplied as an example model with the Maccabee package.
- A targeted estimated, in this case the ATE estimand (specified using the ATE value in the `Model` constants group in `Constants`).
- A `DataSource`, in this case one that contains 1000 observations of 5 independent, standard-normal covariates.
- A grid of parameter which specifies the combinations of parameters as various levels of treatment and outcome function non-linearity.

With these choices made, the `benchmark_model_using_sampled_dgp_grid()` function can be used to:

1. Sample Data Generating Processes, defined over the covariates in the data source, which conform to each of the desired parameter combinations.
2. `num_dgp_samples` different DGPs will be sampled and each will be used to generate `num_samples_from_dgp` different data sets.
3. Fit the model and produce the estimated value of the ATE estimand.
4. Compare the estimated value to the ground truth and collect performance metrics.
5. Repeat steps 2-4 `num_sampling_runs_per_dgp` times to determine the variance of the metric estimates (recall that each metric is defined over a sample of estimate values).

With this procedure in mind, we can take a few steps back to understand the various components which are mentioned above.

The Components of Sampled DGP Benchmarks

Model Specification

Although it is not explicitly displayed above, the first step in using Maccabee is to define a `CausalModel`. The `CausalModel` class wraps an arbitrary method of causal inference, ensuring compatibility with the rest of Maccabee's tooling.

An example model definition, for the `LinearRegressionCausalModel` used in the analysis above, is presented below. The class inherits from `CausalModel` and overrides its methods to wrap a scikit linear regression model. Although this is a simple model, arbitrarily complex models can be encapsulated in the exact same way.

All models take a `GeneratedDataSet` object at construction time and implement `fit()` and `estimate_*()` functions. Arbitrary code can be run at fit and estimate time. Fit is run exactly once per model instance prior to estimates. See the docs for the `CausalModel` for more detail.

[5]:

```
from sklearn.linear_model import LinearRegression
from maccabee.modeling.models import CausalModel


class LinearRegressionCausalModel(CausalModel):
    def __init__(self, dataset):
        self.dataset = dataset
        self.model = LinearRegression()
        self.data = dataset.observed_data.drop("Y", axis=1)

    def fit(self):
        """Fit the linear regression model.
        """
        self.model.fit(self.data, self.dataset.Y)

    def estimate_ATE(self):
        """
        Return the co-efficient on the treatment status variable as the
        ATE.
        """
        # The coefficient on the treatment status
        return self.model.coef_[-1]
```

Data Sources

The second step is supplying a data source. Fundamentally, a `DataSource` is defined by a set of covariate observations. Under the hood, the `DataSource` object is responsible for concretizing stochastically defined covariate specification and for the data normalization and management required for DGP sampling. The vast majority of users will not need to worry about the specifics of these processes because the `data_source_builders` module contains a number of convenient `DataSource` generator functions. These correspond to:

1. High-quality empirical data - accessible via `build_lalonde_datasource()` and `build_cpp_datasource()` (*with more to come*). See Chapter 5 of the theory paper for a discussion on these datasets .
2. Random normal covariates with user-controlled degree of pair-wise correlation. See `build_random_normal_datasource()`.

3. Utilities for loading covariates from CSV files and automating the normalization and processing - see `build_csv_datasource()`.

For these common use cases, building a `DataSource` is as simple as using the code below.

[26]:

```
from maccabee.data_sources import build_lalonde_datasource
data_source = build_lalonde_datasource()
```

Parameter Specification

The final step in running a sampled DGP benchmark is providing the parameter specification which controls the DGP sampling process. At this stage, specification can only be done by specifying a `scikit-learn` style parameter-grid. See the `benchmark_model_using_sampled_dgp_grid()` function for more detail on this specification format.

The valid parameters are in the `AxisNames` constant group in `Constants`. The listing of the parameters can be accessed using the `all()` helper method defined for each constant group in `Constants`.

[27]:

```
from maccabee.constants import Constants
Constants.AxisNames.all()
```

[27]:

```
{'OUTCOME_NONLINEARITY': 'OUTCOME_NONLINEARITY',
 'TREATMENT_NONLINEARITY': 'TREATMENT_NONLINEARITY',
 'PERCENT_TREATED': 'PERCENT_TREATED',
 'OVERLAP': 'OVERLAP',
 'BALANCE': 'BALANCE',
 'ALIGNMENT': 'ALIGNMENT',
 'TE_HETEROGENEITY': 'TE_HETEROGENEITY'}
```

Each of these parameters correspond to a distributional problem space axis and therefore contribute to controling the distributional setting of the observed data produced by the sampled DGPs. See the DGP Sampling Parameterization section of the Design and Implementation documentation and Chapter 5 of the theory paper for a discussion on how these parameters correspond to the axes of the causal inference distributional problem space.

The parameter grid given below would explore every combination of parameters available in Maccabee - exploring the full distributional problem space. It is highly likely that a much smaller subset of combinations would suffice for the purpose of most evaluation.

[37]:

```
LOW, MEDIUM, HIGH = Constants.AxisLevels.LEVELS

complete_param_grid = {
    Constants.AxisNames.OUTCOME_NONLINEARITY: [HIGH, MEDIUM, LOW],
    Constants.AxisNames.TE_HETEROGENEITY: [HIGH, MEDIUM, LOW],
    Constants.AxisNames.TREATMENT_NONLINEARITY: [HIGH, MEDIUM, LOW],
    Constants.AxisNames.PERCENT_TREATED: [HIGH, MEDIUM, LOW],
    Constants.AxisNames.OVERLAP: [HIGH, MEDIUM, LOW],
    Constants.AxisNames.BALANCE: [HIGH, MEDIUM, LOW],
    Constants.AxisNames.ALIGNMENT: [HIGH, MEDIUM, LOW]
}
```

Analyzing Benchmark Results

By default, when running on the ATE estimand, the benchmark function collects and returns all of the performance metrics from the `performance_metrics` module. See the documentation for that module for more detail on the metrics themselves.

The metrics are returned as a dictionary of lists, with keys corresponding to the metric names and list entries for the average metric value at each parameter combination (along with the standard deviation for the metric value across the set of sampled dgps). This dictionary also includes lists that contain the corresponding parameter values.

The upshot is that it is possible to build a `DataFrame` object and then use this object to explore the metric values for every parameter combination. This is exactly what was done above. Building the `DataFrame` is simple:

[40]:

```
results_df = pd.DataFrame(results)
results_df
```

[40]:

	param_outcome_nonlinearity	param_treatment_nonlinearity	RMSE	RMSE (std)	AMBP	AMBP (std)	MABP	MABP (std)
0	HIGH	LOW	0.081	0.030	17.892	25.949	21.771	25.519
1	MEDIUM	LOW	0.043	0.020	2.644	3.316	3.654	3.986
2	LOW	LOW	0.017	0.001	0.100	0.067	0.995	0.445

As you can see, not all metrics may be interesting for all use cases, especially if the sample size is too small to produce a meaningfully narrow metric estimate interval. This appears to be the case for the `MABP` and `AMBP` estimates in the high outcome nonlineairty setting in the table above.

The `DataFrame` can be used to select, sort, and filter the metrics and result rows - in this case, removing the result row with large intervals and the MABP metric results.

```
[44]:
```

```
results_df[
    results_df["param_outcome_nonlinearity"]!=HIGH
].drop(["MABP", "MABP (std)"], axis=1)
```

```
[44]:
```

	param_outcome_nonlinearity	param_treatment_nonlinearity	RMSE	RMSE (std)	AMBP	AMBP (std)
1	MEDIUM	LOW	0.043	0.020	2.644	3.316
2	LOW	LOW	0.017	0.001	0.100	0.067

Conclusion

In order to run a Maccabee benchmark, you will need to supply a `CausalModel` class, a `DataSource` instance and a combination of parameter values. The flexibility of the model and data source classes mean that users can apply the power of sampled DGP benchmarking to a virtually limitless set of causal inference estimators and source covariate datasets. For detailed documentation of the objects and methods mentioned above, see the maccabee package reference.

Benchmarking with a Concrete DGP

The Benchmarking with Sampled DGPs section demonstrated how to evaluate a causal estimator using DGPs sampled from a location in the distributional problem space. While this approach is central to Maccabee's benchmarking philosophy, it is also possible to use this package to run benchmarks using concretely specified DGPs. This will be useful if you want to compare sampled DGP results to previous results from concrete DGPs or if you want to make use of Maccabee's result analysis and parallelization tools.

The Maccabee DSL for Specifying Concrete DGPs

Using a concrete DGP requires manual specification of the data generating process. Maccabee provides a light DSL - a domain specific language - which wraps standard python methods to make custom DGP specification as easy as possible. The DSL allows Maccabee to handle most of the boilerplate operations involved in sampling from the DGP and ensures the compatibility of the DGP with the rest of the functionality in the package.

The code below specifies a concrete DGP with three normally distributed covariates, a linear outcome function and a linear treatment assignment logit function

```
[2]:
```

```
from maccabee.data_generation import ConcreteDataGeneratingProcess, data_generating_method
from maccabee.constants import Constants
from maccabee.data_generation.utils import evaluate_expression
import numpy as np
import sympy as sp
import pandas as pd

DGPVariables = Constants.DGPVariables

class CustomConcreteDataGeneratingProcess(ConcreteDataGeneratingProcess):
    def __init__(self, n_observations):

        super().__init__(n_observations, data_analysis_mode=False)

        # Three covariates - A, B and C.
        self.n_vars = 3
        self.covar_names = ["A", "B", "C"]
        self.A, self.B, self.C  = sp.symbols(self.covar_names)

        # Linear treatment assignment logit
        self.treatment_assignment_function = 1/(1 + sp.exp(-1*(5*self.A + -7*self.B)))

        # Linear untreated outcome function.
        self.base_outcome_function = 6*self.C

    @data_generating_method(DGPVariables.COVARIATES_NAME, [])
    def _generate_observed_covars(self, input_vars):
        X = np.random.normal(loc=0.0, scale=1.0, size=(
          self.n_observations, self.n_vars))

        return pd.DataFrame(X, columns=self.covar_names)

    @data_generating_method(DGPVariables.PROPENSITY_SCORE_NAME,
                            [DGPVariables.COVARIATES_NAME])
    def _generate_true_propensity_scores(self, input_vars):
        observed_covariate_data = input_vars[DGPVariables.COVARIATES_NAME]

        return evaluate_expression(
          self.treatment_assignment_function,
          observed_covariate_data)

    @data_generating_method(
      DGPVariables.POTENTIAL_OUTCOME_WITHOUT_TREATMENT_NAME,
      [DGPVariables.COVARIATES_NAME])
    def _generate_outcomes_without_treatment(self, input_vars):
        observed_covariate_data = input_vars[DGPVariables.COVARIATES_NAME]

        return evaluate_expression(
          self.base_outcome_function,
          observed_covariate_data)

    @data_generating_method(DGPVariables.OUTCOME_NOISE_NAME, [])
    def _generate_outcome_noise_samples(self, input_vars):
        return np.random.normal(loc=0, scale=0.25, size=self.n_observations)

    @data_generating_method(
      DGPVariables.TREATMENT_EFFECT_NAME,
      [DGPVariables.COVARIATES_NAME])
    def _generate_treatment_effects(self, input_vars):
        return 2
```

Note that the `CustomConcreteDataGeneratingProcess` inherits from **`ConcreteDataGeneratingProcess`** and overrides the parent class's `_generate_*()` methods. Each `_generate_*()` method in the class is decorated using **`data_generating_method()`**. The decorator is used to indicate the DGP variable

that the decorated method produces and the DGP variables on which it depends. The variables on which a method depends are automatically passed into the method at execution time. For more detail on the Maccabee DSL see the `maccabee.data_generation.data_generating_process` module reference.

Generating Data

With the DGP specified, we can perform a quick manual data generation to ensure things are working the way we intended.

First, we instantiate the DGP (supplying the desired number of observations) and then generate a dataset:

[11]:

```
concrete_dgp = CustomConcreteDataGeneratingProcess(n_observations=100)
dataset = concrete_dgp.generate_dataset()
```

We can then look at the observed data property of the sampled `GeneratedDataSet` instance. The observed data contains the three covariates and a treatment and outcome status. Probing the `ATE` property of the dataset reveals the expected average treatment effect.

[12]:

```
dataset.observed_data.head()
```

[12]:

	A	B	C	T	Y
0	-1.616131	-0.764650	-0.191657	0	-0.937014
1	-0.577143	-0.209127	-0.554967	1	-0.993029
2	-0.247293	1.054256	-0.307911	0	-1.812844
3	-0.645216	0.106769	-1.083359	0	-6.651235
4	0.915464	0.672721	0.082086	0	0.801945

[15]:

```
# ground truth
dataset.ATE
```

[15]:

```
2.0
```

Given the linearity of the model, we would expect a logistic regression to recover the true ATE and, indeed, it does:

```
[16]:
```

```
from maccabee.modeling.models import LinearRegressionCausalModel

# Build and fit model
model = LinearRegressionCausalModel(dataset)
model.fit()

# estimate
model.estimate_ATE()
```

```
[16]:
```

```
1.9379849734528494
```

Running a Benchmark

We're now ready to run a benchmark. The code is only loosely analogous to the sample-based benchmark in the Benchmarking with Sampled DGPs section. We still supply a model class, estimand and number of samples to take from the DGP. But the concrete specification of the DGP means we only supply the DGP instance rather than sampling parameters and a data source

```
[17]:
```

```
from maccabee.benchmarking import benchmark_model_using_concrete_dgp

aggregated_results, raw_results, _, _ = benchmark_model_using_concrete_dgp(
  dgp=concrete_dgp,
  model_class=LinearRegressionCausalModel,
  estimand=Constants.Model.ATE_ESTIMAND,
  num_sampling_runs_per_dgp=10,
  num_samples_from_dgp=250)
```

As one would expect for such a simple DGP and distributional setting, the RMSE and AMBP are both close to zero.

```
[20]:
```

```
aggregated_results["RMSE"], aggregated_results["AMBP"]
```

```
[20]:
```

```
(0.083, 0.229)
```

Customizing DGP Sampling

The procedure outlined in the Benchmarking with Sampled DGPs tutorial provides a decent default mechanism for operationalizing sampled-DGP based benchmarking. In that tutorial, DGPs are sampled based on selected parameter levels which describe a location in the distributional problem space. The sampling of data from the DGPs and fitting of models was handled by benchmarking functions.

This tutorial describes how to customize the DGP sampling process. Primarily, this means directly specifying DGP sampling parameters instead of relying on levels along distributional problem space axes. These parameters can then be passed to benchmarking functions, analogous to those in the prior tutorials, which then results in the exact benchmarking procedure previously outlined.

However, we will go beyond this procedure, and also look at how to manually sample DGPs, allowing for fine-grained control over the entire end-to-end process. This fine-grained control may be particularly helpful for the purpose of debugging/diagnosis.

Specifying Parameters

DGP sampling parameters are stored in `ParameterStore` instances. These instances encapsulate much of the complexity of the parameter specification. Complete documentation on how to construct `ParameterStore` instances is available in the class reference documentation. In this tutorial, we cover two of the three methods (omitting parameter schema files):

Customizing the Default Parameters

The easiest way to construct a `ParameterStore` instance is to start from Maccabee's default parameter values. This can be done by importing the `build_default_parameters()` function and using it as below:

```
[29]:
```

```
from maccabee.constants import Constants
from maccabee.parameters import build_default_parameters

# Build the parameters
dgp_params = build_default_parameters()

dgp_params
```

```
[29]:
```

```
<maccabee.parameters.parameter_store.ParameterStore at 0x7f7b247bc650>
```

This instance can then be customized using its `set_parameters()` method as demonstrated below.

See the `parameter_store` module docs for detail on the `ParameterStore` API and the parameters available for customization.

```
[6]:
```

```
dgp_params.set_parameters({
    "ACTUAL_CONFOUNDER_ALIGNMENT": 0.25,
    "POTENTIAL_CONFOUNDER_SELECTION_PROBABILITY": 0.7
})

dgp_params.ACTUAL_CONFOUNDER_ALIGNMENT
```

```
[6]:
```

```
0.25
```

Customizing the Axis Level Parameters

The second method of building custom parameters involves starting from the distributional problem space axis level parameters and customizing them as above. This can be done using the `build_parameters_from_axis_levels()` function as below. Any axes omitted from the dictionary will be instantiated to their default values.

```
[9]:
```

```
from maccabee.constants import Constants
from maccabee.parameters import build_parameters_from_axis_levels

# Build the parameters
dgp_params = build_parameters_from_axis_levels({
  Constants.AxisNames.OUTCOME_NONLINEARITY: Constants.AxisLevels.LOW,
  Constants.AxisNames.TREATMENT_NONLINEARITY: Constants.AxisLevels.LOW,
})

dgp_params
```

[9]:

```
<maccabee.parameters.parameter_store.ParameterStore at 0x7f7b054172d0>
```

With custom parameters in hand, there are two ways to proceed. Either using the built in benchmarking tools or manually sampling DGPs. These two paths are outlined below.

Using Benchmarking Functions with Custom Parameters

The procedure for running a benchmark is nearly identical to the one outlined in Benchmarking with Sampled DGPs. However, the function `benchmark_model_using_sampled_dgp()` is used to run a benchmark using the `ParameterStore` instance constructed above in place of the parameter levels grid.

[17]:

```
from maccabee.constants import Constants
from maccabee.data_sources.data_source_builders import build_random_normal_datasource
from maccabee.benchmarking import benchmark_model_using_sampled_dgp
from maccabee.modeling.models import LinearRegressionCausalModel
import pandas as pd

normal_data_source = build_random_normal_datasource(
    n_covars=5,
    n_observations=1000)

results = benchmark_model_using_sampled_dgp(
    dgp_sampling_params=dgp_params,
    model_class=LinearRegressionCausalModel,
    estimand=Constants.Model.ATE_ESTIMAND,
    data_source=normal_data_source,
    num_dgp_samples=2,
    num_sampling_runs_per_dgp=5,
    num_samples_from_dgp=10)

results[0]
```

[17]:

```
{'RMSE': 0.022,
 'RMSE (std)': 0.003,
 'AMBP': 0.61,
 'AMBP (std)': 0.4,
 'MABP': 1.832,
 'MABP (std)': 0.321}
```

It is clear that the return value for this function is slightly different. It has more components - designed to provide lower-level insight into the results - and doesn't lend itself to direct conversion into a `pandas.DataFrame`. See the `benchmark_model_using_sampled_dgp()` function reference docs for more detail.

Manually Sampling DGPs

The second approach involves manually sampling DGPs. This gives direct, low-level access to the data generation process. There code below introduces the `DataGeneratingProcessSampler` class. This is the class used by the benchmarking functions to sample DGPs using the parameters from a `ParameterStore` instance and the covariate data from a `DataSource` instance.

The code below provides the basic template for manual DGP sampling. Consult the relevant subsections of the maccabee package reference section for details on how to interact with the classes and objects which appear below.

```
[24]:
```

```
from maccabee.data_sources import build_random_normal_datasource
from maccabee.data_generation import DataGeneratingProcessSampler

# Build the data source
covar_data_source = build_random_normal_datasource(
  n_covars = 5, n_observations=1000)

# Build a DGP Sampler, supplying params and data.
dgp_sampler = DataGeneratingProcessSampler(
  parameters=dgp_params,
  data_source=covar_data_source)

# Sample a DGP.
dgp = dgp_sampler.sample_dgp()

dgp
```

```
[24]:
```

```
<maccabee.data_generation.data_generating_process.SampledDataGeneratingProcess at
0x7f7b248db4d0>
```

Now that we have a sampled `DataGeneratingProcess` instance, we proceed as outlined in the Benchmarking with a Concrete DGP tutorial - sampling data, fitting a causal model and producing a treatment effect estimate.

[25]:

```
# Generate a data set.
dataset = dgp.generate_dataset()
dataset.observed_data.head()
```

[25]:

	X0	X1	X2	X3	X4	T	Y
0	0.042946	-0.244024	0.352224	-0.385605	-0.297958	0	0.065378
1	-0.035489	-0.130587	0.168029	-0.508966	0.172846	1	-1.160852
2	-0.100376	-0.000438	-0.134698	0.158655	0.133745	1	-1.293867
3	-0.198569	-0.354449	0.093455	-0.113408	-0.307699	1	-0.491382
4	0.250105	-0.005793	-0.202563	-0.027322	0.524938	0	-0.318331

[26]:

```
from maccabee.modeling.models import LinearRegressionCausalModel

# Fit the model
model = LinearRegressionCausalModel(dataset)
model.fit()
```

[27]:

```
# Ground Truth
dataset.ATE
```

[27]:

```
-0.6129999999999998
```

[28]:

```
# Estimate
model.estimate(estimand=Constants.Model.ATE_ESTIMAND)
```

[28]:

```
-0.6095040438593615
```

Using Empirical Covariate Data

Maccabee offers the ability to benchmark causal inference methods using empirical covariate data, while keeping the same control over distributional setting enabled by synthetic DGPs.

There are two ways to use empirical covariate data in benchmarks. First, Maccabee includes a number of built in empirical data sets. Second, Maccabee provides tool for loading external covariate data from appropriately formatted CSVs.

Using Built-in Covariate Data

The empirical data sets included with Maccabee can be loaded as `DataSource` instances using helper functions from See `data_source_builders`. The underlying data sets are prepared versions of well-known publicly-available data sets. The preparation and cleaning procedure is accessible for transparency and reproducibility. See `data_source_builders` for more on this.

Loading a data set is as simple as calling the `DataSource` builder. The code below creates a `DataSource` corresponding to the well known Lalonde data set. Building on Dehejia & Wahba (2002) and Smith & Todd (2005), Maccabee uses a selected subset of the Lalonde experimental data (with real income in 1974 available) and the PSID observational controls.

Crucially, the data set **only includes the empirical covariates** not the treatment assignment and outcomes. These are produced by sampling a (sampled) DGP defined over the empirical covariates.

```
[2]:
```

```
from maccabee.data_sources.data_source_builders import build_lalonde_datasource
lalonde_data_source = build_lalonde_datasource()

lalonde_data_source
```

```
[2]:
```

```
<maccabee.data_sources.data_sources.StaticDataSource at 0x7f9119fdb790>
```

This data source can then be used exactly as seen in the Benchmarking with Sampled DGPs tutorial. Like in that tutorial, the code below benchmarks a linear regression model for DGPs with low, medium and high outcome nonlinearity. Unlike in that tutorial, the DGPs are know defined over empirical rather than standard normal covariate data.

[3]:

```
from maccabee.constants import Constants
from maccabee.benchmarking import benchmark_model_using_sampled_dgp_grid
from maccabee.modeling.models import LinearRegressionCausalModel
import pandas as pd

LOW, MEDIUM, HIGH = Constants.AxisLevels.LEVELS

param_grid = {
    Constants.AxisNames.TREATMENT_NONLINEARITY: [LOW],
    Constants.AxisNames.OUTCOME_NONLINEARITY: [HIGH, MEDIUM, LOW]
}

results = benchmark_model_using_sampled_dgp_grid(
    model_class=LinearRegressionCausalModel,
    estimand=Constants.Model.ATE_ESTIMAND,
    data_source=lalonde_data_source,
    dgp_param_grid=param_grid,
    num_dgp_samples=10,
    num_sampling_runs_per_dgp=1,
    num_samples_from_dgp=10)
```

[4]:

```
pd.DataFrame(results)
```

[4]:

	param_outcome_nonlinearity	param_treatment_nonlinearity	RMSE	RMSE (std)	AMBP	AMBP (std)	MABP	MA (s
0	HIGH	LOW	0.096	0.077	20.528	30.686	21.523	30.32
1	MEDIUM	LOW	0.030	0.015	159.336	464.794	160.574	464.3
2	LOW	LOW	0.015	0.002	1.224	1.726	4.551	6.341

Using Custom Covariate Data

It is also possible to use arbitrary covariate data with Maccabee's sampled DGPs. There are two approaches for this. If the covariate data is static - one, fixed sample - then it can be loaded from a CSV file. If the covariate data is dynamic - extracted from a large database for example - then it can be supplied as a stochastic data source.

Using Static Custom Covariate Data

Static covariate data can be loaded using the `build_csv_datasource()` helper function. This function takes the path to a CSV, with the first row containing covariate names, and optionally a list of the names of the discrete covariates. (Discrete covariates are not included in normalization and some of the subfunctions used to build the sampled DGPs). See the `build_csv_datasource()` helper function reference docs for more detail.

The code below creates a CSV using a numpy array and then loads the data as a static data source. This data source can then be used with the benchmarking code above.

[5]:

```
from maccabee.data_sources.data_source_builders import build_csv_datasource
import numpy as np

# Define and save data
data = np.array([
    [1.07, 0],
    [3.5, 1],
    [5.17, 0]
])
file_name = "data.csv"
np.savetxt(file_name, data, delimiter=',', header="Age, Gender")
```

[6]:

```
# Load data as DataSource
static_datasource = build_csv_datasource(file_name, ["Gender"])

static_datasource
```

[6]:

```
<maccabee.data_sources.data_sources.StaticDataSource at 0x7f90ee705f50>
```

[7]:

```
# Show the covariate data
static_datasource.get_covar_df()
```

[7]:

	Age	Gender
0	-1.000000	0.0
1	0.185366	1.0
2	1.000000	0.0

Using Stochastic/Dynamic Covariate Data

It is also possible to create a data source which executes a function to access (potentially stochastic) covariate data. This is useful if there is a large set of (assumed) independent and identically distributed covariate data and benchmarks only need to use a sample from it. For this

reason, this kind of data source is a `StochasticDataSource`.

A `StochasticDataSource` instance can be constructed using the `build_stochastic_datasource()` by supplying a generator function that returns a 2D `numpy.ndarray`, a list of covariate names for each column in the covariate array, and a list of discrete column names.

An example is given below. This is a toy example that generates a mix of standard normal and bernoulli binary covariates. A real examples would include more expensive/interesting generator functions.

[16]:

```
from maccabee.data_sources.data_source_builders import build_stochastic_datasource

N_covars = 10
N_obs = 1000
binary_col_indeces = [0, 3]
covar_names = [f"X{i}" for i in range(N_covars)]

def generate_data():
    covar_data = np.random.normal(loc=0.0, scale=1.0, size=(
            N_obs, N_covars))

    # Make binary columns.

    for var in binary_col_indeces:
        covar_data[:, var] = (covar_data[:, var] > 0).astype(int)

    return covar_data

stochastic_datasource = build_stochastic_datasource(
        generate_data,
        covar_names=covar_names,
        discrete_covar_names=["X0", "X3"])

stochastic_datasource
```

[16]:

```
<maccabee.data_sources.data_sources.StochasticDataSource at 0x7f91192f2d10>
```

We can verify that the data source does indeed generate data inline with expectations:

[18]:

```
stochastic_datasource.get_covar_df().head()
```

[18]:

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9
0	1.0	0.397143	-0.286716	1.0	-0.104719	0.009483	0.348603	-0.158236	-0.021799	-0.909613
1	0.0	-0.166077	-0.546770	1.0	-0.058030	-0.382312	-0.222217	0.299738	0.028531	-0.665068
2	1.0	0.452277	-0.666157	0.0	-0.130254	-0.167185	0.463603	-0.100672	0.155738	-0.480558
3	0.0	0.500995	0.070425	0.0	0.074098	0.646198	0.510876	0.392154	0.309803	-0.076971
4	1.0	-0.534493	-0.425811	0.0	-0.290319	-0.010616	0.072021	0.023280	0.055812	-0.296402

Analyzing Distributional Setting

The location of the some observed data's distributional setting in the distributional problem space is central to Maccabee's philosophy. This tutorial demonstrates the tools available for analyzing the data generated by sampled/concrete DGPs in terms of the axes of the problem space.

This tutorial builds on the tools covered in the Customizing DGP Sampling tutorial - comparing the distributional setting of data generated under two different parameterizations of the DGP sampling process.

The process involves selecting which data metrics to observe, running a benchmark configured to collect these metrics, and then analyzing the resultant data.

Selecting Data Metrics to Observe

The first step in this analysis involves selecting the data metrics to collect. As explained in the `maccabee.data_analysis` module docs, each distributional problem space axis has a number of associated data metrics which measure the *position* of data along the axis.

> **Note**
>
> The various data metrics are partially overlapping in what they measure and their usefulness/applicability is dependent on the nature of the data. Experimentation and thoughtfulness are required when using these metrics. See the `data_analysis` documentation for more detail on the data metrics.

The full set of metrics is available in the `AXES_AND_METRICS` dictionary in the `data_metrics` module. This contains a large number of metrics. In this tutorial, we'll focus on the outcome nonlinearity axis of the distributional problem space, so let's use the dictionary to look at the set of outcome nonlinearity metrics

```
[2]:
```

```
from maccabee.data_analysis.data_metrics import AXES_AND_METRICS
from maccabee.constants import Constants

for i, metric in enumerate(AXES_AND_METRICS[Constants.AxisNames.OUTCOME_NONLINEARITY], 1):
    print(i, metric["name"])
```

```
1 Lin r2(X_obs, Y)
2 Lin r2(X_true, Y)
3 Lin r2(X_obs, Y1)
4 Lin r2(X_obs, Y0)
5 Lin r2(X_true, Y1)
6 Lin r2(X_true, Y0)
7 Lin r2(X_obs, TE)
8 Lin r2(X_true, TE)
```

For our purposes, metrics one and four look interesting. These respectively measure the R^2 value of a linear regression of the observed outcome and the untreated potential outcome on the covariates.

In order to measure these two metrics, we construct a data metric specification that maps axis names to select metrics. The format of this grid is explained in detail in the documentation for the `calculate_data_axis_metrics()` function.

[3]:

```
# Define axes and metrics to analyze
DATA_METRICS_SPEC = {
    Constants.AxisNames.OUTCOME_NONLINEARITY: [
        "Lin r2(X_obs, Y)",
        "Lin r2(X_obs, Y0)",
    ]
}
```

Collecting Data Metric Values

With the data metric specification specified, we're ready to collect the metric values. Fr demonstrative purposes, we'll manually collect the metrics for low and high outcome nonlinearity. In a real application, the `benchmark_model_using_sampled_dgp_grid()` would be a more convenient mechanism for executing the analysis below.

In the code below, we create parameter specification for low and high outcome nonlinearity and then run a benchmark for each set of parameters to collect the data (and performance) metrics.

Data metrics are designed to complement standard benchmarking and so they are collected as part of a standard model benchmark by setting `data_analysis_mode=True` and optionally suppling the data metric spec in `data_metrics_spec`. The default specification collects **all** metrics and will only be useful for initial exploratory work.

[4]:

```
from maccabee.parameters import build_parameters_from_axis_levels

# Build the parameters
low_nonlin_params = build_parameters_from_axis_levels({
  Constants.AxisNames.OUTCOME_NONLINEARITY: Constants.AxisLevels.LOW,
})

high_nonlin_params = build_parameters_from_axis_levels({
  Constants.AxisNames.OUTCOME_NONLINEARITY: Constants.AxisLevels.HIGH,
})
```

```
[5]:
```

```
from maccabee.data_sources.data_source_builders import build_random_normal_datasource
from maccabee.benchmarking import benchmark_model_using_sampled_dgp
from maccabee.modeling.models import LinearRegressionCausalModel

# Build a random normal data source
normal_data_source = build_random_normal_datasource(
    n_covars=5,
    n_observations=1000)

# Run the benchmarks for both sets of params
results = []
for params in [low_nonlin_params, high_nonlin_params]:
    result = benchmark_model_using_sampled_dgp(
        dgp_sampling_params=params,
        model_class=LinearRegressionCausalModel,
        estimand=Constants.Model.ATE_ESTIMAND,
        data_source=normal_data_source,
        num_dgp_samples=32,
        num_sampling_runs_per_dgp=1,
        num_samples_from_dgp=40,
        data_analysis_mode=True, # SET DATA ANALYSIS MODE
        data_metrics_spec=DATA_METRICS_SPEC, # PROVIDE SPEC
        n_jobs=8)

    results.append(result)
```

Analyzing Data Metric Values

Now that we have benchmark results, we can explore the data metric values for each parameterization. According to the docs for the `benchmark_model_using_sampled_dgp_grid()` function, the data metric values are the fourth and fifth return values. We extract these values for the low outcome nonlinearity results.

The first result contains the data metric value averaged over all the sampled DGPs (internally averaged over all data samples from the DGP). The second result contains the DGP-level averages, averaged over all data samples from the DGP.

```
[6]:
```

```
_, _, _, low_data_metric_agg, low_data_metrics_raw, _ = results[0]
_, _, _, high_data_metric_agg, high_data_metrics_raw, _ = results[1]
```

[7]:

```
low_data_metric_agg
```

[7]:

```
{'OUTCOME_NONLINEARITY Lin r2(X_obs, Y)': 0.853,
 'OUTCOME_NONLINEARITY Lin r2(X_obs, Y0)': 1.0}
```

[8]:

```
low_data_metrics_raw["OUTCOME_NONLINEARITY Lin r2(X_obs, Y)"][:5]
```

[8]:

```
[0.978, 0.994, 0.975, 0.675, 0.952]
```

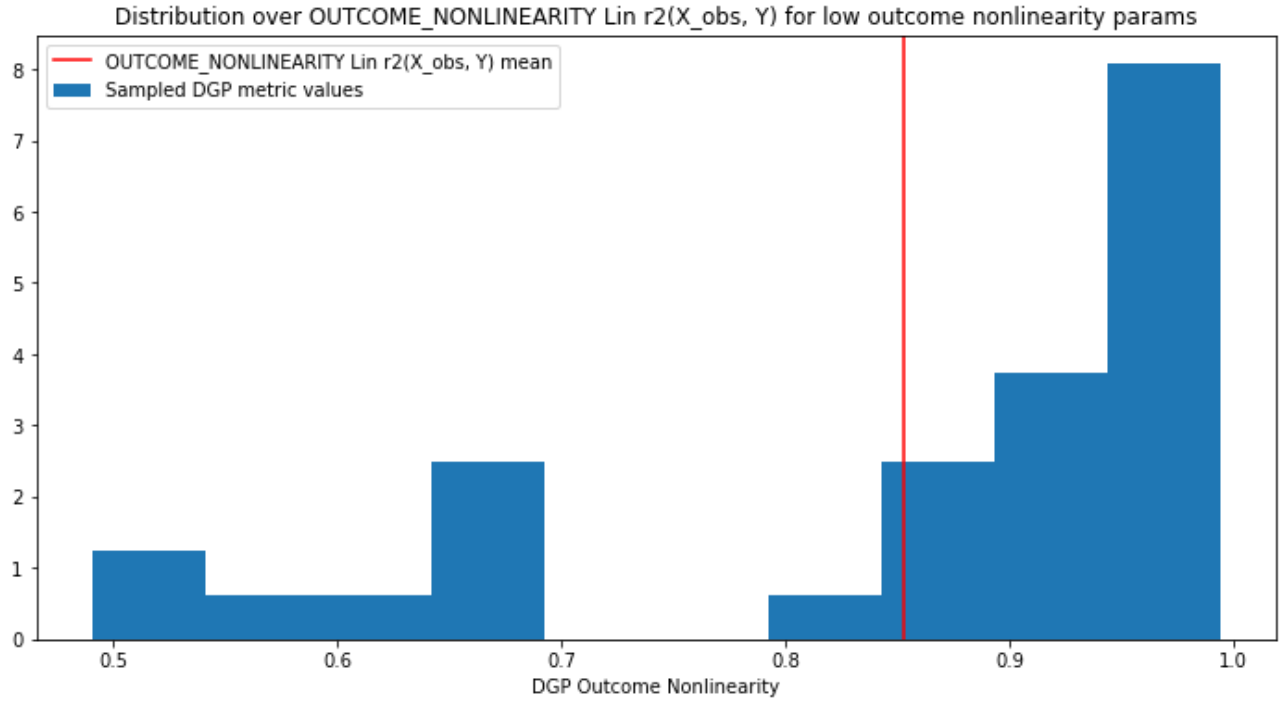
As one might expect, the value of the regression of the observed covariates on the untreated potential outcome is perfect. This makes sense as there is no noise (at this stage). The regression on the observed outcome included noise and the treatment effect and so is not perfect.

We can combine the aggregated and DGP level metric values into a visualization to get a sense of the variance of the sampled DGPs on this axis:

[9]:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
metric = "OUTCOME_NONLINEARITY Lin r2(X_obs, Y)"
plt.hist(low_data_metrics_raw[metric], density=True, label="Sampled DGP metric values")
plt.axvline(x=low_data_metric_agg[metric], c="r", label=f"{metric} mean")
plt.xlabel("DGP Outcome Nonlinearity")
plt.title(f"Distribution over {metric} for low outcome nonlinearity params")
plt.legend()
plt.show()
```



We can also compare the metric values for the low and high outcome nonlinearity parameters to ensure that the generated data is inline with expectations. Looking at the plot, we see that the data metrics do agree with expectation: there is much a much lower R^2 for the high outcome nonlinearity sampling parameters.

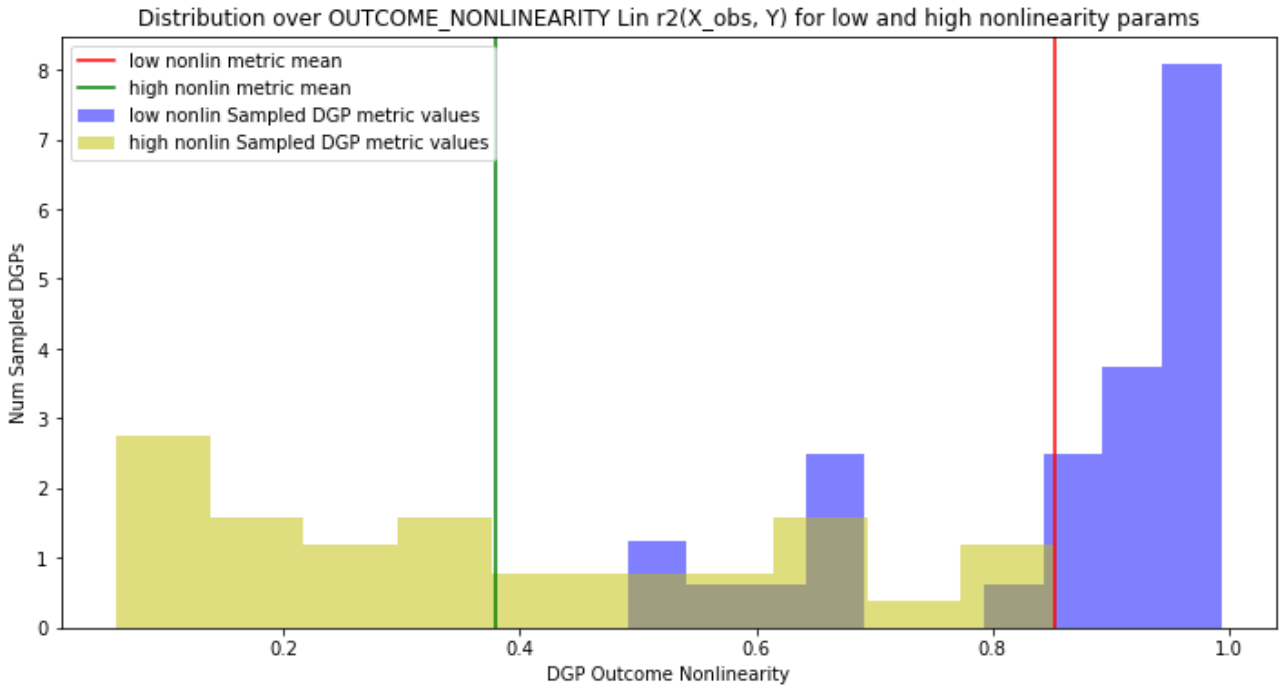
[10]:

```
plt.figure(figsize=(12, 6))
metric = "OUTCOME_NONLINEARITY Lin r2(X_obs, Y)"

plt.hist(low_data_metrics_raw[metric], density=True,
         color="b", label="low nonlin Sampled DGP metric values", alpha=0.5)
plt.axvline(x=low_data_metric_agg[metric],
            c="r", label="low nonlin metric mean")

plt.hist(high_data_metrics_raw[metric], density=True,
         color="y", label="high nonlin Sampled DGP metric values", alpha=0.5)
plt.axvline(x=high_data_metric_agg[metric],
            c="g", label="high nonlin metric mean")

plt.xlabel("DGP Outcome Nonlinearity")
plt.ylabel("Num Sampled DGPs")
plt.title(f"Distribution over {metric} for low and high nonlinearity params")
plt.legend()
plt.show()
```



Advanced Tutorials

R Models

Installing Dependencies

Maccabee includes support for benchmarking models written in R. This is experimental functionality which requires the Rpy2 package.

In order to use this functionality, you must install Maccabee with the optional R dependencies:

```
pip install maccabee[r]
```

For most users, this should be all that is required to proceed with this tutorial. However, on some systems, configuring Rpy2 may require additional setup. If you encounter problems, there are two options. Either consult the Rpy2 documentation or use Maccabee's pre-built docker image which includes a pre-installed distribution of Rpy2. See Installing Maccabee for instructions on this.

Defining R Models

Defining R models for use with Maccabee proceeds in much the same way as covered in the Benchmarking with Sampled DGPs tutorial - the definition of a custom class containing the model logic.

There are two primary differences. Firstly, the model inherits from `CausalModelR` instead of `CausalModel`. Second, Rpy2 is used in the model to access R functions.

R functions can be access by loading existing R packages or loading functions from an R file into a named psuedo-package. In either case, the helper functions return a python object with functions accessible as attribute methods.

> **Note**
>
> This automatic translation has caveats. For example, . separated names are translated to _ separated. So the *glm.fit* function in the *stats* package will bbe *stats.glm_fit*. See the Rpy2 docs for detailed instructions on translating code and accessing return results.

> **Warning**
>
> The attributes of the `GeneratedDataSet` can generally be passed to the R functions only after conversion to `numpy.ndarrays` instances from `pandas.DataFrame` or `pandas.Series` objects. This is **not** done automatically.

Using an Existing R Package

The code below defines an R model which uses both Scikit Learn regression and the *Matching* R package to perform a basic Logistic Propensity Score Matching. The Matching package is imported using the `_import_r_package()` instance method.

```
[19]:
```

```
from maccabee.modeling.models import CausalModelR
from sklearn.linear_model import LogisticRegression

class LogisticPropensityMatchingCausalModel(CausalModelR):
    def fit(self):

        # Import the Matching R package
        matching = self._import_r_package("Matching")

        # Fit the logistic propensity model.
        logistic_model = LogisticRegression(solver='lbfgs', n_jobs=1)
        logistic_model.fit(
            self.dataset.X.to_numpy(), self.dataset.T.to_numpy())
        class_proba = logistic_model.predict_proba(
            self.dataset.X.to_numpy())
        propensity_scores = class_proba[:, logistic_model.classes_ == 1].flatten()

        # Run matching on prop scores using the R match package.
        self.match_out = matching.Match(
            Y=self.dataset.Y.to_numpy(),
            Tr=self.dataset.T.to_numpy(),
            X=propensity_scores,
            estimand="ATT",
            replace=True,
            version="fast")

    def estimate_ATT(self):

        # Return the ATT by extracting it from the match out result.
        return np.array(self.match_out.rx2("est").rx(1,1))[0]
```

With the model defined, we can proceed exactly as in the Benchmarking with Sampled DGPs tutorial, benchmarking the R model. The results are in line with expectations, the high outcome nonlinearity datasets have larger error across all performance metrics. In each outcome nonlinearity level, the estimator performs better with the more linear treatment mechanisms.

```
[14]:
```

```
from maccabee.constants import Constants
from maccabee.data_sources.data_source_builders import build_random_normal_datasource
from maccabee.benchmarking import benchmark_model_using_sampled_dgp_grid
import pandas as pd

LOW, MEDIUM, HIGH = Constants.AxisLevels.LEVELS

param_grid = {
    Constants.AxisNames.TREATMENT_NONLINEARITY: [HIGH, LOW],
    Constants.AxisNames.OUTCOME_NONLINEARITY: [HIGH, LOW]
}

normal_data_source = build_random_normal_datasource(
    n_covars=5,
    n_observations=1000)

results = benchmark_model_using_sampled_dgp_grid(
    model_class=LogisticPropensityMatchingCausalModel,
    estimand=Constants.Model.ATT_ESTIMAND,
    data_source=normal_data_source,
    dgp_param_grid=param_grid,
    num_dgp_samples=5,
    num_sampling_runs_per_dgp=1,
    num_samples_from_dgp=32)
```

```
[15]:
```

```
pd.DataFrame(results)
```

```
[15]:
```

	param_outcome_nonlinearity	param_treatment_nonlinearity	RMSE	RMSE (std)	AMBP	AMBP (std)	MABP	MABP (std)
0	HIGH	HIGH	0.051	0.010	6.008	6.610	8.834	6.048
1	HIGH	LOW	0.050	0.011	2.604	1.752	9.059	4.469
2	LOW	HIGH	0.039	0.011	4.107	4.169	11.299	8.213
3	LOW	LOW	0.042	0.015	1.085	0.571	10.111	11.049

Using Custom R Code

We could also write the R model using an R file. This allows us to avoid some of the complexity of the Rpy2 conversion and also allows for arbitrarily complex R code to be executed. The contents of an R file can be loaded using `_import_r_file_as_package()` instance method.

Below, this functionality is used to replicate the model above but running the matching through a custom R function. Notice that extracting the result is simpler in this format than in the python conversion case above.

```
[20]:
```

```
r_prog = """# Custom R program
library("utils")
capture.output(library("Matching"))

p_score_match <- function(Y, Tr, X){
    out <- Match(
        Y=Y,
        Tr=Tr,
        X=X,
        estimand="ATT",
        replace=TRUE,
        version="fast")

    return(out[["est"]][1][1])
}
"""

with open("r_prog.R", "w") as file:
    file.write(r_prog)
```

```
[18]:
```

```
from maccabee.modeling.models import CausalModelR
from sklearn.linear_model import LogisticRegression

class LogisticPropensityMatchingCausalModel(CausalModelR):
    def fit(self):

        # Import the custom R file
        matching = self._import_r_file_as_package("r_prog.R", "MatchingCode")

        # Fit the logistic propensity model.
        logistic_model = LogisticRegression(solver='lbfgs', n_jobs=1)
        logistic_model.fit(
            self.dataset.X.to_numpy(), self.dataset.T.to_numpy())
        class_proba = logistic_model.predict_proba(
            self.dataset.X.to_numpy())
        propensity_scores = class_proba[:, logistic_model.classes_ == 1].flatten()

        # Run matching on prop scores using the R match package.
        self.att = matching.p_score_match(
            Y=self.dataset.Y.to_numpy(),
            Tr=self.dataset.T.to_numpy(),
            X=propensity_scores)

    def estimate_ATT(self):
        # Return the ATT by extracting it from the match out result.
        return np.array(self.att)
```

Parallelization and Compilation

Maccabee is designed to benchmark methods of causal inference using *large* sample-size Monte Carlo simulations - with hundreds of DGP samples and hundreds of data samples from each DGP.

In general, large sample sizes are important for accurate Monte Carlo estimates. The entire support of the distribution must be explored to ensure there is no asymptotic bias in the estimated value and, even if bias is zero, larger sample sizes reduce the variance/interval width for aggregate estimates.

Using large sample sizes is even more important in Maccabee given the different sources of (significant) variance present in the method. There is stochasticity associated with the sampling DGPs, the sampling of data drawn from each DGP, and (potentially) the the estimation of an estimand in a data set (if the estimator employs a non-deterministic estimation scheme - eg a Neural Network trained by Stochastic Gradient Descent).

With this in mind, Maccabee has two tools designed to speed up the execution of benchmarks and enable larger sample sizes in a fixed amount of run time. The first is parallelization - exploiting the embarrassingly parallelizable nature of Monte Carlo trials to speed up execution by a factor determined by access to computing resources. The second is compilation - which trades off higher once-off set up time for faster execution times of each trial. If a sufficient number of trials are run, then the amortized cost of compilation is justified relative to compilation-free execution.

The use of each of these tools is discussed below.

Parallelization

All of the benchmarking functions in the `benchmarking` module offer a simple parallelization interface via the `n_jobs` keyword argument accepted by each function. The functions will then run the benchmarking using the supplied number of CPU cores. If `-1` is supplied, the number of cores returned by `multiprocessing.cpu_count()` will be used.

Generally speaking, this will produce a decrease in execution time by a factor approximately linear to the number of jobs - provided that each job can be executed in parallel on the computer.

> **Note**
>
> The speed up is likely to be slightly sub-linear due to overhead from parallelization but the degree of overhead is system dependent.

> **Warning**
>
> Using an `n_jobs` value larger than the number of processes your computer can handle is allowed but will generally result in **worse** performance than using the maximum number of parallel processes that can be handled at once.

The code below demonstrates the speed up offered by parallelization. `run_benchmark` runs a benchmark that samples 8 DGPs, and performs 8 sampling runs of 40 datasets each. As you can see, the parallelization speed up is approximately 2x. This means that approximately half the potential linear speed up is consumed by system-specific overhead.

```
[54]:
```

```
%%time

_ = run_parallelization_benchmark(n_jobs=1)
```

```
CPU times: user 12.4 s, sys: 487 ms, total: 12.9 s
Wall time: 1min 10s
```

```
[53]:
```

```
%%time

_ = run_parallelization_benchmark(n_jobs=4)
```

```
CPU times: user 11.7 s, sys: 554 ms, total: 12.3 s
Wall time: 34 s
```

Fine Tuning Parallelization

To get the best results from multitasking it is useful to fine-tune the number of samples to be a multiple of the number of jobs. This prevents wasting capacity by ensuring each execution cycle uses all available resources. (This becomes increasingly important as the number of cores increases)

In order to do this, ensure that, for sampling based benchmarks, `num_dgp_samples` is equal to a multiple of `n_jobs` and, for all benchmarks, that `num_sampling_runs_per_dgp` is also multiple of `n_jobs`. This accounts for the parallelization at the DGP and data sampling levels.

> **Note**
>
> If `num_sampling_runs_per_dgp` is less than `n_jobs`, parallelization occurs at the level of individual data sampling. So you should set `num_samples_from_dgp` to be a multiple of `n_jobs`.

Compilation

Background

Maccabee offers the ability to compile the functions which make up sampled DGPs into C code prior to execution. This speeds up each instance of data sampling at the expense of a once-off penalty when each DGP is sampled. Compilation is activated using the `compile_functions=True` keyword argument which is accepted by each benchmarking function.

Compilation is **not** guarenteed to improve execution time. There are two factors to consider. These are briefly outlined below with a more thorough discussion on this topic in the Design and Implementation documentation.

- First, the number of terms which make up the sampled treatment and outcome functions (this depends on the subfunction sampling. The execution time of uncompiled functions is exponential in the number of terms while the execution time of compiled functions is effectively constant in the number of terms. So, with a large number of terms, the savings from compilation can be substantial. The number of terms primarily depends on the linearity of the treatment/outcome with more nonlinear functions tending to contain (many) more terms.
- Second, the number of data samples **per DGP**. The up-front time penalty of compilation is justified if spread out over enough, faster data samples. If the number of samples per DGP is small then it is almost guaranteed that compilation is **not** worthwhile.

How to decide

In order to decide if compilation is worthwhile, you should run a benchmark that uses a **one** DGP sample, and the **intended number** of sampling runs and data samples. Run this benchmark with and without compilation. If compilation yields a speed up, then use it for the benchmark over multiple DGP samples. If you decide to use compilation, you can keep using it if you later increase/preserve the total number of data samples per DGP - IE `num_sampling_runs_per_dgp * num_samples_from_dgp`.

Demonstration

The code block below runs a benchmark - made up of one DGP sample, 10 sampling runs and different numbers of data samples - with and without compilation. It is clear that compilation is not justified until `num_sampling_runs_per_dgp * num_samples_from_dgp` is large enough, after which compilation becomes increasinglg important

```
[ ]:
```

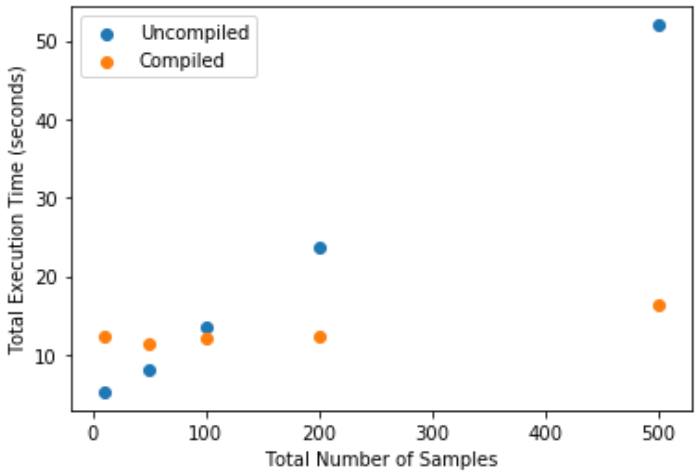
```
from time import time
from collections import defaultdict

N_samples = [10, 50, 100, 200, 500]
times = defaultdict(list)
for n_samples in N_samples:
    for compiled in [True, False]:
        s = time()
        run_compilation_benchmark(
            num_samples_from_dgp=n_samples,
            compile_functions=compiled)
        e = time()
        exec_time = e - s
        times[compiled].append(exec_time)
```

```
[48]:
```

```
import matplotlib.pyplot as plt

plt.scatter(N_samples, times[False], label="Uncompiled")
plt.scatter(N_samples, times[True], label="Compiled")
plt.legend()
plt.ylabel("Total Execution Time (seconds)")
plt.xlabel("Total Number of Samples")
plt.show()
```



Custom Metrics

It's possible to add both custom performance and custom data metrics to Maccabee. These metrics can then be collected alongside the built in metrics during benchmarking. The process for adding each kind of metric is outlined below.

Custom Performance Metrics

The function `add_performance_metric()` function can be used to add new performance metrics. This function takes a metric aggregation level, a metric name and a metric callable. See the function reference docs for more details on these arguments.

The code below adds a new performance metric. The new metric is the mean absolute error for average effect estimates. The mathematical formula for this metric is given below, using the notation from the `performance_metrics` module.

$$\frac{1}{N}\sum_i \left|\hat{\tau_i} - \tau_i\right|$$

```
[2]:
```

```
from maccabee.modeling.performance_metrics import add_performance_metric
from maccabee.constants import Constants


perf_metric_name = "MAE"

def mean_absolute_error(avg_effect_estimate_vals, avg_effect_true_vals):
    import numpy as np
    return np.mean(np.abs(avg_effect_estimate_vals - avg_effect_true_vals))

add_performance_metric(
    aggregation_level=Constants.Model.AVERAGE_ESTIMANDS,
    metric_name=perf_metric_name,
    metric_callable=mean_absolute_error)
```

Custom Data Metrics

The function `add_data_metric()` function can be used to add new data metrics. This function takes an axis name, a metric specification dictionary. See the function reference docs for more details on these arguments.

The code below adds a new data metric. The new metric is based on a function which combines the untreated potential outcome, the outcome noise and a constant offset and then regresses the result onto the observed covariates. This demonstrates the ability to access DGP variables and static values and combine them using arbitrary Python code to create new metrics.

```
[4]:
```

```
# Define axes and metrics to analyze
from maccabee.data_analysis.data_metrics import add_data_metric
from maccabee.constants import Constants
from sklearn.linear_model import LinearRegression

# Metric for the outcome nonlinearity axis
data_axis = Constants.AxisNames.OUTCOME_NONLINEARITY

# Define the callable function
def noisy_biased_outcome_linearity_metric(X, Y0, outcome_noise, bias):
    target = Y0 + outcome_noise + bias
    lr = LinearRegression().fit(X, target)
    return lr.score(X, target)

# Define the metric dict
data_metric_name = "lin (X, Noisy, Biased Y)"
metric_dict = {
    "name": data_metric_name,
    "args": {
        "X": Constants.DGPVariables.COVARIATES_NAME,
        "Y0": Constants.DGPVariables.POTENTIAL_OUTCOME_WITHOUT_TREATMENT_NAME,
        "outcome_noise": Constants.DGPVariables.OUTCOME_NOISE_NAME
    },
    "constant_args": {
        "bias": 42
    },
    "function": noisy_biased_outcome_linearity_metric
}

add_data_metric(data_axis, metric_dict)
```

Accessing Custom Metrics

We can now run a benchmark and show that our new performance and data metrics are immediately usable. The custom performance metric is automatically evaluated for all appropriate estimands. The custom data metric is selectable when running the benchmarking in *data_analysis_mode*.

```
[6]:
```

```
from maccabee.data_sources.data_source_builders import build_random_normal_datasource
from maccabee.benchmarking import benchmark_model_using_sampled_dgp
from maccabee.modeling.models import LinearRegressionCausalModel
from maccabee.parameters import build_default_parameters

# Build the parameters
params = build_default_parameters()

# Build a random normal data source
normal_data_source = build_random_normal_datasource(
    n_covars=5,
    n_observations=1000)

# Select the new data metric (and an old one for good measure)
DATA_METRICS_SPEC = {
    Constants.AxisNames.OUTCOME_NONLINEARITY: [
        data_metric_name,
        "Lin r2(X_obs, Y0)",
    ]
}

# Run a benchmark
perf_agg_metrics, _, _, data_agg_metrics, _, _ = benchmark_model_using_sampled_dgp(
    dgp_sampling_params=params,
    model_class=LinearRegressionCausalModel,
    estimand=Constants.Model.ATE_ESTIMAND,
    data_source=normal_data_source,
    num_dgp_samples=10,
    num_sampling_runs_per_dgp=1,
    num_samples_from_dgp=16,
    data_analysis_mode=True, # SET DATA ANALYSIS MODE
    data_metrics_spec=DATA_METRICS_SPEC, # PROVIDE SPEC
    n_jobs=8)
```

[7]:

```
# New performance metric
perf_agg_metrics[perf_metric_name]
```

[7]:

```
0.023
```

[8]:

```
# New data metric
data_agg_metrics[f"{data_axis} {data_metric_name}"]
```

[8]:

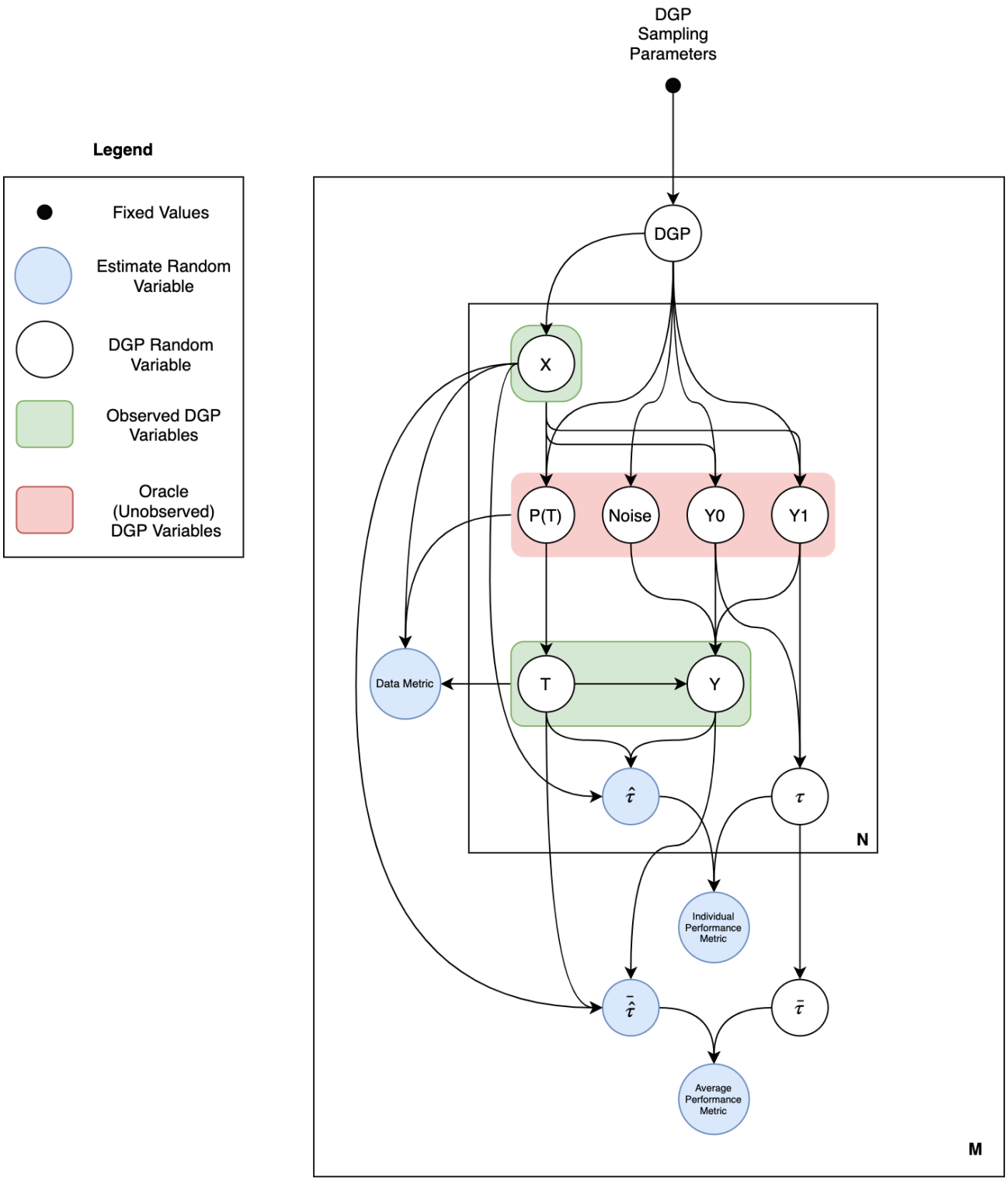
```
0.476
```

Design and Implementation

The goal of this page is to explain how Maccabee's theoretical approach to benchmarking - as laid out in Chapter 5 of the theory paper - is implemented using the functions and classes of the Maccabee package. This page therefore assumes a base level of familiarity with the theoretical approach although a brief summary is provided in the first section below.

Benchmarking Approach Summary

The graphical model below represents the complete statistical model of a Maccabee Monte Carlo benchmark.



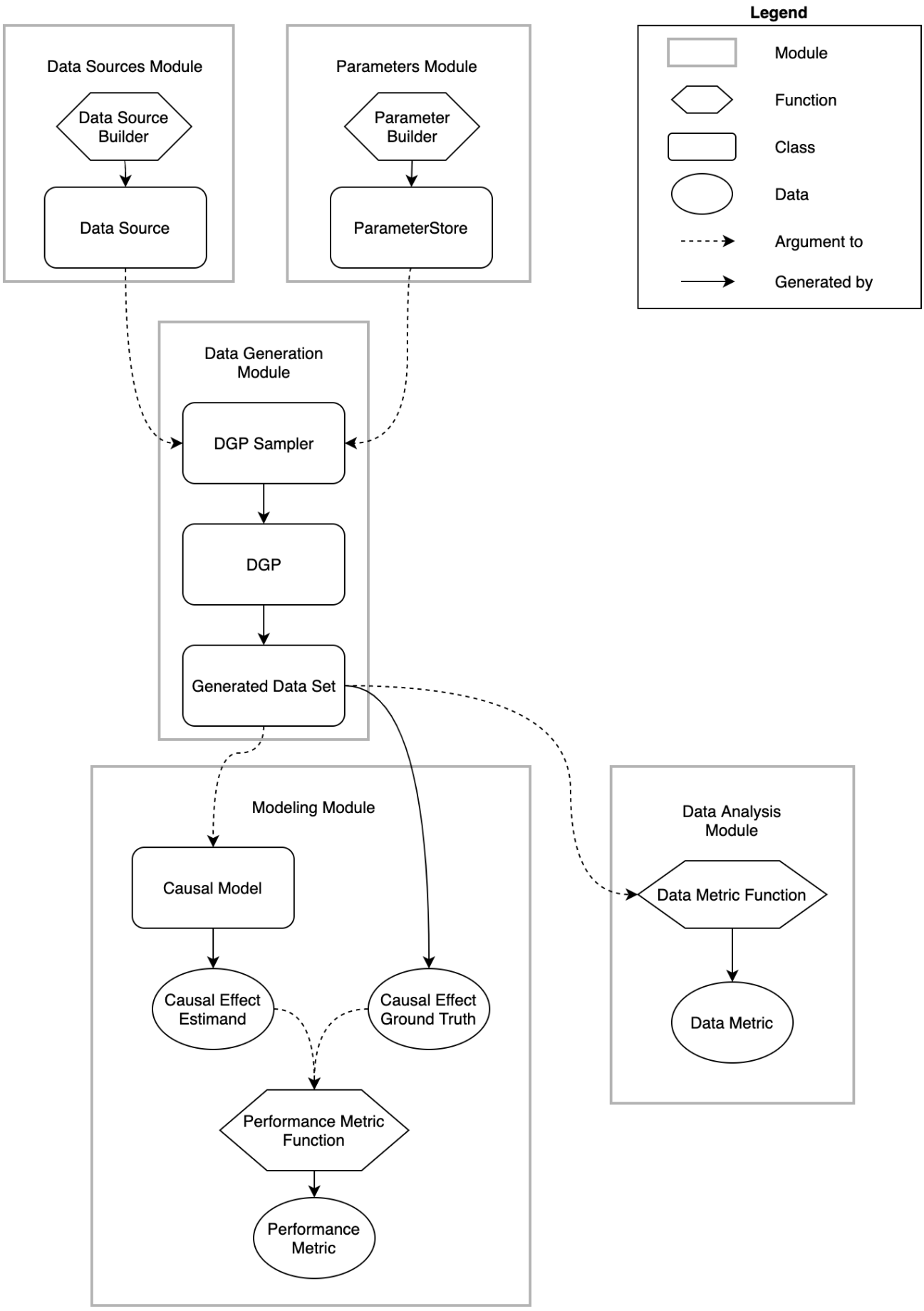
Briefly summarizing the sampling procedure implied by this model:

- A set of M DGPs is sampled based on supplied *DGP Sampling Parameters*.
- For each DGP, N sets of individual observation variables consisting of observed covariates and a set of associated treatment assignment, outcome and causal effect variables (both observed and unobserved). The complete set of variables is defined in `DGPVariables`.
- For each DGP, causal estimand values are sampled (perhaps deterministically) at either the individual observation (for individual effects) or dataset level (for average effects). These values are conditioned on all N of the X, T and Y observations.

- M Individual or Average Performance Metric values are calculated (deterministically sampled) at the dataset level by combining the causal effect estimate values with the appropriate ground truth value(s). Optionally, M Data Metrics are calculated by combining some/all of the covariate data with the observed and oracle outcome data.

Implementation Overview

The section above used a graphical model to describe the benchmarking approach at the level of the data (random variables). This section describes the components used to implement, and sample from, this model. This description is at the level of implemented functions and classes.



Core Sampling Execution Flow

The figure below shows how all of Maccabee's classes and functions fit together to perform a single sample of all of the random variables that appear in the graphical model above. Modules containing the closely related components are indicated using boxes. From top to bottom:

- A *data source builder* function from the `data_source_builders` module is used to build a `DataSource` instance. This class encapsulates the code needed to load and prepare empirical or synthetic covariate observations.

- A *parameter store builder* function from the `parameter_store_builders` module is used to build a `ParameterStore` instance. This class encapsulates the code used to modify and access the DGP sampling parameters.

- The two instances from the steps above are provided to a `DataGeneratingProcessSampler` instance. This class encapsulates the code required to sample Data Generating Processes defined over the covariate data from the `DataSource` instance based on the parameters in the `ParameterStore` instance.

- The `DataGeneratingProcessSampler` instance is used to sample `DataGeneratingProcess` instance. The `SampledDataGeneratingProcess` subclass, produced by the `DataGeneratingProcessSampler`, encapsulates the logic needed to sample datasets given the sampled components of a DGP as described in theory paper.

- The `SampledDataGeneratingProcess` instance is used to sample `GeneratedDataSet` instance. The `GeneratedDataSet` class encapsulates the logic used to access generated DGP variables (the observed and unobserved variables listed in `DGPVariables` over which the DGP is defined). This includes logic to access ground truth estimand values derived from the DGP variables.

- A `GeneratedDataSet` instance is passed to a `CausalModel` instance. This class encapsulates the (user-supplied) modeling logic that estimates causal estimands. The causal model class provides abstracts the details of the model and allows for simple external access to one or more estimands.

- The estimated causal estimand values (from the `CausalModel` instance) and the ground truth values (from the `GeneratedDataSet` instance) are passed to *performance metric functions* from the `performance_metrics` module (this is a submodule of the `modeling` module). Given the relative simplicity of the performance metric calculation, the functions from the `performance_metrics` module are used directly by code outside the module.

- Optionally, the `GeneratedDataSet` instance is passed to *data metric functions* from the `data_analysis` module. The data metric code is complex enough that the calculation of data metrics using *data metric functions* is encapsulated by the `calculate_data_axis_metrics()` function.

Sampling Execution Flow Notes

A few details are missing from the description of the sampling execution flow in the section above.

Firstly, most of the process above is not implemented directly by users. Rather, it is implemented in *benchmarking functions* from the `benchmarking` module. The exact process above is implemented in the `benchmark_model_using_sampled_dgp()` which accepts a `DataSource` instance and a `ParameterStore` instance from the user and then implements the rest of the process (sampling many `SampledDataGeneratingProcess` instances and many `GeneratedDataSet` instances from each DGP). The other functions in the `benchmarking` module support different use cases and these covered in the tutorials.

Second, there is nuance around how covariate data is handled relative to the formal statistical model. In the model, the covariates are sampled directly from a DGP. In the package, covariate sampling is encapsulated in a `DataSource` instance which is provided to the `DataGeneratingProcessSampler` instance. This is done for two reasons.

1. This encapsulates the complex logic needed to load empirical datasets or sample stochastic joint distributions and then normalize the resultant observations. Under the hood, the `DataGeneratingProcess` samples covariates from the `DataSource` as one would expect.
2. A sample of the covariate data is actually used by the `DataGeneratingProcessSampler` when normalizing the sampled treatment and outcome functions. This means the `DataGeneratingProcessSampler` needs access to covariate data **before** DGPs can be sampled.

If the user defines a custom `DataGeneratingProcess` class to represent a concrete DGP, then the choice of whether to use a `DataSource` or sample covariates directly in the `DataGeneratingProcess` is up to the user.

Finally, it is worth discussing the philosophy behind the choice to use classes vs functions to represent different components. In general, code that is stateless (doesn't preserve any information between runs) is implemented using functions. This applies to the *_builder* functions, metric

calculation functions, and the benchmarking functions. Note that, where possible, code executed directly by users is designed to be stateless to allow for execution without the overhead of instance creation and management. Code that is stateful, and called repeatedly, is implemented using classes. Both the functional and class based components are customizable. For example, users can inject their own performance/data metrics as demonstrated in Custom Metrics and subclass the `DataGeneratingProcess` class to benchmark using concrete DGPs as demonstrated in Benchmarking with a Concrete DGP.

DGP Sampling Parameterization

The theory paper outlined the detailed steps used to sample DGPs. This section explains how the paramterization of these steps is implemented in Maccabee. The DGP sampling procedure from the theory paper is summarized below, with the concrete Maccabee parameters given alongside each step. For full explanations of each step, see the theory paper. For details on the specification and constraints for each parameter see the parameter_schema.yml file. For more on how to use the parameters listed below to run benchmarks see the usage tutorials and the documentation for the `parameters` module.

Note: in the current implementation, functional parameters (sampling distributions) are hard coded and therefore are indicated with lower case function names. Dynamic functional parameters are planned for future releases.

1. A set of potential confounders is selected from all of the covariates based on a single, uniform selection probability. This selection probability is the `POTENTIAL_CONFOUNDER_SELECTION_PROBABILITY` parameter.
2. The treatment assignment and untreated outcome functions are sampled from two independently parameterized distributions. Each distribution is parameterized by the probability of selecting each *term type* from the universe of valid terms for that *term type* given the covariates. These probabilities are given in the `TREAT_MECHANISM_COVARIATE_SELECTION_PROBABILITY` and `OUTCOME_MECHANISM_COVARIATE_SELECTION_PROBABILITY` dictionaries respectively. The co-efficients in all of the terms of two functions are initialized by drawing samples from the global co-efficient distribution using the functional parameter `sample_subfunction_constants`.
3. An alignment (confounding) adjustment is performed. This step involves randomly selecting terms to add/remove from the sampled treatment assignment and untreated outcome functions to meet a target alignment parameter. This parameter specifies the target probability for a term in the untreated outcome function to also appears in the treatment assignment function. This target probability is the `ACTUAL_CONFOUNDER_ALIGNMENT` parameter.
4. The treatment effect mechanism is then sampled. First, a constant treatment effect is sampled from a parameterized, global treatment effect distribution using the functional parameter `sample_treatment_effect`. Then, terms from the untreated outcome function are sampled based on a treatment effect heterogeneity parameter. This parameter specifies the uniform probability that a term in the untreated outcome function appears in the treatment effect function. This probability is the `TREATMENT_EFFECT_HETEROGENEITY` parameter.

5. Numerical normalization and linear modification (by multiplicative/additive constants) of the treatment assignment function is used to meet target parameters for the expected max value of the treatment probability. This expected value is the `TARGET_PROPENSITY_SCORE` parameter.
6. Numerical normalization and linear modification (by multiplicative/additive constants) of the untreated outcome function is applied to adjust the signal-to-noise ratio of the observed outcome. This is an automatic adjustment without parameterization.
7. Finally, some components of the DGP sampling occur at data generation time. On each instance of data generation:

    1. A fixed proportion of the covariates from a data source sample are sampled. This proportion is given by the `OBSERVATION_PROBABILITY` parameter.
    2. Observed outcome noise is sampled from a parameterized, global outcome noise distribution using the functional parameter `sample_outcome_noise`.
    3. A parameterized target proportion of the units in the treated and control groups with the lowest/highest probability of treatment respectively are swapped between the groups. This proportion is controlled by the `FORCED_IMBALANCE_ADJUSTMENT` parameter.

Implementation Details

This section of the documentation covers the details of the implementation in Maccabee.

- Pandas for data management
- Abstract Syntax Trees for equation construction
- Process-based Parallelism
- OOP practices.

`maccabee` package reference

`maccabee.benchmarking`

The benchmarking module contains the code responsible for running Monte Carlo trials and collecting metrics which measure estimator performance (performance metrics) and data distributional settings (data metrics). This module is responsible for *execution* of the experiments and metric functions. The metric functions themselves are defined alongside the objects which they measure. See `maccabee.modeling.performance_metrics` for performance metrics and `maccabee.data_analysis.data_metrics` for data metrics

> **Note**
>
> For now, this module is made up of one submodule. In future releases, it may be broken into smaller, more specialized submodules. For convenience, all functions from the single submodule of this module can be imported directly from the module itself.

`benchmarking.benchmarking`

This submodule consists of a series of three independent but functionally 'nested' benchmarking functions. Each function can be used on its own but is also used by the functions higher up the nesting hierarchy.

- `benchmark_model_using_concrete_dgp()` is the basic benchmarking function. It takes a single `DataGeneratingProcess` instance and evaluates a estimator using data sets sampled from the DGP represented by the instance. The collected metrics are aggregated at two levels: the metric values for multiple samples are averaged and the resultant average metric values are then averaged across sampling runs. This two-level aggregation allows for the calculation of a standard deviation for the performance metrics that are defined over multiple sample estimand values. For example - the absolute bias is found by averaging the signed error in the estimate over many trials. The standard deviation of the bias estimate thus requires multiple sampling runs each producing a single bias measure.

- `benchmark_model_using_sampled_dgp()` is the next function up the benchmarking hierarchy. It takes DGP sampling parameters in the form of a `ParameterStore` instance and a `DataSource` instance. It then samples DGPs based on the sampling parameters and uses `benchmarkingbenchmark_model_using_concrete_dgp()` to collect metrics for each sampled DGP. This introduces a third aggregation level, with metrics (and associated standard deviations) being averaged over a number of sampled DGPs.

- `benchmark_model_using_sampled_dgp_grid()` is the next and final function up the benchmarking hierarchy. It takes a grid of sampling parameters corresponding to different levels of one of more data axes and then samples DGPs from each combination of sampling parameters in the grid using `benchmark_model_using_sampled_dgp()`. There is no additional aggregation as the metrics for each parameter combination are reported individually.

---

benchmark_model_using_concrete_dgp(*dgp, model_class, estimand, num_sampling_runs_per_dgp, num_samples_from_dgp, data_analysis_mode=False, data_metrics_spec=None, n_jobs=1*)

Sample data sets from the given DGP instance and calculate performance and (optionally) data metrics.

> Parameters

- **dgp** (`DataGeneratingProcess`) – A DGP instance produced by a sampling procedure or through a concrete definition.
- **model_class** (`CausalModel`) – A model instance defined by subclassing the base `CausalModel` or using one of the included model types.

- **estimand** (*string*) – A string describing the estimand. The class `maccabee.constants.Constants.Model` contains constants which can be used to specify the allowed estimands.
- **num_sampling_runs_per_dgp** (*int*) – The number of sampling runs to perform. Each run is comprised of *num_samples_from_dgp* data set samples which are passed to the metric functions.
- **num_samples_from_dgp** (*int*) – The number of data sets sampled from the DGP per sampling run.
- **data_analysis_mode** (*bool*) – If `True`, data metrics are calculated according to the supplied *data_metrics_spec*. This can be slow and may be unecessary. Defaults to True.
- **data_metrics_spec** (*type*) – A dictionary which specifies which data metrics to calculate and record. The keys are axis names and the values are lists of string metric names. All axis names and the metrics for each axis are available in the dictionary `maccabee.data_analysis.data_metrics.AXES_AND_METRIC_NAMES`. If None, all data metrics are calculated. Defaults to None.
- **n_jobs** (*int*) – The number of processes on which to run the benchmark. Defaults to 1.

Returns

A tuple with four entries. The first entry is a dictionary of aggregated performance metrics mapping names to numerical results aggregated across runs. The second entry is a dictionary of raw performance metrics mapping metric names to lists of numerical metric values from each run (averaged only across the samples in the run). This is useful for understanding the metric value distribution. The third and fourth entries are analogous dictionaries which contain the data metrics. They are empty dicts if *data_analysis_mode* is `False`.

Return type

tuple

Raises

UnknownEstimandException – If an unknown estimand is supplied.

`benchmark_model_using_sampled_dgp`(*dgp_sampling_params, data_source, model_class, estimand, num_dgp_samples, num_samples_from_dgp, num_sampling_runs_per_dgp=1, data_analysis_mode=False, data_metrics_spec=None, data_metric_intervals=False, dgp_class=<class 'maccabee.data_generation.data_generating_process.SampledDataGeneratingProcess'>, dgp_kwargs={}, n_jobs=1, compile_functions=False*)

Short summary.

Parameters

- **dgp_sampling_params** (`ParameterStore`) – A `ParameterStore` instance which contains the DGP sampling parameters which will be used when sampling DGPs.
- **data_source** (`DataSource`) – a `DataSource` instance which will be used as the source of covariates for sampled DGPs.

- **model_class** (`CausalModel`) – A model instance defined by subclassing the base `CausalModel` or using one of the included model types.
- **estimand** (*string*) – A string describing the estimand. The class `maccabee.constants.Constants.Model` contains constants which can be used to specify the allowed estimands.
- **num_dgp_samples** (*int*) – The number of DGPs to sample. Each sampled DGP is benchmarked using `benchmark_model_using_concrete_dgp()`.
- **num_samples_from_dgp** (*int*) – See `benchmark_model_using_concrete_dgp()`.
- **num_sampling_runs_per_dgp** (*int*) – See `benchmark_model_using_concrete_dgp()`. Defaults to 1.
- **data_analysis_mode** (*bool*) – See `benchmark_model_using_concrete_dgp()`. Defaults to False.
- **data_metrics_spec** (*dict*) – See `benchmark_model_using_concrete_dgp()`. Defaults to None.
- **dgp_class** (`maccabee.data_generation.data_generating_process.SampledDataGeneratingProcess`) – The DGP class to instantiate after function sampling. This must be a subclass of `maccabee.data_generation.data_generating_process.SampledDataGeneratingProcess`. This can be used to tweak aspects of the default sampled DGP. Defaults to SampledDataGeneratingProcess.
- **dgp_kwargs** (*dict*) – A dictionary of keyword arguments to pass to the sampled DGPs at instantion time. Defaults to {}.
- **n_jobs** (*int*) – See `benchmark_model_using_concrete_dgp()`. Defaults to 1.
- **compile_functions** (*bool*) – A boolean indicating whether sampling DGP functions should be compiled prior to execution. Defaults to `False`.

Returns

A tuple with four entries. See `benchmark_model_using_concrete_dgp()` for a description of the entries but note that, in this func, the aggregate metric values are averaged across dgp samples and sampling runs and the raw metric values correspond to averages over sampling runs for each sampled DGP. This means each entry in the raw metrics list corresponds to the aggregated result of the `benchmark_model_using_concrete_dgp()` function.

Return type

tuple

Raises

UnknownEstimandException – If an unknown estimand is supplied.

---

benchmark_model_using_sampled_dgp_grid(*dgp_param_grid, data_source, model_class, estimand, num_dgp_samples, num_samples_from_dgp, num_sampling_runs_per_dgp=1, data_analysis_mode=False,*

data_metrics_spec=None, data_metric_intervals=True, param_overrides={}, dgp_class=<class 'maccabee.data_generation.data_generating_process.SampledDataGeneratingProcess'>, dgp_kwargs={}, n_jobs=1, compile_functions=False)

This function is a thin wrapper around the `benchmark_model_using_sampled_dgp()` function. It is used to run the sampeld DGP benchmark across many different sampling parameter value combinations. The signature is the same as the wrapped function with *dgp_sampling_params* replaced by *dgp_param_grid* and the new *param_overrides* option. For all other arguments, see `benchmark_model_using_sampled_dgp()`.

Parameters

- **dgp_param_grid** (*dict*) – A dictionary mapping data axis names to a list of data axis levels. Axis names are available as constants in `maccabee.constants.Constants.AxisNames` and axis levels available as constants in `maccabee.constants.Constants.AxisLevels`. The `benchmark_model_using_sampled_dgp()` function is called for each combination of axis level values - the cartesian product of the lists in the dictionary.
- **param_overrides** (*dict*) – A dictionary mapping parameter names to values of those parameters. The values in this dict override the values in the grid and any default parameter values. For all available parameter names and allowed values, see the parameter_schema.yml file.

Returns

A `DataFrame` containing one row per axis level combination and a column for each axis and each performance and data metric (as well as their standard deviations).

Return type

`DataFrame`

`maccabee.constants`

This module contains constants which are used throughout the package. The constants defined here serve two purposes: First, to simplify and standardize user interaction with the external APIs by providing users a convenient way to refer to common concepts/values. Second, to centralize the configuaration constants which control the internal operation of the package, giving more advanced users the ability to control/modify operation by making changes in one location.

All Maccabee constants are stored as attributes of the `maccabee.constants.Constants` class. Within this class, constants are nested into subclasses of the `ConstantGroup` class (IE, the only attributes of the `Constants` class are `ConstantGroup` subclasses). These subclasses store actual parameters as their attributes. So, for example, the axis names for the axes of the distributional problem space can be accessed as the attributes of the class `Constants.AxisNames`.

All of the `ConstantGroup` classes can be introspected to view the constant names/values stored in the group. This is done by calling the `.all()` method which will return a name/value dictionary. If the `print=True` option is supplied this will (pretty) print the dictionary.

So, to access the axis name constants mentioned above, one first imports the constants class and then uses the `AxisNames` attribute. Because this attribute is a subclass of the `ConstantGroup` class, the `all()` method can be used to introspect the values.

```
>>> from maccabee.constants import Constants
>>> Constants.AxisNames.all()
    { 'ALIGNMENT': 'ALIGNMENT',
      'BALANCE': 'BALANCE',
      'OUTCOME_NONLINEARITY': 'OUTCOME_NONLINEARITY',
      'OVERLAP': 'OVERLAP',
      'PERCENT_TREATED': 'PERCENT_TREATED',
      'TE_HETEROGENEITY': 'TE_HETEROGENEITY',
      'TREATMENT_NONLINEARITY': 'TREATMENT_NONLINEARITY'}
```

class **Constants**

Bases: `object`

As discussed above, this class contains the constants used throughout the package. All of the `ConstantGroup` attribute classes are listed below. Constants which are predominantly for internal use are marked [INTERNAL] and do not have explanations here. Advanced users interested in modifying the value of these constants should see the source code and inline comes in constants.py.

class **ParamFilesAndPaths**

Bases: `object`

[INTERNAL] Constants related to the location and parsed content of the YAML parameter specification files which control the parameter schema, default parameter values and metric-level parameter values. See the docs for the `maccabee.parameters` module for more detail.

class **ParamSchemaKeysAndVals**

Bases: `maccabee.constants.ConstantGroup`

[INTERNAL] Constants related to the keys and values of the parameter specification files mentioned under `ParamFilesAndPaths`.

class **DGPSampling**

Bases: `maccabee.constants.ConstantGroup`

[INTERNAL] Constants related to the sampling of the subfunctions which make up the sampled DGP treatment and outcome functions.

class **DGPVariables**

Bases: `maccabee.constants.ConstantGroup`

Constants related to the naming of the variables over which DGPs are defined. These are used when specifying concrete DGPs using the `ConcreteDataGeneratingProcess` class and when interacting with the data in the `GeneratedDataSet` instances produced when sampling from any `DataGeneratingProcess` instance.

`COVARIATES_NAME` *= 'X'*

The collective name for the observed covariates for each individual observation in the data set.

`TRANSFORMED_COVARIATES_NAME` *= 'X_transformed'*

Transformed covariates are produced by applying all of the sampled subfunctions to the original covariates. Because the treatment and outcome functions are made up of some combination of these subfunctions, the values produced by each subfunction can be thought of as the true covariates of the treatment and outcome functions.

`PROPENSITY_LOGIT_NAME` *= 'logit_p_score'*

The logit of the true probability of/propensity for treatment.

`PROPENSITY_SCORE_NAME` *= 'p_score'*

The true probability of/propensity for treatment.

`TREATMENT_ASSIGNMENT_NAME` *= 'T'*

The treatment assignment/status

`OBSERVED_OUTCOME_NAME` *= 'Y'*

The observed outcome.

`POTENTIAL_OUTCOME_WITHOUT_TREATMENT_NAME` *= 'Y0'*

The potential outcome without treatment, understood in terms of the Rubin-Neyman causal model.

`POTENTIAL_OUTCOME_WITH_TREATMENT_NAME` *= 'Y1'*

The potential outcome with treatment, understood in terms of the Rubin-Neyman causal model.

`TREATMENT_EFFECT_NAME` *= 'treatment_effect'*

The true treatment effect, the different between the potential outcome with and without treatment.

`OUTCOME_NOISE_NAME` *= 'Y_noise'*

The noise in the observation of the units potential outcomes.

class **AxisNames**

Bases: `maccabee.constants.ConstantGroup`

Constants for the names of the axes of the distributional problem space. Maccabee allows for the sampling of DGPs (and associated data samples) at different 'levels' (locations) along these axes. See the Design doc for more on Maccabee's theoretical approach.

OUTCOME_NONLINEARITY= *'OUTCOME_NONLINEARITY'*

The outcome nonlinearity axis - controls the degree of nonlinearity in the outcome mechanism.

TREATMENT_NONLINEARITY= *'TREATMENT_NONLINEARITY'*

The treatment nonlinearity axis - controls the degree of nonlinearity in the treatment assignment mechanism.

PERCENT_TREATED= *'PERCENT_TREATED'*

The percent treated axis - controls the percent of units that are exposed to treatment.

OVERLAP= *'OVERLAP'*

The overlap axis - controls to covariate distribution overlap in the treated and control groups. WARNING: not currently supported in sampling.

BALANCE= *'BALANCE'*

The balance axis - controls the degree of similarity between the covariate distribution in the treated and control group.

ALIGNMENT= *'ALIGNMENT'*

The alignment axis - controls the degree of overlap of appearance of covariates in the treatment and outcome mechanisms. Effectively controlling the number of confounders and ratio of confounders to non-confounders.

TE_HETEROGENEITY= *'TE_HETEROGENEITY'*

The treatment effect heterogeneity axis - controls the degree to which the treatment effect varies per unit.

class **AxisLevels**

Bases: `maccabee.constants.ConstantGroup`

Constants related to the 'levels' of each axis at which Maccabee can sample DGPs. These levels represent the existing presets but do not preclude sampling at any other point by manually specifying sampling parameters. See `AxisNames` for links to further explanation.

LOW= *'LOW'*

The constant for a ‘low’ level on a specific axis.

MEDIUM= *'MEDIUM'*

The constant for a ‘medium’ level on a specific axis.

HIGH= *'HIGH'*

The constant for a ‘high’ level on a specific axis.

LEVELS= *('LOW', 'MEDIUM', 'HIGH')*

The constant for conveniently loading all of the level constants.

class **DataMetricFunctions**

Bases: `maccabee.constants.ConstantGroup`

[INTERNAL] Constants related to the functions used to calculate the metrics which quantify the location of data in the distributional problem space.

class **ExternalCovariateData**

Bases: `maccabee.constants.ConstantGroup`

[INTERNAL] Constants related to external covariate data. See the doc on the empirical data sets built into Maccabee for more detail.

class **Model**

Bases: `maccabee.constants.ConstantGroup`

Constants related to models and estimands.

ITE_ESTIMAND= *'ITE'*

The Individual Treatment Effect estimand.

ATE_ESTIMAND= *'ATE'*

The Average Treatment Effect estimand.

ATT_ESTIMAND= *'ATT'*

The Average Treatment Effect for the Treated estimand.

ALL_ESTIMANDS= *['ITE', 'ATE', 'ATT']*

A list of all estimands supported by the package.

AVERAGE_ESTIMANDS= *['ATE', 'ATT']*

A list of all estimands which target a sample-level average effect

> **INDIVIDUAL_ESTIMANDS**= *['ITE']*
>
> A list of all estimands which target an observation-level individual effect

`maccabee.data_analysis`

The data analysis module contains the code responsible for calculating data metrics - metrics which quantify the location of a data set in the distributional problem space. More on the theory behind these metrics can be found in Chapter 3 of the theory paper.

The module is not responsible for actually executing these calculations, that is handled by the `maccabee.benchmarking` module. Rather, this module is responsible for defining the actual metrics used to quantify the location of a data set on each distributional problem space axis and providing wrapper functionality to calculate multiple metrics given a `GeneratedDataSet` instance.

This module is split into two submodules. `data_metrics` contains the metric definitions and `data_analysis` contains the code which calculates these metrics given a `maccabee.data_generation.generated_data_set.GeneratedDataSet` instance and code which plots calculated metric results.

> **Note**
>
> For convenience, all the classes and functions that are split across the submodules below can be imported directly from the parent `maccabee.data_analysis` module.

`data_analysis.data_metrics`

This submodule contains the definitions for the metrics used to quantify the position of a data set on each of the axes of the distributional problem space. Each axis has one or more metrics, each of which operates on (potentially overlapping) components of the data set to output a single, real metric value that measures the data's position on the associated axis.

The module uses a dictionary-based data structure to define data metrics. IE, rather than concretely defining metrics for each axis as python functions which extract the relevant data from a `GeneratedDataSet` instance and perform some calculation, metrics are defined using dictionaries specify which data and functions to (re)use. This allows different concrete metrics to share the same data and calculation functions without code repetition. It also allows package users to inject new metrics at run time by simply modifying the dictionaries outlined below.

The module actually uses, and exposes, three dictionaries which work together to define the data metrics. The main dictionary is the `AXES_AND_METRICS` dictionary. The other two dictionaries support the specification and use of the main dictionary. `AXES_AND_METRIC_NAMES` summarizes the content of `AXES_AND_METRICS` by mapping the axis names to a list of unique metric names. This can be used for convenient selection of the metrics to record when running a benchmark (see the

`benchmarking` module for more). The `AXIS_METRIC_FUNCTIONS` maps function name constants from `maccabee.constants.Constants.DataMetricFunctions` to generic calculation functions/callables defined in this module.

The main `AXES_AND_METRICS` dictionary defines the data metrics by mapping each axis name from `maccabee.constants.Constants.AxisNames` to a list of *metric definition dictionaries*. Each metric definition dictionary has three components:

- The unique name of the metric. This is the name which appears in the `AXES_AND_METRIC_NAMES` dictionary and is used when specifying which metrics to collect during a benchmark.
- The generic metric function used to calculate the metric. This is specified as a function name constant from `maccabee.constants.Constants.DataMetricFunctions`. As mentioned above, the dictionary `AXIS_METRIC_FUNCTIONS` maps these names to callables defined in this module. This structure is used because the functions are typically repeated across many different concrete metrics. So it is efficient to define generic functions once and reuse the same callable repeatedly. For example, there is a linear regression R^2 function which regresses the vector arg y against the matrix arg X.
- The arguments to the generic metric function. These concretize what the metric measures by applying the generic function to specific data. For example, by passing the original covariates and observed outcome as the arguments X and y of the linear regression function, one can construct a metric for the linearity of the outcome. The arguments are specified by a dictionary which maps the generic functions (generic) argument names to DGP data variable names from `maccabee.constants.Constants.DGPVariables`. These constant names are then used to access the corresponding data from `GeneratedDataSet` instances.

AXES_AND_METRICS - *{axis_name: [{"name": "...", "function": "...", "args": {...}}]}*

The dictionary mapping axis names to a list of metric definition dictionaries. Each metric definition dictionary has a name, calculation function, and argument mapping that specifies which DGP variables to supply as each function arg.

AXES_AND_METRIC_NAMES - *{axis_name: [metric_name]}*

The dictionary mapping axis names to a list of available metric names.

AXIS_METRIC_FUNCTIONS - *{metric_function_name: metric_function_callable}*

The dictionary mapping constant metric function names to function callables from this module.

add_data_metric(*axis_name, metric_dict*)

Add a data metric specified by the components of *metric_dict* to the metrics for the axis in *axis_name*.

Parameters

- **axis_name** (*str*) – The name of an axis from `AxisNames`.

- **metric_dict** (*dict*) – A dict, as described above, which contains keys for the name, args and function that is used to calculate the metric.

`data_analysis.data_analysis`

This submodule contains the functions responsible for calculating the metrics used to quantify the position of a data set on each of the axes of the distributional problem space. As described in the sibling `data_metrics` submodule, each axis may have multiple metrics. The primary function in this module takes a `maccabee.data_generation.generated_data_set.GeneratedDataSet` instance and an *observation_spec* which selects which axes and associated metrics to calculate. It then executes the calculation using the dictionary-based metric definitions from `data_metrics`.

`calculate_data_axis_metrics`(*dataset, observation_spec=None, flatten_result=False*)

This function takes a `maccabee.data_generation.generated_data_set.GeneratedDataSet` instance and calculates the data metrics specified in *observation_spec*. It is primarily during the benchmarking process but can be used as a stand alone method for custom workflows.

Parameters

- **dataset** (`GeneratedDataSet`) – A `GeneratedDataSet` instance generated from a `DataGeneratingProcess`.
- **observation_spec** (*dict*) – A dictionary which specifies which data metrics to calculate and record. The keys are axis names and the values are lists of string metric names. All axis names and the metrics for each axis are available in the dictionary `maccabee.data_analysis.data_metrics.AXES_AND_METRIC_NAMES`. If None, all data metrics are calculated. Defaults to None.
- **flatten_result** (*bool*) – indicates whether the results should be flattened into a single dictionary by concatenating the axis and metric names into a single key rather than returning nested dictionaries with axis and metric names as the keys at the first and second level of nesting.

Returns

A dictionary of axis names and associated metric results. If flatten_result is `False`, then this is a dictionary in which axis names are mapped to dictionaries with metric name keys and real valued values. If flatten_result is `True`, then this is a dictionary in which the keys are the concatenation of axis and metric names and the values are the corresponding real values.

Return type

dict

Raises

UnknownDGPVariableException – if a selected metric function specifies an unknown DGP variable as an arg to its calculation function.

`maccabee.data_generation`

This module contains the classes and functions responsible for data generation. The `DataGeneratingProcess` class is central to the data generation process: `DataGeneratingProcess` instances are used to sample `DataGeneratingProcess` instances (sampled DGPs). `DataGeneratingProcess` instances - either sampled as above or concretely defined - are then used to sample `GeneratedDataSet` instances (sampled data sets). Models are then benchmarked against these sampled data sets.

This module is comprised of three submodules which align with the three components of the data generation process as outlined above.

> **Note**
>
> For convenience, all the classes and functions that are split across the three submodules below can be imported directly from the parent `maccabee.data_generation` module.

`data_generation.data_generating_process_sampler`

This module contains the `DataGeneratingProcess` class which is used to sample DGPs given sampling parameters which determine where in the distributional problem space the `DataGeneratingProcess` targets for sampling.

class `DataGeneratingProcessSampler`(...)

The `DataGeneratingProcessSampler` class takes a set of sampling parameters and a data source (a `DataSource` instance) that provides the base covariates. It then samples the treatment assignment and outcome functions which, in combination with the observed covariate data from the `DataSource` completely specify the DGP. The two functions are sampled based on the provided sampling parameters to target a location in the distributional problem space.

The `DataGeneratingProcessSampler` class is designed to work for most users and use cases. However, it is also designed to be customized through inheritance in order to cater to the needs of advanced users. This is achieved by breaking down the DGP sampling process into a series of steps corresponding to class methods which can be overridden individually. Users interested in this should see the linked source code for extensive in-line guiding comments.

> Parameters

- **parameters** (`ParameterStore`) – A `ParameterStore` instance which contains the parameters that control the sampling process. See the `maccabee.parameters` module docs for more detail on how to build a `ParameterStore` instance.
- **data_source** (`DataSource`) – A `DataSource` instance which provides observed covariates. See the `data_sources` module docs for more detail.

- **dgp_class** (`SampledDataGeneratingProcess`) – A class which inherits from `SampledDataGeneratingProcess` . Defaults to `SampledDataGeneratingProcess` . This is only necessary if you would like to customize some aspect of the sampled DGP which is not controllable through the sampling parameters provided in *parameters*.
- **dgp_kwargs** (*dict*) – A dictionary of keyword arguments which is passed to the sampled DGP at instantiation. Defaults to {}.

`sample_dgp`()

This is the primary external method of this class. It is used to sample a new DGP. Internally, a number of steps are executed:

- A set of observable covariates is sampled from the `DataSource` supplied at instantiation.
- A subset of the observable covariates is sampled based on the observation likelihood parameterization.
- Potential confounders are sampled. These are the covariates which could enter either or both of the treatment and outcome function. Covariates not sampled here are nuisance/non-predictive covariates.
- The treatment and outcome subfunctions, the transformations which make up the two main functions, are sampled according to the desired parameters for function form and alignment (the degree of confounding).
- The treatment and outcome functions are assembled and normalized to meet parameters for target propensity score and treatment effect heterogeneity.
- The DGP instance is assembled using the class and kwargs supplied at instantiation time and the components produced by the steps above.

Returns

A `SampledDataGeneratingProcess` instance representing a sampled DGP.

Return type

`SampledDataGeneratingProcess`

`data_generation.data_generating_process`

This module contains the `DataGeneratingProcess` base class that is used to represent sampled and concrete DGPs. Within this class, the DGP is represented as a series of data generating methods which each produce a DGP variable and can depend on other, previously generated, DGP variables. The methods are called in a predetermined order in the main `generate_dataset()` method in order to sample a data set.

The base `DataGeneratingProcess` class and its inheriting classes make use of a minimal DSL which is used to specify the data flow in the DGP. This DSL reduces boilerplate code by automatically managing dgp method dependencies, outputs, and execution. In the future, it will allow for

advanced features like dependency resolution and parralelism. The DSL is used by decorating all data generating methods in a `DataGeneratingProcess` class with the `data_generating_method()` decorator. This decorator is parameterized with the DGP variables which the method required, the DGP variable it produces, and other options. See the documentation below for more detail.

The documentation below explains the `data_generating_method()` decorator and the `DataGeneratingProcess` class (and its data generating methods) in more detail.

`@data_generating_method`(*generated_var, required_vars, optional=False, data_analysis_mode_only=False, cache_result=False*)

This DGP DSL decorator is applied to all of the `_generate_*` methods which comprise the definition of a `DataGeneratingProcess` class. The decorator takes parameters which describe the DGP variables which the generating method requires and generates and a number of other parameters relevant to execution.

> **Parameters**

- **generated_var** (*string*) – A string DGP variable name from `DGPVariables`. This is the DGP variable which the decorated method generates.
- **required_vars** (*list*) – A list of string DGP variable names from `DGPVariables`. These are DGP variables which the decorated method requires to generate its variable. The values of these variables are passed as a dictionary to the decorated method as the first position argument.
- **optional** (*bool*) – Indicates whether the decorated method is optional. If `True`, the method will only run if its requirements are satisfied and there will not be an exception raised if its requirements are missing. If `False`, there will be an exception if required variables are missing at execution time. Defaults to `False`.
- **data_analysis_mode_only** (*bool*) – Indicates if the decorated method should only be run if the DGP is in data analysis mode. IE, the generated DGP variable is required only for data metric calculation. Defaults to False.
- **cache_result** (*bool*) – Indicates whether the result of the decorated method should be cached so that all samples from the DGP after the first will have the same value of the generated variable. Defaults to False.

Warning

If `cache_result=True` and the decorated method depends on other variables which change (IE, they are not cached), the changes to these variables will not reflect in the variable generated by the decorated method.

> **Raises**

DGPVariableMissingException – if a non-optional decorated method is missing its required variables at execution time.

---

class **DataGeneratingProcess**(*n_observations*, *data_analysis_mode=False*)

This class represents a Data Generating Process. A DGP relates the DGP Variables - defined in the constants group `DGPVariables` - through a series of stochastic/deterministic 'data generating functions'. The nature of these functions defines the location of the resultant data sets in the distributional problem space.

This is the base DGP class. It defines the data generating functions which make up a DGP by generating all of the required DGP variables. These functions are defined without providing concrete implementations (with exceptions made for the data generating functions that are reasonably generic). Parameterized DGP DSL decorators are provided for guidance (see the source code). Inheriting classes are expected to provide the data generating function implementations and to redecorate implemented methods (repeating the generated variable and specifying the correct dependencies and execution options). All data generating functions with concrete implementatins are marked with [CONCRETE].

This class does define a concrete `generate_dataset()` method which specifies the order of execution of the data generating functions and constructs a `GeneratedDataSet` instance from the DGP variables produced by the data generating functions.

Parameters

- **n_observations** (*int*) – The number of observations which will be present in sampled data sets. This value is used throughout Maccabee to build the correct data structures (and it is useful throughout this class) so it must be specified priori to sampling.
- **data_analysis_mode** (*bool*) – Indicates whether the DGP should be run in data analysis mode. This will execute all data generating methods marked as *data_analysis_mode_only* in order to generate the dgp variables which are only used in calculating data metrics. Defaults to False.

n_observations

data_analysis_mode

generate_dataset()

This is the primary external API method of this class. It is used to sample a data set (in the form of a `GeneratedDataSet` instance) from the DGP.

Returns

a sampled `GeneratedDataSet` instance.

Return type

`GeneratedDataSet`

Raises

DGPVariableMissingException – If the execution order of the data generating methods is in conflict with their specified requirements such that a method's dependencies haven't been generated when it is executed.

`_generate_observed_covars`(...)

Generate the observed covariates (`DGPVariables.COVARIATES_NAME`). It is likely that this function can be implemented by using the `get_covar_df()` method of a `DataSource` instance.

Returns

a `DataFrame` containing the covariate observations. It must contain *n_observations* rows.

Return type

`pandas.DataFrame`

`_generate_transformed_covars`(...)

Generate the transformed covariates (`DGPVariables.TRANSFORMED_COVARIATES_NAME`). This is only possible if the treatment/outcome functions are additive functions of arbitrary covariate transforms. The method is typically used in data analysis mode only as this DGP variable is not required for causal inference.

Returns

a `DataFrame` containing the transformed covariate observations. It must contain *n_observations* rows.

Return type

`pandas.DataFrame`

`_generate_true_propensity_scores`(...)

Generate the true propensity scores for each observed unit (`DGPVariables.PROPENSITY_SCORE_NAME`). This implements the treatment assignment function if the function naturally generates probabilities of treatment. It is not necessary as treatment assignments can be generated directly. See `_generate_treatment_assignments()` .

Returns

an array containing the probability of treatment. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

`_generate_true_propensity_score_logits`(...)

[CONCRETE] Generate the true propensity score logits for each observed unit (`DGPVariables.PROPENSITY_LOGIT_NAME`). A concrete implementation is provided which simply calculates the logit of the propensity scores generated by `_generate_true_propensity_scores()`. In most common use cases, this will be a data analysis only mode function as the logits are not required for causal inference or treatment assignment (if the propensities are known).

Returns

an array containing the logit probability of treatment. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

`_generate_treatment_assignments`(...)

[CONCRETE] Generate the treatment assignment for each observed unit (`DGPVariables.TREATMENT_ASSIGNMENT_NAME`). A concrete implementation is provided which assigns treatment based on the propensity scores generated by `_generate_true_propensity_scores()`. This function can be used to assign treatment even if propensity scores are never generated/known.

Returns

an array containing the treatment assignment as a integer. 1 for treatment and 0 for control. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

`_generate_outcome_noise_samples`(...)

[CONCRETE] Generate the outcome noise for each observed unit (`DGPVariables.OUTCOME_NOISE_NAME`). A concrete implementation is provided which generates a zero noise vector.

Returns

an array containing the outcome noise for each observation as a real value. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

`_generate_outcomes_without_treatment`(...)

Generate the potential outcome without treatment for each observed unit (`DGPVariables.POTENTIAL_OUTCOME_WITHOUT_TREATMENT_NAME`). This implements the base outcome function - without noise or treatment effect.

Returns

an array containing the potential outcome without treatment for each observation as a real value. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

`_generate_treatment_effects`(...)

Generate the treatment effect for each observed unit (`DGPVariables.TREATMENT_EFFECT_NAME`). This implements the treatment effect function.

Returns

an array containing the treatment effect for each observation as a real value. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

`_generate_outcomes_with_treatment`(...)

[CONCRETE] Generate the potential outcome with treatment for each observed unit (`DGPVariables.POTENTIAL_OUTCOME_WITH_TREATMENT_NAME`). A concrete implementation is provided. It sums the potential outcome without treatment generated by `_generate_outcomes_without_treatment()` and the the treatment effect generated by `_generate_treatment_effects()` .

Returns

an array containing the potential outcome with treatment for each observation as a real value. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

`_generate_observed_outcomes`(...)

[CONCRETE] Generate the observed outcome for each observed unit (`DGPVariables.OBSERVED_OUTCOME_NAME`). A concrete implementation is provided. It selects either the the potential outcome with or without treatment generated by `_generate_outcomes_with_treatment()` / `_generate_outcomes_without_treatment()` based on the treatment assignment generated by `_generate_treatment_assignments()` and then adds the outcome noise generated by `_generate_outcome_noise_samples()`

Returns

an array containing the observed outcome for each observation as a real value. It must contain *n_observations* entries.

Return type

`numpy.ndarray`

---

class **`SampledDataGeneratingProcess`**(*params, observed_covariate_data, outcome_covariate_transforms, treatment_covariate_transforms, treatment_assignment_function, treatment_effect_subfunction, untreated_outcome_subfunction, treatment_assignment_logit_func=None, outcome_function=None, data_source=None, data_analysis_mode=True, compile_functions=False*)

Bases: `maccabee.data_generation.data_generating_process.DataGeneratingProcess`

EDIT THIS: In Maccabee, a `DataGeneratingProcess` combines a covariate `DataSource` and concrete/sampled treatment and outcome functions. These two components provide all the information required to draw sampled data sets.

---

class **`ConcreteDataGeneratingProcess`**(*n_observations, data_analysis_mode=False*)

Bases: `maccabee.data_generation.data_generating_process.DataGeneratingProcess`

`data_generation.generated_data_set`

This submodule contains the `GeneratedDataSet` class which represents data sampled from `DataGeneratingProcess` instances, providing the API used to interact with data sets generated by sampled or concrete DGPs.

---

class **`GeneratedDataSet`**(*dgp_variable_dict*)

This class is used to interact with the data generated by Maccabee DGPs. It encapsulates the internal data structure produced by `DataGeneratingProcess` instances and provides a clean (and field familiar) API which can be used to interact with the data sets sampled from DGPs.

A `GeneratedDataSet` instance provides access to two logically-distinct sets of variables via its attributes:

- First, it provides access to all of the DGP variables which are generated by the DGP as attributes of the same name. This includes both observable and oracle DGP variables. Note that these attributes, listed below, are named by the **value** of the constants in `maccabee.constants.Constants.DGPVariables`. This is designed to aid in creating readable and succinct models. `get_dgp_variable()` can be used to access a DGP variable by name, thus allowing the use of the constant names (rather than values) in `DGPVariables`. These properties are marked with **[DGP VARIABLE]** below.
- Second, it defines attributes which return ground-truth estimand values. These attributes are backed by functions which calculate the ground-truth values from the generated data. These properties are marked with **[ESTIMAND]** below.

Note

Behind the scenes, the DGP variable attributes are actually accessor functions which access the internal data structure to return the correct value for each DGP variable in `DGPVariables`. These functions are injected into the `GeneratedDataSet` class through the `DGPVariableAccessor(type)` class which is used as type/metaclass for `GeneratedDataSet`. Users who plan to add their own DGP variables should see the source code and in line comments for the `GeneratedDataSet` to ensure they understand this mechanism.

get_dgp_variable(*self*, *dgp_var_name*)

Returns the DGP variable given as *dgp_var_name*.

Parameters

dgp_var_name (*string*) – The name of a DGP variable from `DGPVariables`.

Returns

the `DataFrame` or *numpy.ndarray* containing the DGP variable observations.

Return type

object

Raises

- **UnknownDGPVariableException** – if an unknown DGP variable is requested.
- **DGPVariableMissingException** – if the DGP variable was not generated by the DGP.

ground_truth(*estimand*)

Returns the ground truth for the estimand given in *estimand*.

Parameters

estimand (*string*) – An estimand from `maccabee.constants.Constants.Model`

Returns

the ground truth value for the given estimand.

Return type

float

Raises

UnknownEstimandException – if an unknown estimand is supplied.

property **ATE**

[ESTIMAND]

This property accesses the ATE estimand

property **ATT**

[ESTIMAND]

This property accesses the ATT estimand

property **ITE**

[ESTIMAND]

This property accesses the ITE estimand

property **T**

[DGP VARIABLE]

This property accesses the DGP variable T - `TREATMENT_ASSIGNMENT_NAME`)

property **X**

[DGP VARIABLE]

This property accesses the DGP variable X - `COVARIATES_NAME`)

property **X_transformed**

[DGP VARIABLE]

This property accesses the DGP variable X_transformed - `TRANSFORMED_COVARIATES_NAME`)

property **Y**

[DGP VARIABLE]

This property accesses the DGP variable Y - `OBSERVED_OUTCOME_NAME`)

property **Y0**

[DGP VARIABLE]

This property accesses the DGP variable Y0 - `POTENTIAL_OUTCOME_WITHOUT_TREATMENT_NAME`)

property **Y1**

[DGP VARIABLE]

This property accesses the DGP variable Y1 - `POTENTIAL_OUTCOME_WITH_TREATMENT_NAME`)

property **Y_noise**

[DGP VARIABLE]

This property accesses the DGP variable Y_noise - `OUTCOME_NOISE_NAME`)

property **logit_p_score**

[DGP VARIABLE]

This property accesses the DGP variable logit_p_score - `PROPENSITY_LOGIT_NAME`)

property **observed_data**

[DGP VARIABLE GROUP]

This property returns a DataFrame containing all of the observable data: the observable covariates, the treatment assignment and the observed outcome. This is the data on which causal inference will be performed.

property **observed_outcome_data**

[DGP VARIABLE GROUP]

This property returns a DataFrame containing the observable outcome data: the treatment assignment and the observed outcome.

property **p_score**

[DGP VARIABLE]

This property accesses the DGP variable p_score - `PROPENSITY_SCORE_NAME`)

property **treatment_effect**

[DGP VARIABLE]

This property accesses the DGP variable treatment_effect - `TREATMENT_EFFECT_NAME`)

`data_generation.utils`

This submodule contains utility functions that are used during both DGP sampling and the sampling of data from DGPs. The functions in this module may be useful for users writing their own concrete DGPs.

evaluate_expression(*expression*, *data*)

Evaluates the Sympy expression in *expression* using the `pandas.DataFrame` in *data* to fill in the value of all the variables in the expression. The expression is evaluated once for each row of the DataFrame.

Parameters

- **expression** (*Sympy Expression*) – A Sympy expression with variables that are a subset of the variables in columns data.
- **data** (`DataFrame`) – A DataFrame containing observations of the variables in the expression. The names of the columns must match the names of the symbols in the expression.

Returns

An array of expression values corresponding to the rows of the *data*.

Return type

`ndarray`

`initialize_expression_constants`(*constants_sampling_distro, expressions, constant_symbols={a, c}*)

Initialize the constants in the expressions in *expressions* by sampling from *constants_sampling_distro*.

Parameters

- **constants_sampling_distro** (*function*) – A function which produces *n* samples from some distribution over real values when called using a size keyword argument as in `constants_sampling_distro(size=n)`.
- **expressions** (*list*) – A list of Sympy expressions in which the constant symbols from *constant_symbols* appears. These are initialized to the values sampled from *constants_sampling_distro*.
- **constant_symbols** (*list*) – A list of Sympy symbols which are constants to be initialized. Defaults to `{sympy.abc.a, sympy.abc.c}`.

Returns

A list of the sympy expressions from *expressions* with the constant symbols from *constant_symbols* randomly intialized.

Return type

list

Examples

```
>>> from sympy.abc import a, x
>>> import numpy as np
>>> initialize_expression_constants(np.random.normal, [a*x], [a])
0.1*x
>>> initialize_expression_constants(np.random.normal, [a*x], [a])
-0.3*x
```

`select_objects_given_probability`(*objects_to_sample, selection_probability*)

Samples objects from *objects_to_sample* based on *selection_probability*.

Parameters

- **objects_to_sample** (list or `numpy.ndarray`) – List of objects to sample. If dimensionality is greater than 1, selection is along the primary (row) axis.

- **selection_probability** (*list or float*) – The probability with which to sample objects from *objects_to_sample*. The value or values supplied should be between 0 and 1. If a list of probabilities is supplied, it should be the same length as the primary axis of the list in *objects_to_sample* and will be the per-object/row selection probability. If float, then this is the selection probability for all objects. In this case, `int(len(objects_to_sample)*selection_probability)` objects will be sampled if this value is greater than 0. Otherwise the single probability will be the selection probability for each object.

Returns

An array of the selected objects.

Return type

`numpy.ndarray`

Examples

```
>>> select_objects_given_probability(["a", "b", "c"], [0.5, 0.1, 0.001])
["a"]
```

```
>>> select_objects_given_probability(["a", "b", "c"], 0.1)
["a"]
```

`maccabee.data_sources`

This module is comprised of the submodules listed below. The `data_sources` module contains the `DataSource` classes which define the internal data handling logic and external API. The `data_source_builders` module contains utility functions which can be used to build `DataSource` instances that correspond to commonly used covariate data sources.

Note

For convenience, all the classes and functions that are split across the two submodules below can be imported directly from the parent `maccabee.data_sources` module.

`data_sources.data_sources`

This module contains `DataSource`-derived objects that standardize access to and management of different sources of covariate data and meta-data used by Maccabee DGPs.

class **DataSource**(*covar_names*, *discrete_covar_names*, *normalize=True*)

Bases: `object`

An abstract class that defines the encapsulation logic used to store, process and access covariate data and meta-data. The external API provides clean access to the data required for the sampling and application of treatment/outcome functions. Concrete implementations are responsible for the different loading/sampling and normalization schemes required to handle different static/stochastic covariate data.

- The primary (abstract) method is `_generate_covar_df()` which returns an unnormalized `DataFrame` that contains the covariate observations and covariate names.
- The concrete method `_normalize_covariate_data()` provides a default normalization scheme for the data in the covariate `DataFrame`.
- The methods `get_covar_df()`, `get_covar_names()`, and `get_discrete_covar_names()` provide the external API which is used to access the covariate data/meta-data during DGP and data sampling.

Parameters

- **covar_names** (*list*) – *covar_names* is a list of the string names of the covariates present in the `DataFrame` produced by `_generate_covar_df()`.
- **discrete_covar_names** (*list*) – *discrete_covar_names* is a list of the string names of the discrete covariates present in the `DataFrame` produced by `_generate_covar_df()`.
- **normalize** (*bool*) – *normalize* indicates whether the covariates in the `DataFrame` returned by `_generate_covar_df()` should be normalized prior to use by applying the `_normalize_covariate_data()` method. The default normalize scheme provided by this method assumes a normal distribution over the data in each continuous covariate and leaves discrete covariates as is. Defaults to *True*.

covar_names

a list of the string names of the covariates present in the covariate `DataFrame` produced by *_generate_covar_df*.

discrete_covar_names

list of the string names of the discrete covariates present in the covariate `DataFrame` produced by `_generate_covar_df()`.

normalize

indicates whether the covariate `DataFrame` returned by `_generate_covar_df()` will be normalized prior to use.

_generate_covar_df()

Abstract method which, when implemented, returns a `DataFrame` that contains the covariate observations and covariate names as the column names. This may involve sampling a joint distribution over the covariates, reading a static set of covariates from disk/memory etc.

Returns

a `DataFrame` that contains the covariate observations and covariate names as the column names.

Return type

A `DataFrame`

Raises

NotImplementedError – This is an abstract function which is concretized in inheriting classes.

`_normalize_covariate_data`(*covar_df*)

This method normalizes the covariate data returned by `_generate_covar_df()` in preparation for DGP sampling. If the `DataSource` is to be used with sampled DGPs then all continuous covariates should be 0 mean and have an approximate standard deviation of 1. Symmetry in the covariate distributions is not required but will improve the ability of the sampling process to achieve the desired distributional setting. See the docs for the `maccabee.data_generation.data_generating_process.DataGeneratingProcess` for more detail).

Parameters

covar_df (`DataFrame`) – The `DataFrame` which contains the unnormalized covariate observations.

Returns

a `DataFrame` which contains the normalized covariate observations.

Return type

`DataFrame`

`get_covar_df`()

Main API method that is used by external classes to access the generated covariate data.

Returns

The `DataFrame` containing normalized covariate observations and covariate names.

Return type

`DataFrame`

`get_covar_names`()

Accessor method for `covar_names`.

Returns

The list of string names of the covariates in this data source.

Return type

list

Examples

```
>>> data_source = DataSource(covar_names=["X1", "X2", "X3"], discrete_covar_names=
["X1"])
>>> data_source.get_covar_names()
["X1", "X2", "X3"]
```

`get_discrete_covar_names`()

Accessor method for `discrete_covar_names`.

Returns

The list of string names of the discrete covariates in this data source.

Return type

list

Examples

```
>>> data_source = DataSource(covar_names=["X1", "X2", "X3"], discrete_covar_names=
["X1"])
>>> data_source.get_discrete_covar_names()
["X1"]
```

---

class **`StaticDataSource`**(*static_covar_data, covar_names, discrete_covar_names, normalize=True*)

Bases: `maccabee.data_sources.data_sources.DataSource`

A concrete implementation of the abstract `DataSource`, which can be used for sampling static sources of covariate data (ones which do not change).

Parameters

- **static_covar_data** (`numpy.ndarray`) – a 2D `numpy.ndarray` of covariate data.
- **covar_names** (*list*) – see `DataSource`.
- **discrete_covar_names** (*list*) – see `DataSource`.
- **normalize** (*bool*) – see `DataSource`.

`_generate_covar_df`()

Concretized implementation of `DataSource._generate_covar_df()` which returns a `DataFrame` containing the data supplied in *static_covar_data* at initialization time.

Returns

a `DataFrame` containing static covariate observations.

Return type

`DataFrame`

---

class **`StochasticDataSource`**(*covar_data_generator*, *covar_names*, *discrete_covar_names*, *normalize=True*)

Bases: `maccabee.data_sources.data_sources.DataSource`

A concrete implementation of the abstract `DataSource`, which can be used for sampling stochastic sources of covariate data by automatically using a supplied sampling function for each call to `get_covar_df()`.

Parameters

- **covar_data_generator** (*function*) – a function which samples some joint distribution over covariates and returns a 2D `numpy.ndarray` of covariate data.
- **covar_names** (*list*) – see `DataSource`.
- **discrete_covar_names** (*list*) – see `DataSource`.
- **normalize** (*bool*) – see `DataSource`.

`_generate_covar_df`()

Concretized implementation of `DataSource._generate_covar_df()` which calls the *covar_data_generator* supplied at initialization, and returns a `DataFrame` containing the data returned by it and the `covar_names`.

Returns

a `DataFrame` containing sampled covariate observations.

Return type

`DataFrame`

`data_sources.data_source_builders`

This module contains utility functions of the form `build_*_datasource` which provide a convenient way to instantiate `DataSource` instances corresponding to commonly used/useful data sets.

`build_cpp_datasource`()

Builds a datasource using the CPP data set of empirical covariates. See Chapter 5 of the theory **paper** for more on this data set.

Returns

A `DataSource` instance which will generate the covariates from the CPP data set when sampled.

Return type

`DataSource`

`build_csv_datasource`(*csv_path, discrete_covar_names=[]*)

Builds a datasource using the CSV of covariates at *csv_path*. This method expects a CSV with covariate names in the first row.

Parameters

- **csv_path** (*string*) – The path to a CSV.
- **discrete_covar_names** (*list*) – A list of string covariate names corresponding to the discrete covariates. Defaults to [].

Returns

A `DataSource` instance which will generate the covariates from the CSV when sampled.

Return type

`DataSource`

`build_lalonde_datasource`()

Builds a datasource using the Lalonde data set of empirical covariates. See Chapter 5 of the theory **paper** for more on this data set.

Returns

A `DataSource` instance which will generate the covariates from the Lalonde data set when sampled.

Return type

`DataSource`

`build_random_normal_datasource`(*n_covars=20, n_observations=1000, partial_correlation_degree=0.0*)

Builds a datasource using random normal covariates.

Parameters

- **n_covars** (*int*) – The number of random normal covariates. Defaults to 20.
- **n_observations** (*int*) – The number of observations in the data set. Defaults to 1000.
- **partial_correlation_degree** (*float*) – The degree of partial correlation between the covariates. Full independance at `0.0` and perfect correlation at `1.0`. A random covariance matrix is generated based on this parameter by approximating the random

- **method. Defaults to 0.0.** (*vine*) –

> **Returns**
>
> A `DataSource` instance which will generate random normal covariates when sampled.
>
> **Return type**
>
> `DataSource`

`build_stochastic_datasource`(*generator_func*, *covar_names*, *discrete_covar_names*)

Builds a datasource which generates covariates using the function in *generator_func*.

> **Parameters**
>
> - **generator_func** (*function*) – A function which returns a 2D `numpy.ndarray` that contains covariates as columns with covariate observations as rows.
> - **covar_names** (*list*) – A string list of covariate names corresponding to the columns of the `numpy.ndarray` generated by *generator_func*.
> - **discrete_covar_names** (*type*) – A list of string covariate names corresponding to the discrete covariates. Defaults to [].
>
> **Returns**
>
> A `DataSource` instance which will generate the covariates from the CSV when sampled.
>
> **Return type**
>
> `DataSource`

`maccabee.modeling`

This module contains the code used to define and evaluate causal models. Causal models are responsible for estimating causal effects from observational data (a subset of the data available as part of `GeneratedDataSet` instance).

The code in this model is split into two submodules. The `maccabee.modeling.models` submodule defines the base `CausalModel` class which all concrete models inherit from. It also contains some derived example models. Second, the `maccabee.modeling.performance_metrics` submodule defines the metrics used to evaluate all causal models.

`modeling.models`

This submodule contains the `maccabee.modeling.models.CausalModel` class which is the base class that defines the interface Maccabee expects from all models. IE, all models benchmarked using Maccabee should inherit from this class.

The submodule also contains some example concrete implementations. These serve as an implementation guide for user-defined models and can be used as a baseline for custom model performance.

class **CausalModel**(*dataset*)

Bases: `object`

The base `maccabee.modeling.models.CausalModel` class presents a minimal interface. This is important because many models, with diverse operation/characteristics, are expected to conform to this interface. It takes a `GeneratedDataSet` instance which contains the data to be used for estimation. It has an abstract `fit()` method which, when called on inheriting classes, should prepare the model to produce an estimate. This preparation could mean pre-processing data, training a neural network etc. Finally, it has a concrete `estimate()` method which expects to find a defined method with the `estimate_*` where * is an estimand name. It is up to the inheriting class to define the appropriate estimator methods depending on the estimands which will be evaluated.

Parameters

dataset (`GeneratedDataSet`) – A `GeneratedDataSet` instance produced by a `DataGeneratingProcess` .

Attributes

dataset: the data set supplied at initialization time.

estimate(*estimand*, **args*, ***kwargs*)

The estimate method is called to access an estimand value. It is a convenience method which gives external classes access to any of the (different) estimand functions using a single parameterized function. It does not handle the calculation of the estimand itself but rather delegates to an `estimate_*` method on the instance where * is the name of the estimand.

Parameters

- **estimand** (*string*) – The name of the estimand to return. This should be one of the estimands in the constants list `ALL_ESTIMANDS` .
- ***args** (*list*) – All position args are collected and passed to the estimand function.
- ****kwargs** (*dict*) – All keyword args are collected and passed to the estimand function.

Returns

The value of the estimand.

Return type

float

Raises

UnknownEstimandException – if an unknown estimand is requested.

`fit`()

The fit method is expected to be called once per model instance. It prepares the model for estimating any of the implemented estimands.

Returns

the fit function does not have any meaningful return value.

Return type

None

Raises

NotImplementedError – this is an abstract implementation.

class `CausalModelR`(*dataset*)

Bases: `maccabee.modeling.models.CausalModel`

This class inherits from `maccabee.modeling.models.CausalModel` and implements additional tooling using to write causal models with major components in R.

`_import_r_file_as_package`(*file_path*, *package_name*)

Helper function to import an R file as a psuedo-package. The functions from the R file are translated as exported methods of a package called *package_name*.

Parameters

- **file_path** (*str*) – The path to the R file to import.
- **package_name** (*str*) – The name to be used for the psuedo-package.

Returns

A python object representing the R package with all functions as attribute methods of the object.

Return type

object

`_import_r_package`(*package_name*)

Helper function to import a package pre-installed in the system's R language.

Parameters

package_name (*str*) – The string name of the package, as would be used in the R *load* command.

Returns

A python object representing the R package with all functions as attribute methods of the object.

Return type

object

---

class **LinearRegressionCausalModel**(*dataset*)

Bases: `maccabee.modeling.models.CausalModel`

This class inherits from `maccabee.modeling.models.CausalModel` and implements a linear-regression based estimator for the ATE and ITE using SciKit Learn linear regression model. The ITE is a dummy estimand in this case given the linear model assumes a homogenous effect amongst all units.

estimate_ATE()

Return the co-efficient on the treatment status variable as the ATE.

estimate_ITE()

Return the difference between the model's predicted value with treatment set to 1 and 0 as the ITE. This will be a constant equal to the ATE given that this is a linear model.

fit()

Fit the linear regression model.

`modeling.performance_metrics`

This submodule contains the definitions for the performance metrics used to quantify the performance of causal estimators implemented using the model classes from `models`.

The metrics defined in this submodule are divided into those which quantify the performance of average effect estimators and those which quantify the performance of individual effect estimators. These two classes of metrics are operationally very similar. They define some measure of the estimator quality based on a comparison of the estimate value and the ground truth. And, crucially, they are both defined over samples of estimate values rather than a single estimate-ground-truth pair (although some are well defined in this individual limit). The difference between them is that average effect metrics operate over a pair of real numbers for the estimated/true effect per data set while individual effect metrics operate over a pair of real-valued estimate/truth vectors.

The submodule exposes these two sets of metrics as dictionaries - one for average effect metrics and one for individual effect metrics. These dictionaries map metric names to callables that calculate the metrics given the inputs outlined above. These dictionaries are used by the `benchmarking` module to calculate performance metrics for models when applied to `GeneratedDataSet` instances produced by concrete/sampled `DataGeneratingProcesses`. The dictionaries, and the metrics contained by each, are outlined below.

In the documentation below, assume a sample of N data sets each of which contain M observations. The estimated effects will be referred to as $\hat{\tau}$ and the ground truth effects as τ. For average effects, of which there is one estimate/ground-truth pair per data set, the i th pair of values will be referred to as $\hat{\tau}_i / \tau_i$ respectively. For individual effects, of which there are M per data set, the j th observation in the i th data set will be referred to as $\hat{\tau}_{ij} / \tau_{ij}$.

Average Effect Metrics

AVG_EFFECT_METRICS - *{...}*

The dictionary containing the average effect metrics. It contains the following metrics:

- `RMSE` - the Root Mean Squared Error.
- `AMBP` - Absolute Mean Bias Percentage.
- `MABP` - Mean Absolute Bias Percentage.

---

root_mean_sqaured_error(...)

The Root Mean Squared Error (RMSE):

$$\sqrt{\frac{1}{N} \times \sum_{i=1}^{N} \left(\hat{\tau}_i - \tau_i\right)^2}$$

The RMSE is a well-known measure which captures the sum of estimator bias and variance. A zero RMSE implies both zero variance and bias. But a non-zero RMSE does not indicate wether bias or variance is the source of the error. This metric is, therefore, paired with the Absolute Mean Bias Percentage, which measures the bias. If bias is low but the RMSE is high, then the source of this high RMSE is isolated to estimator variance.

Parameters

- **avg_effect_estimate_vals** (`numpy.ndarray`) – an array of sampled average effect estimates.
- **avg_effect_true_vals** (`numpy.ndarray`) – an array of sampled average effect ground truths.

Returns

The Root Mean Squared Error.

Return type

float

---

absolute_mean_bias_percentage(...)

The Absolute Mean Bias Percentage (AMBP):

$$100 \times \left| \frac{1}{N} \times \sum_{i=1}^{N} \frac{\hat{\tau}_i - \tau_i}{\tau_i} \right|$$

This metric quantifies the degree of systematic bias in the estimator. An unbiased estimator will not favor estimates either above or below the ground truth. If this is true, when many estimates are averaged, their mean will be zero. A non-zero mean indicates bias exists. Taking the absolute value of this bias and scaling as a percentage of the ground-truth effects reflects the equal severity of positive/negative bias and allows the bias to be averaged/compared for different magnitudes of effect.

Parameters

- **avg_effect_estimate_vals** (`numpy.ndarray`) – an array of sampled average effect estimates.
- **avg_effect_true_vals** (`numpy.ndarray`) – an array of sampled average effect ground truths.

Returns

The Absolute Mean Error Percentage.

Return type

float

`mean_absolute_bias_percentage`(...)

The Mean Absolute Bias Percentage (MABP):

$$100 \times \frac{1}{N} \times \sum_{i=1}^{N} \left(\left| \frac{\hat{\tau}_i - \tau_i}{\tau_i} \right| \right)$$

This metric quantifies the average bias in the estimator. It measures the mean absolute value of this bias in the estimate as a percentage of the ground-truth effects. Note: an unbiased estimator may still have a large MAB if each estimate tends to be far from the ground truth.

Parameters

- **avg_effect_estimate_vals** (`numpy.ndarray`) – an array of sampled average effect estimates.
- **avg_effect_true_vals** (`numpy.ndarray`) – an array of sampled average effect ground truths.

Returns

The Absolute Mean Error Percentage.

Return type

float

Individual Effect Metrics

INDIVIDUAL_EFFECT_METRICS - *{...}*

The dictionary containing the individual effect metrics. It contains the following metrics:

- `PEHE` - The Precision in Estimation of Heterogenous (Treatment) Effects

precision_in_estimating_heterogenous_treat_effects(...)

The Precision in Estimation of Heterogenous (Treatment) Effects (PEHE):

$$\frac{1}{N} \times \sum_{i=1}^{N} \left(\sqrt{\frac{1}{M} \times \sum_{j=1}^{M} \left(\hat{\tau_{ij}} - \tau_{ij} \right)^2} \right)$$

This metric, first introduced by Hill (2011) [1], measures the quality of the estimation of the individual treatment effects in a data set. It measures the RMSE of the individual effect estimate within data sets and then averages this value across data sets. As such, it measures both the bias and variance of an individual effect estimator in producing individual effect estimates.

> **Parameters**
>
> - **indv_effect_estimate_vals** (`numpy.ndarray`) – a 2D array of sampled average effect estimates. Each row representing the individual effect estimate data for a sampled data set.
> - **indv_effect_true_vals** (`numpy.ndarray`) – a 2D array of sampled average effect ground truths. Each row representing the individual effect ground truth data for a sampled data set.
>
> **Returns**
>
> Precision in Estimation of Heterogenous (Treatment) Effects
>
> **Return type**
>
> float

Adding New Metrics

add_performance_metric(*aggregation_level, metric_name, metric_callable*)

Adds a performance metric for individual or average estimands to the benchmarks.

> **Parameters**
>
> - **aggregation_level** (*str*) – The estimand aggregation level to which the metric applies. One of either: `Constants.Model.INDIVIDUAL_ESTIMANDS` or `Constants.Model.AVERAGE_ESTIMANDS`.
> - **metric_name** (*str*) – The name of the metric. Must be unique for the aggregation level.

- **metric_callable** (*fynction*) – A callable which accepts two arguments - one for the estimand estimate values and one for the ground truth values. If at the average effect aggregation level, each argument is a 1D `numpy.ndarray` with each entry corresponding to an estimand value in a different dataset of a sampling run. If at the individial effect aggregation level, each argument is a 2D `numpy.ndarray` with each row corresponding to individual effect estimand values for a single dataset (one column per individual).

Footnotes

1

Hill, J. L. (2011). Bayesian nonparametric modeling for causal inference. Journal of Computational and Graphical Statistics, 20(1), 217–240. https://doi.org/10.1198/jcgs.2010.08162

`maccabee.parameters`

This module contains the classes and files which are used to specify the sampling parameters that control aspects of the DGP sampling process. IE, the parameters which control the operation of `DataGeneratingProcessSampler` class which produces `SampledDataGeneratingProcess` instances.

The parameters which control the DGP sampling process are non-trivial both in number and kind. The parameters can take the form of:

- Single values with different valid values.
- Dictionaries with sets of required keys
- Calculated parameters which are derived from other parameters in a non-trivial way through some one-off calculation.
- Functional parameters (specified using python code) which allow for the injection of arbitrary analytical expressions at appropriate points in the DGP sampling process. For example, different sampling distributions can be specified using functional parameters that contain arbitrary sampling code.

The complexity of the parameterization necessitates an encapsulation layer so that the `DataGeneratingProcessSampler` class can consume parameters without worrying about the detail of their specification. The `ParameterStore` class serves this role, acting as a unified store for all of the parameters and providing a simple access interface.

Instances of the `ParameterStore` class can be created in a variety of ways depending on user requirements. Typically, instances are created automatically by the `maccabee.benchmarking` functions based on the user's desired position for sampled DGPs in distributional problem space in terms of levels/positions along the axes of the space. This allows users to specify, at a high-level, the parameterization they want and leave the detailed parameter value specification to Maccabee. Under the hood, this approach uses the `build_parameters_from_axis_levels()` function. See the docs for that function or the `benchmarking` docs for more on this approach.

Beyond specifying axis levels, there are also ways to specify parameter values more directly. The `build_default_parameters()` function returns a `ParameterStore` instance containing a set of default parameter values. This instance can then act as a starting point for small modifications using the `set_parameter()` / `set_parameters()` methods. Finally, the `build_parameters_from_specification()` function can be used to generate a `ParameterStore` instance from a parameter specification file. This allows for granular control over parameter values. Users interested in setting custom parameter values should look at the parameter_schema.yml file. This contains the names, validity conditions and descriptions of all the sampling parameters.

See the `ParameterStore` docs for detail on the set of sampling parameters, parameter specification files, and the parameter schema file.

> **Note**
>
> For convenience, the classes and functions in the `maccabee.parameters.parameter_store` submodule can be imported directly from this module.

`parameters.parameter_store`

This submodule defines the `ParameterStore` class. If you haven't already read the overview of sampling parameterization provided in the docs for the `maccabee.parameters` module you should read those docs before proceeding to read the content below.

---

class **`ParameterStore`**(*parameter_spec_path*)

The Design of the ParameterStore

In order to understand the `ParameterStore` class, it is important to understand the motivation behind its design. The goal of the class is to provide a simple interface to specificy and access a complex set of parameters. As mentioned in the parent module docs, the DGP sampling parameters can be of very different types with different validity conditions and access workflows (simple data retrieval, once-off calculation, dynamic calculation). This means standard data structure based storage, for example in a dictionary, would be very difficult and require tight coupling between the consumption of the parameters (in the `DataGeneratingProcessSampler`) and their specification format. It would also make the task of specifying the parameters laborious as large dictionaries are hard to organize and parse by visual inspection.

In theory, using a vanilla parameter class would allow for all of this complexity to be encapsulated behind a simple interface. Verification of parameters, execution of arbitrary code etc are easy to implement using standard class design principles/mechanisms. Unfortunately, users are likely to experiment with many different parameterizations as they explore the distributional problem space and using a vanilla class for parameter storage is not good for experimentation. In order to experiment with many different sets of parameter values using classes requires either:

1. The maintenance of many separate classes with different methods/values. This strategy introduces a lot of boilerplate code to define and manage different classes. Even if inheritance is used, defining many new classes is cumbersome and produces parameter specifications which are hard to grok by visual inspection.
2. Using a single class and changing the methods/attributes by manual run-time parameterization. This strategy makes it hard to switch between different parameterizations and risks loss of reproducibility if specific parameterizations are lost/forgotten.

Further, it is possible and likely that the parameters will be added by users of this package. This should, ideally, be facillitated without requiring code changes to the package itself. Vanilla classes would necessitate such changes.

With the above context in mind, the `ParameterStore` class pursues a hybrid approach. Parameter values are specified using structured YML files referred to as parameter specification files. These files are easy to read and allow for low-overhead, reliable storage, duplication, and editing. The parameter specification file is read by the `ParameterStore` class at instantiation time. Its content is interpretted based on a package-level parameter schema file that specifies the set of expected parameters and their format/handling mechanism and validity conditions. This allows a single set of parameter handling code in the `ParameterStore` class to be (re)used in storing and accessing an arbitrary set of parameters which are defined in the schema (arbitrary up to the pre-defined set of handling mechanisms which require code changes to modify).

After instantiation, the parameters are available as **attributes** of the `ParameterStore` instance as if they had been coded directly into the class definition (despite being stored in a specification file which stores their values for reproducibility and inspection).

Building ParameterStore Instances

While it is possible to build a `ParameterStore` instance using the constructor, this is not recommended. There are three supported ways to build instances. The first two are useful if direct control of parameter values is useful/required. The third is designed to build instances using higher-level specification of the desired parameterization (as outlined in the documentation of the parent module).

- The helper function `build_parameters_from_specification()` can used to easily construct a `ParameterStore` instance from a parameter specification file.
- The helper function `build_default_parameters()` builds an instance using the default_parameter_specification.yml file
- Finally, the helper function `build_parameters_from_axis_levels()` builds an instance using a specification of location in the distributional problem space. See its documentation for detail.

Using ParameterStore Instances

As mentioned above, after instantiation, the sampling parameters are available as attributes of the instance. They can therefore be *accessed* as standard attributes of an instance. However, when *setting* the value on an attribute, it is important to use the

`set_parameter()` / `set_parameters()` methods so that calculated parameters are updated appropriately.

Instance Method Documentation

> Parameters
>
> **parameter_spec_path** (*string*) – The path to a parameter specification file.

`set_parameter`(*param_name*, *param_value*, *recalculate_calculated_params=True*)

Set the value of the param with the name *param_name* to the value *param_value*.

> Parameters
>
> - **param_name** (*string*) – The name of the parameter to set.
> - **param_value** (*type*) – The value to set the parameter to.
> - **recalculate_calculated_params** (*bool*) – Indicates whether calculated params should be recalculated. Defaults to True.

> Examples

```
>>> from maccabee.parameters import build_default_parameters
>>> params = build_default_parameters()
>>> params.set_parameter("TREATMENT_EFFECT_TAIL_THICKNESS", 42)
>>> params.TREATMENT_EFFECT_TAIL_THICKNESS
42
```

`set_parameters`(*param_dict*, *recalculate_calculated_params=True*)

Set the value of the params with the names of the keys in the *param_dict* dictionary to the corresponding value in the dictionary.

> Parameters
>
> - **param_dict** (*dict*) – A dictionary mapping parameter names to parameter values.
> - **recalculate_calculated_params** (*bool*) – Indicates whether calculated params should be recalculated. Defaults to True.

> Examples

```
>>> from maccabee.parameters import build_default_parameters
>>> params = build_default_parameters()
>>> params.set_parameters({"TREATMENT_EFFECT_TAIL_THICKNESS": 42})
>>> params.TREATMENT_EFFECT_TAIL_THICKNESS
42
```

`parameters.parameter_store_builders`

`build_default_parameters()`

Return a `ParameterStore` instance based on the default parameter specification file.

Returns

A `ParameterStore` instance.

Return type

`ParameterStore`

Examples

```
>>> from maccabee.parameters import build_default_parameters
>>> build_default_parameters()
<maccabee.parameters.parameter_store.ParameterStore at ...>
```

`build_parameters_from_axis_levels`(*metric_levels, save=False*)

Return a `ParameterStore` instance based on the provided parameter specification file.

This function uses the values in the metric_level_parameter_specifications.yml file to change the parameter values in order to achieve the desired level/position on the given axes.

Parameters

parameter_spec_path (*dict*) – A dictionary mapping data axis names to a data axis level. Axis names are available as constants in `maccabee.constants.Constants.AxisNames` and axis levels available as constants in `maccabee.constants.Constants.AxisLevels`.

Returns

A `ParameterStore` instance.

Return type

`ParameterStore`

Examples

```
>>> from maccabee.parameters import build_parameters_from_axis_levels
>>> from maccabee.constants import Constants
>>> # define the axis level for treatment nonlinearity.
>>> # all others will remain at default.
>>> axis_levels = { Constants.AxisNames.TREATMENT_NONLINEARITY:
Constants.AxisLevels.HIGH }
>>> build_parameters_from_axis_levels(axis_levels)
<maccabee.parameters.parameter_store.ParameterStore at ...>
```

`build_parameters_from_specification`(*parameter_spec_path*)

Return a `ParameterStore` instance based on the provided parameter specification file.

Parameters

parameter_spec_path (*string*) – A string path to the parameter specification file.

Returns

A `ParameterStore` instance.

Return type

`ParameterStore`

Raises

ParameterSpecificationException – if there is a problem with the parameter specification. More detail will be provided by the exception subclass and message.

Examples

```
>>> from maccabee.parameters import build_parameters_from_specification
>>> build_parameters_from_specification("./param_spec.yml")
<maccabee.parameters.parameter_store.ParameterStore at ...>
```