Hybrid Monte Carlo Benchmarks for Methods of Causal Inference

Josh Broomberg

February 29, 2020

Contents

Part I

Introduction

Chapter 1

New Approaches, New Challenges

The last decade has seen enormous progress in the development of advanced algorithms that extract useful information from rapidly expanding stores of data (Lecun, Bengio, & Hinton, 2015). These algorithms - referred to as methods of machine learning - broadly operate in the paradigm of predictive inference: implicitly or explicitly learning a joint probability distribution over independent and identically distributed (IID) data and using the implied correlations to make conditional predictions (Schölkopf, 2019). As a result of this reliance on statistical correlation to infer the relationship between variables, these methods are useful but inherently limited. As is pointed out in Pearl (2009), the accuracy of predictive methods fail when, rather than observing and predicting IID from outside of a static data generating process, the target of prediction is an (intended) intervention that changes the data generating mechanism and, as a result, the observed data distribution. Here is a simple example of this failure: In an observed data set, the presence of rain and a wet roof are correlated. Knowing the roof is wet will reduce one's uncertainty about the weather. But, understanding that actively wetting the roof will not result in changes to the weather - despite the high *predictive* power of a wet roof for the presence of rain - requires understanding that rain *causes* the wet roof and not the reverse. This pattern holds in less contrived settings. If a new medicine tends to be prescribed to healthier (less at risk) patients during a trial period, then there will be a strong correlation between the new medicine and quick recovery regardless of the efficacy of the medicine. Given this correlation, predictive methods would (correctly) infer that prescribing the medicine increases the probability of recovery (*within the IID observed data)*. But, upon intervening by giving the medicine to less healthy patients, one would find no positive effect - the predicted recover was the result of the previous treatment policy, not the medicine itself. These help establish an important truth: predicting the result of an intervention that changes some part of an observed system requires information beyond pure statistical correlation. I will show below that correlation does, in fact, play an important role in inferring the outcome of intervention but must be augmented.

The challenge presented by this limitation is that intervention, of one kind or another, lies at the heart of many of the most important questions we would like to answer with growing stores of data. Social, economic and health policy is designed to *change* systems to produce better outcomes for members of society. Companies take actions that *change* market conditions to produce commercially-desirable outcomes. The intention to intervene means that predictive inference, applied to observed examples of different policies/actions, cannot be used to guide choices of which policy/action is best. Rather, what is required is methods of *causal inference.* These methods can be thought of as uncovering correlations at the fundamental, mechanistic level rather than the correlations which appear at the (arbitrary) phenomenological level. These fundamental correlations persist under interventions that change the observed setting and, therefore, allow for the evaluation of the *true,* causal effect of that intervention (Schölkopf, 2019).

Historically, randomized experiments have been the gold standard in uncovering these fundamental, mechanistic correlations (Meldrum, 2000). Randomizing the 'intervention' to which some unit is exposed, means that any observed (average) difference in outcome can only and will only reflect the effect of differential exposure to interventions and no other underlying cause. The problem with this approach is that running an experiment is often impossible (for nation-scale policy changes), expensive, slow, and quite possibly unethical (in cases where there is a reasonable expectation the intervention will mitigate suffering but to an unknown extent) (Meldrum, 2000). Even if experimentation is viable, the validity of the effect recovered is limited to the sub-population which is targeted for the experiment which, in many cases, will not be representative of the broader population and, further, is only correct on average but not for any single individual (Rothwell, 2006). For these reasons, there is great value in methods that are able to infer fundamental, mechanistic effects from non-experimental, non-randomized, *observational* data at both the average and individual level. Methods that are capable of this inference are able to guide policy/action decisions based on the reservoirs of data already available to governments and corporations without the need for experimentation.

The last few decades have seen the rapid development of just such methods of *observational causal inference.* As a general rule, these methods build on existing tools of predictive inference but - through a combination of weak assumptions and modified estimands - recover fundamental mechanistic relations. These methods can be divided into two classes based on the nature of the targeted estimand. First, there is a mature literature on the estimation of *average causal effects.* Methods in this class estimate the average effect of an intervention by undoing the *confounding bias* in (average) group outcomes induced by factors that affect both the outcome of some intervention and the likelihood of receiving it (as in the medical treatment example above). This estimand is useful for government policy evaluation which focuses on the macro-efficacy of some intervention. Common approaches include inverse probability weighting (Hirano, Imbens, & Ridder, 2003; Horvitz & Thompson, 1952) and matching estimators (Rosenbaum & Rubin, 1983; Rubin, 1974)). See Imbens and Wooldridge (2009) and Athey and Imbens (2017) for extensive reviews of the numerous methods which target average causal effects. The second class of methods is comprised of a newer body of work focused on estimating *individual causal effects* - the effect of an intervention on specific individuals with the possibility of heterogeneous effects across the population of individuals. Estimation of individual effects requires disambiguating the confounding bias, as defined above, which operates at the level of intervention groups, from the heterogeneity of individuals' responses to intervention at the sub-group level. The individual effect estimate is important where there is known heterogeneity in the response to intervention and the impact on individuals is relevant to policy decisions. For example, medical treatment often involves choices between hundreds of available options with highly varied individual responses and, as such, requires an understanding of how each treatment will affect a specific individual (Lu, Sadiq, Feaster, & Ishwaran, 2018). As a result of the inferential challenge implied by both confounding bias and heterogeneous response, methods for individual effect estimation tend to be newer and use modern, flexible semi/nonparametric estimators that require substantial computational resources. These include: regression trees (Athey & Imbens, 2016; Su, Edu, Wang, Nickerson, & Li, 2009), Random Forests (Athey, Tibshirani, & Wager, 2019; Wager & Athey, 2018), Least Absolute Shrinkage and Selection Operator (Lasso) (Chen, Tian, Cai, & Yu, 2017; Qian & Murphy, 2011; Tian, Alizadeh, Gentles, & Tibshirani, 2014), Neural Networks (Fredrik D. Johansson, Kallus, Shalit, & Sontag, 2018; Fredrik D Johansson, Shalit, & Sontag, 2016; Künzel et al., 2018; Li & Fu, 2017; Schwab, Linhardt, & Karlen, 2018) or Bayesian machine learning (Hill, 2011; Taddy, Gardner, Chen, & Draper, 2016).

The diversity in methods, both for average and individual effect estimation, is to be expected. Causal inference fundamentally rests on the same inferential mechanics as predictive inference: accurately estimating distributions, or expectations over those distributions, from observed data. This presence of predictive inference is clear in both of the dominant frameworks for causal inference: the Rubin-Neyman Potential Outcomes Framework (Holland, 1986) and Pearl's Structural Causal Modelling (SCM) (Pearl, 2009). Causal inference is, however, distinguished from purely predictive inference by the addition of mechanisms and assumptions to counteract the confounding bias induced by the shift in observational setting created by a (potential) intervention. These mechanisms and assumptions are simply combined with the more fundamental inference tasks rather than replacing them. This framing of causal inference methods, which will be formalized in the chapters below, provides an intuitive justification for the existence of many equivalent estimators for the average and individual effect estimands. There are two distinct sources of equivalency evident in the framing above. First, for any given mechanism (and set of assumptions) for reversing observational bias, there tend to be many equivalent predictive inference methods that can then be used to operationalize the estimation of the target effect. Second, for any given data set, there are usually multiple equivalent mechanisms (and requisite assumptions) that can be used to mitigate confounding bias. Much like in pure predictive inference, different, equivalent inference techniques and bias-mitigation mechanisms may exhibit very different performance and properties in finite samples while all converging to the same estimands in the asymptotic limit.

This raises the challenge that is the core focus of this paper. Given numerous equivalent methods for observational causal inference, how does one know which one is best? This question is crucially important - both for researchers who are working to improve these methods and for practitioners who wish to apply the best method available to make important policy decisions in their fields. It is also a question that is hard to answer. Unlike in predictive inference, the ground-truth is not present in the training data. Under the Potential Outcomes framework, this stems from the fact that we never see the same unit under different interventions and thus can never know the true causal effect of the intervention on that unit. Equivalently, in Pearl's SCMs, the observed data is distributed differently to the data under intervention. In either case, the result is the same - the training data does not contain the ground-truth. Further, while some methods may lend themselves to formal, asymptotic convergence proofs, these are not universally available, rely on assumptions which often render them mutually incomparable between methods and, importantly, do not apply to performance in the finite data regime which is of the most relevance for method selection (Knaus et al., 2018). This situation is further complicated by a large space of different causal inference challenges corresponding to different distributional settings in the observed data. It will be demonstrated that inference methods are likely to perform quite differently on finite samples corresponding to different locations in the *problem space* of different distributional settings. This implies there may not be one method of causal inference which is universally superior across all finite data samples. So, in short, answering the question of which causal inference algorithm is best is hard because it requires finite sample evaluation, without access to ground-truth, and with the challenge of heterogeneous performance under different distributional settings. This paper tackles this challenge by proposing an evaluation method that allows for finite sample evaluation of arbitrary causal inference methods across an arbitrary selection of distributional settings.

Researchers have typically addressed the evaluation problem outlined above through two complementary evaluation methods, both of which provide access to a ground-truth causal effect with some trade-off in evaluative efficacy. Empirical evaluation methods use (randomized) experimental datasets and test an estimator's ability to reproduce the experimental result when using a non-randomized control group (to simulate an observational setting). This strategy was pioneered by Lalonde (1986) with iterative development by Heckman and Todd (1998), R. H. Dehejia and Wahba (1999), R. H. Dehejia and Wahba (2002), R. Dehejia (2005), and Smith and Todd (2005) with notable contributions from Hill, Reiter, and Zanutto (2005) and Shadish, Clark, and Steiner (2008). The data used in these evaluations is realistic - in so far as it is drawn from data collected as part of formal, real-world studies - but only a small number of such experimental datasets (with appropriate observational controls) are available and these cover a small and poorly defined subset of the problem space[1]. In contrast, the second approach, synthetic evaluation, relies on synthetic data generated by a hand-crafted data generating process (DGP). These DGPs can simulate any location in the distributional problem space but are often unrealistic in terms of the number and type of variables used as well as the functional forms used to simulate the outcome and intervention data. The mechanisms behind these two approaches - and the strengths and weaknesses that these mechanisms imply - are formalized in the chapters below.

The primary contribution of this paper is a hybrid evaluation method designed to overcome the weaknesses of both of the evaluation methods above. The method proposed is based on the *sampling* of synthetic DGPs defined over real, observational data. This hybrid approach combines the diagnostic clarity of synthetic evaluation methods with the realistic distributions of empirical evaluation methods. Appropriate parameterization of the sampling process allows for the generation of DGPs which are in well-defined, specific locations in problem space without the problematic implications of hand-crafted, 'targeted' (but potentially biased) design. Additionally, the sampling approach also allows for the generation of many distinct instances of evaluative datasets in the same problem class, allowing for repeat evaluation of a method and, thus, for convergence to an accurate distribution of performance - revealing average/best/worst case results rather than just single-point samples from an unknown distribution.

The hybrid approach proposed is not entirely novel. A number of other authors have proposed similar methods albeit with small but significant idiosyncrasies. These methods are reviewed in Chapter 4 below. A common shortcoming of all the existing work in this subfield is the lack of tooling to make the proposed evaluation method available to others such that it can be applied to methods not covered in the original papers and used with base data and problem-instance settings relevant to different academic fields of study. Existing code, if available, is tightly coupled to specific empirical source data and inflexibly/confusingly parameterized. With this context in mind, this paper makes

[1] The true data generating process in these datasets is unknown which means it is unclear what distribution properties the data displays.

two contributions. First, a thorough comparative analysis of the hybrid approaches proposed in the literature with the goal of selecting a single, strong hybrid evaluation method (or a synthesis of multiple methods). And, second, an accompanying tool that makes it easy to apply the selected method of hybrid evaluation to arbitrary causal inference methods using arbitrary base data and arbitrary problem-instance settings. The goal of this tool is to provide a consistent, universal means by which causal inference researchers - and users - can develop, compare and select methods of causal inference.

The rest of this paper proceeds as follows: Chapter 2 introduces causal inference and formalizes the notation and terminology used throughout this work. Chapter 3 establishes the causal inference problem space - the space of distributional settings that impact the performance of different causal inference estimators. This is the space over which estimators should be evaluated. Chapter 4 presents a review of the literature on causal estimator evaluation, comparing asymptotic, Monte Carlo and Hybrid Monte Carlo evaluation methods. Chapters 5 and 6 introduce Maccabee - a package for Hybrid Monte Carlo Benchmarking. Chapter 5 presents the design of the Maccabee benchmark, which builds on the best method from the review in Chapter 4. Chapter 6 explains the implementation used to operationalize this design. Chapter 7 present preliminary validation results. Chapter 8 concludes.

Part II

Causal Inference Theory

Chapter 2

A Framework for Causal Inference

This chapter provides a theoretical overview of observational causal inference. This serves as a foundation for the rest of the work, introducing the notation and framing which are used throughout. The framing of causal inference introduced in this chapter is primarily based on Rubin's Potential Outcome Framework as outlined in Holland (1986) with augmentation from Pearl's Structural Causal Models as outlined in Pearl (2009).

Per Holland, observational causal inference applies statistical methods to measure the "effects of causes" in non-experimental settings. IE, methods of observational causal inference seek to estimate the quantitative effect of some *treatment* on *units* with the potential - realized or not - to be exposed to that treatment without the use of randomized exposure[1]. Note that if the unit has no potential to be exposed to the treatment or is always exposed, then understanding the effect of the treatment on that unit is philosophically impossible: the effect of a cause is defined based on the observed outcome relative to some (hypothetical) *counterfactual* in which the cause did not occur. These ideas can be formalized by expressing them through a quantitative, statistical lens as follows.

2.1 Notation and Estimands

Let there be a population of units - U - with individual units from U indexed as u_i . Let $Z(u_i) = Z_i$ be an indicator variable that tracks whether a unit was exposed to the treatment [2]. Based on the binary nature of the treatment, each unit has two *potential outcomes* - $Y_1(u)$ and $Y_0(u)$ - which measure the outcome the unit experiences under treatment or absence of treatment (referred to as the *control* condition, following experimental nomenclature). Further, each unit has a (potentially vector-valued) covariate measurement X_i . The name covariate is used to imply that the variable X may co-vary (or be *correlated*) with the outcomes, Y_1 and Y_0 , and the treatment status Z across the population U . In the real world, a correlation between these variables may

[1] The term treatment is adopted from experimental literature for clarity.

[2] This implies a binary treatment regime in which treatment is either present or absent for each unit but the framework present in the text can be extended to multiple discrete or single continuous treatments.

result from the covariates in X causally affecting the outcome/treatment assignment or the inverse. For simplicity, I assume for the rest of this section that the covariates are measured pre-treatment and, thus, are not causally affected by a unit's treatment assignment or outcome(s) [3]. There are two types of covariates depending on the underlying causal mechanism: *confounders* are causally related to both the treatment and outcome of units in the population while *sources of heterogeneity* are related to either, but not both, the treatment or outcome. The reason for this naming will become clear shortly.

Given the above, the effect of a cause on an individual can be defined as $\tau_i = Y_1\left(u_i\right) - Y_0\left(u_i\right)$. *The Fundamental Problem of Causal Inference,* per Holland (1986), is that "it is impossible to observe the value of $Y_1\left(u_i\right)$ and $Y_0\left(u_i\right)$ for the same unit and, therefore, it is impossible to observe the effect of t on u" . For any given unit, we only observe a single outcome defined by $Y\left(u\right) = Z_i \times Y_1\left(u\right) + \left(1 - Z_i\right) \times Y_0\left(u\right)$ - the outcome under the treatment which the unit experienced. At this point, it may appear that the idea of causal inference is hopeless. However, introducing statistical tools (more specifically expectations), provides a path forward. The first step is to define average effect estimands in terms of observed and unobserved potential outcomes. These causal inference *estimands* can then be targeted by causal inference *estimators* defined over only observed data. Define the *population average treatment effect* (PATE) as:

$$PATE = E\left[\tau\right] = E\left[Y_1 - Y_0\right] = E\left[Y_1\right] - E\left[Y_0\right]$$

This is the most basic causal estimand. It is possible to construct analogous expressions for samples rather than populations and for the treated/control subgroups of the population/sample. However, for the purposes of this paper, the only other important estimator is the *conditional average treatment effect* (CATE) which refers to the causal effect conditioned on a covariate observation X_i :

$$CATE = E\left[Y_1 - Y_0 | X_i\right] = E\left[Y_1 | X_i\right] - E\left[Y_0 | X_i\right]$$

Note that the PATE can be recovered by averaging the CATE over all units in the population based on the formula below. This allows us to focus our attention on this estimand.

$$PATE = \tfrac{1}{n} \textstyle\sum_{i=1}^{n} E\left[Y_1 - Y_0 | X_i\right]$$

Given than two units with an identical observed X_i are effectively indistinguishable in experimental terms, the CATE is often referred to as the *individual average treatment effect* [4].

[3] Resolving the truth of this assumption is crucially important in practice - see Pearl (2009) for the risks of using covariates which are causally affected by treatment/outcome (even if the measurement is pre-treatment). However, this consideration is not important for establishing the fundamental operation of causal inference.

[4] The word average is used to indicate that units with identical X_i values may, in fact, have different treatment effects in which case the CATE is an average over these effects for units indistinguishable on the observed covariates.

2.2 The Challenge of Accurate Causal Inference

The estimands above imply that if we can accurately approximate the expectation of the two potential outcomes across the population, or a covariate level, then we can approximate the average treatment effects. IE, it is possible to estimate the unobservable counterfactual at the individual level using expectations for the potential outcomes defined over the population of units. However, estimating these expectations is challenging: In any experimental or observational setting - rather than observing $E\left[Y_1\right]$ and $E\left[Y_0\right]$, we observe $E\left[Y_1|Z=1\right]$ and $E\left[Y_0|Z=0\right]$. This expresses the idea that we observe the outcome conditioned on treatment assignment. There is no guarantee that $E\left[Y_1\right]=E\left[Y_0|Z=1\right]$ and, in fact, this is highly unlikely in observation settings. This is where the notion of selection bias become important.

Following Holland, let us call the average treatment effect arrived at from the observed outcomes the treatment effect prima facie:

$$T_{pf}=E\left[Y_1|Z=1\right]-E\left[Y_0|Z=0\right]$$

Observe that if the treatment assignment is independent of the potential outcomes - $P\left(Z|Y_1,\ Y_0\right)=P\left(Z\right)$ - then $E\left[Y_1|Z=1\right]$ does indeed equal $E\left[Y_1\right]$ and the true average treatment effect is recovered by the treated/control group estimates. This explains the power of randomized control trials. If treatment/control is assignment randomly then the condition above holds and the group outcome averages allow recovery of the true effect. In the absence of randomization, it is possible for the treat/control group average outcomes to be ‘biased’ by the systematic inclusion of units which have a better/worse potential outcome under treatment. This bias can be neatly formalized. Assume a constant treatment effect T . Then for the treated group we have: $E\left[Y_1|Z=1\right]=E\left[Y_0|Z=1\right]+T$. Substituting this into the formula for T_{pf} , we arrive at:

$$T_{pf}=T+\left(E\left[Y_0|Z=1\right]-E\left[Y_0|Z=0\right]\right)$$

The prima facie effect is the true effect plus the expected difference in the outcome under control for the treated and control groups. In the absence of randomization, systemic differences between the treatment and control group induce bias. The challenge of observation causal inference is how to undo the bias induced by these differences. Before proceeding to strategies for achieving this, it is important to explore the mechanisms which create systemic difference in the first place. To do this, I draw on Structural Causal Models (SCMs) introduced by Pearl and summarized in Pearl (2009). SCMs are useful because they provide an intuitive way to communicate the causal mechanisms which give rise to the bias discussed above. This understanding, in turn, makes it easier to understand the construction of bias-free causal inference estimators.

Pearl's SCMs relate the variables defined above - outcome, treatment and covariates - through a directed acyclic graph. The nodes in this graph are the variables and edges between nodes indicate causal dependency between the child node variable (the target of the directed edge) and its parent(s) (the origin of the edge(s)). More precisely, the value of each variable node is defined in terms of a function that depends solely on the values of the parents of the node in the graph. This implies that knowing the value of all the parent nodes of a child node fully determines the value of the child node[5]. As

[5] In a complete SCM each node has a parent which presents a source of uncorrelated noise which

a result of this causal dependence, variables will be correlated, to some degree, with their parents in the graph, to the ancestors of those parents, and to any other nodes which share common parents/ancestors. With this definition established, I present the simple SCM in Figure 1 below to explain the origin of selection bias in causal effect estimation.

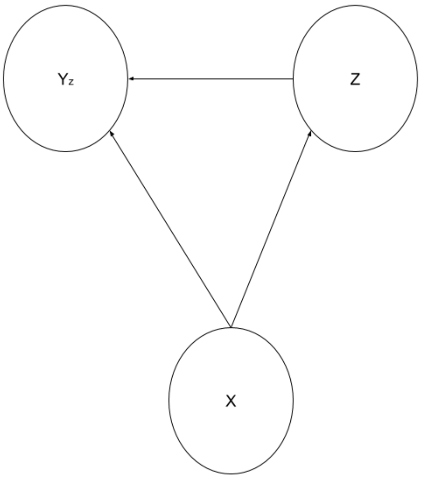


Figure 1: SCM displaying causal relations between the outcome, treatment status and covariates which gives rise to selection bias.

This SCM displays the simplest possible situation in which selection bias is a problem. The edge between Z and Y_z is the primary effect of interest. The value of Y_z will increase by T if $Z = 1$ relative to its value if $Z = 0$. Further, both Y_z and Z are causally related to the covariate X. IE, the treatment status and the outcome are determined by a function which depends only on X and noise. As a result of this causal mechanism, it is easy to see that $P\left(Z|Y_1,\ Y_0\right) \neq P\left(Z\right)$. So, the treatment status is not independent of the outcome as a result of the shared dependence on the value of X and the two groups will have systematically different base outcomes. This simple example provides insight into a more general pattern: where treatment assignment depends on the same covariates as the outcome, there will be a systematic difference in outcome between the groups. We say that the covariates which affect both treatment assignment and outcome confound the estimation of causal effects - hence the name confounders. Beyond clarifying the mechanism behind selection bias, the model in Figure 1 also hints at a viable path for counteracting the bias. If one conditions on X, then for fixed values of X, the only correlation left between Y_z and Z is due to the edge between them. If we treat X_i as an observation of a full set of covariates which full identify a unit, this process is intuitively clear. By conditioning on the covariates, the correlation between Y_z and Z is estimated on subpopulations of identical units. Any change in outcome in these subpopulations must occur as a result of the change in the treatment status. But note that this only holds if we observe all of the confounding covariates related to both the treatment assignment and outcome.

The process outlined here, and the assumptions required to operationalize it on real

explains the absence of perfect predictive power of the observed parents. For simplicity, this noise node is omitted from the models presented in the text. This implies that all the variables are fully determined by their parents.

data, are outlined in the next section.

2.3 Causal Inference as Predictive Inference under Assumptions

The ideas expressed above can be formalized in the Potential Outcome framework. The only nuance is that, under this framework, the outcomes under treatment and control are fixed and therefore there is no directed edge, in the language of SCMs, between treatment assignment and outcome. Despite this small difference, statistical dependency between the potential outcomes and treatment assignment is induced by confounders exactly as above.

Returning to the formalism of the estimands defined above, recall that finding the PATE requires estimating $E\left[Y_1\right]$ and $E\left[Y_0\right]$ but that, in an observational setting, one only has access to $E\left[Y|Z=1\right]$ and $E\left[Y|Z=0\right]$. If one assumes that the covariate vector X_i contains all confounders we have the following:

$$E\left[Y_1\right] = E_x\ \left[E_{Y_1}\left[Y_1|X\right]\right] = E_x\left[E_{Y_1}\left[Y_1|X,Z=1\right]\right] = E_x\left[E_{Y_1}\left[Y|X,Z=1\right]\right]$$
$$E\left[Y_0\right] = E_x\ \left[E_{Y_0}\left[Y_0|X\right]\right] = E_x\left[E_{Y_0}\left[Y_0|X,Z=0\right]\right] = E_x\left[E_{Y_0}\left[Y|X,Z=0\right]\right]$$

This result implies that:

$P\left(Y_1,Y_0|X,Z\right) = P\left(Y_1,Y_0|X\right) =$

This is the statement that the potential outcomes are conditionally independent of the treatment status given the covariates:
$Y_1,\ Y_0\ \perp\!\!\!\perp Z|X$
And this is exactly what was demonstrated to be true using the SCM above under the assumption that X does indeed include all confounders. Note that, under this framing, the problem of observational causal inference is reduced to a standard conditional estimation task $E\left[Y_z|X,Z=z\right]$ for the two treatment groups ($z\ \in \{0,1\}$). This is discussed in detail shortly.

Formally, the Rubin Causal Model makes 4 assumptions to recover an unbiased estimate of causal effects:

- **Assumption 1 - Stable Unit Treatment Value Assumption:**

  $Y\left(u\right) = Z_i \times Y_1\left(u\right) + \left(1-Z_i\right) \times Y_0\left(u\right)$. IE, each unit has a fixed outcome under treatment and control and there is no interaction between the outcomes of different units.

- **Assumption 2 - Exogeneity of Covariates:**

  $X_i\ |Z_i = 1\ \ = X_i|\ Z_i = 0$. IE, covariates are independent of the treatment assignment.

- **Assumption 3 - Conditional Independence:**

$Y_1,\ Y_0\ \perp\!\!\!\perp\ Z|X$. IE, the covariate observation X contains all confounders and there is conditional independence between outcomes and treatment assignment conditional on X .

- **Assumption 4 - Common Support:**

  $0 < P\left(Z=1|X=x\right) < 1$ for all $x \in \mathit{support}\left(X\right)$. IE, there is some probability that all possible units are exposed to both treatment and control. Philosophically, this is required for the existence of Y_1 and Y_0 because in the absence of the potential for exposure it is unclear what the (impossible) potential outcome would mean. Practically, this ensures that the outcome estimands in each group have common support and can be combined and averaged without inducing bias. This assumption can be relaxed but this must be done with extreme care.

Under these assumptions, it is possible to estimate $E\left[Y_z|X\right]$ using $E\left[Y|X,Z=z\right]$. So, the estimands defined above become:

$$CATE\left(X=x\right)=E\left[Y|X=x,Z=1\right]-E\left[Y|X=x,Z=0\right]$$

$$PATE=\sum_x P\left(X=x\right)\times\left[E\left[Y|X=x,Z=1\right]-E\left[Y|X=x,Z=0\right]\right]$$

These expressions make it clear that, under certain assumptions, observational causal inference reduces to estimation tasks which are typically *predictive* in that they rely only on standard statistical correlation. In the framing above, the first step of causal inference is to estimate the quantity $E\left[Y|X,Z\right]$ - the *response surface* in the words of Hill (2011) - given the observed data from the two treatment groups. This is a standard predictive estimand. Under causal assumptions, the resultant estimated response surface(s) can then be combined to find average/conditional treatment effects.

While this two step process does clarify the connection between causal and predictive inference, it is a severe oversimplification. Different inference methods can be used to operationalize the 'standard' inference of the response surface - ranging from simple means across covariates blocks to fully-fledged function fitting. Moreover, one can define equally unbiased expressions for the estimands above which depend exclusively on 'standard' inference but have meaningfully different mathematical forms. The different estimators implied by these two complications may be more or less efficient - producing 'better' results on the same dataset - and they may exhibit different performance under different distributional settings. Understanding these differences is crucial - the goal of research into causal inference estimators is to produce methods which are more efficient and work in specific, or across many, distributional settings. The next chapter explores different causal inference estimators and how their performance is affected by (a formalized notion of) the distributional setting of observed data. The formalization of distributional setting is particularly important as this is the constant background against which new estimators must be validated.

Chapter 3

The Causal Inference Problem Space

The previous chapter ended with the claim that many causal estimators can be constructed to target the same causal estimands and that these estimators may display varied performance under different *distributional settings.* The primary goal of this chapter is to formalize the notion of a *distributional setting* and motivate why the *distributional setting*, as defined, affects estimator performance. In so doing, I will introduce the notion of a causal inference problem space which is the space of all performance-relevant distributional settings. I also discuss techniques for measuring the location of some observed data in this space.

I will proceed in three steps. First, I will provide a brief overview of the landscape of causal inference estimators. While this overview will include instructive examples of different approaches, it is not meant as a complete (or even partial) review of different methods. Rather, the analysis will provide initial insight into the importance of the properties of the underlying data distribution for the performance of different estimators. The discussion of distributional properties will remain informal at this stage. Second, I will formalize the notion of a distributional setting by expressing causal problems in terms of a joint distribution over observed data. In this context, a distributional setting can then be defined as a joint distribution over the observed data with some set of properties that affect estimator performance. Finally, I will establish the axes of the Causal Inference Problem Space. This is an abstract space over distributional settings with each axis of the space corresponding to the possible values taken on by some property of the joint distribution.

3.1 The Sensitivity of Causal Estimators to Properties of the Observed Data

In the previous chapter, I arrived at an expression for the average causal effect estimands in terms of $E[Y|X, Z]$ and referenced this as the 'response surface'. I also asserted that it was possible to find expressions for the same estimands using meaningfully different, but purely statistical/predictive, estimators. An intuitive explanation for this statement is that causal processes are made up of different *causal mechanisms* that relate the various variables[1]. And, in order to recover the unbiased effect, it is sufficient to accurately model only one of these mechanisms. Hill (2011) echos this explanation and refers to two mechanisms: the *treatment assignment mechanism* - which relates the covariates X to the treatment status Z - and the *response mechanism* - which relates the covariates X and the treatment status Z to the observed outcome Y . The authors point out that an accurate model of either mechanism is sufficient to recover average treatment effects. Künzel, Sekhon, Bickel, and Yu (2019) further divide the *response* mechanism into an *outcome mechanism* which relates the covariates X to the outcome without treatment Y_0 and the *treatment effect mechanism* which relates X and Z to τ which, in this framing, is the change in outcome (from Y_0 to Y_1) as a result of treatment [2]. Hypothetically, accurate specification of any of the now three mechanisms is sufficient to recover causal effects. But, given that it is impossible to directly model the treatment effect mechanism without a model of one of the other two mechanisms, it is only useful in so far as it clarifies a potential source of heterogeneity in the combined (outcome and treatment effect) response mechanism as defined by Hill (2011). With this idea of causal mechanisms in mind, different causal estimators can be understood as arising from the targeting of either, or both, the treatment assignment and response mechanisms. The targeted mechanism also explains the sensitivity of different estimators to different properties of the observed data distribution. With this established, I proceed to the brief overview of different estimators in order to make the points above more tangible.

Looking first at methods which target the treatment assignment mechanism: Matching ((Abadie & Imbens, 2006; Rosenbaum & Rubin, 1983) aims to mitigate selection bias by creating treatment and control groups with the same distribution of covariates, reversing the effect of the treatment assignment mechanism. This is done by pairing units with similar covariate values but different treatment exposure to create two groups with similar overall distributions[3] [4] [5]. This process is non-parametric relative to the treatment assignment mechanism because it targets the result of the mechanism without attempting to directly model it[6]. This means it is not sensitive to the exact functional form of the mechanism. However, it is still sensitive to other aspects of the observed data. Broomberg (2017) describes the potential sensitivity as follows. First, as the number of covariates grows, the probability of finding similar units, for any fixed notion of similarity, shrinks exponentially, thus requiring exponentially larger observed donor groups to achieve balance. This holds even in the absence of strong selection pressure producing different covariate distributions between the groups. Second, if

[1] Mathematically, these mechanisms take the form of functions which take as inputs some subset of the variables for some unit and output the value of some other variable for that unit. These functions represent some real generative process which relates the measured quantities in the real world. In order to represent real stochastic processes the functions may themselves be stochastic and take the form of distributions parameterized by the input variables.

[2] The covariates appear in the treatment effect mechanism because the treatment effect may be non-homogeneous/non-constant in which case it will depend on the covariate values of the exposed unit.

[3] The reality of the pairing is slightly more nuanced. The standard case is that observed units from a treated group are matched with control units from a 'donor pool'. This produces an estimate of the average effect for the treated.

[4] Defining a metric in covariate space is a complex task, so there is no single definition of 'similar' units. A naive approach is simply to measure the Euclidian distance between covariate vectors in real-number space. Ultimately, a successful metric can be anything which, when applied to large donor pools of observations, results in matches that produce equal covariate distributions across the groups.

[5] Imai, King, and Stuart (2008) show that if the distribution in the two groups is indeed the same (balanced), then a simple average over the observed outcome is an unbiased estimator of the average treatment effect.

[6] Matching methods may indeed have parameters which determine their operation but the correctness of these parameters is independent of the functional form of the treatment assignment mechanism.

there is strong selection pressure, then it is likely that the distribution of covariates will be meaningfully different between the groups. The larger the imbalance, the less likely it will be to find equivalent units in the opposite group and the larger the set of observed units that will be required to find matches. So, in short, matching is sensitive to the number of observed units, the number of covariates, and the functional form of the treatment assignment mechanism (in so far as it affects balance).

Propensity score based methods address some of the weaknesses above by explicitly modeling the treatment assignment mechanism and estimating a probability (propensity) of treatment for each observation. Austin (2011) summarizes the various ways in which the propensity score can be used to remove bias. The four classes of methods reviewed by the authors are matching/stratification on the propensity score, inverse probability of treatment weighting, and propensity-based covariate adjustment. These methods share at least two failure modes related to the observed data: one, Hill (2011) points out that the common methods used for estimation of the treatment mechanism are parametric and therefore sensitive to the functional form of the underlying mechanism and the number of covariates. If this form is misspecified, or there are too many covariates, then the resultant propensity scores will not properly mitigate bias. Two, Knaus et al. (2018) points out that small propensity scores - which inevitably arise when there is an imbalance in covariate distribution across groups - can produce high variance (or even degenerate) estimates in weighting-based methods. So, again, there is sensitivity to the number of covariates, the functional form of the treatment assignment mechanism, and the covariate balance.

Turning to methods which estimate the response surface, a similar pattern of sensitivity is evident. The simplest methods in this class are referred to as conditional mean estimations by Knaus et al. (2018) or as T-learners by Künzel et al. (2019). These methods estimate the conditional mean $E\left[Y|X, Z=z\right]$ separately for both the treatment and control groups (a method which was motivated in Chapter 2). Much like with direct modeling of the treatment assignment, this approach is sensitive to specification of the function form used for the estimation. This means there is sensitivity to the underlying functional form (and complexity) of the response mechanism. This sensitivity has inspired a huge number of flexible semi/nonparametric estimation schemes. See, for example, Causal Forests (Athey, Imbens, & Wager, 2018), BART (Hill, 2011), and X-learner (Künzel et al., 2019). But even these flexible estimation methods are sensitive to aspects of the data distribution. In the case of outcome function fitting, imbalance means that, in both groups, there are certain regions of the covariate support with fewer observations. This makes it hard to accurately fit functions in these regions. This is less problematic in parametric methods with fewer degrees of freedom but seriously affects non-parametric methods which rely on the data for accurate local estimation. Further, response mechanism-based approaches only work if they correctly capture the response dependency on all confounders and the treatment assignment (the treatment effect). This means that, much like in any function fitting task, there is sensitivity to the explainability of the response with respect to the data (the signal-to-noise ratio) and the number of, and redundancy between, predictive variables (the covariates). Incorrect inference of the response mechanism is possible in cases where there is a low signal-to-noise ratio and/or the presence of many covariates with similar but non-identical correlations with the prediction target

The final class of methods explicitly, or implicitly, combines the modeling of both the treatment and response surface, usually through weighted estimation targets. Examples which include explicit models of the two mechanisms are Rubin and Thomas

(2000) and Robins and Rotnitzky (1995). Scharfstein, Rotnitzky, and Robins (1999) show that if semiparametric estimation is used for modeling both the treatment assignment and response mechanisms, then the resultant estimators are valid if either the treatment assignment *or* the response mechanism are correctly modeled, leading to the name "Double Robust" estimators for this class of methods. Note that while this double robustness is appealing, the models of the two mechanisms are still sensitive to the same aspects of the data as outlined above. The combination simply improves the probability of valid results by hedging the accuracy of the one model with the other. More modern methods may implicitly combine models of both mechanisms rather than explicitly combining them. For example, Fredrik D Johansson et al. (2016) jointly optimize a neural network based estimator to produce balanced covariate representations and accurate outcome predictions using this representation. While demonstrating exact sensitivity is hard for complex estimators such as this, it is reasonable to assume that the same sensitivities outlined above will have some impact on accuracy. This is based on the idea that these sensitivities represent fundamental differences in the amount of information available to estimators and the complexity of extracting this information. No estimator, regardless of flexibility or complexity, is immune to the impact of decreasing predictive information.

The synthesis above is, by no means, a complete picture of all the causal estimators present in the literature. Rather, it serves to make a simple point: All estimators - regardless of the mechanism(s) they model and the parametric/non-parametric inference method used to perform the modeling - are sensitive to aspects of the observed data. From the examples above, it appears that the list below is a minimal subset of the performance-relevant aspects of the observed data:

- The number of observations
- The functional forms of the treatment and response mechanism
- The balance of the covariate distribution across groups
- The number of covariates
- The predictive power of the covariates for the treatment assignment/outcome (signal-to-noise ratio and overlap/correlation between predictors)

The *distributional setting* of a particular causal inference problem collectively refers to the value of all of these performance-relevant aspects of the observed data. The goal of the next section is to formalize the notion of a distribution setting as a set of properties of the joint distribution over the observed data.

3.2 Distributional Settings as Joint Distributions over Observed Data

The section above observed that different estimators target different underlying, component mechanisms and that this targeting produces sensitivity to different aspects of the observed data. This section formalizes this idea by showing that the two component mechanisms outlined above can be thought of as acting together to produce a joint distribution over the observed data, with the properties of this distribution affecting estimator performance. In this framing, a *distributional setting* is a joint distribution over the observed data with specific marginal/collective properties that are relevant to performance. Many joint distributions may have the same property values and therefore represent roughly the same distributional setting.

The first step in establishing this framing is to express the component mechanisms discussed above as probability distributions over the variables involved. This is the paradigm used in Pearl's Structural Causal Models (SCM) (Pearl, 2009) which builds on the more general idea of representing generative processes as factorized probability distributions that relate causally-connected variables. Readers unfamiliar with these concepts should see Bishop (2006) Chapter 8 for an introduction to graphical models and their interpretation as representing causal, generative processes. In this paradigm, each of the mechanisms above is represented as a conditional distribution:

- **The Treatment Assignment Mechanism:** $P\left(Z|X\right)$

- **The Response Mechanism:** in a classic SCM, this would be $P\left(Y|Z,X\right)$. For consistency with the Potential Outcomes Framework, I further factor $P\left(Y|Z,X\right)$ as $P\left(Y|Z,X\right) = P\left(Y_1|\mathrm{Z}\ ,\ \mathrm{Y}_0,\ \tau\right)P\left(Y_0|Z,X\right)P\left(\tau|Z,X\right) = P\left(Y_1|Y_0,\tau\right)P\left(Y_0|X\right)P\left(\tau|X\right)$. This isolates the two sub-mechanisms identified by Künzel et al. (2019) - the *outcome mechanism* and the *treatment effect* mechanism.

I add to these mechanisms the idea of an *observation mechanism* which produces the sample of units - both treated and untreated - that are available for study. The *observation mechanism* is not directly useful in revealing the causal effects for the units within the sample but it is important for generalizing the results beyond this group. That is, one can only generalize an inferred effect to a wider population if one assumes the observation mechanism is producing representative samples of the wider population. Further, even if it is not used directly in the causal estimators, this mechanism is an important part of the generative process which produces the observed data and can thus affect estimator performance. For simplicity, I will ignore potential conditioning on a wider population and sampling process and simply refer to a generative distribution over observed units X .

- **The Observation Mechanism:** $P\left(X\right)$

These three conditional distributions - representing the component causal mechanisms - combine to produce a joint distribution over the observed variables:

$$P\left(X,Y,Z\right) = P\left(Y|X,Z\right)P\left(Z|X\right)P\left(X\right)$$
$$P\left(X,Y,Z\right) = P\left(Y_1|\mathrm{Z}\ ,\ \mathrm{Y}_0,\ \tau\right)P\left(Y_0|Z,X\right)P\left(\tau|Z,X\right)P\left(Z|X\right)P\left(X\right) \text{ given that}$$
$$Y = Z \times Y_1 + (1-Z) \times Y_0$$

It is trivially true than any observed dataset can be described by some joint distribution $P\left(X,Y,Z\right)$ over the variables X, Y and Z . Above, I have built this joint distribution from the 'bottom up' by defining different component sub-mechanisms of the general causal mechanism, expressing these mechanisms as conditional distributions, and combining them into the joint distribution. An equivalent process for this analysis would be to start from the joint distribution and assert, based on conditional independence implied by some generative model, that the joint distribution can be factored in the way above. The simple but representative generative model in Figure 1 of the previous chapter (Chapter 2) produces the exact factorization above under the conditional independence relations implied by its structure. The equivalent generation model for the potential outcome based framing is given below in Figure 2.

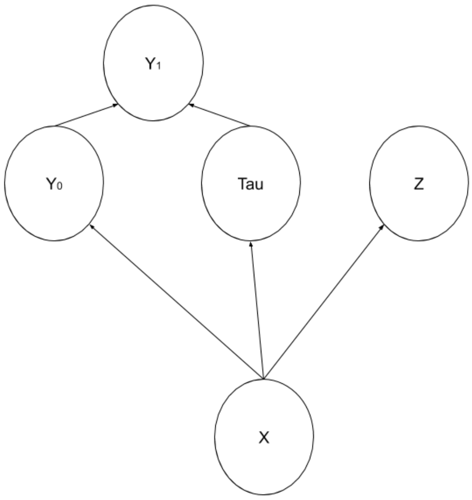


Figure 2: Graphical (Generative) Model for the variables in Rubin's potential outcomes framework.

With this joint distribution and its factorization in hand, the next conceptual step is to connect the aspects of the observed data discussed above to properties, and factored components, of the distribution. This is relatively straight forward. Some of the aspects of the observed data discussed above relate to properties stemming from a single (factorized) component of the joint distribution. For example, the non-linearity of the outcome mechanism dependents on the functional form of $P\left(Y_0|X\right)$. Others stem from the combined effect of multiple components. For example, the covariate balance in each group can be described by $P\left(X|Z\right)$ which by a simple application of Bayes Theorem is given by the formula below. This formula implies that as the treatment assignment mechanism varies away from simple random assignment $P\left(Z|X\right) = 0.5 \forall\, X$, the covariate balance in the groups will diverge (as one expects based on intuition).

$$P\left(X|Z\right) \propto P\left(Z|X\right) P\left(X\right)$$

A complete mapping between the various aspects of the observed data discussed above and properties of the joint distribution is provided later. For now, I simply assert that such a mapping exists based on the motivating examples above. The question then becomes why framing the different aspects of the data discussed above in terms of the joint distribution is useful.

There are three reasons for this. Firstly, it unifies conceptually disparate aspects of the observed data - functional form, covariate balance, predictive power etc - into a single theoretical object. The joint distribution fully determines the aspects of the data which impact performance. Thus, the *distributional setting* can be used to coherently describe a class of joint distributions which reliably result in similar estimator performance characteristics. All joint distributions with similar, quantitatively-defined properties - that is, with the same distributional setting - should exhibit similar performance. Second, this framing provides a clear connection between the process used to generate the observed data, the observed data itself, and the performance of estimators. The introduction briefly mentioned that Monte Carlo methods can sample from flexibly-specified joint distributions. The dependence of estimators on the joint distribution over X, Y , and Z - as established here - is what results in Monte Carlo being a useful evaluation tool. Monte Carlo sampling can be used to generate joint distributions corresponding to different distributional settings in which estimators can be expected to perform quite differently relative to each other and to their performance in other settings. Third, combining these two reasons, a concrete notion of distributional setting allows for the construction of an explorable causal inference problem space. The abstract problem space contains all of the meaningfully different distributional settings (those with different impact on estimator performance) and the use of Monte Carlo generative processes allows for the exploration of this space in order to validate estimators in a (theoretically) exhaustive set of circumstances. Establishing this space is the focus of the next section.

3.3 The Causal Inference Problem Space

The first section in this chapter used examples of estimators to highlight sensitivity to aspects of the observed data. The second section introduced the idea that the performance-relevant aspects of the observed data can be fully described in terms of a *distributional setting* which describes the properties of the joint distribution $P(X, Y, Z)$. This section brings these ideas together by describing the *axes* which define the space of distributional settings eluded to at the end of the last section. The axes introduced below represent the properties of the joint distribution which can vary and, in doing so, produce changes in the performance of estimators when applied to data from the distribution. The resultant problem space is intended to be exhaustive, implying that the axes should span all *distinct distributional settings* - classes of joint distributions which yield different estimator performance. The claim that the defined space is exhaustive is impossible to prove concretely. Indeed, I expect that iteration on the axes may be required as new methods are developed. However, it is worth noting two points which motivate the choice of axes below:

- The axes were selected by examining the properties of the joint distribution affected by manipulation of the underlying mechanisms described in the last two sections - primarily different covariate distributions, and treatment assignment/outcome mechanisms. This is not a guarantee that all relevant distributional properties have been discovered but it does mean those selected are non-arbitrary and arise from varied causal mechanisms.

- The axes selected by this process end up spanning the evaluation settings present in the benchmarking literature that is reviewed in Chapter 4. Primarily, agreeing with the distributional properties measured by Dorie, Hill, Shalit, Scott, and Cervone (2019). This means that, at a minimum, these axes appear to span the space of distributional settings present in a reasonably wide sample of the relevant literature.

Without further ado, I present the proposed axes of the causal inference problem space. For each axis, I explain which distributional component(s) of the overall causal mechanism affect its value. Following each definition (or group of related definitions), I provide simple toy datasets and estimators to demonstrate the impact of the axis on estimator performance. The toy datasets are based on a single covariate X and use the simplest possible setting for all axes but the one in focus.

1) **The Nonlinearity (Modeling Complexity) of the (Untreated) Outcome Mechanism** which defines the distribution over the value of Y_0 (the outcome without treatment) based on a unit's covariates. This corresponds to the functional form of the distribution $P\left(Y_0|X\right)$.

2) **The Nonlinearity (Modeling Complexity) of the Treatment Effect Mechanism** which defines the distribution over the treatment effect for each unit. This corresponds to the form of the distribution $P\left(\tau|X\right)$ and specifically the dependence on the covariates X .

   Toy Example for the Nonlinearity of the Response Mechanism

   The example below demonstrates the impact of a nonlinear response mechanism - created by nonlinearity in either or both of the untreated outcome or the treatment effect - when using parametric (potentially misspecified) conditional mean regression. This example is inspired by a similar plot found in Hill (2011). It is evident that the misspecification of the regression model used to model the outcomes in each group produces an estimation error.

   It is worth noting that this toy example glosses over some nuance. First, the error depicted only applies to estimates of the individual treatment effect. Estimates of average effects will be unbiased in this case as a result of the balance of the covariates across the treatment groups. (The average effect estimate in the example below is, indeed, 0). Second, depending on the modeling approach used, nonlinearity in the treatment effect mechanism may not have the same result as nonlinearity in the outcome mechanism. The point of this example is simply to demonstrate that there is sensitivity to the nonlinear of the outcome mechanism.

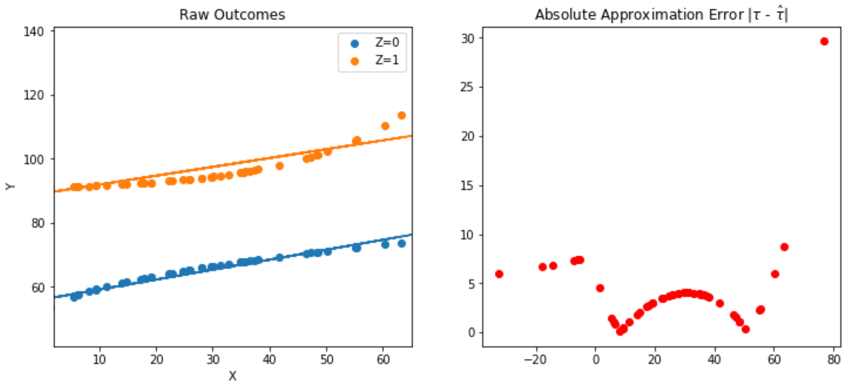


Figure 3: Left panel: the covariate and outcome values for the nonlinear response mechanism toy example. The data is divided into treatment groups with a linear outcome model plotted for each group. Right panel: the absolute error in the predicted individual treatment effect for each observation in the dataset. It is clear that a nonlinear outcome mechanism, combined with a misspecified outcome model, produces bias in the effect estimation.

3) **The Magnitude of the Treatment Effect:** the magnitude of treatment effect, measured relative to noise in the outcome mechanism (as defined above). This corresponds to the magnitude of the values produced by $P\left(\tau|X\right)$ and the magnitude of the noise in $P\left(Y_0|X\right)$ - IE, the changes in Y_0 not correlated with changes in X.

Toy Example for Magnitude of Treatment Effect

The example below demonstrates the impact of a treatment effect which is small relative to the noise in the outcome mechanism. The base outcome is linear and the treatment effect is constant. This is a classically easy problem but, in this case, the noise makes correctly-specified parametric inference highly inaccurate.

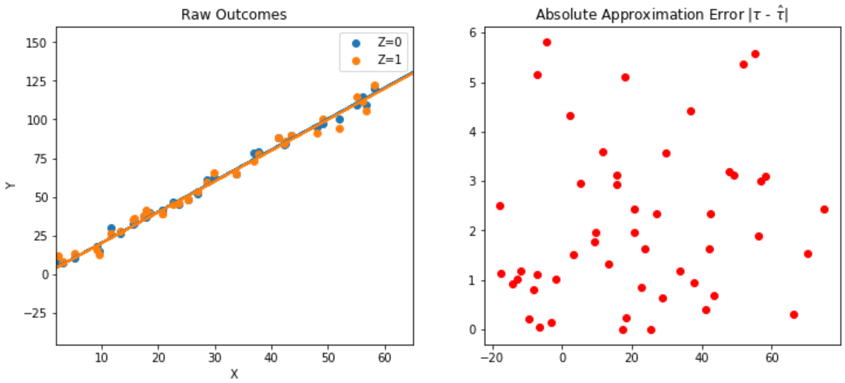


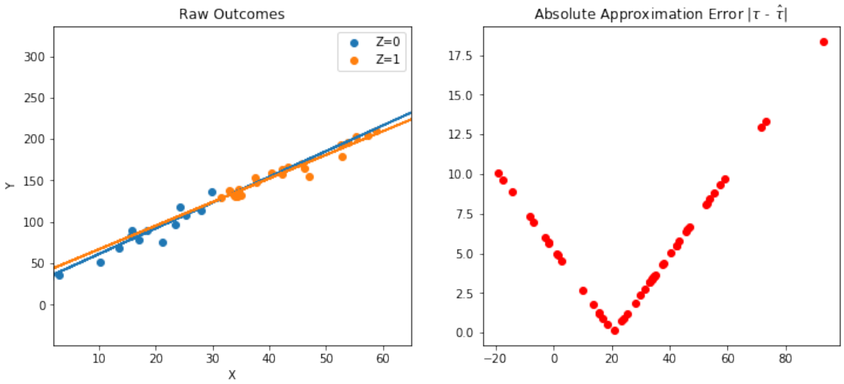
Figure 4: Left panel: the covariate and outcome values for the magnitude of treatment effect toy example. The data is divided into treatment groups with a linear outcome model plotted for each group. Right panel: the absolute error in the predicted individual treatment effect for each observation in the dataset. It is clear that a low signal-to-noise ratio produced by treatment effect values that are small relative to the outcome noise produces bias in the effect estimation.

4) **The Nonlinearity/Complexity of the Treatment Assignment Mechanism** which defines the distribution over the treatment assignment value based on a unit's covariates. This corresponds to the form of $P\left(Z|X\right)$, the treatment assignment mechanism per the definitions above.

   This can have an effect in one of two ways. In methods which directly model the treatment assignment mechanism to reverse selection bias, misspecification (of a more complex function) will induce bias. In methods which don't model the treatment assignment, a nonlinear assignment will (likely) result in more extreme propensity scores. This will produce regions of the covariate space with imbalance/the absence of overlap.

   The second situation is displayed in the toy example below. A non-linear treatment assignment logit - in this case a simply cubic around the mean value of the covariate X - produces treatment groups with little to no overlap. This results in error despite the correctly specified outcome model.

   Toy Example for Nonlinearity of Treatment Assignment Mechanism



5) **The Covariate Balance:** The (dis)similarity of the distributional of covariates in the two treatment groups. As above, this is determined by $P(Z|X)$ and $P(X)$.

6) **Covariate Distribution Overlap** this is conceptually similar to balance but refers to whether the domain of the covariate distribution in each of the treatment groups is the same. Much like balance it is determined by $P(Z|X)$ and $P(X)$.

In finite samples, the impact of imbalance and lack of overlap is similar, so a single toy example is given below. However, in the asymptotic limit, appropriate methods can reverse the bias induced by imbalance but will not reverse the bias induced by lack of overlap. This is why overlap was taken as a strong assumption of valid causal inference in Chapter 2.

Toy Example for the Balance and Overlap Axes

The example below demonstrates the effect of covariate imbalance induced through strong selection pressure. The example outcome is a simple linear model with a constant treatment effect and moderate noise. Strong selection pressure is simulated by filtering the treated/control observations which fall outside of the indicated overlap region. This produces treatment groups with an unbalanced distribution of the covariate X. This results in conditional regressions that extrapolate beyond the bounds of the outcome information. When combined with moderate noise, this produces large error outside of the region of overlap between the groups.

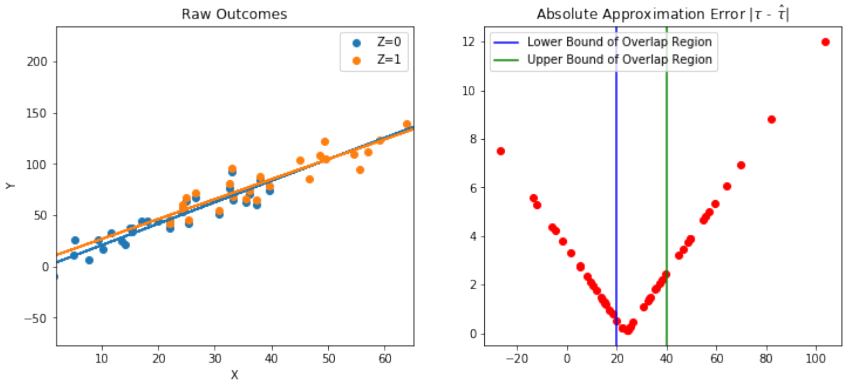


Figure 5: Left panel: the covariate and outcome values for the balance/overlap toy example. The data is divided into treatment groups with a linear outcome model plotted for each group. Right panel: the absolute error in the predicted individual treatment effect for each observation in the dataset. It is clear that, with a finite sample size, imbalance between the treatment groups' covariate distributions produces bias in the effect estimation.

7) **The Percent of Units Treated:** the (dis)similarity of the number of units in each group stemming from systematically extreme (low or high) values of the probability for treatment for all observed units. Again, primarily determined by $P\left(Z|X\right)$ and $P\left(X\right)$.

Toy Example for the Percent of Units Treated

The example below demonstrates the effect of a small percent of units treated. It was created by giving all units a small probability of treatment - set at $P(T|X) = 0.1$.The result is similar to the case of imbalance, in that there is imperfect overlap that results in extrapolation. But, in this case, this lack of overlap is created simply by a small treated group even though covariate balance holds in the asymptotic limit. IE, if there were many more observations, and the same fixed but small propensity for treatment, the bias would dissapear.

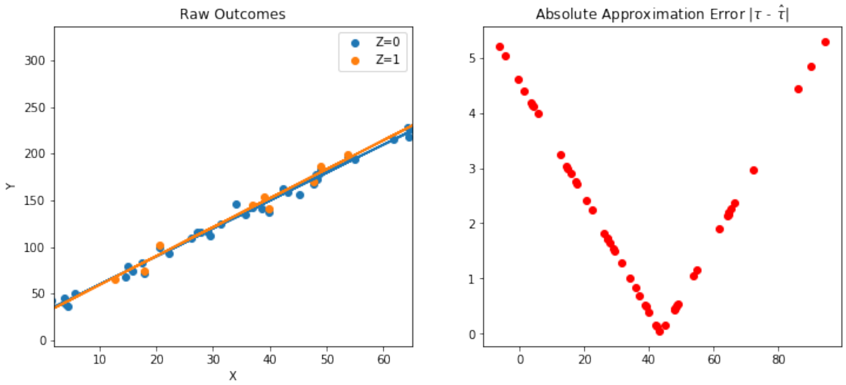


Figure 6: Left panel: the covariate and outcome values for the small percent of treated units toy example. The data is divided into treatment groups with a linear outcome model plotted for each group. Right panel: the absolute error in the predicted individual treatment effect for each observation in the dataset. It is clear that, with a finite sample size, a small treated group produces bias in the effect estimation.

8) **The Degree of Alignment Between the Treatment Assignment and Outcome Mechanism:** the number of confounding covariates - that appear in both the treatment assignment and outcome mechanisms - relative to the number of non-confounding, but still, predictive covariates - that appear in at least of the two mechanisms. The term alignment, to reflect this degree of confounding, is taken from Kern, Stuart, Hill, and Green (2016).

   The impact of this axis is subtle. If there are a small number of confounders relative to the variables that are predictive of either (but not both) of the treatment assignment and outcome, then controlling for confounding becomes a difficult variable selection problem. It is possible that a model may accurately model both the treatment and outcome mechanism without controlling for the confounding effect of a few variables even if the mechanism by which these variables affect treatment assignment/outcome is relatively simple. Conversely, if there is a large number of confounders relative to the predictive covariates then variable selection is easier but modeling the treatment/outcome mechanisms (or reversing bias by other means) is more challenging. This complexity means that this axis does not lend itself to simple demonstration through a toy example. I defer quantitative demonstration for Chapter 7, which presents results from a benchmark that varies the alignment axis of the distributional setting.

9) **Other Axes:** there are two further properties of observed data that are relevant to estimator performance but are not strictly properties of the joint distribution. These properties are ignorability - whether all covariates which make up the treatment and outcome mechanisms are observed - and the number of observed samples - that, as was clear from the above, modulates the effect of many of the other axes. Given their close relation to the joint distribution, and their impact on performance, I treat these two properties as axes of the distributional setting problem space.

3.4 Measuring the Location of Data in the Problem Space

Dorie et al. (2019) propose the use of *metrics*, defined over the various variables discussed above, to measure the location of a DGP along an axis. These metrics work by assigning a numerical score to data drawn from a DGP, with this score proportional in some way to the property described by the target axis. Note that this means that these metrics are *estimates* of the position along an axis for two reasons. First, the metric itself is only proportional to the underlying axis *concept* and there may be many, roughly equivalent functions that share this proportionality property but do not produce values that are absolutely comparable. Second, the metric is calculated using data sampled from the DGP. This means there is sampling bias associated with the metric value. That said, it is not crucial that these metrics are perfectly *accurate* because there is no hard ground truth against which accuracy is measured. The problem space is a *conceptual* space. Estimators are expected to be sensitive to the approximate location of data in this space but it is the region of the data in the space more than the (undefined) exact location that matters. This means heuristic metrics, that measure relative position, are perfectly acceptable.

With this established, I proceed to describe the metric scheme proposed by Dorie et al. (2019). Rather than describe each metric, I focus on the methods they use to measure concepts which are common to multiple axes above (for example, linearity). Before doing this, I first draw the distinction between the oracle and observed variables over which the metrics are defined. A complete list of the metrics proposed by the authors can be found in the supplementary material of their paper. These metrics can be used to measure all of the axes proposed above.

3.4.1 Observed vs Oracle Variables

The sections above established that the axes of the problem space correspond to properties of the distributional setting - the joint distribution over the observed data. While this is true, it is clear that distributions over unobserved data play a critical role in creating/defining the distributional setting of the observed data. For example, the linearity of the outcome mechanism depends on the distribution over the potential outcome without treatment - $Y0$ - conditioned on the observed covariates.

Accurate measurement of the position of observed data along the axes above therefore requires access to unobserved data. Without this access, it is still possible to *approximately* measure the location but it may be hard to disambiguate location along related axes. For example, low linearity of the observed outcome with respect to the covariates could imply nonlinearity in the untreated outcome mechanism or the treatment effect mechanism. This doesn't mean that all metrics over observed data are approximate. For example, an estimate of the linearity of treatment assignment may be accurate given that both true treatment assignment and covariates are observed.

If the true DGP is known, then one has access to both observed and unobserved variables. This means that metrics including both can be calculated. To draw a distinction between (sometimes approximate) metrics over the observed data and (accurate) metrics over some unobserved data, Dorie et al. (2019) refer to the former as observed metrics and the later as oracle metrics. I adopt this nomenclature in the rest of the paper when referencing these classes of variables.

3.4.2 Functions for Measuring Axes Concepts

While the axes above each correspond to a unique property of the distributional setting, there is a smaller number of unique concepts that appear across the axes. For example, the concept of linearity appears directly in three of the axes (an indirectly in one more). The combination of a concept and the data to which the concept is applied forms a unique axis.

Metrics to measure the position of data along each axis use functions that measure these concepts applied to some subset of the observed/oracle variables. Below, I describe the three concepts that cover all of the nontrivial metrics in Dorie et al. (2019) and outline the different functions that can be used to produce quantified (but heuristic) estimates of each concept.

1) **Linearity:** the linearity of the function which relates some (potentially vector-valued) variable A to a real valued variable B can be quantified using the R^2 value of a linear regression of B on A. This captures how much of the variance in B can be explained through a linear relation with A. If B is categorical, the linear regression can be replaced by a logistic regression to achieve the same effect in the probability space of the relationship between A and B. Metrics that use this approach are applied to measure the axes:

    - The Nonlinearity (Modeling Complexity) of the (Untreated) Outcome Mechanism
    - The Nonlinearity (Modeling Complexity) of the Treatment Effect Mechanism
    - The Nonlinearity/Complexity of the Treatment Assignment Mechanism
    - The Degree of Alignment Between the Treatment Assignment and Outcome Mechanism [7].

2) **Distributional distance:** the distributional distance between two sets of observations of the same variables can be measured in a number of ways. **Mean-based**

    Distance Measures

    - The L2 (vector-space) distance between the mean of variables in each set of observations. This approach summarizes each group of observations using the mean of each variable and then calculated the distance between the means. This is only useful if the mean is a good summary and if the variables have roughly similar scales. Otherwise, a (relatively) small difference in the value of the mean of a variable with a (relatively) large scale may overwhelm all the very similar mean values of variables with small scales and produce a large overall distance.
    - The Mahalanobis distance between the group means. This is equivalent to the L2 distance with each covariate scaled by its empirical standard deviation. This accounts for the scaling problem assuming the empirical standard deviation is an appropriate scale factor.

    Distribution-based Distance Measures

[7] Measuring the correlation between the treatment and observed/oracle outcome variables can be used as a proxy for ratio of confounding to non-confounding variables in the treatment and outcome mechanism

- Both the L2 and Mahalanobis distance can be applied to nearest neighbor pairs across the two sets of observations - rather than the mean - with the result averaged across the data set. This produces a measure of distributional similarity that doesn't rely on the mean summary and is empirically weighted by the number of observations in each region of the domain.
- The Wasserstein distance measures the absolute difference in the cumulative distribution function for the distribution across the domain. Conceptually, this is the amount of probability mass that would have to be moved in one of the distributions to equalize the two distributions.

Metrics that use the distance measures above are applied to measure the axes:

- Covariate Balance
- Covariate Distribution Overlap [8]

3) **Relative Heterogeneity**: the heterogeneity of some variable A, relative to another variable B, can be measured using the ratio of the standard deviation of A to the standard deviation of B. Metrics that use this approach are applied to measure the axes:

- Treatment Effect Heterogeneity [9]

3.5 Conclusion

This chapter laid the theoretical groundwork necessary for the robust evaluation of causal inference estimators. Different estimators target different components of the causal mechanism and this produces sensitivity to different properties of the joint distribution over the observed data. The Causal Inference Problem Space is a space defined by ten axes along which the properties of the joint distribution vary. Understanding this space is important because different estimators are likely to perform differently on distributions which lie in different parts of it. With this foundation established, the next chapter goes on to review the methods used to evaluate causal inference methods in the literature.

[8] It is impossible to know whether the distribution of the empirical data in the treatment groups overlaps, so distance measures at the distributional level are used as a proxy

[9] The standard deviation in the treatment effect relative to the standard deviation in the untreated outcome measures can be used as a proxy for the heterogeneity in the treatment effect.

Part III

Literature Review

Chapter 4

Approaches for Benchmarking Causal Inference

This review explores and synthesizes existing approaches for the evaluation of causal inference estimators. I aim to achieve three primary goals. First, to justify the importance of finite sample evaluation based on Monte Carlo methods by exploring the landscape of alternatives. Second, to establish principles for the validity of finite sample evaluation. Finally, third, to analyze the validity of existing approaches in terms of these principles in order to motivate a better approach. This literature review is comprised of four sections that work toward these goals. First, I briefly establish definitions for statistical inference and estimators which perform this inference. Second, I explore the landscape of different approaches to evaluating the performance of estimators - comparing theoretical, asymptotic evaluation and Monte Carlo-based finite-sample evaluation. Third, I establish the general principles for the validity of finite sample evaluation. Finally, I apply these principles to analyze existing approaches to evaluating causal inference estimators. The first three sections present this author's framing of the literature with light reference to the underlying works. The final section is a close review of the existing methods of Monte Carlo based evaluation.

4.1 Statistical Inference and Estimators

All statistical inference can be characterized in the following simple terms, due to Pearl (2009): There is some true data generating process (DGP) characterized by a set of (random) variables, functions (possibly probability distributions) that relate these variables, and parameters that determine the exact form of the functions. The DGP gives rise to a joint distribution over the variables which make up the DGP. Any sample of observed data can then be thought of as a sample from this joint distribution. The goal of inference is to estimate the functional relationships and parameters of the original data-generating process using sample data from the joint probability distribution.

As summarized in Calder (1953), estimators operationalize inference by providing a mathematical mapping between a sample of data from the true joint distribution and an estimate of a target parameter - an *estimand* - that is (or is concretely related to) a parameter from the true DGP. Given that a sample is inherently stochastic, estimators give rise to a *sampling distribution* over the estimand rather than a single value. The quality of an estimator is defined by some quality metric that depends on this sampling distribution. Concretely defining the abstract notion of the 'quality' of an estimator is a non-trivial task, discussed below.

4.2 Evaluating the Performance of Statistical Estimators

Ultimately, the goal of evaluation is to determine the quality of an estimator. There is no single measure for quality. The metric used is sensitive to the eventual application of the estimator and the real-world, normative value associated with various kinds of error. Bishop (2006) Chapter 1 outlines most of the commonly encountered quality metrics and in which circumstances they tend to be most useful. The common thread which unifies these metrics is that they depend on the estimator's sampling distribution over the estimand. All metrics are, at some level of abstraction, calculated in reference to this sampling distribution (Calder, 1953). So, the estimator itself, the sample size available, and the underlying joint data distribution affect the value quality metrics by affecting the sampling distribution.

Given the central role of the sampling distribution, one can think about evaluation approaches in terms of how they derive the sampling distribution for a given estimator. In this framing, there are two approaches that I will refer to as *theoretical* and *experimental*. Theoretical methods derive an analytical expression for the sampling distribution based on the estimator, sample size, and underlying data distribution. Experimental approaches *estimate* the sampling distribution by repeatedly applying the estimator to samples from an underlying data distribution with a known value for the estimand. Repeat sampling results in convergence to the expected value of any metric defined over the sampling distribution (Paxton, Curran, Bollen, Kirby, & Chen, 2001). This provides *experimental* insight into the performance of the estimator in the sense that data is collected to estimate the quality of the metric under researcher-controlled settings of the sample size and underlying distribution. In general, the process of using samples from a distribution to estimate the value of parameters over that distribution is referred to as Monte Carlo estimation (Hastings, 1970). This is why sampling-based evaluation is referred to as Monte Carlo Evaluation in the literature.

These two approaches - theoretical and experimental - are not mutually exclusive and, for the evaluation of an arbitrary estimator, can both provide useful information. However, honing in on our specific context, Paxton et al. (2001) provide a compelling analysis of the specific usefulness of Monte Carlo evaluation in evaluating econometric estimators. They raise three concerns with the usefulness of theoretical evaluation which are outlined below along with corresponding points made by authors who reach the same conclusions with regard to evaluating causal inference estimators specifically.

1) In many cases, the theoretical sampling distribution for an estimator is not known. As pointed out in Knaus et al. (2018), more complex estimators tend to make for much more challenging theoretical analysis. This fact, combined with the trend toward semi/nonparametric causal estimators outlined in the introduction, means that theoretical sampling distributions are missing for many causal inference estimators which have proven promising in practice.

2) Theoretical evaluation tends to rely heavily on simplifying assumptions - the most common of which is the assumption of the normality of the underlying data distribution. These assumptions limit the applicability of the theoretical sampling distribution: it is unlikely that real-world data distributions meet these assumptions and thus the theoretical metrics may not hold on real data. Even if the assumptions behind the theoretical analysis of different methods are available, Knaus et al. (2018) point out that the assumptions behind the analysis for different estimators may not overlap such that comparing theoretically derived metric values is impossible.

3) Theoretical analysis tends to hold in the asymptotic limit of large sample sizes. The implications of theoretical results for finite samples is not necessarily well defined. This is echoed in Huber, Lechner, and Wunsch (2013).

Taken together, these three weaknesses mean that Monte Carlo analysis - which can be performed using arbitrary sample sizes and on any underlying data distribution - plays a crucial role in the evaluation of causal inference estimators. The rest of this review focuses exclusively on these methods of evaluation, exploring the principles which define a good Monte Carlo evaluation and how existing variants measure up against these principles.

4.3 Monte Carlo Evaluation of Causal Inference Estimators

4.3.1 General Principles

This section proposes a synthesized set of principles for what makes a *good* Monte Carlo-based evaluation method. Much like is the case for evaluating individual estimator quality, there is no universal metric for the quality of an evaluation method. Two thematic axes of quality are common in the literature. I refer to these axes as the *axis of specific validity* and the *axis of general validity.* I use the word axis to represent the non-discrete nature of validity. We will see that there appears to be a trade-off between specific and general validity and that different methodological choices will have different expected levels of validity on these axes. The levels of validity are hard to quantify but this should not be confused with the absence of a continuum.

4.3.1.1 The Axis of Specific Validity

Specific validity refers to whether a Monte Carlo evaluation provides a valid measure of an estimator's performance in some specific distributional setting. A distributional setting is defined by a given DGP (joint distribution of the data) and the sample size drawn from this DGP. On the surface, internal validity appears to hold (trivially) for any Monte Carlo evaluation. There is some joint distribution from which a fixed size sample is drawn, the estimator is applied to the sample and the estimand value is collected. If this is repeated enough times, the distribution over the estimand for the distributional setting can be approximated with arbitrary precision. The problem is revealed by the nuance that researchers do not target arbitrary DGPs. The power of Monte Carlo - as established above - is that it does not rely on simple joint DGPs/joint distributions and, thus, can be used to validate *realistic* DGPs (DGPs which could be found in the real world). This means that Monte Carlo evaluations are (and should be) designed with DGPs which aim to be representative of the real world. The quality with which the Monte Carlo DGP represents the *specific* real-world setting targetted is the source of *specific validity.* It is possible for a researcher to target a realistic setting but fail to use a DGP which is representative and, therefore, end up producing a result that is not valid in the *specific* setting. This is one of a set of related failure modes for specific validity which will be explored below. The others include biased specification and the creation of DGPs which do not ensure convergence to the true sampling distribution.

Concern for this validity is present in various papers that evaluate causal inference estimators using methods derived from Monte Carlo evaluation. Dorie et al. (2019) refer to the importance of the "calibration of the DGP to the real world" . They observe that if the types of covariates, their marginal and joint distribution, and the functional forms which relate them to the treatment and outcome variables are too simple, then the DGP will not be representative of the real world. Paxton et al. (2001) refer to the need for example-data-driven DGP design in order to avoid the creation of DGPs purpose-built to confirm the efficacy of an estimator despite having "little resemblance to [data and] models encountered in practice" . The common concern here is that a positive evaluation result from an unrealistic DGP, or a DGP which poorly represents some real DGP, will not correspond to evidence that the estimator works in a specific, real-world setting. This is where the idea of *specific validity* originates.

4.3.1.2 The Axis of General Validity

General validity refers to whether a Monte Carlo evaluation provides insight into whether an estimator generalizes well across different distributional settings. Dorie et al. (2019) observe that "while it is natural to make sure that a method works in the specific scenarios for which it is designed, this doesn't necessarily help a general researcher understand how it might perform more broadly" . The implicit premise in this observation is that it is uncommon for practitioners to know the DGP which underlies a given set of observational data. This means it is important to know how a given estimator will perform across a range of relevant distributional settings. The assumption being that an estimator that performs well (or better than other estimators) across a range of relevant distributional settings is the one which is most likely to be best when the true setting is unknown. While superior performance is not guaranteed - unless the estimator is tested against an exhaustive set of distributional settings - it does appear that performance on a range of settings is a sensible and useful heuristic to determine general robustness. This is where the idea of *general validity* originates.

4.3.1.3 The Trade-off between Specific and General Validity and the Optimal Design Curve

Both Paxton et al. (2001) and Wendling et al. (2018) point out that "there is a trade-off between realism and control in any Monte Carlo design" (Paxton et al., 2001). This implies a trade-off in specific and general validity. General validity requires evaluation of a representative set of well-defined distributional settings but the control required to specify the DGPs that comprize this well-defined set tends to result in less realistic, and therefore less *specifically valid*, evaluations. The mechanistic reasons for this are explored in the next section.

For now, I observe that there is no single answer to the right balance between general and specific validity. Researchers working on causal Inference methodology may care more about specific validity in one distributional setting in order to demonstrate that a new method achieves its design purpose. Researchers working on applied problems in social and political science may care more about general validity to ensure robust performance in unknown settings.

One could imagine a set of designs representing optimal trade-offs where, for any given design, the external validity is maximized relative to a minimum constraint placed on internal validity. This would trace out an abstract 'optimal frontier' curve in the design space - as in Figure 7A below. Any design along this curve could not improve its validity on either axis without sacrificing validity on the other. Designs on this curve would all be equally *good* in the sense that they may be useful in different circumstances. Unfortunately, as will be seen in the next section, the majority of the actual designs do not lie on this optimal curve and, with specific design improvements, one/both the specific and general validity could be improved without decreasing the validity of the other. The location of such a design - and a path for unambiguous improvement - is demonstrated in Figure 7B. The rest of this review is focused on locating existing approaches to Monte Carlo evaluation in this design space and exploring how they could be moved closer to the optimal frontier.

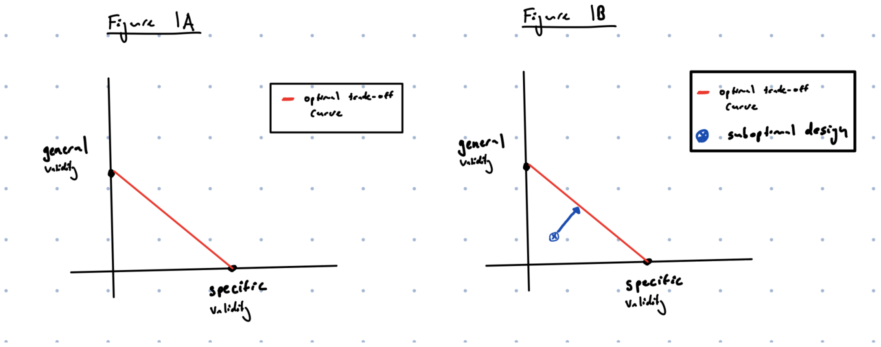


Figure 7: Left panel: A point which is not on the optimal trade off curve. Right panel: A point on the optimal trade off curve.

4.3.2 (Sub)Optimal Approaches

At this stage, it's clear that Monte Carlo based evaluation is a particularly useful method of assessing the performance of causal inference estimators. In the real world, practitioners deal with finite data samples from unknown but complex DGPs. Monte Carlo is capable of evaluating estimators in these settings while theoretical analysis is hard or impossible. It is also clear that Monte Carlo evaluation can vary in validity along two axes - specific and general validity - and that different designs may make (sub)optimal trade-offs between validity on these axes. The analysis which established these two points applied to Monte Carlo evaluation in an abstract sense. This section reviews concrete implementations of Monte Carlo evaluation of causal inference estimators from the literature, analyzing them in light of the axes of validity introduced above. This move toward analyzing concrete implementations brings with it vocabulary which is specific to causal inference. I refer the reader to Chapter 2 for clarification of any unfamiliar terms.

The analysis in this section divides Monte Carlo evaluation methods into two categories based on the underlying design of their DGP: *pure* and *hybrid*. *Pure* methods are further subdivided into purely *synthetic* and *empirical* DGP designs. *Hybrid* designs are hybrids because they combine the two pure designs strategies in a single DGP. This is a 'novel' ontology introduced to summarize the existing variants of Monte Carlo. All of the designs share the same fundamental structure - in the sense that the same variables must be sampled so that a causal inference estimator can be evaluated - but exhibit different validity characteristics when judged through the lens of specific and general validity. I proceed by, first, establishing a generic DGP - an abstract representation of the DGP used in all Monte Carlo evaluation methods. I then express the various designs found in the literature in terms of different strategies for the concretizations of this generic DGP, making comparative analysis straightforward. Finally, I analyze the implication of different concretization strategies in terms of the axes of validity from above.

4.3.2.1 A Generic DGP for the Monte Carlo Evaluation of Causal Inference Estimators

This sections builds towards a formal specification scheme for the DGPs which underpin Monte Carlo evaluation of causal inference estimators. The scheme is defined in terms of a generic DGP, the functional components of which can then be defined/instantiated to describe a specific, concrete DGP design.

First, Table 1 below recapitulates the variables over which a causal inference problem-instance is defined and provides the associated dimensionality and definition. Full explanations for these variables are presented in Chapter 2 above.

$X \in R^{(n,d)}$	X is the covariate matrix that contains measurements of d covariates for a population of n individuals
$T \in \{0,1\}^n$	T is the treatment status of the n individuals for a binary treatment
$Y_0 \in R^n$	Y_0 is the outcome under control (absence of treatment) referred to as the untreated outcome
$Y_1 \in R^n$	Y_1 is the outcome under treatment referred to as the treated outcome
$TE \in R^n$	TE is the individual treatment effect
$Y \in R^n$	Y is the observed outcome

Table 1: Variables which define a causal inference problem instance

The variables in Table 1 are related through four generic functions given in Table 2 [1]. A concrete Monte Carlo evaluation design can, thus, be fully specified by the concrete instantiation of these four functions.

[1] Note that the functions specified here may be stochastic functions, functions which do not necessarily produce the same output for the same input on every evaluation.

$\rho : () \rightarrow R^{(n,d)}$	The **covariate population sampler** which represents the joint distribution over the d covariates and produces samples of the covariate matrix X. Strictly speaking, X contains sampled covariate observations for n individuals such that a 'sampled' X is constructed by repeatedly sampling the underlying joint distribution over the covariates
$\Omega : \ R^{(n,d)} \ \rightarrow R^n$	The **treatment assignment sampler** which assigns a treatment status T to the individuals in X
$\Phi : \ R^{(n,d)} \ \rightarrow R^n$	The **untreated outcome sampler** which assigns an outcome under control (Y_0) to the individuals in X
$\tau : \ R^{(n,d)} \ \rightarrow R^n$	The **treatment effect sampler** which assigns a treatment effect (τ) to the individuals in X

Table 2: Functions which generate and relate the variables in a causal inference problem instance

A complete (but generic) DGP is then captured by the procedure presented in Table 3. This procedure relates the variables defined in Table 1 using the functions defined in Table 2.

Generic DGP Procedure
$X \leftarrow \rho\,()$
$T \leftarrow \Omega\,(X)$
$Y_0 \leftarrow \Phi\,(X)$
$TE \leftarrow \tau\,(X)$
$Y_1 \ \leftarrow Y_0 + TE$
$Y = T \times Y_1 + (1 - T) \times Y_0$

Table 3: The procedure for a Monte Carlo evaluation DGP defined in terms of the variables from Table 1 and functions from Table 2

Note three facts about this generic DGP:

- The designer has a maximum of four degrees of freedom in specifying the DGP. These degrees of freedom correspond to selecting the four functions ρ, Ω, Φ, τ .

- This generic DGP provides all of the information required to evaluate a causal inference estimator. The estimator receives X, T, and Y and produces an estimate of either $E\,[TE|X]$ - the conditional treatment effect - or $E\,[TE]$, the average treatment effect. The ground-truth (sampled) values of TE are then be used to evaluate the estimates.

- This DGP is constructed to ensure ignorability. Following Dorie et al (2019): Assume Φ, Ω and τ are probability distributions, then we have the following factorization as a result of the DGP form above (and specifically as a result of the absence of T as an argument to Φ, τ): $p\,(Y_0,\ Y_1, T|X) = p\,(Y_0,\ Y_1|X) \times$

$p\left(T|X\right)$ which implies $p\left(Y_0,\ Y_1|T,X\right)=p\left(Y_0,\ Y_1|\ X\right)$ or $Y_0,\ Y_1 \perp\!\!\!\perp T|X$. Non-ignorability can be induced by dropping some of the d covariates in X but, by default, it is guaranteed.

Taken together, these three facts mean that specifying the four functions is sufficient to generate a causal inference problem-instance. In fact, the minimal level of constraints on the four degrees of freedom means that this generic DGP can be used to specify a concrete DGP design corresponding to any joint distribution which factors in the way outlined above.

Having established this generic DGP, I turn to the idea of concretization strategies. Different strategies for specifying the four base functions - thus concretizing the generic DGP - give rise to the different design categories which were outlined above. There are three mutually-exclusive and exhaustive strategies for concretely specifying a function:

1) A function can be manually specified by constructing a mathematical (distribution) function from elementary mathematical building blocks and specifying all the parameters which appear in its definition. This produces functions with known form and parameter values. I will label functions specified in this way as $f_{synthetic}$.

2) A function can be estimated from empirical data by specifying a fixed functional form and inferring the unknown parameter values. This produces functions with a known form but unknown ground-truth parameters. Functions produced by this strategy will be labeled $f_{estimated}$.

3) A function can be ‘specified’ to exactly match empirical data. In this case, the mathematical form of the function and any associated parameter values are unknown. This strategy produces functions labeled $f_{empirical}$.

The latter two strategies necessitate access to empirical data from some target DGP. Variables that are sourced empirically for this purpose will be labeled as $A_{empirical}$ where $A\ \in\{X,T,Y\}$ (the set of observable variables).

With the generic DGP and the three concretization strategies in mind, I proceed to review and categorize the designs found in the literature.

4.3.2.2 Pure Designs for Monte Carlo Evaluation

Pure designs are pure because they rely exclusively on sampling functions of the type $f_{synthetic}$ - in the case of synthetic designs - or $f_{empirical}$ - in the case of empirical designs. Below, I outline each of the resultant design approaches and how they tend to appear in the literature.

Synthetic Monte Carlo Designs Synthetic designs rely on hand crafted functions to specify the four degrees of freedom which control the distribution over covariates, treatment assignment, untreated outcome and treatment effect. These designs are common in literature which introduces new estimators designed to tackle specific distributional settings as the manual specification allows careful control over the resultant joint distribution. For recent examples, see Fredrik D. Johansson et al. (2018), Fredrik D Johansson et al. (2016), Künzel et al. (2018). Designs which fit into this class are used extensively in the literature beyond this small sample of recent papers. Kang and Schafer (2007b) introduce a benchmark based on four independent, normally distributed covariates, a linear outcome with Gaussian noise, a constant treatment effect and a linear expit for the propensity score. Imbalance is created by variable censoring of individuals based on assignment treatment. This benchmark is cited in over eight hundred works at the time of writing, the majority of which use the proposed DGP for evaluating causal inference estimators or use results from analysis of this kind by others. Kallus (2016) introduces a similar benchmark which attempts to more accurately model non-linearities present in real data. The benchmark presented is that work is based on two covariates drawn from a uniform distribution, a non-linear propensity score expit and normally distributed outcomes with means derived from the covariates. This benchmark is cited in 10 other works, mostly those applying semi/nonparametric estimators. In both cases, the stated design principle is to create a DGP that is representative of the kinds of data found in practice.

The top left panel of Table 4 below represents the synthetic design using the function labels and generic DGP defined above. All four degrees of freedom are specified to be synthetic functions and the consequence is a DGP devoid of empirical grounding. The bottom left panel expresses the Kang and Schafer (2007a) DGP design as an assignment of the four base functions.

Empirical Monte Carlo Designs In contrast to synthetic designs, empirical designs specify the four functional degrees of freedom using real-world data. The challenge is how to do this in a way that still provides access to the treatment effect which is required to evaluate an estimator. Empirical designs overcome this challenge by using data randomized control trials, replacing the control group with non-randomized survey/administrative observations over the same covariates that were observed in the experiment (Huber et al., 2013). The average effect from the randomized control trial provides the 'ground truth' target for the average effect estimand. Examples of this design for evaluation include (R. H. Dehejia & Wahba, 1999; Flores & Mitnik, 2009; Fraker & Maynard, 1987; Friedlander & Robins, 1995; Heckman & Todd, 1998; Lalonde, 1986; Smith & Todd, 2005).

There are three things about this design worth noting. First, the empirical specification of the functional degrees of freedom means the true distributional setting is not known. Second, and related, the ground truth is itself an estimate rather than a known value. This produces additional uncertainty in the results which come from this evaluation method. Finally, this design can, by default, on evaluate estimators for the average treatment effect. The conditional average treatment effect must be estimated from the data using the very methods which we would like to evaluate using a Monte Carlo benchmark. This means the evaluation of conditional average treatment estimators is impossible under empirical designs of the kind described here.

The top right panel of Table 4 below represents the empirical design using the function labels and generic DGP defined above. All four degrees of freedom are specified to empirical functions (although some of these are unknown). The consequence is a DGP grounded in empirical data but with an unknown joint distribution over X, T, and Y and an unknown values for the true treatment effect and potential outcomes. The bottom left panel expresses the Lalonde (1986) DGP design as an assignment of the four base functions.

There is an obvious concern worth addressing in calling this a Monte Carlo design. The component functions are specified by data rather than sampling functions which means that the DGP cannot be used to generate multiple samples from the joint distribution over the variables. As a result, this method does not provide access to a sampling distribution for the estimand but rather only a single estimate. I believe the Monte Carlo classification is justified for two reasons despite this concern. Firstly, we can conceive of the empirical design as a single sample from some latent DGP which could - in theory - be used produce more samples. This design might be weakened by the small (single) sample size but this doesn't change the fundamental principle behind its operation. Second, the analysis below will introduce hybrid designs which share common components with the pure empirical design but do allow sampling. In this sense, the pure empirical design can be thought of the extreme (and somewhat degenerate) end of a Monte Carlo design spectrum.

	Synthetic Monte Carlo	**Empirical Monte Carlo**
Concretized DGP	$\rho_{synthetic} \leftarrow f_{synthetic}$ $\Omega_{synthetic} \leftarrow f_{synthetic}$ $\Phi_{synthetic} \leftarrow f_{synthetic}$ $\tau_{synthetic} \leftarrow f_{synthetic}$ $X \leftarrow \rho_{synthetic}()$ $T \leftarrow \Omega_{synthetic}(X)$ $Y_0 \leftarrow \Phi_{synthetic}(X)$ $TE \leftarrow \tau_{synthetic}(X)$ $Y_1 \ \leftarrow Y_0 + TE$ $Y = T \times Y_1 + (1-T) \times Y_0$	$\rho_{empirical} \leftarrow f_{emp} = X_{empirical}$ $\Omega_{empirical} \leftarrow f_{emp} = T_{empirical}$ $\Phi_{empirical} \leftarrow f_{emp} =?$ $\tau_{empirical} \leftarrow f_{emp} =?$ $X \leftarrow X_{empirical}$ $T \leftarrow T_{empirical}$ $Y_0 \leftarrow ?$ $TE \leftarrow ?$ $Y_1 \ \leftarrow ?$ $Y = Y_{empirical}$
Example Concretization	Kang and Schafer (2007) $\rho \leftarrow N(X \mid 0,1)$ $\Omega \ \leftarrow expit(-X_1 + 0.5X_2$ $... - 0.25X_3 - 0.1X_4)$ $\Phi \leftarrow N(Y \mid 210 + 27.4X_1$ $... + 13.7(X_2 + X_3 + X_4), 1)$ $\tau \leftarrow \ C \ \in R$	Lalonde (1986) $\rho \leftarrow \{X_{experiment}, \ X_{survey}\} \sim ?$ $\Omega \leftarrow \ T_{empirical} \ \sim ?$ $\Phi \leftarrow ?$ $\tau \leftarrow ?$
Joint Distribution over Observables	$P(X,Y,T) \sim \rho \times \Omega \times \Phi$	$P(X,Y,T) \sim ?$

Table 4: Synthetic vs Empirical Monte Carlo Designs expressed as concretized versions of the generic DGP along with examples framed in terms of assignments to the four functional degrees of freedom.

Comparing the Designs The framing proposed above clarifies how the theoretical trade-off between specific and general validity manifests in the pure design strategies. Synthetic designs provide granular control of a well-defined distributional setting because all of the component degrees of freedom are set by hand with functions of known form and parameterization. This enables easy testing of estimators across a range of settings, promoting general validity. But, the hand-crafted nature of the function specification raises concerns about the correspondence of the DGP to reality, thereby weakening specific validity. Empirical designs are realistic and correspond to a valid evaluation (with some caveats) of a specific distributional setting (specific validity) but do not allow for easy control of the distributional setting, weakening general validity.

This comparison between the designs, grounded in concretization strategies, provides some insight but is still too abstract. It is unclear what is meant by the ideas of realism and control which appear throughout the paragraph above. Below, I synthesize the work in Paxton et al. (2001), Huber et al. (2013) and Dorie et al. (2019) to compare the properties of synthetic and empirical designs that contribute to specific and general validity. Many of the points which appear below are implicit or explicit in the analysis above but this complete enumeration will prove useful later when considering hybrid designs which mix the properties of the pure designs.

Synthetic designs tend to have weak specific validity for three reasons which stem from the manual specification of functions:

- Unrealistic covariate distributions: the number, type (continuous, categorical, binary) and joint distribution (over the covariates) is often poorly calibrated to the data researchers encounter in practice. It is unlikely that covariates in the real-world are drawn from a multivariate normal distribution.

- Unrealistic treatment and outcome functions: the complexity of the functional forms - in terms of interactions, non-linearity etc - is often poorly calibrated to mechanisms likely to be found in the real world. It is unlikely that the treatment policy in the real world is a logistic function with a linear expit.

- The potential for specification bias: because covariates and treatment/outcome functions are selected by hand there is the potential for intentional or unintentional bias in the design such that the DGP favors the method being evaluated. This implies DGPs with similar properties would or could lead to worse results. Consider that many researchers may change the DGP specification until their method produces the results they *expect* to see from it under the assumption that their DGP specification, and not their method, is at fault for unexpected results. This would induce the kind of bias described here.

However, synthetic designs have three characteristics which counter-balance those above when considering specific validity:

- The DGP can be guaranteed to meet the assumptions required for observational causal inference, specifically ignorability and overlap. This avoids false negatives in which an otherwise valid estimator produces poor results as a result of violated assumptions rather than inherent flaws.

- The DGP can be used to produce many samples and, therefore, to ensure convergence to a true sampling distributions for the estimand. Further, this provides access to a distribution over performance which is useful for quantifying best and worst case performance.

- The DGP provides access to the ground truth of the individual treatment effect and average treatment effect. This precludes the need for additional estimation of the ground truth and means that a wider range of estimators can be evaluated.

Moving on to consider general validity, synthetic designs have two properties which make for strong general validity:

- The joint distribution over covariates, treatment and outcomes is well-defined and known. This means the performance of the estimator is relative to a known distributional setting.

- The distributional setting is controllable such that the estimator can be tested over a range of settings.

Empirical designs are effectively polar opposite to synthetic designs when examined in terms of the eight properties above. In order of the points above: They possess inherently realistic covariate distributions and outcome/treatment functions and, by extension, unbiased specification (given that the researcher has no control). However, they do not guarantee that causal inference assumptions are met, do not allow repeat sampling and do not provide access to the ground truth of the average or individual treatment effect without estimation. Further, they use DGPs with an unknown distributional setting and do not allow for researcher control.

Table 5 provides a summary of the eight properties outlined above, which of the two validity axes they affect, the functional degrees of freedom from the generic DGP that affect the possession of the property, and whether each of the two pure designs possesses the (positive) property.

Property	**Validity Axis**	**Relevant functional degrees of freedom**	**Synthetic Designs**	**Empirical Designs**
Realistic Covariate Distribution	Specific	ρ	🟥	🟩
Realistic Treatment/Outcome Functions	Specific	$\Omega,\ \Phi, \tau$	🟥	🟩
Guaranteed Unbiased Specification	Specific	$\rho,\ \Omega,\ \Phi, \tau$	🟥	🟩
Access to Ground Truth	Specific	Φ, τ	🟩	🟥
Guaranteed to Obey Causal Assumptions	Specific	$\Omega,\ \Phi, \tau$	🟩	🟥
Allows Repeat Sampling	Specific	$\rho,\ \Omega,\ \Phi, \tau$	🟩	🟥
Known Distributional Setting	General	$\rho,\ \Omega,\ \Phi, \tau$	🟩	🟥
Controllable Distributional Setting	General	$\rho,\ \Omega,\ \Phi, \tau$	🟩	🟥

Table 5: The eight properties of Monte Carlo designs which affect specific and general validity of evaluation along with the functional degrees of freedom which impact this property

The analysis above establishes a more granular understanding of the properties of the two pure designs which produce the differences in their specific and general validity. It is clear that these two designs are polar opposites but this does not necessarily mean that they are suboptimal in terms of the trade-off between specific and general validity. If one could not do better in one without sacrificing the other, then these two designs would represent a (discrete) optimal frontier with researchers being forced to choose one or the other depending on their needs. Fortunately, this is not the case.

4.3.2.3 Hybrid Designs for Monte Carlo Evaluation

Hybrid designs combine the concretization strategies from synthetic and empirical designs to produce a Monte Carlo evaluation method with more of the desirable properties enumerated above. The unifying principle behind hybrid designs is the maximal use of data to inform specification of the DGP - to improve realism and specific validity - while maintaining the control required to allow general validity.

The earliest examples of thought in the direction of blending synthetic and empirical designs appear in Abadie and Imbens (2002) and Diamond et al. (2012). Both papers use DGPs designed to mimic the data in Lalonde (1986) but both use synthetic, simplified covariate sampling and hand-crafted treatment/outcome functions which are 'inspired' by the data but not directly informed by it.

Huber et al. (2013) formalize this idea and introduce what they call *Empirical Monte Carlo* (EMC). The context for the introduction of their method is instructive. Their paper attempts to resolve a disagreement between Frölich (2004) and Busso, Dinardo, and Mccrary (2014), both of whom evaluate the same set of propensity-score based estimators using synthetic Monte Carlo but find quite different results. Huber et al. (2013) points out that both papers use DGPS with poor specific validity as a result of arbitrary synthetic design. This inspires the design of a method which uses a "DGP not entirely [built] on relations specified by the researcher, but [one which] exploits real data as much as possible instead, [using] observed outcomes and covariates instead of simulated ones as well as an observed selection process" . The authors propose the following design: They define a population of observations as a dataset covering all German nationals with social insurance in the years between 1990 and 2006 (millions of observations). The treatment is participation in any employment assistance program during the observed period and the outcome is the employment status in the 36 months following participation. EMC then involves four steps:

1) A propensity score model is estimated based on the entire population using a logistic regression with a linear expit.
2) A random sample of the non-treated individuals is drawn from the population. This means the observed outcome - with an unknown functional relation to the covariates - serves as the value Y_0 .
3) Each individual is assigned a treatment status based on a draw from a Bernoulli distribution parameterized by the propensity score for the individual under the model from step 1.
4) Each individual is subjected to a 'placebo' treatment to construct Y_1. This is treatment with a constant, zero treatment effect.
5) Steps 2 through 4 are repeated for multiple samples.

Table 6 expresses this design using the generic DGP and functional building blocks from above.

Empirical Monte Carlo - Huber et al (2013)

$\rho \leftarrow f_{empirical}$
$\Omega \leftarrow f_{estimate}$
$\Phi \leftarrow f_{empirical}$
$\tau \leftarrow f_{synthetic}$
$X \leftarrow X_{empirical}$
$T \leftarrow \text{LogisticFit}\left(T_{empirical} \sim X_{empirical}\right)$
$Y_0 \leftarrow Y_{empirical}$
$TE \leftarrow 0$
$Y_1 \ \leftarrow Y_0 + TE$
$Y = T \times Y_1 + (1 - T) \times Y_0$

Table 6: Concretization of the generic DGP under the design of Huber, Lechner, and Wunsch (2013)

Analyzing the assignment of the functional degrees of freedom from Table 6 in light of Table 5 makes analyzing the validity properties of this design relatively straightforward. These are listed below with a summary given in Table 9. It is clear from these properties that hybrid designs are promising - the use of data provides realism while the simulated components of the treatment and outcome provide access to ground truth and ensure assumptions are obeyed. However, as is evident from the analysis below, the EMC design provides only partial conformance to some of the desirable properties of validity.

- **Realistic Covariate Distribution - Yes:** The use of observed covariates means the covariate distribution is realistic (in the specific domain of labour data).

- **Realistic Treatment/Outcome Functions - Partial:** The use of the observed outcome mechanism means this too is realistic. However, the treatment assignment and treatment effect are both hand crafted and too simplistic.

- **Guaranteed unbiased specification - No:** The hand selection of the two functions above means there is potential for researcher bias (which holds even if more complex functions are chosen).

- **Access to ground truth - Yes:** the simulated treatment means the ground truth is known at the individual and average effect level.

- **Guaranteed to meet causal assumptions - Yes:** the treatment assignment mechanism is known to include only the observed covariates so we have selection on observables and therefore ignorability. The authors also include an offset in the treatment assignment expit to ensure that the probability of treatment is bounded away from 0 and 1, meaning there is overlap. So both important causal assumptions are met.

- **Allows repeat sampling - Yes:** the large population of observations allows repeat samping as in a synthetic Monte Carlo design:

- **Known distributional setting - Weak:** the joint distribution over the covariates and the outcome is unknown. The treatment assignment mechanism is known.

- **Controllable distributional setting - Weak:** the treatment assignment mechanism can be altered but the joint distribution over the covariates and the outcome mechanism is fixed by the data.

Knaus et al. (2018) build on EMC. They use a very similar dataset as Huber et al. (2013) - but covering Swiss administrative data on labour and employment rather than German. They also share the same logistic estimation for the treatment assignment. But, unlike Huber et al, they specify a non-zero, non-constant treatment effect mechanism. This is specified synthetically rather than estimated because "estimation may favor [functionally] similar estimators under investigation" . It is unclear why the authors are concerned about this problem in outcome mechanism but are ok with fixed functional form estimation in the treatment assignment mechanism. This paper is designed to test methods which infer individual treatment effects, so the authors specify the treatment effect mechanism with the goal of "making disambiguating selection and treatment effect heterogeneity hard" . To this end, they include the propensity score in the synthetic treatment effect, apply a highly non-linear transform, and add random noise. The treatment effect from this function is added to the observed outcome (much like in EMC, only non-treated individuals are selected into samples so the observed outcome equals the untreated potential outcome Y_0). In addition to the synthetic treatment effect, the authors parameterize the population sampler, treatment assignment mechanism and treatment effect function to generate 24 datasets with different sample sizes, treatment effect magnitude, treatment effect heterogeneity, and counts of treated vs control individuals.

Knaus et al (2018)

$\rho \leftarrow f_{empirical}$

$\Omega \leftarrow f_{estimate}$

$\Phi \leftarrow f_{empirical}$

$\tau \leftarrow f_{synthetic}$

$X \leftarrow X_{empirical}$

$T \leftarrow \text{LogisticFit}\left(T_{empirical} \sim X_{empirical}\right)$

$Y_0 \leftarrow Y_{empirical}$

$TE \leftarrow f_{synthetic}$

$Y_1 \ \leftarrow Y_0 + TE$

$Y = T \times Y_1 + (1 - T) \times Y_0$

Table 7: Concretization of the generic DGP under the design of Knaus et al. (2018)

The concretized DGP is given in Table 7. The makes the similarity with Huber et al obvious. But the changes do represent a moderate improvement over EMC when examining the validity properties. The outcome mechanism is arguably more realistic as a result of the complex synthetic treatment effect function combined with the empirical control outcome. But, the synthetic component of the specification opens this design to the same criticisms of realism and potential bias as any other synthetic design.

The primary contribution of these authors is the parameterization of the functions to generate datasets with different, well-defined distributional settings. This allows the evaluation of estimators over a partial subspace of all possible distributional settings. These changes are reflected in the 5th column of Table 9 which upgrades the known and controllable distributional setting properties from weak under EMC to partial under Knaus et al.

The two approaches analyzed above use an empirical covariate population and empirical outcome with an estimated treatment assignment mechanism. There is a loosely analogous set of designs which use an empirical treatment and specify a synthetic outcome mechanism. Hill (2011) develops a method to test their new causal estimator based on non-parametric Bayesian regression. The proposed method uses experimental data from the Infant Health and Development Program (IHDP) - a "randomized experiment which targeted low birth-weight, premature infants and provided the treatment group with high-quality individual care" (Brooks-Gunn, Klebanov, & Liaw, 1991). Per Hill (2011), using an experimental dataset guarantees overlap (and balance) in the covariate distributions and simulating the outcome using only the observed covariates ensures ignorability. The outcome mechanism (the outcome function Φ and treatment effect function τ) are specified by hand - the author chooses two conditions, one linear outcome and constant treatment effect and one non-linear outcome with heterogenous treatment effect. Starting from the position of balanced covariate distribution allows the author to induce imbalance by non-randomly censoring individuals from the treat/control group. This is the first instance of an intentional specification of the level of balance.

The concretization of the generic DGP proposed by Hill (2011) is summarized in Tbale 8. Ultimately, as a result of the parallel between simulating outcome and treatment, this method has very similar validity properties to those analyzed above. Unlike the methods in Huber et al. (2013) and Knaus et al. (2018), the simulated (untreated) outcome provides greater control over the distribution of outcomes and the experimental datasets allows for the easy control of balance (although a similar censoring method could be applied to non-experimental data). This means the method has arguably better definition and control of the distributional setting but this control is still worse than under purely synthetic treatment assignment and outcome mechanisms.

Hill (2011)

$\rho \leftarrow f_{empirical}$

$\Omega \leftarrow f_{empirical}$

$\Phi \leftarrow f_{synthetic}$

$\tau \leftarrow f_{synthetic}$

$X \leftarrow X_{empirical}$

$T \leftarrow T_{empirical}$

$Y_0 \leftarrow f_{synthetic}$

$TE \leftarrow f_{synthetic}$

$Y_1 \ \leftarrow Y_0 + TE$

$Y = T \times Y_1 + (1 - T) \times Y_0$

Table 8: Concretization of the generic DGP under the design of Hill (2011)

Two papers iterate on the design proposed by Hill (2011) but change the specification of the outcome mechanism (the outcome and treatment effect functions) in order to improve the realism of the synthetic outcomes and reduce the potential for bias. Wendling et al. (2018) uses four experimental datasets from the medical field but, instead of manually specifying the outcome and treatment effect functions, estimates these using neural networks. This technically avoids the need to specify a functional form because neural networks can represent an arbitrary set of functional forms. This is a move back toward the estimation used in EMC with $\Phi \leftarrow f_{estimated}$ and $\tau \leftarrow f_{estimated}$. While this may more accurately model the true outcome function, the estimation method could still favor specific estimators - for example those based on neural networks which will more naturally recover the same estimated function from the same data. It also gives up on control over the outcome aspect of the distributional setting because it is not clear what level of interaction/non-linearity is represented by the estimated functions nor how to change these functions to change the level of these aspects of the distribution. The first point results in improved realism of the outcome/treatment but no change to the Guaranteed Unbiased Specification property. The second point results in worsened scores on Known and Controllable distributional setting - see Table 9.

Kern et al. (2016) pursue a different approach - they specify the outcome and treatment effect functions manually but use a “saturated parametric model” with some desired level of nonlinearity and interaction. This means that, for example, all covariates may appear in both linear and quadratic form as well as in every possible (pairwise) interaction with lower/higher powers and interaction levels possible. The motivation for this is to create a principled approach for building complex functions with desired properties but without allowing the function to be hand crafted with arbitrary form and/or modified to produce desired results. This moves towards a design that allows for control over the outcome mechanism while also partially mitigating specification bias. Note that the the full saturation is itself fairly arbitrary (although it is closed to respecification in order to achieve desired results). On balance, the result of this change is a design which appears to have some degree of positive across-the-board ‘performance’ when assessed on the properties of validity - see Table 9. This is not a claim to formal superiority but rather an indication of a promising design direction.

	Pure Designs		**Hybrid Designs**				
Property	**Synthetic Designs**	**Empirical Designs**	**Huber et al (2013)**	**Knaus et al (2018)**	**Hill (2011)**	**Wendling (2019)**	**Kern et al (2016)**
Realistic Covariate Distribution	No	Yes	Yes	Yes	Yes	Yes	Yes
Realistic Treatment/Outcome Functions	No	Yes	Partial	Partial	Partial	Partial [2]	Partial
Guaranteed Unbiased Specification	No	Yes	No	No	No	No	Partial
Access to Ground Truth	Yes	No	Yes	Yes	Yes	Yes	Yes
Guaranteed to Obey Causal Assumptions	Yes	No	Yes	Yes	Yes	Yes	Yes
Allows Repeat Sampling	Yes	No	Yes	Yes	Yes	Yes	Yes
Known Distributional Setting	Yes	No	Weak	Partial	Partial	Weak	Partial
Controllable Distributional Setting	Yes	No	Weak	Partial	Partial	Weak	Partial

Table 9: Comparison of validity properties across hybrid designs in the literature.

The methods explored up until this point share a few key properties. They all use empirical covariate data from which samples are drawn. In all cases, either the treatment assignment or outcome mechanism is simulated to provide access to ground truth and allow for control over the distributional setting. Some methods, like EMC, trade off control for realism but newer approaches propose designs which allow for some degree of realism and control - either by approximation with non-parametric functions (Wendling et al., 2018) or by the principled specification of functions from simpler components (Kern et al., 2016). The ideal method would improve on the methods above by improving the realism of the treatment assignment/outcome mechanism while also increasing control over the distributional setting.

Dorie et al. (2019) propose a method which moves toward this ideal, building on the

[2] The semi-parametric neural network estimation of outcome function and empirical treatment assignment makes this approach arguably more realistic, to the extent that one believes that semi-parametric methods are an a priori better method of function estimation.

ideas of Hill (2011) by simulating functions on top of experimental data and Kern et al. (2016) by providing a principled way to construct functions while maintaining control over desired properties. The authors explicitly set out to create a "testing ground" to help applied researchers choose between different causal estimators - a goal which is shared by this paper. There are two notable design contributions in this paper. First, the authors motivate for the simulation of both the outcome and treatment assignment mechanisms. This allows for control over the full range of distribution settings including "balance, overlap, and nonlinearity" . Second, the authors mitigate the (potential) decrease in realism by employing a meta-sampling approach: The functions for untreated outcome, treatment effect and treatment assignment are sampled randomly from the space of "generalized additive functions" by (stochastically) selecting terms from a set of covariate transform building blocks - polynomial powers, discontinuous-valued 'steps', discontinuous-gradient 'kinks' - and combining these through the (stochastic) use of the operators addition, exponentiation, and multiplication. By parameterizing the distribution over the base terms and combination operators, the (distribution over) the functional properties of the sampled functions can be modified. This provides a principled way to construct functions with desired properties - as in Kern et al. (2016) - but allows testing over a large number of different functions which are more likely to capture performance on realistic DGPs by, one, sampling from space large enough to contain realistic DGPs and, two, averaging results over many samples from this space to maximize accuracy and remove idiosyncrasies. Combined, this makes for strong realism. Further, the authors parameterize the function selection mechanism to allow control over six "knobs" which cover most of the evaluation space from the previous section. The level of each knob is measured through the use of well-defined metrics. This allows the relatively precise specification of a distributional setting of a simulation in terms of targeted metric values and a corresponding sample of functions. This allows for close control over the distributional setting without giving method evaluators control over the actual functional forms.

The six knobs are:

1) The degree of nonlinearity in treatment assignment/outcome
2) The treatment effect heterogeneity
3) The treatment effect magnitude
4) The percentage treated
5) The degree of treat/control group covariate overlap
6) The degree of alignment between treatment and control mechanism

The design of the evaluation method is presented in Table 10 below. Note that there are effectively two Monte Carlo processes happening in this design. The first is a sample of DGPs from the space of DGPs defined by the function sampling parameters. The second is a sample of the data defined by the sampled DGP. The authors do not run repeat sampling at the data level, instead conflating both sampling processes into a single sample run repeatedly. As pointed out by the authors, this should have no effect on the convergence of the results to the true sampling distribution of the targeted estimand.

Dorie et al (2019)
$\rho \leftarrow f_{empirical}$
$\Omega \leftarrow f_{synthetic} \ \sim P\,(...)$
$\Phi \leftarrow f_{synthetic} \ \sim P\,(...)$
$\tau \leftarrow f_{synthetic} \ \sim P\,(...)$
$X \leftarrow X_{empirical}$
$T \leftarrow f_{synthetic}$
$Y_0 \leftarrow f_{synthetic}$
$TE \leftarrow f_{synthetic}$
$Y_1 \ \leftarrow Y_0 + TE$
$Y = T \times Y_1 + (1 - T) \times Y_0$

Table 10: Concretization of the generic DGP under the design of Dorie, Hill, Shalit, Scott, and Cervone (2019)

Based on the properties above, it appears that the design proposed by Dorie et al. (2019) has all of the desired validity properties - although the degree of validity may be contingent on design details like the size of the function space. Table 11 compares the validity properties of the three methods based on experimental data and principled function construction.

	Pure Designs		**Hybrid Designs**		
Property	**Synthetic Designs**	**Empirical Designs**	**Hill (2011)**	**Kern et al (2016)**	**Dorie et al (2019)**
Realistic Covariate Distribution	No	Yes	Yes	Yes	Yes
Realistic Treatment/Outcome Functions	No	Yes	Partial	Partial	Yes [3]
Guaranteed Unbiased Specification	No	Yes	No	Partial	Yes
Access to Ground Truth	Yes	No	Yes	Yes	Yes
Guaranteed to Obey Causal Assumptions	Yes	No	Yes	Yes	Yes
Allows Repeat Sampling	Yes	No	Yes	Yes	Yes
Known Distributional Setting	Yes	No	Partial	Partial	Yes
Controllable Distributional Setting	Yes	No	Partial	Partial	Yes

Table 11: Comparison of validity properties across designs based on experimental data and principled function construction.

4.4 Conclusion

Hybrid Monte Carlo evaluation designs - which combine empirical data and simulated functional forms - provide the foundation for evaluation methods with both specific and general validity. Not all hybrid designs are optimal. Many of the proposed designs - like those by Hill (2011), Huber et al. (2013), Kern et al. (2016), Knaus et al. (2018), Wendling et al. (2018) - do allow for some degree of specific and general validity but are still subject to criticism on both axes. Dorie et al. (2019) provide a sampling-based approach which appears to establish a near optimal middle ground in the design space by allowing careful control over the distributional while using leveraging the Monte Carlo sampling of designs to improve realism and mitigate bias.

[3] Provided the space of functions which can be sampled is large enough to contain realistic functions and enough samples are taken to allow for effective averaging.

Part IV

Introducing Maccabee: a Tool for Hybrid Monte Carlo Benchmarking

Chapter 5

Maccabee's Benchmarking Design

The goal of this chapter is to provide a self-standing description of the benchmarking approach implemented in the Maccabee package. This is presented in three sections. The first section provides a brief summary of the methods reviewed in Chapter 4. This sets the context for the design proposed in this chapter. The second section outlines Maccabee's benchmarking approach, which is based on the design proposed by Dorie et al. (2019). First, looking at the high level approach used by Dorie et al. (2019) and then at how this approach is concretized in Maccabee's DGP sampling procedure. The third section provides a formal statistical specification of a Maccabee benchmark. Finally, the fourth section discusses the empirical data sets that are included in the package. Taken together, the content of these four sections provides a complete picture of the theoretical design of the Maccabee package.

5.1 Design Goal: Better Benchmarking for Causal Inference Methods

As established in Chapter 4, most existing approaches to benchmarking methods for causal inference fall into one of two *pure* design categories:

1) **Empirical methods** use real, observed covariate and outcome data. This data is typically drawn from randomized experiments so that ground truth effect values are known (although experiments only provide average effect estimates). In these benchmarks, there is a true Data Generating Process (DGP) but it is latent and its properties are unknown. It is unclear if the data meets the typically required causal inference assumptions, especially after modifications like replacing the random control group with non-random data to create a *psuedo-observational* setting in which randomization, and the resultant expected covariate balance, doesn't hold.

2) **Synthetic methods** use covariate and outcome data generated from synthetic (hand-specificed rather as opposed to observed/empirical) distributions and functions. By definition, the DGP is known. As a result the true causal effects - both at the average and individual level - can be calculated directly and it is known if causal assumptions are met.

While both of these approaches have strengths and weaknesses, explored in detail in Chapter 4, they share a common flaw: their sample size is, effectively, one. In either design, a new causal inference method is tested against one or a few DGP (per targeted distributional setting). In empirical datasets, the properties of this DGP and, most importantly, its place in the distributional problem space, are unknown. The causal inference method is validated against a single DGP 'sample' from an unknown location in the problem space. In synthetic datasets, the DGP properties - and its location in problem space - are known but, again, there is usually only one or a few DGPs per distributional setting.

In the best case, either of these approaches risks missing important aspects of estimator performance in the selected region of problem space by failing to average over variance in performance present in structurally similar DGPs. In the worst case, the results are biased - either by failure of empirical data to conform to basic causal assumptions or by manual specification using unrealistic data or unrepresentative/simplistic versions of DGP from the selected region (unintentially). In either scenario, benchmarking results are unlikely to generalize as indicators of consistent, real-world performance.

The primary design goal of the Maccabee is to enable a benchmarking approach that mitigates the flaws of the common approaches analyzed above. This is done by adopting the theoretical design proposed Dorie et al. (2019).

5.2 Approach: Flexible, Reusable Hybrid Benchmarking

This section outlines Maccabee's benchmarking approach. The high-level approach is inspired by, and closely follows, the benchmarking process proposed in Dorie et al. (2019). The low-level design specifics of the approach, and the corresponding implementation described in Chapter 6, are de novo.

5.2.1 Dorie et al's General Approach to Hybrid Benchmarking

The benchmarking process proposed by Dorie et al. (2019) mitigates the weaknesses outlined in Section 5.1 by *sampling* Data Generating Processes. These sampled DGPs have treatment and outcome mechanisms defined over arbitrary covariates and sampled from parameterized distributions over a subspace of functions. The parameterization of these distributions allows for control over the expected location of sampled DGPs in the distributional problem space (established in Chapter 3). This approach allows for:

- More thorough exploration of the distributional problem space (relative to empirical benchmarks) because DGPs can be sampled from many different (known) locations in the problem space.
- More realistic covariate distributions (relative to synthetic benchmarks) because sampled DGPs can (but do not have to) consist of sampling functions defined over empirical/observed sets of covariate data.
- More robust results (relative to either of the pure approach) because performance evaluations are performed by averaging across many DGPs selected, in a principled manner, from a region of problem space.

Maccabee, following Dorie et al. (2019), samples a treatment assignment, an untreated outcome function, and a treatment effect function and uses these functions as the components Ω, Φ and τ to concretize the generic DGP outlined in Chapter 4. The sampled functions are drawn from the subspace of *generalized, additive functions* - additive functions with terms that are 1st, 2nd or 3rd degree polynomials, two or three way interactions, and linear functions with jump discontinuities in their value or derivative.

The fourth degree of freedom in the concretization process - the joint distribution over covariates, ρ - is specified to be either an empirical distribution over observed covariates or an arbitrary joint distribution. In either case, the three sampled functions are defined over the covariates from ρ.

5.2.2 Concretizing Dorie et al's Approach

While the approach described above makes sense, it does not prescribe a specific sampling scheme, and associated set of sampling parameters, that could be used to sample DGPs from a desired location in the distribution problem space. Dorie et al. (2019) does have such a scheme implemented in R (available here). But this code is designed to generate a static set of benchmarking datasets - using a single empirical covariate dataset and a single set of sampling parameterizations. Further, this code is not designed to execute benchmarks or perform metric calculation, it is only designed to generate the benchmarking data itself. Finally, the code is poorly documented which makes parsing the exact parameterization used by the authors non-trivial. Given this, Maccabee proposes its own parameterization and sampling scheme, based on the general approach outlined above. This scheme is then implemented such that it can be used with any covariate data and with arbitrary, easy-to-specify parameterizations.

The goal of this explanation is to show how a set of *DGP Sampling Parameters* can be used to sampled DGPs from a region of the distributional problem space. As established in Chapter 3, this means the parameters must (approximately) control the location of sampled DGPs along the seven problem space axes. Discussion of how sampled DGPs can then be used to implement benchmarking is covered in the formal specification in Section 5.3.

In Maccabee, the distribution over functions is central to this control. Sampled DGP functions are made up of the linear combination of the set of *term types* listed in Section 5.2.1 above. These term types are: 1st, 2nd or 3rd degree polynomials, two or three way interactions, and linear functions with jump discontinuities in their value or derivative. Covariates may appear in multiple terms, so the *universe* of concrete terms for each *term type* is all of the combinations of covariates that could appear in the form of the *term type* (there are some restrictions, noted below). The distribution over the functions is specified by the set of (uniform) probabilities that each term in the universe of terms for each term type *term type* is selected to appear in the function. This means the function distribution is parameterized by seven probability parameters, one per *term type*. In addition to this, each term has a co-efficient sampled from a co-efficient distribution. While this distribution is central to the Maccabee design, the complete DGP sampling procedure must be complicated to allow for control over the complete set of axes.

The procedure is given below. The parameters which influence the operation of each step are given in bold at the end of each step description.

1) A set of *potential confounders* is selected from all of the covariates based on a single, uniform selection probability. This is the set of confounders that could

enter either/both of the sampled treatment/outcome mechanisms. Fewer selected covariate produces a lower signal-to-noise ratio in the data and the makes the modeling of the treatment/outcome mechanisms harder.

2) The treatment assignment and untreated outcome functions are sampled from two independently parameterized distributions of the kind described above. The universe of each *term type* is specified by the combination of all the potential confounders selected in step 1). The co-efficients in all of the terms of two functions are initialized by drawing samples from a parameterized, global co-efficient distribution.

   Note that, in the generic DGP described in Chapter 4, the treatment assignment is sampled using a single function referred to as Ω. In this concretization, Ω is split into a treatment probability and treatment assignment function. The treatment probability function is sampled in the manner described above and the treatment assignment produces assignments conditioned on the probability. For simplicity, this difference is not made explicit in the explanations below.

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. Lower alignment thus implies a more disjoint set of covariates appearing across the functions. This produces weaker confounding of the treatment assignment and outcome variables present in the generated data.

4) The treatment effect mechanism is then sampled. First, a constant treatment effect is sampled from a parameterized, global treatment effect distribution. 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 means the treatment effect function can be thought of a subset of the untreated outcome function terms (plus a constant offset) that interact linearly with the binary treatment assignment.

5) Numerical normalization and linear modification (by multiplicative/additive constants) of the treatment assignment function is used to meet target parameters for the min/expected/max value of the treatment probability

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 only occur at data generation time. On each instance of data generation:

   a) A fixed proportion of the covariates are sampled with the default being that all observations are sampled. This step can be used to mimic small datasets (or heavy, at-random censorship) when using empirical datasets. This will change the number of observations in each generated dataset.

   b) Observed outcome noise is sampled from a parameterized, global outcome noise distribution.

c) 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 creates a larger imbalance in the co-variate distribution between the groups.

By controlling the probability with which different term types appear in the treatment assignment, untreated outcome, and treatment effect functions, as well as the overlap of terms between the functions, one can (approximately) control linearity, degree of confounding and causal effect heterogeneity. By controlling the distributions from which the treatment effect, outcome noise, and constant co-efficients are sampled, and applying normalization techniques, one can (approximately) control the location and scale distribution over outcome values and probabilities of treatment (as well as the signal-to-noise ratio of these quantities with respect to the covariates). Finally, by swapping units between groups, one can (approximately) control the imbalance in the distribution of covariates across the treated/control groups can be controlled.

Note that the degrees of freedom listed above correspond exactly to the axes that make up the distributional problem space. Therefore, by using the parameters that control these degrees of freedom, it is possible to generate DGPs that are - in expectation - located in different regions of the space. This, in turn, allows for better evaluation of new causal inference methods. Both by evaluating them across more regions in the problem space and more robustly in each region by sampling multiple, structurally similar DGPs and averaging out any function-specific effects.

5.3 Formal Specification of the Maccabee Benchmark

Section 5.2 above provided an overview of the approach taken by Maccabee, focusing on the DGP sampling procedure. This section provides a formal specification of the end-to-end benchmarking procedure. The specification takes the form of a statistical model, depicted as a graphical model, that describes a complete Maccabee Monte Carlo benchmark. The DGP sampling procedure from the previous section is only a sub-component of the statistical model depicted here.

5.3.1 Statistical Model

Figure 8 contains a graphical model that provides a formal, statistical description of Maccabee's benchmarking process. It explicates the relations between all of the (fixed) parameters and (sampled) random variables that are combined to produce a benchmark result (a Monte Carlo estimate of one or more metric defined over the estimand sampling distribution). This can be thought of as a description of the benchmarking process at the level of data that is completely generic with respect to the functions/distributions that relate/produce the data.

It is important to note that this model extends (and slightly modifies) the generic DGP statistical model proposed in Chapter 4 (as well as the concretized, sampled-function version from Section 5.2). Those models included only the *DGP Random Variables* (and the relations between them). The model below extends this set of variables to include *Estimand Random Variables*: The causal estimand random variables - samples from the estimand sampling distribution - and the performance metrics calculated over this distribution. As well as the data metrics that quantify the position of the DGP in the distributional problem space. Three notational tools are used to make this extension:

- *DGP Random Variables* - that are part of the original generic DGP framework - are indicated in white while *Estimate Random Variables* - either estimands or metrics derived from these estimates - are in blue.
- Plate notation is used to indicate that each dataset includes N observations and that M such datasets, each drawn from a different DGP - are present in the benchmark described by the model. Each dataset has an associated set of performance and data metrics (that are themselves random variables). There is one metric value (for each metric) per dataset.
- The *DGP* random variable - sampled based on the *DGP Sampling Parameters* - represents the treatment and outcome mechanisms that connect the observed covariates to the *oracle* (latent/unobserved) treatment and outcome random variables. This variable stretches graphical notation, so it worth a brief discussion.

  Ordinarily the functions/distributions relating the covariate variable X to the treatment/outcome variables would be represented by the edges between these variables - each defined by fixed, parametric functional forms. In the Maccabee benchmarks, these functions/distributions are themselves sampled based on underlying parameters and distributions. As such, they are best represented as a Random Variable which conditions the value of the random variables that are related/generated by the DGP. The single *DGP* random variable can thus be thought of as a variable that abstracts lower level functional detail.

Finally, it is worth noting that the estimator used to generate average/individual causal effect estimates from the observed outcome data is not explicitly depicted in this model. This estimator - whether stochastic or deteministic - is represented by the edges connecting the observed DGP variables to the individual/average estimate values.

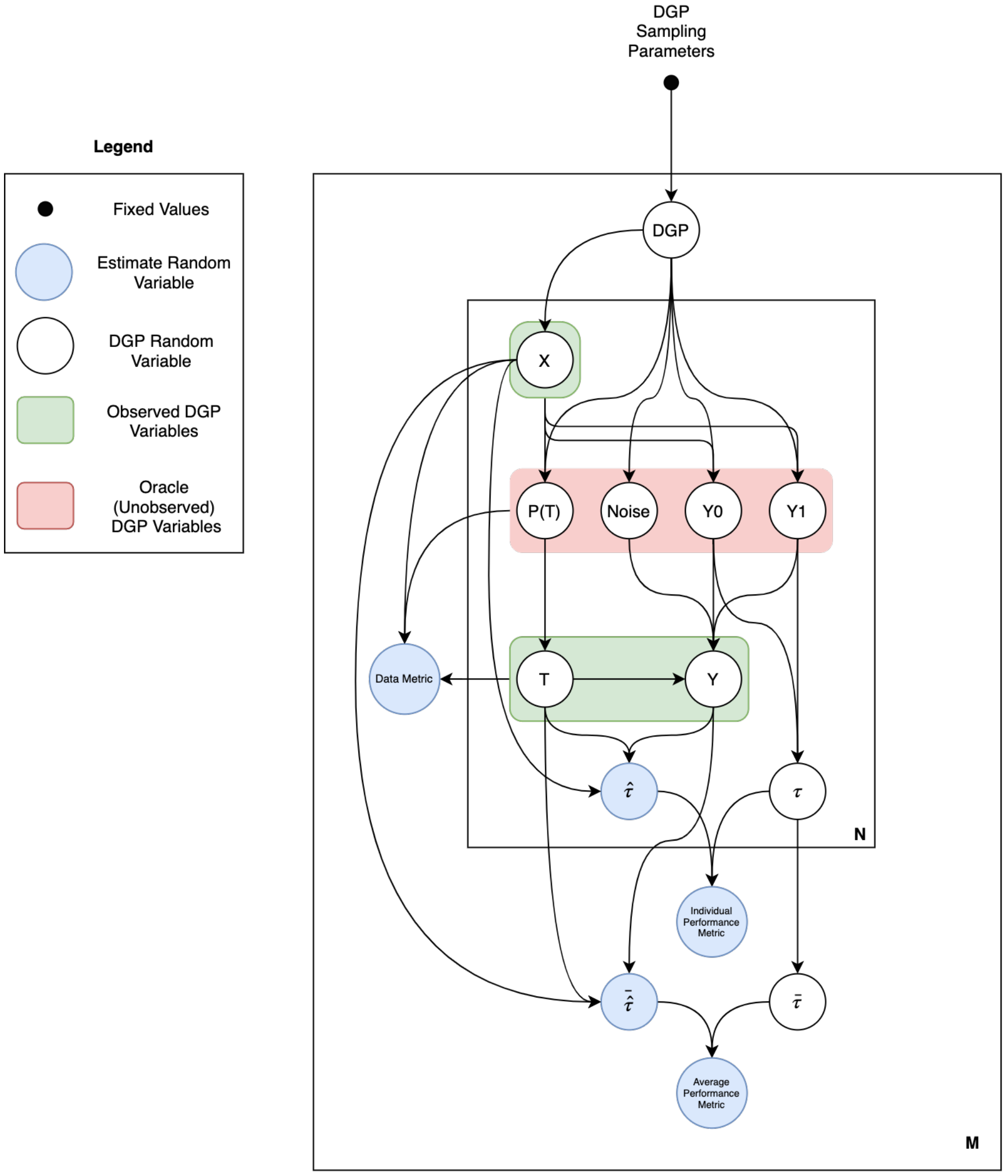


Figure 8: A Graphical Model depicting the complete statistical model of the a Maccabee Monte Carlo benchmark.

5.3.2 Sampling Procedure

The graphical model in Figure 8 concretely specifies the sampling procedure used to execute a Maccabee Monte Carlo benchmark. Moving from the top of the model to the bottom:

1) A set of M DGPs is sampled from a sampling distributions parameterized by the fixed *DGP Sampling Parameters* described above.

2) A set of N covariate observations - each represented by the variable X - is drawn for each *DGP* (actually from the subcomponent of the DGP representing the joint distribution over covariate observations, the ρ distribution in the theory

paper).

3) Each covariate observation X has an associated set of treatment assignment, outcome and causal effect random variables. Added to the observed covariates, these variables represent the complete observed and unobserved information about each individual observation in the dataset. Following the nomenclature from Chapter 3, I interchangeably refer to unobserved variables as oracle variables.

   Note that, for expositional clarity, the dependencies between the individual observation variables depicted in this model are slightly different to the actual relations implied by the sampled DGPs described in Section 5.3 above.

   With that established, the sampling procedure for each individual observation is given below:

   - The *oracle* (unobserved) treatment probability ($P(T)$) is sampled conditioned on the covariate observation X and the treatment probability function (defined over X) from the sampled DGP. The *observed* treatment assignment - T - is then sampled conditioned on $P(T)$.
   - The *oracle* (unobserved) potential outcome variables ($Y1$ and $Y0$) are sampled conditioned on the covariate observation X and the outcome functions (defined over X) from the sampled DGP. This is true to Rubin's Potential Outcome framework as described in Chapter 2 but is not in line with procedure implied by the structure of the sampled DGPs from Section 5.3 above. In a sampled DGP, only the untreated outcome, $Y0$, is sampled (from the untreated outcome function, Φ). This value is them combined with a treatment effect (sampled from the treatment effect function, τ) to produce the treated outcome, $Y1$. This difference simplifies the graphical model and is irrelevant for the remainder of the sampling procedure.
   - The *oracle* (unobserved) outcome noise is sampled from the outcome noise distribution from the sampled DGP.
   - The *observed* outcome variable - Y - is sampled conditioned on the treatment assignment, potential outcomes and outcome noise.
   - The individual causal effect variable - τ - is (deterministically) sampled conditioned on the potential outcome variables. Note, again, that in the sampled DGP, the individual treatment effect is sampled from the treatment effect function, τ.

4) Causal estimand values can be sampled at the individual observation or dataset level. At the individual level, N individual effect estimates $\hat{\tau}$ are sampled from a (deterministc/stochastic) estimator conditioned on each X, T and Y. At the dataset level, a single average effect estimand $\bar{\hat{\tau}}$ is sampled from a (deterministic/stochastic) estimator conditioned on all N of the X, T and Y observations.

5) Following estimand sampling, 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) - τ or $\bar{\tau}$ respectively.

6) Finally, and optionally, M Data Metrics are calculated by combining some/all of the covariate data with the observed and oracle outcome data.

5.4 Empirical Data Sets in Maccabee

The sections above discuss the joint distribution over covariates, the ρ component of the DGP, entirely generically. In reality, the selection and construction of this distribution is an important aspect of Maccabee's core design. As mentioned by section 5.2.1 above, and justified in Chapter 4, the design advantage of Maccabee comes from the combination of *empirical* covariates and sampled treatment and outcome mechanisms. This section discusses the use of empirical covariate distributions in Maccabee.

As implied by the name, an empirical covariate distribution is a joint distribution over observed covariates with an unknown parameterization, ergo one which is defined, empirically, by the data itself. This data may be an incomplete/biased sample so that the empirical covariate distribution defined by it is not the same as its true underlying joint distribution.

Maccabee is designed with both a set of included, high quality empirical covariate distributions as well as tools to allow for the use of user-supplied data to define custom empirical covariate distributions. These two approaches are described in the sections below.

5.4.1 Included Empirical Covariate Distributions

Maccabee includes empirical covariate distributions defined by data sets from the literature on causal inference benchmarking. These data sets where selected because they are:

- Used widely, indicating a social consensus that they are high quality - representative, useful - data sets for the evaluation of causal inference methods.
- Varied in the number of observations and covariates, allowing for benchmarking of methods under different circumstances.
- Open source, meaning the data can be accessed and used by others to ensure that any results reproduce as expected.

5.4.1.1 The National Support Work (NSW) Data Set

This is a data set corresponding to the NSW data, introduced by Lalonde (1986), that appears throughout the early causal inference literature. Building on R. Dehejia (2005) and Smith and Todd (2005), the package uses a selected subset of the Lalonde (1986) experimental data (the subset with real income in 1974 available) and the *PSID* observational controls with the data downloaded from here. In the experimental data, there are 297 treated units and 425 control units. The control units are dropped in favor of the 2490 PSID units to produce a non-randomized observational setting [1]. Real Income in 1978 is the measured outcome. All other variables are measured pre-treatment and serve as covariates.

5.4.1.2 The Collaborative Perinatal Project (CPP) Data Set

The CPP data set is extracted directly from the benchmarking implementation provided by Dorie et al. (2019). The authors describe the data set as follows.

[1] Note that, even though the treatment and outcome mechanisms will be sampled, it is necessary to drop the experimental controls to reproduce the covariate distributions in the Lalonde data set

> The Collaborative Perinatal Project (Niswander and Gordon, 1972), is a massive longitudinal study that was conducted on pregnant women and their children between 1959 to 1974 with the aim of identifying causal factors leading to developmental disorders. The publicly available data contains records of over 55,000 pregnancies each with over 6500 variables. Variables were selected by considering a subset that might have been chosen for a plausible observational study. Given the nature of the data set we chose to consider a hypothetical twins study examining the impact of birth weight on a child's IQ. We chose covariates that a researcher might have considered to be confounders for that research question. After reducing the data set to complete cases, 4802 observations and 58 covariates remained. Of these covariates, three are categorical, five are binary, 27 are count data and the remaining 23 are continuous.

5.4.2 Custom Empirical Covariate Distributions

In Maccabee, almost any data to be used to defined an empirical covariate distribution. This is useful for benchmarking methods that are designed to work well in very specific covariate settings. The tools that enable the use of custom empirical data sets are described in the package documentation here.

It is crucial to note that the properties of the sampled DGPs, measured along the axes from Chapter 3, are sensitive to the empirical covariate distribution. This means that, in order to produce DGPs in the desired region of the distributional problem space, it is usually necessary to both lightly pre-process the covariate data and fine tune the other DGP sampling parameters. There are limits on how much of these two steps can be automated by the package. By default, Maccabee applies normalization to the covariates appropriate for pair-wise independent, symmetric distributions of each covariate. The default sampling parameters are also tuned for to expect this form of distribution. If any other data is used, care will need to be taken. Fortunately, the formal metrics used to measure the position of sampled DGP in the distributional problem space can be used to guide this process without any fear of overfitting/compromising benchmark validity. DGP sampling is done prior to any causal estimation, so pre-processing and sampling parameter values can be adjusted until metric values indicate that DGP samples are being drawn from the desired region of the problem space.

Chapter 6

Maccabee's Benchmarking Implementation

The source of truth for the implementation of the Maccabee package is the package documentation. A detailed explanation of the implementation of the sampling procedure outlined in Chapter 5 is located on the design page of this documentation.

The documentation is also included in this document as Appendix A. Accessibility is worse in the PDF format given the absence of pagination and cross-linking, so the online documentation is recommended.

Chapter 7

Validating Maccabee

NOTE TO GRADER: the results in this section are preliminary and reflect quick initial validation. I plan to spend all of the next three weeks fleshing out the results in this chapter by looking at a wider variety of case studies.

This chapter presents results and analysis aimed at validating the approach and the implementation of the Maccabee package. The content is split into two sections. The first section presents results aimed at validating the DGP sampling parameterization by showing that Maccabee is able to sample DGPs from specified locations in the distributional problem space. The second section presents results aimed at validating the benchmarking approach by looking at a number of applicable case studies.

It is worth noting, explicitly, that validating this package with any level of certainty is a hard problem. First, and most generally, because there is no good way to objectively and quantitatively compare different benchmarking approaches. Producing different results for some method relative to other benchmarking approaches is not indicative of either success/failure without a deeper understanding of why the results differ. Second, as the reader will see below, validating the Maccabee approach relies on quantifying the position of sampled DGPs in the distributional problem space by using a set of *data metrics* that were introduced in Chapter 3. These metrics are noisy/imprecise measurements of the true distributional setting of a DGP. In fact, Dorie et al. (2019) find almost no correlation between the metrics the propose (the same ones used in this work) and estimator performance even though estimator performance is strongly correlated with the parameterization that they used to control distributional setting in their paper. This means the changes in the distributional setting created by different parameterization of the DGP sampling process are not clearly reflected in changes in the value of metrics. The first section below relies, unavoidably, on the metrics. However, in the second section, I have made an effort to define a validation scheme that is not overly reliant on the metric values.

7.1 Validating the DGP Sampling Parameterization

Chapter 5 established that one of Maccabee's key design goals is to allow the sampling of DGPs from specified locations in the distributional problem space through the use of a parameterized DGP sampling procedure. Chapter 6, and the documentation linked therein, discussed the different ways in which the parameters of the sampling process can be specified by the user. These can be summarized as either *preset* - specifying parameters by selecting a *high*, *medium* or *low* level on each supported axis of the problem space and using the low-level parameter values built into the package. Or, *customized* - specifying the low-level parameters directly. This section focuses on validating the first approach. If custom low-level parameters are required, users should follow the advanced tutorials in the package documentation in order to refine parameter values using the relevant data metrics.

Figure 9 displays the primary DGP sampling validation results. Each subplot corresponds to a single axis of the problem space. The Y axis is the value of a metric which measures the location of sampled DGPs along the problem space axis and the X axis is the user-specified target *level* for the axis. In each plot, the parameters corresponding to all other axis are kept at package default values. The metric value at each level is evaluated by sampling 32 DGPs and then sampling 100 datasets from each DGP. The value of the data metric is averaged across the datasets for each DGP and then averaged across all DGPs. The error bars correspond to the standard deviation of the metric value between DGPs.

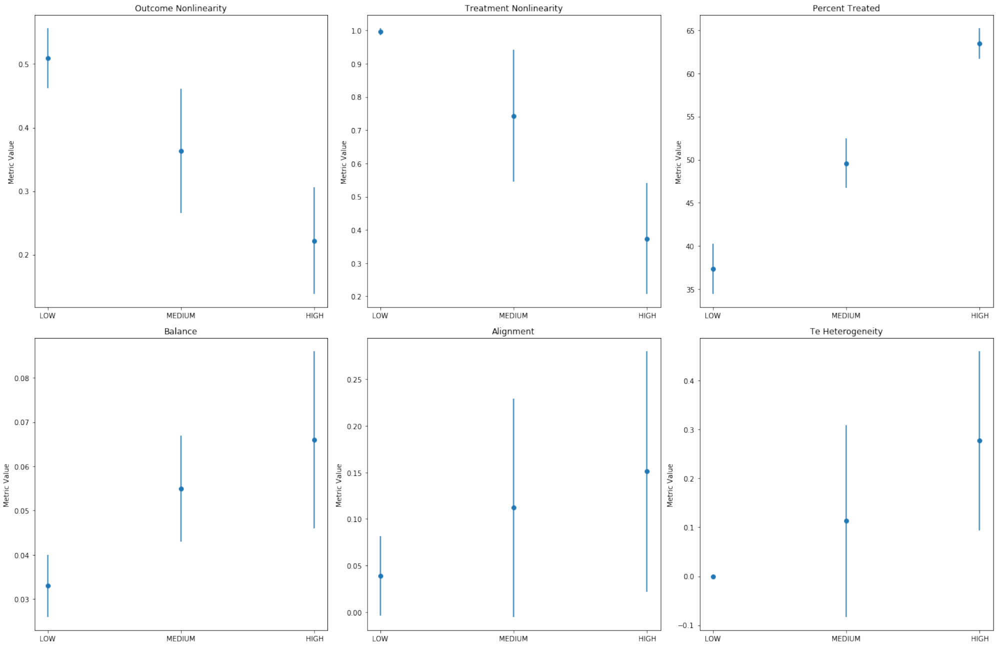


Figure 9: DGP sampling parameterization results - metric values for each axis when DGPs are sampled at the axis level on the X axis.

The plots indicate that the level-based specification provides for reasonable control over the position of sampled DGPs along each axis. Across all axes, the mean of the metric value at each level increases/decreases in the direction expected given the change in level[1]. Below, I describe the metric used to measure each value, see Chapter 3 for a thorough theoretical introduction to the construction of these metrics. Working from top left to bottom right:

- **Outcome Nonlinearity** measured by the R^2 value of a linear regression of the observed outcome on the observed covariates. The value of this metric decreases

[1] Note that, for each axis, the meaning of low, medium, and high is taken from the natural semantic interpretation of a low, medium and high level of that axis. So, for example, low imbalance means a more imbalanced covariate distribution in the treatment and control group while a low outcome nonlinearity means a more linear outcome mechanism.

as the level of the axis is changed from low to high, as would be expected under decreasing linearity of the outcome mechanism.

- **Treatment Assignment Nonlinearity** measured by the R^2 value of a linear regression of the oracle treatment probability logit on the observed covariates. The value of this metric decreases as the level of the axis is changed from low to high, as would be expected under decreasing linearity of the treatment assignment mechanism.
- **Percent Treated** measured by the percent of units in the treated group. The value of this metrics increases, as expected, as the percent treated axis level is changed from low to high.
- **Imbalance** measured by the Wasserstein Distance[2] between the covariate distributions in each treatment group. A higher metric value (larger Wasserstein Distance) indicates a larger the distance between the distributions in each group, meaning greater imbalance. The values of this metrics increase, as expected, as the imbalance axis level is changed from low balance to high balance.
- **Alignment** measured by a linear regression of the observed outcome on the oracle treatment probability logit. A higher value of the metric means there is greater overlap in covariates between the treatment and outcome mechanisms. The values of the metric increase, as expected, as the alignment axis level is changed from low to high.
- **Treatment Effect Heterogeneity** measured by the ratio of the standard deviation of the treatment effect to the standard deviation of the observed outcome. The values of the metric increase as the axis level is changed from low to high, indicating an increasing portion of the individual outcome variance stems from changes in the treatment effect between individuals.

Note that, despite the promising trend in the mean of each metric, there is a weakness evident in the plot: The variance of the sampling process appears to be substantial relative to the scale of the differences in metric values at each level. Some of this variance may be equivalent to *measurement* noise[3] rather than indicating variance in the DGP sampling process itself. This can be tested by increasing the number of samples of data from each DGP to see if the standard deviation of the metric across DGPs decreases. Doubling the sample count from 100 to 200 does decrease the average standard deviation across all metrics and levels by about 30%, which means that at least some of variance can be attributed to measurement noise. However, this does seem to confirm that the source of some of the variance is the DGP sampling process itself. IE, at each parameterization, it is possible to sample DGPs which are quite dissimilar in their position along any given axis.

7.2 Validating the Benchmarking Approach

This section focuses on validating Maccabee's Hybrid Monte Carlo benchmarking approach. This is done by looking at a series of case studies that demonstrate key advantages. These case studies are based on comparisons to the benchmarks used in Diamond

[2] The Wasserstein Distance is an *integral probability metric* that measures the absolute value of the difference between the CDF of each distribution, integrated across the domain. This can be approximated by formulating a minimum flow optimization problem using the observed covariate values in each group. See the implemented metric code for detailed documentation.

[3] This noise could stem from the imprecise nature of the metric calculations and the stochasticity of the data sampled from each DGP.

et al. (2012). This paper was chosen because it benchmarks a new method for causal inference - Genetic Matching - using a series of three benchmarking approaches that align nicely with the landscape of approaches established in Chapter 4:

1) **Pure, Synthetic Benchmark**: First, Diamond et al. construct a series of synthetic DGPs based on the following scheme: Ten covariates are sampled from a joint distribution consisting of a mix of ten independent Bernoulli binary (with $p = 0.5$) and standard normal variables. They specify a fixed outcome mechanism with a constant treatment effect. Finally, they hand-specify a set of seven treatment assignment (logit) functions (all generalized, additive functions) with escalating levels of nonlinearity (introduced through quadratic terms) and non-additivity (introduced through two-way interaction terms). This produces a total of seven synthetic DGPs labeled A-G in order of increasing treatment mechanism complexity.

2) **Pure, Empirical Benchmark**: Second, Diamond et al. perform a pure empirical benchmark using the data set from Lalonde (1986).

3) **Hybrid Benchmark** Finally, the authors specify a hybrid benchmark that uses the covariates from the Lalonde (1986) empirical benchmark and hand-specified treatment and outcome functions.

The results collected from these three benchmarks provide a useful baseline against which results from Maccabee can be compared. With each of the three benchmarks providing insight into different aspects of Maccabee's sampled-based Hybrid Monte Carlo approach. The sections below focus on making these comparisons. In each case, I customize the operation of the Maccabee package to provide a narrow comparison that maximizes the set of 'constants' which are kept the same between the two methods.

7.2.1 Sampled vs Static Synthetic DGPs

This section focuses on validating the idea of sampling synthetic treatment assignment and outcome mechanisms. This is done by comparing a sample-based synthetic DGP benchmark, executed using Maccabee, to Diamond et al.'s static synthetic benchmarking approach.

7.2.1.1 Comparison Explanation

In order to focus the comparison, the standard Maccabee benchmark is customized to sample covariates from an the same distribution as used in Diamond et al. (2012) and to use the exact outcome mechanism and (constant) treatment effect used in the paper. Further, the low-level parameterization of the treatment assignment mechanism sampler is set to sample functions that look almost identical to the treatment assignment mechanism in scenario G of Diamond et al. (2012). This means that the number of terms of each type - linear, quadratic and two-way interaction - will be same across sampled and hand-specified functions. The only difference is in which covariates are selected for each term type and what coefficients are used. Finally, coefficient are sampled from a distribution designed to produce coefficients that are very similar to the ones in the hand-specified functions.

Note that this is a much lower variance setting than a standard Maccabee sampled benchmark. A standard sampled benchmark would sample a set of functions with varied functional form to target a given level of nonlinearity. In this comparison, the samples are drawn from the set of functions with the exact same form as the hand-specified function. The implication of this fact is that variance in this class of functions is a subset of wider variance we would expect from the universe of possible functional forms (with roughly similar non-linearity, see Section 7.1) that could have been selected.

With the above established, the (compared) benchmarking procedures are as follows: In the original work, Diamond et al. sample 1000 datasets from the DGP in Scenario G and report the Root Mean Squared Error and the Absolute Bias (Percentage) of a number of estimators (including Genetic Matching)[4]. I repeat this data sampling 30 times, to produce an interval that quantifies the variance of the performance metrics for this single DGP. In order to construct the comparison benchmark, I sample 16 DGPs and, from each, sample 1000 datasets, 30 times. This produces a set of 16 performance metric intervals for comparison with the single hand-specified DGP. On top of the performance metrics, I also collect the data metrics for every data set to allow for comparison of the distributional setting.

NOTE: In the results below, the only estimator used is the Logistic Propensity Matching Estimator. More estimators will be included shortly.

7.2.1.2 Comparison Implementation

All of the above is done using Maccabee. The hand-specifed DGP from Diamond et al. (2012) is sampled using a custom, concrete DGP and the sampled DGPs are constructed using the default Maccabee sampler with custom parameters, a custom data source and a customized DGP class (to keep the outcome mechanism fixed). The case study code is available in a Notebook here. See Chapter 6 for explanations of these implementation concepts.

7.2.1.3 Comparison Results

Table 12 displays the average data metric values for the datasets produced by the Genmatch and Maccabee (sampled) DGPs. The metric values are close to identical for all metrics. This does not mean that all the sampled DGPs have the same metric value (this variance will be discussed shortly) but it does mean that the concrete treatment assignment mechanism selected in the Genmatch paper appears to be a 'typical' function in the set of functions of the same form. Typical, in this setting, means that the metric values for the selected function match the mean of the metric values for the DGPs constructed by sampling from the set of all functions of the same form.

[4] See Chapter 6 for definitions of these performance metrics

Axis	Metric	Genmatch Value	Maccabee Value
Outcome Nonlinearity	Lin r2(X_obs, Y)	0.909	0.905
Outcome Nonlinearity	Lin r2(X_true, Y)	0.91	0.907
Outcome Nonlinearity	Lin r2(X_obs, Y0)	1.0	1.0
Outcome Nonlinearity	Lin r2(X_true, Y0)	1.0	1.0
Treatment Nonlinearity	Lin r2(X_obs, Treat Logit)	0.938	0.796
Treatment Nonlinearity	Lin r2(X_true, Treat Logit)	1.0	1.0
Treatment Nonlinearity	Log r2(X_obs, T)	0.679	0.669
Percent Treated	Percent(T==1)	54.927	50.309
Alignment	Lin r2(Y, Treat Logit)	0.059	0.1
Alignment	Lin r2(Y0, Treat Logit)	0.014	0.096
Alignment	Log r2(Y0, T)	0.552	0.612
Alignment	Log r2(Y, T)	0.665	0.669
Te Heterogeneity	std(TE)/std(Y)	0.0	0.0

Table 12: Data metrics values for the comparison between the Genmatch and Maccabee benchmarks based on pure, synthetic DGPs. Maccabee produces DGPs with mean metric values nearly identical to the metric values for the original Genmatch DGP.

Based on the similar average data metrics, one would expect that the mean performance of the causal estimator on the data from the Genmatch vs sampled Maccabee DGPs would be similar (if the data metrics are indeed indicative of performance). Indeed, this appears to be the case. Comparing the two metrics used by Diamond et al.:

- The Mean Absolute Bias Percentage (MABP) is 7.592% (std 0.11%) in the Genmatch benchmark and 7.054% (std 1.24%) in the Maccabee benchmark.
- The Root Mean Squared Error is 0.038 (std 0.001) in the Genmatch benchmark and 0.035 (std 0.006) in the Maccabee benchmark.

At this point, it appears that there is little difference between the two benchmarking approaches. The only evident difference is in the standard deviation of the MABP. Upon closer examination, this turns out to be a crucial factor in the comparison. Figure 10 shows the distribution over performance metrics in the sampled DGPs compared to the performance metric value for the Genmatch DGP. It is clear, in both panels, that the MABP value for the Genmatch DGP is a 'sample' from a fairly wide distribution over the performance metric. This is initial evidence for the power of sampled DGPs. If the sampled functions are indeed 'like' the concrete, hand-specified function in the paper then it conceivable that a function from anywhere in the distribution could have been selected. The empirical 95% interval around the mean of the sampled distribution ranges from 5.5% to 9.7%. Alone, this interval is not necessarily concerning but if the interval overlaps with the performance interval for another estimator, then the rate of false ordinal ranking from single DGP comparison of the estimators will be proportional to the degree of overlap in the intervals. *Pending work will focus on exploring the extend of this overlap in the context of the different methods compared in the Genmatch paper.*

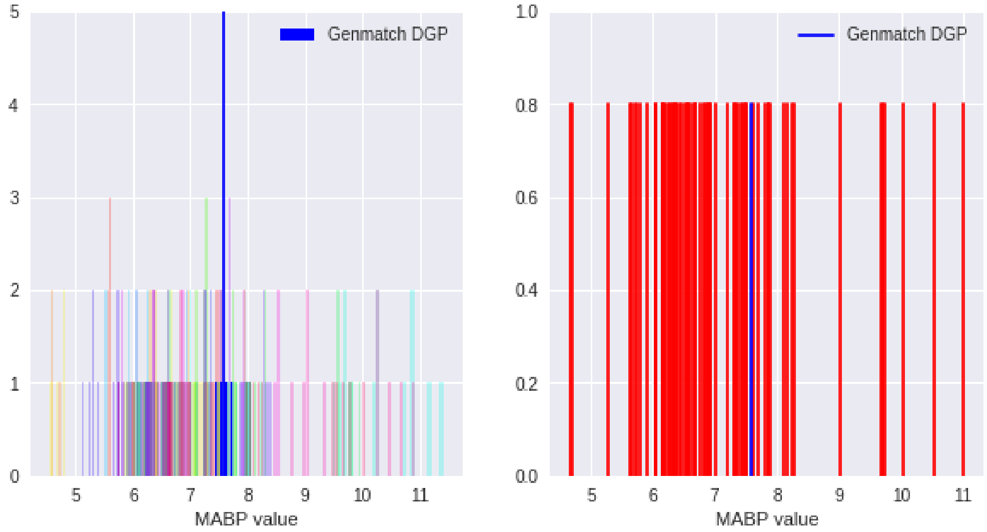


Figure 10: Pure, Synthetic benchmark performance metric comparison. The concrete DGP from the Genmatch paper is a typical sample from a fairly wide distribution over possible performance metrics. Left panel: the distribution over the MABP for each sampling run, for each DGP. Each DGP is in a different color. Right panel: the distribution over the mean value of the performance metric across all sampling runs for each DGP.

The natural question to ask at this point is how the performance metrics are correlated to the data metrics. Figure 11 shows the MABP performance metric plotted against a metric for treatment nonlinearity. This plot is interesting. Firstly, while there is a clear (and significant, with p=0.0001) correlation, this correlation is reversed from what we would expect. One would expect that as the linearity increases (with higher values of the metric corresponding to increased linearity) the performance of the Logistic Propensity Matching estimator - which estimates propensity scores using a linear model - would improve. However, the reverse appears to be true. This may be because, the more linear the function, the more alignment there is between the treatment and outcome functions (the outcome function is linear) and this then means there is a larger degree of confounding to reverse.

Regardless of the trend direction, there are two important observations.

1) Even within a narrow set of functions, there is variance in the position of DGPs in the distributional problem space as measured by data metrics. This variance is significant in that it is meaningfully correlated with changes in the performance of estimators. This means that choosing a single DGP from the set of possible DGPs is risky as it could appear anywhere in the distribution over the possible distributional setting and hence anywhere in the distribution over estimator performance.

2) Even though the data metric is predictive of performance, there is still significant variance in the performance metric at each data metric. This implies that even DGPs with similar measurable distributional settings on one axis may differ in other performance-relevant axes.

Both of these points imply that sampling DGPs to capture, and then average over, the distribution of performance metrics for a given estimator is crucial. Using a single DGP risks picking an arbitrary location in the distribution without any indication of the underlying variance. The first point also motivates an approach that targets specific locations in the distributional problem space by targeting measurable metric values. If these metrics are indeed predictive of performance, then a complete exploration of the performance of an estimator requires exploring its behavior under all and/or known distributional settings.

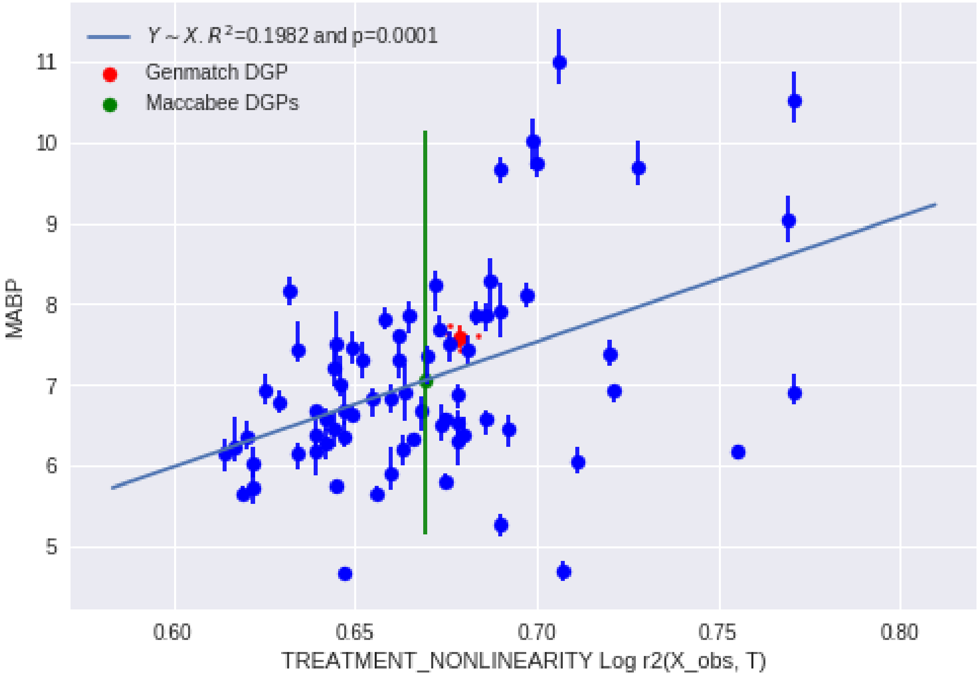


Figure 11: Pure, Synthetic benchmark performance metric plotted against the data metric values for treatment nonlinearity. Maccabee sampled DGPs are in blue with a large dot at the mean performance/data metric value for each DGP and error bars corresponding to the standard deviation of the performance metric across sampling runs. Genmatch, concrete DGPs are in red with a large red dot at the the mean performance/data metric value and an individual, smaller dot for the performance/data metric value of each sampling run. It is clear that the treatment nonlinearity metric is predictive of performance but only explains a small portion of the variance in the performance. It is also clear that the concrete Genmatch DGP displays performance which is typical of its treatment nonlinearity.

7.2.2 Sampled vs Static Hybrid DGPs

Results for this comparison are pending.

7.2.3 Sampled, Hybrid DGPs vs Empirical DGP

Results for this comparison are pending.

Part V

Conclusion

Chapter 8

Conclusion

PENDING: this chapter will conclude.

Part VI

Appendices

Appendix A

Maccabee Package Documentation

The documentation for the Maccabee package is included as a PDF below. It's also available as a webpage here, with cleaner formatting, clickable links, and search tools.

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]:
```

```python
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]:
```

```python
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]
```

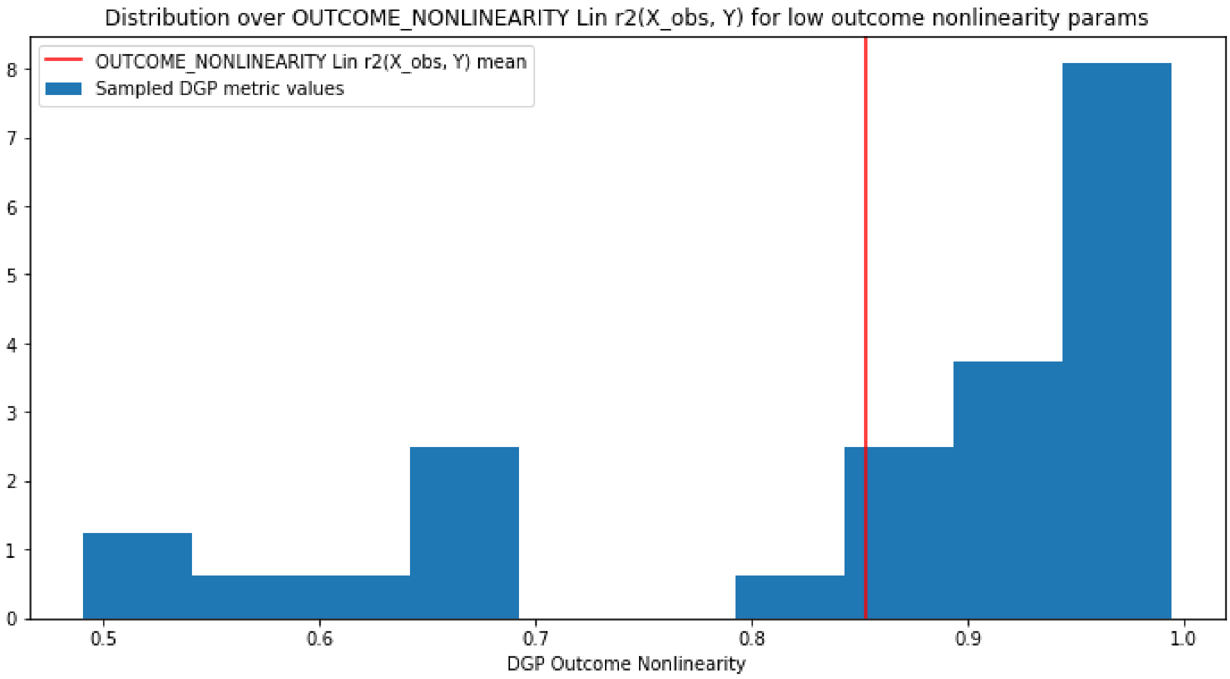
As one might expect, the value of the regression of the observed covariates on the untreated potential outcome is perfect. This makes sense as there is no noise (at this stage). The regression on the observed outcome included noise and the treatment effect and so is not perfect.

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.

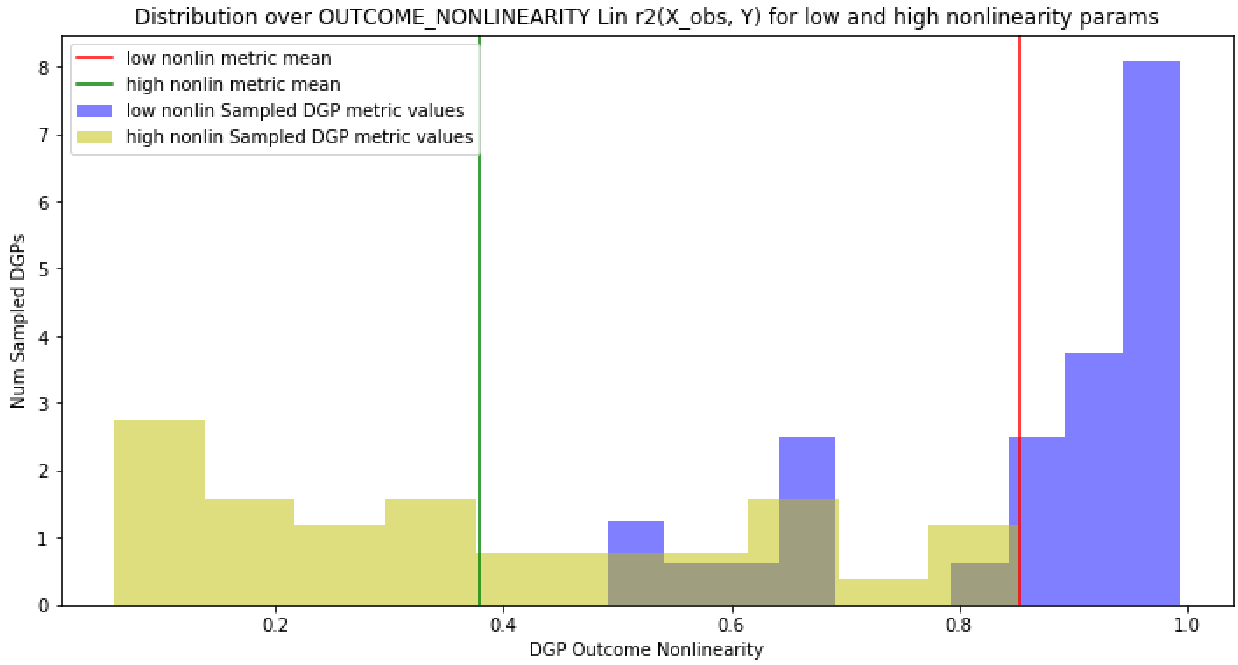
```
[10]:
```

```
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]:
```

```python
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

```
[ ]:
```

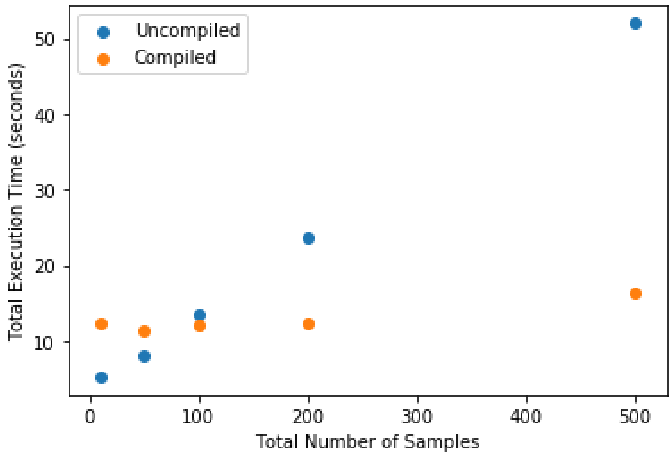
```
from time import time
from collections import defaultdict

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]:
```

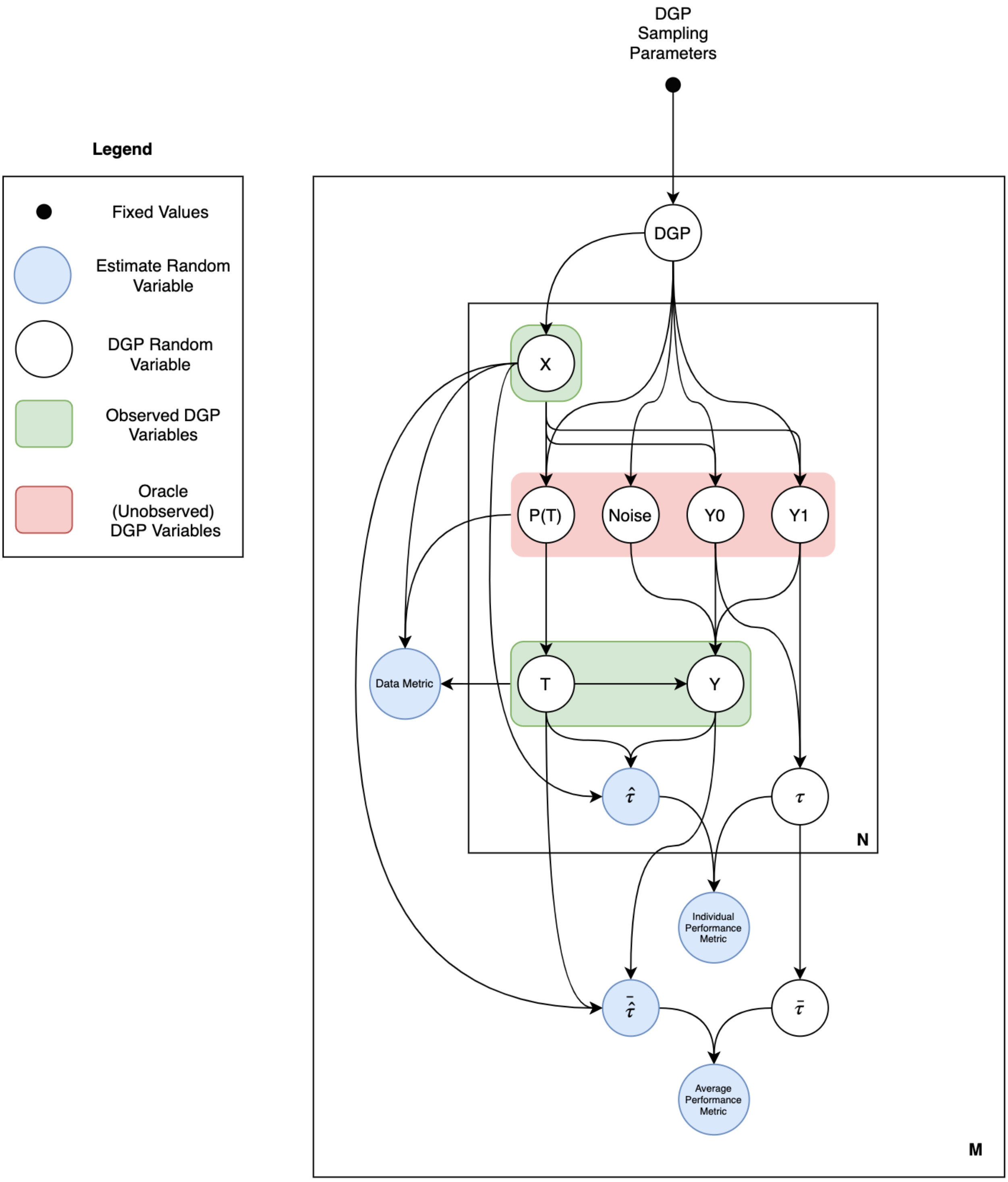
```
0.476
```

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.



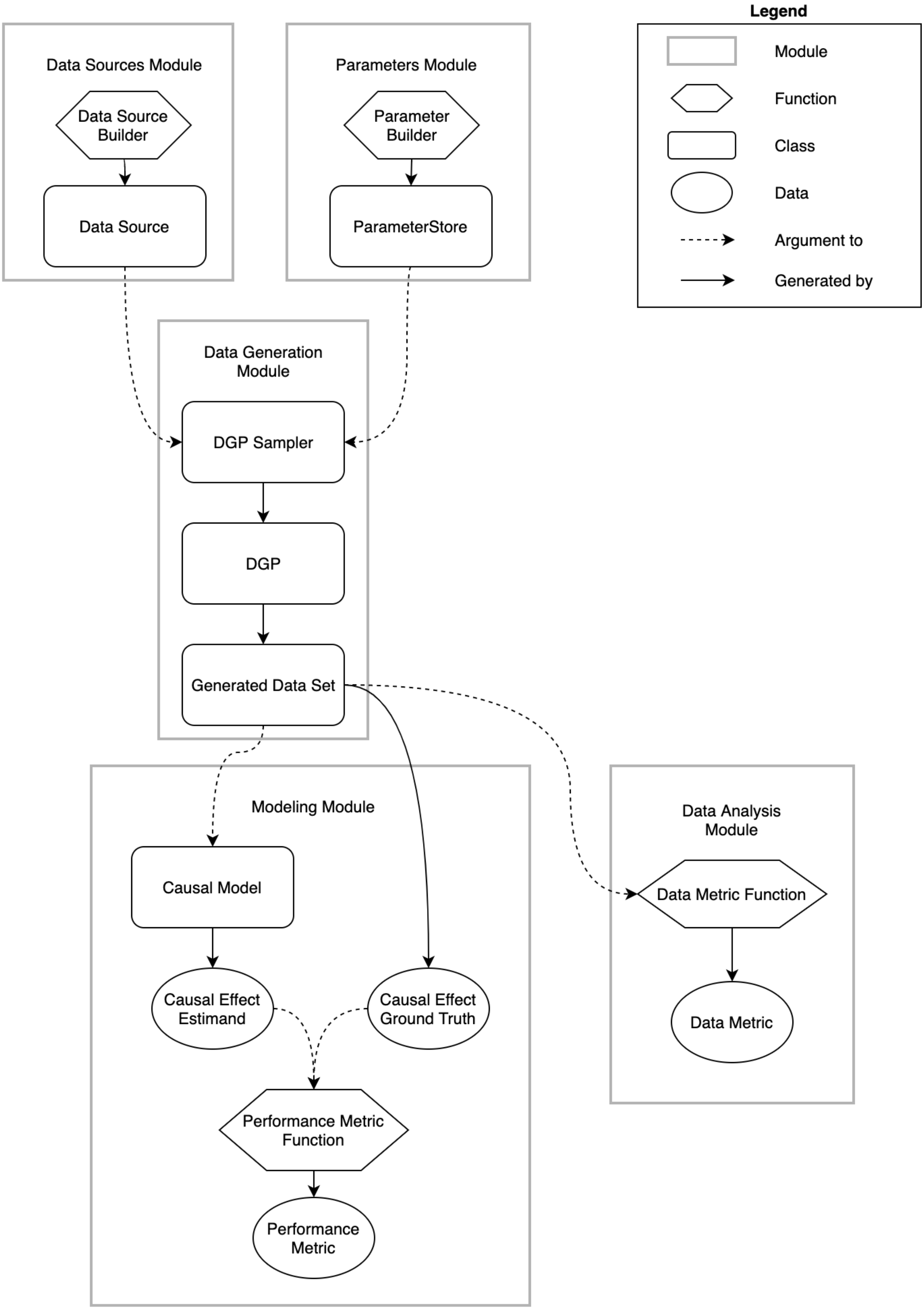
Briefly summarizing the sampling procedure implied by this model:

- 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 ...>
```

Appendix B

HCs and LOs

B.1 HC Application Tags

NOTE: these are rough tags which indicate where the HC is applied. I will invest effort in fleshing these out over the coming 3 weeks.

#**estimation:** Applied to estimating the relative advantage of compilation vs dynamic execution using the Big-O complexity approximation. Documented in Chapter 6.

#**algorithms:** Applied in the construction of the DGP sampling procedure documented in Chapter 6.

#**variables:** Applies extensively in constructing the statistical framework that is used to formalize the concept of a Data Generating Process. See Chapter 3.

#**sampling:** used in construction benchmarks that average over all relevant sources of variance.

#**descriptivestats:** used to calculate performance and data metrics from sampled datasets.

#**distributions:** used extensively in the theoretical design of maccabee (Chapter 5) and the actual implementation.

#**probability:** used extensively in the theoretical foundations in Chapter 2.

#**correlation:** used extensively in the theoretical foundations in Chapter 2 and in the analysis in Chapter 7.

#**breakitdown:** pending

#**dataviz:** used in Chapter 7.

#**gapanalysis:** used in the design of Maccabee (Chap 5) when frameing the design goals in terms of the weaknesses of existing approaches.

#**modeling:** Applies extensively in constructing the statistical framework that is used to formalize the concept of a Data Generating Process. See Chapter 3.

#**testability:** used in Chapter 7.

#**audience:** used in the choice to make real online documentation that is easily consumed.

#**communicationdesign:** used in the choice to make real online documentation that is easily consumed.

#**critique:** used extensively in Chapter 4 to review existing approaches.

#**strategize:** pending

#**selfawareness:** pending

#**confidenceintervals:** used in Chapter 7 when discussing benchmark validation.

#**significance:** used in Chapter 7 when discussing benchmark validation.

#**biasidentification:** used in the review of existing methods when discussing how researchers may introduce bias by selecting functions which suit their own methods.

#**biasmitigation:** used in the review of existing methods when discussing how researchers may introduce bias by selecting functions which suit their own methods.

#**medium:** used in the choice to build online docs.

#**context:** used in the lit review in Chapter 4.

B.2 LO Application Tags

NOTE: because these are custom LOs, their application is fairly obvious. I will write extensive tags soon.

B.2.1 Foundation: Causal Inference

#**CausalInferenceTheory:** applied in Chapter 2.

#**graphicalmodels:** applied in Chapter 2

#**probabilitytheory:** applied in Chapter 2

#**CausalProblemSpace:** applied in Chapter 3

B.2.2 Hybrid MC Benchmarking: Theory/Design

#**CausalBenchmarkTheory:** applied in Chapter 5

#**CausalBenchmarkDesign:** applied in Chapter 5

B.2.3 Hybrid MC Benchmarking: Implementation

#**CodeDesign:** applied in Chapter 6

#**CodeImplementation:** applied in Chapter 6

#**CodeUsability:** applied in Chapter 6

Bibliography

Abadie, A., & Imbens, G. W. [G W]. (2002). Large Sample Properties of Matching Estimators for Average Treatment Effects. *October*, *0136789*(October), 146–146. doi:10.3386/t0283

Abadie, A., & Imbens, G. W. [Guido W.]. (2006). Large Sample Properties of Matching Estimators for Average Treatment Effects. *Econometrica*, *74*(1), 235–267. doi:10.1111/j.1468-0262.2006.00655.x

Athey, S., & Imbens, G. (2016). Recursive partitioning for heterogeneous causal effects. *Proceedings of the National Academy of Sciences of the United States of America*, *113*(27), 7353–7360. doi:10.1073/pnas.1510489113

Athey, S., & Imbens, G. W. [Guido W.]. (2017). The state of applied econometrics: Causality and policy evaluation. In *Journal of economic perspectives* (Vol. 31, *2*, pp. 3–32). doi:10.1257/jep.31.2.3

Athey, S., Imbens, G. W., & Wager, S. (2018). Approximate residual balancing: debiased inference of average treatment effects in high dimensions. *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, *80*(4), 597–623. doi:10.1111/rssb.12268

Athey, S., Tibshirani, J., & Wager, S. (2019). Generalized random forests. *Annals of Statistics*, *47*(2), 1179–1203. doi:10.1214/18-AOS1709

Austin, P. C. (2011). An introduction to propensity score methods for reducing the effects of confounding in observational studies. *Multivariate Behavioral Research*, *46*(3), 399–424. doi:10.1080/00273171.2011.568786

Bishop, C. M. (2006). Pattern recognition and machine learning.

Brooks-Gunn, J., Klebanov, P. K., & Liaw, F. r. (1991). The learning, physical, and emotional environment of the home in the context of poverty: The infant health and development program. *Children and Youth Services Review*, *17*(1-2), 251–276. doi:10.1016/0190-7409(95)00011-Z

Broomberg, J. (2017). *Deep Causal Inference*.

Busso, M., Dinardo, J., & Mccrary, J. (2014). New Evidence on the Finite Sample Properties of Propensity Score Reweighting and Matching Estimators. doi:10.1162/REST{_}a{_}00431

Calder, K. (1953). *Statistical Inference*. Retrieved from https://www.asc.ohio-state.edu/calder.13/stat528/Lectures/lecture21_6slides.PDF

Chen, S., Tian, L., Cai, T., & Yu, M. (2017). A general statistical framework for subgroup identification and comparative treatment scoring. *Biometrics*, *73*(4), 1199–1209. doi:10.1111/biom.12676

Dehejia, R. (2005). Practical propensity score matching: a reply to Smith and Todd. *Journal of Econometrics*, *125*, 355–364. doi:10.1016/j.jeconom.2004.04.012

Dehejia, R. H., & Wahba, S. (1999). Causal Effects in Nonexperimental Studies: Reevaluating the Evaluation of Training Programs. *Journal of the American Statistical Association*, *94*(448), 1053–1062. doi:10.1080/01621459.1999.10473858

Dehejia, R. H., & Wahba, S. (2002). Propensity score-matching methods for nonexperimental causal studies. doi:10.1162/003465302317331982

Diamond, A., Sekhon, J. S., Abadie, A., Brady, H., Caughey, D., Dehejia, R., ... Todd, P. (2012). *Genetic Matching for Estimating Causal Effects: A General Multivariate Matching Method for Achieving Balance in Observational Studies*. Retrieved from http://sekhon.berkeley.edu/,

Dorie, V., Hill, J., Shalit, U., Scott, M., & Cervone, D. (2019). Automated versus Do-It-Yourself Methods for Causal Inference: Lessons Learned from a Data Analysis Competition 1. *Statistical Science*, *34*(1), 43–68. doi:10.1214/18-STS667

Flores, C., & Mitnik, O. (2009). Evaluating Nonexperimental Estimators for Multiple Treatments: Evidence from Experimental Data.

Fraker, T., & Maynard, R. (1987). The Adequacy of Comparison Group Designs for Evaluations of Employment-Related Programs. *The Journal of Human Resources*, *22*(2), 194. doi:10.2307/145902

Friedlander, D., & Robins, P. K. (1995). Evaluating Program Evaluations: New Evidence on Commonly Used Nonexperimental Methods. *American Economic Review*, *85*(4), 923–37.

Frölich, M. (2004). Finite-sample properties of propensity-score matching and weighting estimators. In *Review of economics and statistics* (Vol. 86, *1*, pp. 77–90). doi:10.1162/003465304323023697

Hastings, W. K. (1970). Monte carlo sampling methods using Markov chains and their applications. *Biometrika*. doi:10.1093/biomet/57.1.97

Heckman, J. J., & Todd, P. (1998). *Matching As An Econometric Evaluation Estimator*.

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

Hill, J. L., Reiter, J. P., & Zanutto, E. L. (2005). A Comparison of Experimental and Observational Data Analyses. (pp. 49–60). doi:10.1002/0470090456.ch5

Hirano, K., Imbens, G. W., & Ridder, G. (2003). Efficient estimation of average treatment effects using the estimated propensity score. *Econometrica*, *71*(4), 1161–1189. doi:10.1111/1468-0262.00442

Holland, P. W. (1986). *Statistics and Causal Inference* (tech. rep. No. 396).

Horvitz, D. G., & Thompson, D. J. (1952). *A Generalization of Sampling Without Replacement From a Finite Universe* (tech. rep. No. 260).

Huber, M., Lechner, M., & Wunsch, C. (2013). The performance of estimators based on the propensity score. *Journal of Econometrics*, *175*, 1–21. doi:10.1016/j.jeconom.2012.11.006

Imai, K., King, G., & Stuart, E. A. (2008). *Misunderstandings between experimentalists and observationalists about causal inference* (tech. rep. No. 2).

Imbens, G. W. [Guido W.], & Wooldridge, J. M. (2009). Recent developments in the econometrics of program evaluation. *Journal of Economic Literature*, *47*(1), 5–86. doi:10.1257/jel.47.1.5

Johansson, F. D. [Fredrik D.], Kallus, N., Shalit, U., & Sontag, D. (2018). Learning Weighted Representations for Generalization Across Designs. Retrieved from http://arxiv.org/abs/1802.08598

Johansson, F. D. [Fredrik D], Shalit, U., & Sontag, D. (2016). *Learning Representations for Counterfactual Inference*.

Kallus, N. (2016). A Framework for Optimal Matching for Causal Inference. Retrieved from http://arxiv.org/abs/1606.05188

Kang, J., & Schafer, J. (2007a). Demystifying double robustness: A comparison of alternative strategies for estimating a population mean from incomplete data. *Statistical Science*, *22*(4), 523–539. doi:10.1214/07-STS227

Kang, J., & Schafer, J. (2007b). Demystifying Double Robustness: A Comparison of Alternative Strategies for Estimating a Population Mean from Incomplete Data 1. *Statistical Science*, *22*(4), 523–539. doi:10.1214/07-STS227

Kern, H. L., Stuart, E. A., Hill, J., & Green, D. P. (2016). Assessing Methods for Generalizing Experimental Impact Estimates to Target Populations. *Journal of Research on Educational Effectiveness*, *9*(1), 103–127. doi:10.1080/19345747.2015.1060282

Knaus, M. C., Lechner, M., Strittmatter, A., Graham, B., Santos, A., Schuler, A., ... Zimmert, M. (2018). *Machine Learning Estimation of Heterogeneous Causal Effects: Empirical Monte Carlo Evidence.*

Künzel, S. R., Stadie, B. C., Vemuri, N., Ramakrishnan, V., Sekhon, J. S., & Abbeel, P. (2018). Transfer Learning for Estimating Causal Effects using Neural Networks. Retrieved from http://arxiv.org/abs/1808.07804

Künzel, S. R., Sekhon, J. S., Bickel, P. J., & Yu, B. (2019). Metalearners for estimating heterogeneous treatment effects using machine learning. *Proceedings of the National Academy of Sciences of the United States of America*, *116*(10). doi:10.1073/pnas.1804597116

Lalonde, R. J. (1986). Evaluating the Econometric Evaluations of Training Programs with Experimental Data. *Journal of Chemical Information and Modeling*, *76*(4), 604–620. doi:10.1017/CBO9781107415324.004

Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. doi:10.1038/nature14539

Li, S., & Fu, Y. (2017). Matching on balanced nonlinear representations for treatment effects estimation. In *Advances in neural information processing systems* (Vol. 2017-Decem, pp. 930–940).

Lu, M., Sadiq, S., Feaster, D. J., & Ishwaran, H. (2018). Estimating Individual Treatment Effect in Observational Data Using Random Forest Methods. *Journal of Computational and Graphical Statistics*, *27*(1), 209–219. doi:10.1080/10618600.2017.1356325

Meldrum, M. L. (2000). A brief history of the randomized controlled trial: From oranges and lemons to the gold standard. doi:10.1016/S0889-8588(05)70309-9

Paxton, P., Curran, P. J., Bollen, K. A., Kirby, J., & Chen, F. (2001). Monte Carlo Experiments: Design and Implementation. *Structural Equation Modeling*, *8*(2), 287–312. doi:10.1207/S15328007SEM0802{_}7

Pearl, J. (2009). Causal inference in statistics: An overview. *Statistics Surveys*, *3*, 96–146. doi:10.1214/09-SS057

Qian, M., & Murphy, S. A. (2011). Performance guarantees for individualized treatment rules. *The Annals of Statistics*, *39*(2), 1180–1210. doi:10.1214/10-aos864

Robins, J. M., & Rotnitzky, A. (1995). Semiparametric efficiency in multivariate regression models with missing data. *Journal of the American Statistical Association*, *90*(429), 122–129. doi:10.1080/01621459.1995.10476494

Rosenbaum, P. R., & Rubin, D. B. (1983). The central role of the propensity score in observational studies for causal effects. *Biometrika*, *70*(1), 41–55. doi:10.1093/biomet/70.1.41

Rothwell, P. M. (2006). Factors That Can Affect the External Validity of Randomised Controlled Trials. *PLoS Clinical Trials*, *1*(1), e9. doi:10.1371/journal.pctr.0010009

Rubin, D. B. (1974). Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of Educational Psychology*, *66*(5), 688–701. doi:10.1037/h0037350

Rubin, D. B., & Thomas, N. (2000). Combining propensity score matching with additional adjustments for prognostic covariates. *Journal of the American Statistical Association*, *95*(450), 573–585. doi:10.1080/01621459.2000.10474233

Scharfstein, D. O., Rotnitzky, A., & Robins, J. M. (1999). Adjusting for Nonignorable Drop-Out Using Semiparametric Nonresponse Models: Rejoinder. *Journal of the American Statistical Association*, *94*(448), 1135. doi:10.2307/2669930

Schölkopf, B. (2019). Causality for Machine Learning. Retrieved from http://arxiv.org/abs/1911.10500

Schwab, P., Linhardt, L., & Karlen, W. (2018). Perfect Match: A Simple Method for Learning Representations For Counterfactual Inference With Neural Networks. Retrieved from http://arxiv.org/abs/1810.00656

Shadish, W. R., Clark, M. H., & Steiner, P. M. (2008). Can nonrandomized experiments yield accurate answers? A randomized experiment comparing random and nonrandom assignments. *Journal of the American Statistical Association*, *103*(484), 1334–1343. doi:10.1198/016214508000000733

Smith, J., & Todd, P. (2005). Does matching overcome LaLonde's critique of nonexperimental estimators? *Journal of Econometrics*, *125*(1-2), 305–353.

Su, X., Edu, C., Wang, H., Nickerson, D. M., & Li, B. (2009). *Subgroup Analysis via Recursive Partitioning Chih-Ling Tsai.*

Taddy, M., Gardner, M., Chen, L., & Draper, D. (2016). A Nonparametric Bayesian Analysis of Heterogenous Treatment Effects in Digital Experimentation. *Journal of Business and Economic Statistics*, *34*(4), 661–672. doi:10.1080/07350015.2016.1172013

Tian, L., Alizadeh, A. A., Gentles, A. J., & Tibshirani, R. (2014). A Simple Method for Estimating Interactions Between a Treatment and a Large Number of Covariates. *Journal of the American Statistical Association*, *109*(508), 1517–1532. doi:10.1080/01621459.2014.951443

Wager, S., & Athey, S. (2018). Estimation and Inference of Heterogeneous Treatment Effects using Random Forests. *Journal of the American Statistical Association*, *113*(523), 1228–1242. doi:10.1080/01621459.2017.1319839

Wendling, T., Jung, K., Callahan, A., Schuler, A., Shah, N. H., & Gallego, B. (2018). Comparing methods for estimation of heterogeneous treatment effects using observational data from health care databases. *Statistics in Medicine*, *37*(23), 3309–3324. doi:10.1002/sim.7820