# ABSTRACT

TALAPATRA, KASTURI. Space-filling Exploratory Experimental Design. (Under the direction of Eric Laber and Len Stefanski.)

Monte Carlo simulations are widely used to study and compare statistical methodologies. Space-filling Exploratory Experimental Design (SEED) is a general approach to design extensive experiments to arrive at more general conclusions from a simulation study. SEED obtains performance measures of methods on a very large number of generative models that are systematically varied to ensure coverage in the space of models using mathematical optimization algorithms. Statistical modeling techniques are used to characterize how features of the underlying generative models affect performance. Furthermore through calibration, the fitted statistical model can be used to predict a method's performance on a new dataset. We present a simulation optimization algorithm to search for feature values where methodologies outperform or break down. The issue of reproducibility in research is addressed in SEED by using object oriented programming to design and implement the SEED framework. The modular code structure in SEED makes the simulation experiment easy to understand and modify; and all experimental details are systematically recorded to facilitate reproducibility.

Space-filling Exploratory Experimental Design

by
Kasturi Talapatra

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Statistics

Raleigh, North Carolina

2014

APPROVED BY:

| | |
|---|---|
| Brian Reich | Hua Zhou |
| Eric Laber<br>Co-chair of Advisory Committee | Len Stefanski<br>Co-chair of Advisory Committee |

# DEDICATION

This dissertation is dedicated to my husband Parag. Thank you for your tireless support, unconditional love, and many sacrifices.

# BIOGRAPHY

The author was born in New Delhi, India. She attended University of Delhi from 2002 to 2007 and graduated with a bachelors degree in statistics, and a masters degree in applied operational research. After her studies at University of Delhi she worked with Hewlett Packard Company as a business analyst for two years. In 2009 she moved to Raleigh, North Carolina to pursue a Ph.D. degree in statistics at North Carolina State University. Her doctoral research under Dr. Eric Laber and Dr. Len Stefanski focused on designing a framework to conduct rigorous statistical simulation experiments. After graduation she plans to pursue an industrial career in Cary, North Carolina.

# ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisors Dr. Eric Laber and Dr. Len Stefanski for their guidance, encouragement, and support. Their technical and editorial advice was essential for the completion of this dissertation. They showed me the way to approach research and helped me improve my research skills. They are a source of inspiration and my role models of what a statistician should be.

I would like to sincerely thank my committee members, Dr. Brian Reich and Dr. Hua Zhou for their valuable feedback on my research.

I would like to thank my graduate student friends, Mi Zhou, Kristin Linn, Yiwen Zhang, Gina Maria Pomann, Alana Unfried, and Xiaofei Bai, who made this journey richer and more meaningful. I am deeply grateful to my friend Geetha Rao who made me feel at home in Raleigh and supported me like a family member.

Finally and most importantly, I would like to thank my parents for their love, support, teachings, and for always believing in me.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

# Introduction

In this chapter we will introduce Space-filling Exploratory Experimental Design (SEED), a framework for designing simulation studies. Monte Carlo (MC) simulation studies are an invaluable tool to study finite sampling properties of statistical methods. However, typical MC experiments have serious drawbacks that often lead to simulation results not being trusted fully. These drawbacks are discussed in detail later in this chapter. SEED provides a solution for some of the common problems found in typical MC experiments. The issues we have tried to address with SEED belong to a wide variety, with substantial research potential in each of the problem areas. While SEED does not address all the issues discussed in this chapter, we believe that SEED is a step toward better design, evaluation, and dissemination of simulation studies.

In Section 1.1 and 1.2 we discuss the aspects of generalizability, reproducibility and repeatability in research, that are the motivations behind SEED. In Section 1.3 SEED is introduced in detail. In the final section of this chapter we review the state of the art for simulation studies and reproducible research.

## 1.1 Generalizable, reproducible and repeatable research in science

We define generalizable research as research where qualitative findings from a scientific study can be extended to a larger population of interest based on a representative sample studied (Forster, 2000). The larger the sample, the more it can be generalized to the population it represents. Variables and settings used in an experiment need to be carefully examined before a researcher can make claims of a generalizable result. For example if the research question is, "What percentage of people in North Carolina support the Republican party", a generalizable result needs a study that considers a representative sample of the population in North Carolina, free of sampling bias (`https://www.iwh.on.ca/wrmb/generalizability`; `http://writing.colostate.edu/guides/guide.cfm?guideid=65`). By definition generalizable research findings are applicable to a wider variety of situations, and are more likely to be adopted by others, and contribute to scientific progress.

Reproducible research is defined to be research that contains a sufficiently detailed description of the scientific and experimental process, such that all reported experiments can be reconducted based on the provided information; and doing so leads to the same qualitative conclusions as reported in the original work. In a laboratory setting, to check for reproducibility in experimental findings, a new experiment would be conducted in a different laboratory, by a different researcher and by using different apparatus (ASTM, 2014). Repeatable research is defined to be research where reconducting the experiment in identical conditions would lead to matching numerical results, within reasonable bounds for numerical tolerance. In a laboratory setting, a repeatable experiment is one where an experiment repeated by the same operator, in the same laboratory, and using the same

apparatus, leads to very similar numerical results (ASTM, 2014). Thus reproducibility provides stronger evidence than repeatability, that the reported results are correct and valid.

Both reproducibility and generalizability are important aspects of research. Generalizable results imply high potential impact, but it needs to be reproducible and repeatable to be truly meaningful. Reproducible results that apply to a small number of specific cases considered in the study, may be limited in their scope.

The following quote from Donoho (2010) is pertinent to understanding the aspects of reproducible and repeatable research:

" An article about computational science in a scientific publication is *not* the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment, and the complete set of instructions that generated the figures. "

The importance of reproducible scientific research has garnered a lot of attention recently. A 2010 New Yorker article by Jonah Lehrer highlights how many experimentally established findings do not seem to hold up to repeated investigation. Ledgerwood (2014) cites many scandals involving lack of reproducibility in the fields of psychology, cancer research, neuroscience, and genetics. Transparency and reproducible research has a higher chance of a lasting impact and becoming widely adopted. Section 1.4.2 contains more on the problems caused by lack of reproducibility and repeatability in research and some solutions from the literature.

In the last decade or so, various initiatives have been announced by scientific journals to address the problem of irreproducibility. Along with every original published article, Annals of Internal Medicine provides information regarding how repeatable the article is, by indicating if the study protocols, code, and datasets used are freely avail-

able. Biostatistics favorably marks every article D and/or C if the data and/or code used is made available; and R if the results were repeatable using the provided data and code. Many others like the like Science, Nature, Biometrical Journal, IEEE Transactions on Signal Processing, and The Insight Journal have introduced similar initiatives to propagate a culture of greater transparency and repeatable research in science (`http://reproducibleresearch.net/`).

## 1.2 Generalizable, reproducible and repeatable research in statistics

In statistics we define generalizable research as the property that qualitative results from a study can be extended to a larger population of interest based on the samples used, or the assumptions used in the study. Research studies that arrive at general conclusions about the methods under consideration, examining where methods perform both well and poorly, are referred to as generalizable studies. Reproducible research in statistics, is defined as research that reports a sufficiently detailed description of the methodologies and experimental process using which a reader can reproduce all claims and reach the same qualitative conclusions as presented in the original report. Repeatable research in statistics is research where all claims and results can be regenerated by repeating the exact steps specified by the author. Reproducibility is used in a more general sense than repeatability. To test if a research is reproducible, the reader may work from first principles to reproduce the reported results, whereas to test for repeatability the reader would check if the reported results are obtained by repeating the exact steps, in sequence, as described by the author. This distinction is clearer when we discuss reproducibility

and repeatability for simulation experiments.

### 1.2.1 Theoretical results

A clear detailed proof for a theoretical result is intrinsically repeatable and reproducible. Historically many methodological papers in statistics used theory to establish validity of their methods and then used simulation studies to demonstrate feasibility and underscore theoretical results. However studying properties of statistical methods analytically is difficult, without making assumptions that are too restrictive to be generally informative. Even in cases when analytic derivations are possible, they almost always are asymptotic (large sample size) in nature, and sometimes of questionable relevance in finite samples. Thus even though theoretical results are generalizable to the populations as defined by the assumptions of the theorems, their usefulness is limited if the assumptions are restrictive.

### 1.2.2 Simulation results

MC simulation experiments are used in many scientific fields like statistics, bio medicine, signal processing, operations research, and epidemiology, to study the behavior of processes and methods. In statistics these experiments are generally used to compare competing methodologies. For example, while evaluating a statistic we would be interested to know its bias and precision, among other properties of its sampling distribution. MC studies can be also be used to validate if asymptotic results hold in finite samples and test the sensitivity to violations of assumptions. Thus simulation experiments help us study the operational characteristics of estimation and inference procedures, when theoretical results may be impossible to obtain without strong assumptions. However our simulations experiments do not always seem to be held to the same rigor as our theoretical

results.

In a simulation study we must specify a series of generative models that are used to generate data upon which the methods of interest will be evaluated. Generalizability of results from simulation experiments is closely related to the choice of these generative models. Simulation studies that cover a large range of generative models rather than a few models provide a more informative assessment of a method's properties and performance. Simulation experiments in many research papers consider only a handful of generative models because of convenience. Thus conclusions from the experiment are limited to the few cases considered. Worse still, the few generative models considered in the typical research papers may be chosen because the proposed methodology works well in these settings. If a theoretical proof relies on strong assumptions, it is criticized. The same rigor should be applied when restrictive generative models are used to obtain simulation results.

Typical simulation experiments are also hard to generalize because formal analysis of the results is rarely done. In many settings the performance of a proposed statistical method is uniformly better than that of competitors in the simulation examples considered so no analysis seems warranted. However, uniformly better performance is almost always a symptom of an incomplete set of generative models.

In simulation experiments reproducible research implies that the research report contains sufficiently detailed description of the methodologies and generative models used in the experiment, such that the simulation experiment can be coded from scratch, and the results from such an experiment leads to conclusions that are qualitatively the same as given in the research report. A repeatable simulation study implies that executing the provided simulation code would generate the same exact results as reported in the study. For statistical simulation experiments reproducibility also involves examining results for

input variables and generative models different from the ones used originally, and requires new code to obtain the reported results. Therefore reproducibility implies repeatability but not vice versa, provided the simulation code used is obtain the reported results is made available by the researcher. Reproducibility may also include tests for sensitivity to random number generators and seeds used in the experiment.

### Importance of generalizable, reproducible and repeatable simulation experiments

A lack of vigorous standards for simulation studies can incentivise selective reporting of results. A method that has strikingly good empirical performance relative to established competitors might be viewed favorably by referees and editors increasing its chance of publication. However a deluge of papers with statistical methods aimed at achieving the same goal (e.g., model selection) each outperforming the other on their respective simulated examples presents a logical inconsistency that leads to skepticism and distrust of simulation results. If this distrust becomes widespread, and some believe it already has, then simulated experiments will come to be viewed as either: (i) a best case scenario for the proposed method; or (ii) meaningless. In either case a carefully designed and analyzed simulation study illustrating both where a proposed method does well and where it does not may not be reviewed positively by reviewers.

Reproducibility is a corner stone of science, without which there can be no real scientific progress. Transparency protects researchers against accusations of academic dishonesty; increases the impact of published results; and helps to train junior researchers, graduate students, etc. In statistics, simulation study results found in published articles are often laborious to reproduce. Experimental details provided in the papers are commonly insufficient for numerical results to be verified independently (Burton et al.,

2006b). Many research studies are not repeatable because the code is not provided by default (Peng, Dominici and Zeger, 2006). Even if code is provided it is hard to understand and relating it to the described methods can be difficult (Gentleman and Lang, 2007). Limited or omitted description of non statistical components of the simulation study, including software version, random number generators, seed, etc. also makes typical MC studies hard to repeat or reproduce. For these reasons simulation results are not always trusted fully. This is ironic because we increasingly rely on them to assess performance and compare methods. Lack of reproducibility and repeatability of simulation studies combined with the fact that typical MC experiments are rarely generalizable, is the grave problem we attempt to address using SEED.

## 1.3 SEED

SEED is a framework for generalizable, reproducible and repeatable simulation experiments in statistics. A SEED experiment aims to do the following:

- *Obtain more general conclusions using simulation studies, and improve generalizability of results*: Use a large number of carefully chosen generative models and obtain performance indicators of the methods being studied for each generative model in the MC experiment. Statistical analysis is used to formally model the effect of features of the generative model on a method's performance. Stress testing of methods to find feature values for which a method breaks down or outperforms, gives a clearer picture of its performance, and aids deeper understanding of methods.

- *Facilitate reproducibility and repeatability*: The SEED framework is designed using

object oriented programming (OOP) principles, where each of the major steps of a simulation experiment is carried out in a specified part of the code. Changes to one part has minimal or no effect on the other parts of the SEED code. This makes reproducible research easier. For example a new method may be compared to already existing methods in a SEED experiment, by making changes to only one part of the SEED framework. The modular SEED framework makes it relatively simple to follow, that makes repeatability easier.

### 1.3.1 Generalizability in simulation studies

Let us study the issue of generalizability by taking a closer look at generative models used in MC experiments first. Some notations and definitions used with respect to generative models in SEED is as follows. Let $\theta = (\theta_1, \theta_2, \ldots, \theta_k) \in \Theta$ denote the set of parameters specifying a generative model. Denote by $\Theta$, the space corresponding to the domain of the parameters in $\theta$. Define $f(.) : \Theta \mapsto \mathcal{S}$, $\mathcal{S} \subseteq \mathcal{R}^d, d = 1, 2 \ldots, k$; $f(\theta) = (f_1(\theta), f_2(\theta), \ldots, f_d(\theta))$ as the set of all features of interest in the simulation experiment. Let the space of interest corresponding to the features be $\mathcal{S}^* \subseteq \mathcal{S}$. Consider an experiment involving the random variables $U_1, U_2$, and $U_3$, where $U_i$ is generated from a Uniform$(a_i, b_i)$ distribution, $i = 1, 2, 3$. Here the parameters of the generative model are $a_1, a_2, a_3, b_1, b_2$, and $b_3$. Here $\Theta = \{a_i, b_i : i = 1, 2, 3; a_i \leq b_i; a_i, b_i \in \mathcal{R}\}$ In this example suppose the researcher is interested in studying a method that depends on different values of the probability of the event $U_1 \geq U2 + U3$. Hence the feature of interest $f(\theta) = \Pr(U_1 \geq U2 + U3)$ and $\mathcal{S} \equiv \mathcal{S}^* \equiv [0, 1]$.

For each generative model, there are corresponding values of parameters and features. Once the class of distributional families to be used in the generative model are decided,

specifying the values in $\theta$ defines the generative model, thus data can now be generated from it in the experiment, as long as restrictions on $\theta$ are met.

**Example 1.3.1.** *Comparison of two estimators for a population mean*: Given a sample $X_1, X_2, \ldots, X_n$ from an unknown univariate distribution $h_X(x)$, denote

$$\overline{X} = \frac{\sum_{i=1}^{n} X_i}{n}, \tag{1.3.1}$$

$$\overline{X}_{tr} = \frac{\sum_{i=[\alpha n]+1}^{[\beta n]} X_{(i)}}{[\beta n] - [\alpha n]}; \tag{1.3.2}$$

where $X_{(1)}, X_{(2)}, \ldots, X_{(n)}$ denote the order statistics of the sample, $0 \leq \alpha \leq \beta \leq 1$, and [.] is the greatest integer function (Stigler, 1973). The estimators $\overline{X}$ and $\overline{X}_{tr}$ estimate the unknown expectation $\mathbb{E}(X)$. The researcher is interested to compare the mean squared errors (MSE) of $\overline{X}$ and $\overline{X}_{tr}$ in a simulation experiment, when the sample points $X_1, X_2, \ldots, X_n$ come from a distribution with heavy tails and high skewness. In this case the researcher has to decide the form and parameter values of one univariate distribution $h_X(x)$, to specify the generative model. Suppose the researcher decides to consider a mixture of four Gaussian distributions as the generative model. Hence $\theta = (p_i, \mu_i, \sigma_i; i = 1, 2, 3, 4)$, where $p_i$,$\mu_i$, and $\sigma_i$ denote the weight, mean, and standard deviation respectively, of the $i^{th}(i = 1, 2, 3, 4)$, individual Gaussian distribution in the mixture. The researcher is interested to compare the two estimators $\overline{X}$ and $\overline{X}_{tr}$ for different values of skewness and kurtosis of the generative model, thus the features of interest are $f(\theta) = (\text{skew}_X, \text{kurt}_X)$.

Note that obtaining generative models corresponding to a specific value of $f(\theta)$ is often not directly possible. In Example 1.3.1 the form of the generative model is relatively simple, but obtaining values of $\theta$ for specific values of $f(\theta)$ is not straightforward. This

is illustrated in Chapter two.

As methods become more complex, it becomes harder to choose generative models for a simulation study. One of the reasons for this is that generative models required to study such methods are complex. Constructing features of interest for evaluating a complex method tends to be difficult to express as a function of the parameters of the generative model. Consider the following example.

**Example 1.3.2.** *Comparison of two methods to estimate a dynamic treatment regime in personalized medicine (Laber, Linn and Stefanski, 2014)*: This example is studied in detail in Chapter four. Consider a two-stage randomized treatment trial, with two possible medical treatments coded as -1 and 1, at each stage. For each patient we define $(X_1, A_1, X_2, A_2, Y)$. Here $X_t$ is the vector of length $p_t$ of observed covariates at stage $t$, $A_t \in \{-1, 1\}$ is the treatment assigned at stage $t$ $(t = 1, 2)$, and $Y$ is the observed scalar outcome at the end of the second stage, with larger values implying better results for the patient. At each stage $t$, a decision rule dictates the treatment that should be given to a patient at stage $t$, depending on the observed information until stage $t$, for $t = 1, 2$. At stage one, the treatment decision depends on $X_1$, and at stage two, the treatment decision depends on $(X_1, A_1, X_2)$. A dynamic treatment regime is a sequence of decision rules for assigning treatments to a patient at different stages. A dynamic treatment regime is called the optimal regime if it optimizes the expected outcome of interest (Laber, Linn and Stefanski, 2014). The optimal regime is unknown and is estimated using different methodologies such as $Q$ learning or $IQ$ learning. The details of $Q$ learning and $IQ$ learning are unimportant at the moment, however notice that in order to compare the two methods using a simulation experiment, the generative model involved would be much more complex than in Example 1.3.1. Moreover the distribution of $X_2$ depends

on $X_1$ and $A_1$, and the distribution of $Y$ depends on $X_1$,$A_1$,$X_2$, and $A_2$. Thus deciding the cases to consider is much more difficult than it was in Example 1.3.1. As generative models grow is complexity, it is common for typical simulation studies to consider very simple cases of $f(\theta)$, like $p_t$, the length of the vector $X_t, (t = 1, 2)$. In Chapter five we consider different features ($f(\theta)$) and parameters ($\theta$) of generative models for this example.

Simulation experiments found in most research studies involve generative models that are complex in nature, like in Example 1.3.2. In many such cases no clear justification regarding the choice of the generative models and its parameter values is given (Burton et al., 2006a). This leads to the cases being studied so specific that findings from the simulation study are difficult to generalize.

**Uniform space-filling designs**

In a SEED experiment we consider a very large number of generative models, instead of a handful of them. A researcher first identifies features that are of interest for the method being studied. This may be a calculated guess based on theoretical properties or preliminary simulations. Generative models in SEED are varied in a systematic manner to uniformly cover the feature space of interest, $\mathcal{S}^*$, using a flexible class of generative models. In Chapter two the uniform space-filling design is presented in detail.

Performance of a method is evaluated for every generative model using MC simulations. At the end of the experiment statistical modeling techniques are used to identify interesting relationships between features of the generative models and the methodology under study. We use the term metamodel to refer to statistical models where the response variable is the performance of a methodology (or comparative performance of multiple methodologies), and the explanatory variables are the features of the generative

model under which the performance was calculated. Therefore conclusions from a SEED experiment drawn using data from a large number of generative models and metamodels, corresponding to a feature space of interest, is much more generalizable. This approach is thus much more informative than a table of results given in the case of typical simulation studies.

Metamodels obtained using SEED can be used for performance prediction of methods on a real dataset. The generative models can be chosen in a way to include the data ranges in the observed dataset. Next performance measures are obtained on a massive number of generative models. Using appropriate metamodels, we can then predict the performance for observed real data.

Another use of SEED is finding suitable values for tuning parameters. Many statistical methods use tuning parameters, such as parameters to govern the amount of regularization in penalized estimates, number of clusters in $k$-means cluster analysis, number of neighbors in $k$-nearest neighbors and many other statistical learning techniques. There is not always good scientific theory guiding the choice of tuning parameters. This choice is important because it effects the performance of methods. At times when a new method is proposed and a simulation study is presented to illustrate its performance, certain values of the tuning parameters are chosen without an explanation of how they should be chosen in general. Although we have not considered such examples in this work, the SEED framework is general enough to provide an approach to study the effect of tuning parameter values on a method's performance, by setting tuning parameters as the features of interest.

SEED can also be used to do a factorial experiment with the features as factors. Different levels of the features of the generative model, $f(\theta) = (f_1(\theta), f_2(\theta), \ldots, f_d(\theta))$ are considered and values of the parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$ corresponding to those

factor levels are obtained. A full factorial experiment is a kind of uniform space-filling experiment with the space of interest being the different factor levels of the elements of $f(\theta)$. Note that it may be difficult to control the value of $f(\theta)$ for a generative model directly, since the form of the function $f(.)$ is arbitrary and may not be available in closed form, thus requiring estimation. SEED uses adaptive search algorithms to find a generative model corresponding to a particular value of $f(\theta)$, which is studied in detail in Chapter two.

**Stress testing of methods**

The second part of using SEED to arrive at more general and useful conclusions using MC experiments, is finding conditions of worst or best performance. SEED simulation studies aim to find levels of features where one method performs better than another. Starting with a randomly chosen generative model SEED uses search algorithms to maximize or minimize performance measures comparing two methods, over the space of admissible generative models. This is repeated for multiple starting generative models to arrive at an estimate of the optimal solution. More details are presented in Chapter two.

### 1.3.2 Reproducible and repeatable simulation results

General qualitative conclusions from a simulation study are more likely to be reproducible. So our discussion in Section 1.3.1 in a way also illustrates how SEED facilitates reproducible research. Here we briefly discuss reproducibility and repeatability in SEED from a coding point of view. The underlying structure in SEED is defined using object oriented programming, that facilitates easy changes to various parts of the simulation experiment code. Changes to one part of the code has minimal or no impact on the

other parts. This facilitates reproducible research in the following way. Suppose a reader has studied some standard methods for model selection and has conducted a SEED experiment to compare them. When a new method aimed at achieving the same goal is published, the reader can add code that implements the new method to his or her existing SEED code, by making changes to only one part of the code. Hence the reader can now compare the new method to the other methods for all the features of interest, conduct stress testing for the new method, etc., with minimal effort using SEED.

Along with simulation results from SEED, providing all code used by the SEED experiment helps ensure repeatable results. Running the provided SEED code would regenerate all reported results, as long as version and software requirements are met and the code is correctly written. The SEED code as described in Chapter five contains specific lists, functions, and classes that together with the SEED package code and documentation, provide all details required to repeat experimental results. Changes to values of the features of the generative model, methods being studied, can be made with minimal effort. Considering a feature is also made possible, the details to be provided are described in Chapter five.

The SEED framework controls the inputs and outputs of classes and functions, but not the actual definitions in some cases. Hence it is up to the researcher to write code clearly and with sufficient comments. Ensuring that the methods being studied are implemented correctly is also the responsibility of the user. SEED will not work correctly if the input and output requirements of the pre specified classes and functions, as given in the documentation in Chapter five, are not met. We use the default random number generator in R for the SEED package code. The seeds used in the experiment are recorded along with all simulation experiment settings and results.

In the next section we review the state of the art of simulation experiments and reproducible research, and relate it to the different parts of SEED.

## 1.4 Literature review

We categorize the literature surveyed here into two main categories. The first category includes space-filling designs, simulation optimization, metamodel analysis methods, and overall design of simulation experiments. Works in this category focus on deeper technical details of simulation designs, often concentrating on only one aspect of simulation experiments. The second category of works surveyed discuss the problems due to irreproducibility, and simulation experiments from a reproducible research viewpoint. In some of the works belonging to the first category, certain aspects of MC experiments is explored in greater detail than in SEED. SEED is developed as a more general approach to solving the various problems described in Section 1.2. The works studied here are also greatly relevant for future SEED extensions tailored to specialized research areas.

### 1.4.1 General simulation design and models

Space-filling designs look at different ways to fill the $d$-dimensional input variable space of interest corresponding to $f(\theta)$. Considerations such as total computing cost and prediction quality of metamodels are considered in different types of space-filling designs. Pronzato and Müller (2012) is a survey paper on space-filling designs used in simulation experiments. Different geometric designs and Latin hypercubes are the main approaches studied in this paper. The choice of a Space-filling design depends mainly on the dimension of the feature space, number of feature levels and structural properties desired (Pronzato and Müller, 2012). The methods discussed in Pronzato and Müller (2012)

assume that there are no constraints on the input parameter space to be filled. This implies that for $d$ features of interest, the space corresponding to the domain of the input variables in $f(\theta)$ denoted by $\mathcal{S}^*$ is assumed to be equivalent to $\mathcal{R}^d$, or $\mathcal{S}^* \equiv \mathcal{R}^d$. Maximin and minimax are two geometric designs outlined in the paper, that minimize or maximize distance functions of the input variables. However the geometric approaches do not satisfy good Space-filling properties on subspaces of features (Pronzato and Müller, 2012). This is undesirable because it is common for some features to have an insignificant effect on the performance of methods being studied in the experiment. The authors state that this drawback is partially overcome with Latin hypercube designs that have good Space-filling properties for any one dimensional projection of the input variable space. Pronzato and Müller (2012) provide guidelines on the type of space-filling criterion to use. In SEED we focus on uniformly filling the $d$-dimensional space defined by $f(\theta)$, $d = 1, 2, \ldots, k$. For future work, in cases where $d$ is large and computations for each generative models is expensive, Latin hypercubes would be an interesting option to explore. An advantage of the SEED approach is that unknown constraints in $\mathcal{S}^*$ are automatically taken into account by the search algorithms, since an infeasible combination is discarded. In Section 2.1.2 this property is presented in detail.

Simulation optimization has been used in different contexts, however more commonly it refers to optimization using simulations, in place of mathematical programming. From optimization of computing time to being metamodel based, modern designs using simulation optimization focus on many different criterion (Kleijnen, 2008). In real optimization applications, the objective function is rarely known in closed form. Assuming a closed analytic form for the objective function in such cases is an oversimplification. Simulation optimization is useful in such cases.

The objective function in simulation optimization is random, it needs to be estimated

first in order to be optimized. Since the estimate of the objective function is noisy, it is tricky to apply common optimization algorithms directly. A common approach is to fit an appropriate statistical model to the noisy estimates of the objective function (Gosavi, 2003). For example consider a vector $t = (t_1, t_2, \ldots, t_n)$ and an objective function $g(t)$. The function $g(t)$ contains probability density functions or cumulative density functions, and is difficult to obtain analytically (Gosavi, 2003). Hence $g(t)$ is estimated using simulations, for different values of $t$ sampled carefully from the space of interest (Gosavi, 2003). Next the pairs $(\widehat{g(t)}, t)$ are modeled using different statistical techniques, like linear regression, piecewise regression, response surface methodology (RSM), or neural networks (Gosavi, 2003) to obtain an estimate of the objective function (Gosavi, 2003). Classic RSM uses an estimated response surface as the objective function that has a scalar value, whereas general RSM is designed to handle multiple objective functions and constraints (Kleijnen, 2008).

An advantage of the simulation optimization approach used in SEED is that estimated objective functions are not assumed to have a particular model or structure, the only models assumed in SEED being the generative models. SEED uses a very large number of sample points and estimates the objective function empirically. Paired with parallel processing, SEED is efficient and practical. Using neural networks to model the random objective function as done in Gosavi (2003) provides more flexibility than linear regression, but the estimation becomes time consuming and model verification is necessary. Also neural networks are hard to interpret, unlike the SEED approach which is very intuitive. In Chapter two, the simulation optimization approach used by SEED is explained in detail.

Kleijnen (2008) provides details of different metamodeling methods like univariate and multivariate polynomial regression and kriging. Kriging metamodels, also called

spacial correlation models, model the correlation between outputs, from the simulation experiment, from different generative models (Kleijnen, 2008). Further, validation of metamodels is discussed, using leave-one out cross-validation and coefficient of determination, $R^2$. One of the main concerns discussed in Kleijnen (2008) is the number of generative models to be used in a simulation experiment, when obtaining MC results for each generative model is computationally expensive. Screening designs to identify the most interesting features for an experiment are discussed in the last chapter of Kleijnen (2008), for experiments that have a large number of features that may potentially affect performance. In Chapter two we use a classification tree as a metamodel to analyze the relationship between the features considered and results obtained in the SEED experiment. More generally, the choice of a metamodel for a SEED experiment depends on the research problem, and is left to the user. More details can be found in Chapter two.

Li and Sudjianto (2005) discuss a penalty likelihood method for parameter estimation for Gaussian kriging models for experiments with a small number of MC runs. Ankenman, Nelson and Staum (2010) describe kriging based metamodels for simulation output as functions of controllable input variables and uncontrollable or unknown environment variables. Joseph, Hung and Sudjianto (2008) give a method called blind kriging for meta-models. The authors apply a Bayesian variable selection technique to obtain better predictive performance over ordinary kriging.

The R-package simFrame provides an object oriented framework for simulation experiments and is described in detail in Alfons (2011). Initially developed for simulation studies for survey statistics and missing data problems, this framework uses `S4` classes in R (Alfons, 2011). The examples outlined in Alfons (2011) are for design-based simulations to imitate the realistic scenario of repeatedly sampling from a finite population, as is commonly done in survey statistics, instead of sampling from a distributional model.

Table 1.1: Burton et al. (2006a): A review of simulation experiments given in Statistics in Medicine

| Count | Issue |
|---|---|
| 34 out of 58 | Articles that do not provide justification for data generation |
| 57 out of 58 | Articles that do not state the random number generator used |
| 57 out of 58 | Articles that do not specify the statistical software used to carry out the simulation experiment |
| 57 out of 58 | Articles do not state the dependence of starting seeds |
| 37 out of 58 | Articles used 1000 or less iterations in the simulation step |
| 52 out of 58 | Articles do not provide justification for number of iterations used in the simulation step |

SEED incorporates the `S4` OOP framework in R 3.1.0.

### 1.4.2 Reproducible research

Burton et al. (2006a) discuss the inadequacies found in simulation studies from published papers and the need for better simulation design. Their findings are given in Table 1.1. The authors provide a set of guidelines for simulation experiments to be more systematic, transparent and reproducible. Although the simulation studies considered are mainly from the medical literature, most of the discussion holds true for simulation studies in general.

The topic of reproducible research has received a lot of attention, which is much needed, in the last few years. A Google search on lack of reproducibility in scientific experiments yields more than 4 million results in 0.49 seconds. The areas of biomedical research and cancer research have been focal points of these concerns. Prinz, Schlange and Asadullah (2011) examine the fact that a large number of clinical trials fail in phase II citing lack of efficacy, raising questions about the predictive modeling and target identification done in earlier stages of the trials. Published work for feasible target

identification in the drug discovery process is often relied on heavily (Prinz, Schlange and Asadullah, 2011). But it is common for a pharmaceutical company to conduct their own validation projects that try and reproduce the related published experiments (Prinz, Schlange and Asadullah, 2011). The authors of the article Prinz, Schlange and Asadullah (2011), researchers at Bayer health care, state:

> .. validation projects that were started in our company based on exciting published data have often resulted in disillusionment when key data could not be reproduced. Talking to scientists, both in academia and in industry, there seems to be a general impression that many results that are published are hard to reproduce.

In Prinz, Schlange and Asadullah (2011), results from 67 validation projects were considered. Each of the projects involved reproducing reported results from published literature. The numbers found were dismal. Published results in only 25% of the projects were reproducible in the validation process, even though substantial efforts of multiple employees over multiple months went into the process for each case (Prinz, Schlange and Asadullah, 2011). Lack of resources available to reviewers for detecting errors, and practices that lead to this lack of reproducibility, like clear and complete reporting of conditions has also been discussed in Prinz, Schlange and Asadullah (2011). It has been one of our objectives to shape SEED in a way so that reviewers find it easy to repeat experiments, and also check results for different generative models.

Reproducibility of epidemiological studies is found at Peng, Dominici and Zeger (2006). The authors review 69 published articles from the American Journal of Epidemiology and the Journal of the American Medical Association. In 21 articles, information on how the statistical analysis was implemented on the data is not provided. Code for

statistical implementation or data processing was not available in any of the cases. Peng, Dominici and Zeger (2006) discuss requirements for a reproducible epidemiology study and illustrate an example satisfying all requirements involving data, code to replicate results, and a dynamic document using R and LaTeX.

Rossini, Lumley and Leisch (2003) describe the importance of computing methods in statistical research and reproducible statistical research. Monte Carlo methods are one of the key tools to study methodologies, and take a closer look at the underlying theory. However the use of MC simulations for evaluating methodologies published in papers do not go through a rigorous verification process, like in the case of mathematical proofs, although often implementing the MC experiments can be equally challenging. As a result, tables of numerical results in many publications do not come with a proof of correctness, but play an important role in emphasizing the usefulness of the methodology (Rossini, Lumley and Leisch, 2003). The authors state that verification of MC results can be made possible if all code and details of input models, information of software packages is shared along with the paper. Clear documentation of code is important for this purpose and the need for developing such tools is highlighted in Rossini, Lumley and Leisch (2003). The authors stress on the importance of needing simulation experiments that examine how methods will perform on different generative models, as well as the effect on performance when underlying assumptions are not satisfied. Such simulation experiments can be implemented using SEED. Testing for generalizability of research findings becomes especially relevant when a method is being applied to real data.

Analysis of sensitivity of methods to tuning parameters and data models is important when deciding if reported results are as good as they look. Rossini, Lumley and Leisch (2003) state the following as imperative for reproducing and repeating results:

- a computing environment that is available widely, preferably free, and powerful for statistical analysis.
- All code and data files used to generate results, that are available at website.
- Information on all inputs used, and approaches taken to reproduce the results given in the paper exactly.

The authors also discuss the use of literate programming given in Murdoch and Carey (2001), Knuth (1984), and Leisch (2002) for combining code and its formal documentation. They also stress that providing the above mentioned inputs is not enough if a reviewer requires a high degree of manual effort to reproduce the experiments, thus an automated approach is also highly desired for code verification to be a common practice. With SEED we address a number of issues discussed in this article. We also consider a uniform space-filling design to consider a large number of generative models corresponding to a space of interest, conduct an MC experiment for each of the generative models, and formally analyze results using statistical models. Finally we develop the SEED R package as way to facilitate increased sharing, collaboration, and reproducible results from MC experiments.

Barr et al. (1995) discuss guidelines for computer experiments involving heuristic methods. Heuristic methods have been defined by the authors as "A heuristic method is a well-defined set of steps for quickly identifying a high-quality solution for a given problem, where a solution is a set of values for the problem unknown and "quality" is defined by a stated evaluation metric or criterion. Solutions are usually assumed to be feasible, meeting all problem constraints". As they compare effectiveness of different algorithms, selecting the cases to study is seen an important step in the experiment. If a more general generative model is selected, effective solutions are considered to be

much more valuable. But in many cases of complex methodologies, improvements in effectiveness of a new method is shown for fairly specific cases, which researchers might might hard to prove relevant. The authors of the article Barr et al. (1995) also advocate reporting results for as large a number of features as is feasible, to draw more meaningful conclusions. A feature here is any element in the experiment that can be controlled and may have an impact on the result. This is a key idea in SEED. We consider a large number of generative models that correspond to a range of different levels of features.

As in the case of collecting real sample data in an unbiased and random manner that ensures valid representation of the population, design of experiments principles should be used in computer experiments as well [Barr et al. (1995), Rossini, Lumley and Leisch (2003)]. Inevitably analysis of results from such experiments depends on the experimental design used. Full factorial and Latin square designs have been proposed in Barr et al. (1995) for features in a computer experiment. The authors reiterate the importance of rigorous reporting along with providing program code, and details on inputs and seeds used, in order for a repeatable experiment.

Lenth and Hjsgaard (2011) describe a system Statweave that uses principles of literate programming for repeatable statistical analyses. The Statweave system can work on code written in multiple languages and uses LaTeX or OpenOffice to prepare the final document. Lenth and Hjsgaard (2011) include an example where R and SAS are used together in a data analysis problem. Baumer et al. (2014) describe a software R Markdown, that can be used to teach reproducible data analysis in introductory statistics classes. Since most of the literate programming methods involve a thorough knowledge of R and LaTeX it is not commonly usable for new students of statistics. Nolan and Lang (2010) discuss the need for a bigger role of advanced computational literacy and programming in the statistics curriculum. Statisticians are involved in jobs with an increasingly heavy

dose of programming and methodological applications. Implementation of methods are thus becoming as important as the mathematics behind the methods (Nolan and Lang, 2010). Baumer et al. (2014) have designed R markdown to be a powerful tool for use by undergraduates to work with large datasets, who are new to R programming and LaTeX but are in a computationally intensive statistics curriculum. R markdown is used with R and R studio, and examples are given in Nolan and Lang (2010), along with possible implementation issues. A simpler version of SEED, along with literate programming principles used in R markdown can be a powerful teaching tool in introductory statistics. This can be a potential future research project.

Donoho (2010) describes the observed benefits of reproducible research: greater collaboration and teamwork, more focus on work, greater impact, and continuity in research as training new students for research on a topic explored previously becomes smoother. The author welcomes the repeatability initiative by the journal Biostatistics, such as providing a single R file that replicates all tables and figures reported in the accompanying paper.

Reporting requirements for statistical analyses by the journal Biomarkers is discussed in Lovell (2012). The authors provide a detailed list of guidelines for statistical analysis reporting for different fields in line with reproducible research principles. Specific guidelines have been provided for Animal Research, Genetic Risk Prediction Studies, Meta Analysis, Tumor Marker Prognostic Studies, and Molecular Epidemiology. The journal Biomarkers require use of appropriate experimental design principles for the statistical analyses in the papers, along with sample size justification and power analysis (Lovell, 2012). Lovell (2012) also propose the use of literate programming for documenting data analysis in these scientific fields.

Another step towards reproducible research is found in Wang and Day (2010). This

paper provides an R package for simulation studies for the design of clinical trials. The object oriented programming of `S4` classes in R is used to develop this package, and details of the components, extension details are provided.

In this chapter the motivations, objectives, and principles of SEED was discussed in detail. In the next chapters we will examine implementation of the various aspects of SEED that have been introduced so far.

# Chapter 2

# Generalizable Research using SEED

Monte Carlo (MC) simulation experimentation is invaluable for studying inferential properties of methods in statistics. As methods become more complex, studying properties of methods using theory may be impossible without strong assumptions. Many analytical derivations require asymptotic assumptions, making analytical results of questionable relevance in finite samples. Thus simulation experiments play an increasingly important role in statistical research today.

Simulations experiments in statistics do not always seem to be held to the same rigor as theoretical results. A theoretical result is usually criticized if it relies on strong assumptions, but the same standards are not typically applied to simulation studies, as many published papers contain simulation studies considering a handful number of generative models, and conclusions from such studies are limited to the few generative models considered. In many MC experiments, the choice of generative models is a matter of convenience or worse still, chosen because the proposed methods work well for those models. Rigorous standards for simulation studies can address the issue of selective reporting of results, which if unchecked can lead to skepticism and distrust of simulation

findings.

Statistical analysis of simulation results is rarely done in typical MC experiments. In many cases the performance of a proposed statistical method is uniformly better than that of its competitors in the simulation examples considered so no analysis seems warranted. However, uniformly better performance is almost always a consequence of a small set of generative models.

A generative model is the statistical model used to generate data in an MC study. In most cases performance of methods in an MC study depends on the generative model used. For example the sample mean ($\overline{X}$) being an uniformly minimum variance unbiased estimator for the population mean when the sample is from a normal distribution, would have smaller or equal variance than any other unbiased estimator of the population mean in an MC study, if a normal distribution is used as the generative model. However under a different generative model like a uniform distribution, another unbiased estimator of the population mean may have a lower variance than $\overline{X}$. Considering a large number of generative models that cover a feature space of interest leads to more general conclusions from a simulation study. A feature is a function of the generative model that potentially affects performance of the method being studied in the MC experiment. In many MC experiments only simple factors such as sample size and dimensions of variables in the model are varied to study performance of methods. However a method's theory often points to other features that are likely to have a larger impact on the method's performance, for example the performance of multiple hypotheses testing procedures may be affected by the correlation in test statistics used for testing the hypotheses (Yekutieli and Benjamini, 1999). Considering a range of different values of such features in the MC experiment leads to more useful conclusions from the study. In addition considering feature values where methods break down or perform extremely well, gives a more complete

picture of a method's performance. If conditions where a method is likely to perform poorly is known, appropriate checks can be put in place before using the method in real data applications.

We define a generalizable simulation study as one designed to understand the relationship between features and outcomes of the study, leading to more general conclusions about the performance and properties of the methods under study, in the class of generative models considered. Note that complete generalizability using simulation experiments is impossible. For instance the Central limit theorem (CLT) states that for independent and identically distributed $X_1, X_2, \ldots, X_n$ with mean $E(X_1) = \mu$ and variance $E(X_1) = \sigma^2 < \infty$, $\sqrt{n}(\overline{X} - \mu)$ converges in distribution to a normal distribution with mean 0 and variance $\sigma^2$, as $n \to \infty$. The CLT is completely generalizable for independent and identically distributed $X_1, X_2, \ldots, X_n$ with finite mean and variance. While a simulation study cannot be completely generalizable, SEED is an approach to make simulation studies relatively more generalizable than typical MC studies.

In this chapter we consider a simulation experiment designed to arrive at more general conclusions, using SEED. SEED is a framework to do rigorous statistical simulation experiments; evaluating methods on a large number of generative models, ensuring that the space of features of interest is covered either uniformly or using a factorial design; testing where methods perform outstandingly well as well as badly; and formally analyzing the simulation results. In Section 2.1, we study space-filling experiments in SEED. Simulation optimization in SEED is presented in Section 2.2.

## 2.1 Space-filling experiments

Let $\theta = (\theta_1, \theta_2, \ldots, \theta_k) \in \Theta$ denote the set of parameters specifying a generative model. Denote by $\Theta$, the space corresponding to the domain of the parameters in $\theta$. Define $f(.) : \Theta \mapsto \mathcal{S}$, $\mathcal{S} \subseteq \mathcal{R}^d, d = 1, 2 \ldots, k$; and $f(\theta) = (f_1(\theta), f_2(\theta), \ldots, f_d(\theta))$ as the set of all features of interest in the simulation experiment. We assume the features of the generative model to be scalar functions of the generative model indexed by $\theta$; which can be expressed in closed form in terms of $\theta$ or estimated using simulations. Features can potentially impact the properties or performance of the methods under study, and are chosen based on theoretical properties or preliminary simulations of the methods under study. In SEED, a space-filling experiment is a simulation experiment that considers a large number of generative models and aims to cover the space of features of interest, $\mathcal{S}^* \subseteq \mathcal{S}$.

We consider a toy example of location estimation to illustrate a space-filling simulation experiment. Given a sample of size $n, X_1, X_2, \ldots, X_n$ from an unknown distribution we consider two estimators of the unknown population mean:

$$\overline{X} = \frac{\sum_{i=1}^{n} X_i}{n}, \tag{2.1.1}$$

$$\overline{X}_{tr} = \frac{\sum_{i=[\alpha n]+1}^{[\beta n]} X_{(i)}}{[\beta n] - [\alpha n]}; \tag{2.1.2}$$

where $X_{(1)}, X_{(2)}, \ldots, X_{(n)}$ denote the order statistics of the sample, $0 \leq \alpha \leq \beta \leq 1$, and [.] is the greatest integer function. In this toy example we use $\alpha = \beta = 0.05$.

To compare the performance of these two estimators, we use the estimated ratio of their mean squared errors (MSE), $\text{MSE}(\overline{X})/\text{MSE}(\overline{X}_{tr})$. Both estimators are location invariant. Our objective is to conduct a simulation experiment that leads to general

conclusions regarding the performance of $\overline{X}$ and $\overline{X}_{tr}$.

In the SEED framework, the researcher starts by identifying features that might affect the performance of the methods being studied. In the case of location estimation, the literature suggests that symmetry and heaviness of tails of the $X$ distribution are two such factors. We also explore whether variance affects the performance $\overline{X}$ and $\overline{X}_{tr}$ by considering it as a factor to be varied in the MC experiment. We require a flexible class of generative models in order to cover a range of different values of variance, skewness, and kurtosis. A natural choice of flexible generative models is a mixture of Gaussian distributions since any continuous distribution can be approximated using a finite mixture of Gaussian distributions to an arbitrary degree of closeness (McLachlan and Peel, 2004).

In some cases the features of interest can be varied directly, whereas in others it might not be possible to do so. For example using a Student's $t$-distribution as the class of generative models, we can directly vary the kurtosis by varying the degrees of freedom, however only a small number of kurtosis values can be achieved using the Student's $t$-distribution. Thus, we use a Gaussian mixture distribution as a flexible class of generative models to attain a range of different kurtosis values. For a mixture distribution however, directly controlling the kurtosis by varying the distributional parameters is difficult. Thus, we use an optimization algorithm, that searches for values of $\theta$ in the class of generative models corresponding to a desired value of $f(\theta)$. This is presented in detail in Section 2.1.2.

### 2.1.1 Deriving $f(\theta)$ in closed form

We use a $t$ component Gaussian mixture as the class of generative models in this example. Let $p_i, \mu_i, \sigma_i$ be the probability, mean, and standard deviation respectively of the $i^{\text{th}}$ com-

ponent in the mixture; $i = 1, 2 \ldots, t$. Thus, $X \sim \text{NMix}(p_1, p_2, \ldots, p_t, \mu_1, \mu_2, \ldots, \mu_t, \sigma_1, \sigma_2, \ldots, \sigma_t)$ and $\theta = \{p_1, p_2, \ldots, p_t, \mu_1, \mu_2, \ldots, \mu_t, \sigma_1, \sigma_2, \ldots, \sigma_t\}$. The Gaussian mixture density is given by, $h_X(x) = \sum_{i=1}^{t} p_i \phi(x|\mu_i, \sigma_i)$ where

$$\phi(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right),$$

$\sum_{i=1}^{t} p_i = 1$; $0 \leq p_i \leq 1$ $\forall i = 1, 2, \ldots, k$. The $r^{th}$ moment about the origin is given by, $E(X^r) = \sum_{i=1}^{t} p_i E(Y_i^r)$, where $Y_i \sim N(\mu_i, \sigma_i)$, $\forall i = 1, 2, \ldots, t$. Without loss of generality, we center $X$ at mean 0, and $\mu_i^* = \mu_i - \sum_{i=1}^{t} p_i \mu_i, \forall i = 1, 2 \ldots, t$. Denote $X \sim \text{NMix}(p_1, p_2, \ldots, p_t, \mu_1^*, \mu_2^*, \ldots, \mu_t^*, \sigma_1, \sigma_2, \ldots, \sigma_t)$ without loss of generality. Hence,

$$\begin{aligned}
\text{standard deviation}(X) &= \sigma_X \\
&= \sqrt{E(X^2)} \\
&= \sqrt{\sum_{i=1}^{t} (p_i \mu_i^{*2} + p_i \sigma_i^2)}, \qquad (2.1.3) \\
\text{skewness}(X) &= \tau_X \\
&= \frac{E(X^3)}{E(X^2)^{3/2}} \\
&= \frac{\sum_{i=1}^{t} (p_i \mu_i^{*3} + 3 p_i \mu_i^* \sigma_i^2)}{\sigma_X^3}, \qquad (2.1.4) \\
\text{kurtosis}(X) &= \kappa_X \\
&= \frac{E(X^4)}{E(X^2)^2} \\
&= \frac{\sum_{i=1}^{k} (p_i \mu_i^{*4} + 6 p_i \mu_i^{*2} \sigma_i^2 + 3 p_i \sigma_i^4)}{\sigma_X^4}. \qquad (2.1.5)
\end{aligned}$$

Therefore for this example, $\theta = (p_1, p_2, \ldots, p_t, \mu_1^*, \mu_2^*, \ldots, \mu_t^*, \sigma_1, \sigma_2, \ldots, \sigma_t)$, and $f(\theta) = \{\sigma_X, \tau_X, \kappa_X\}$. Suppose that our goal is to consider generative models where $\sigma_X$ ranges from 0 to 3; $\tau_X$ ranges from $-5$ to 5; and $\kappa_X$ ranges from 0 to 30. Thus $\mathcal{S}^* \equiv [0, 3] \times [-5, 5] \times [0, 30]$. In addition, a constraint on $\mathcal{S}^*$ is found in Johnson (1949) as

$$\kappa_X \geq \tau_X^2 + 1. \tag{2.1.6}$$

This can be shown as follows. Without loss of generality assume $\sigma_X = 1$, then

$$\text{Cov} \begin{bmatrix} X \\ X^2 \end{bmatrix} = \begin{bmatrix} 1 & \tau_X \\ \tau_X & \kappa_X - 1 \end{bmatrix},$$

must be positive definite. Next we attempt to cover $\mathcal{S}^*$ using a Gaussian mixture distribution, our flexible class of generative models.

### 2.1.2 Space-filling in $\mathcal{S}^*$

We would like study the comparative performance of $\overline{X}$ and $\overline{X}_{tr}$ uniformly over the space of interest $\mathcal{S}^*$, for feasible values as given by Equation (2.1.6). Let us first consider a naive approach to get generative models of interest that fill $\mathcal{S}^*$. We start by considering 5000 generative models. The parameters of the generative model are randomly chosen from a wide range of feasible values of $\theta$, using a Uniform distribution. For a Gaussian mixture with four components, $p_1, p_2, \ldots p_4$ are generated from Uniform$(0, 1)$ and scaled to add to 1 hence, we generate uniformly on the set $\{p_i : 0 \leq p_i \leq 1; \sum_{i=1}^4 p_i = 1; i = 1, 2, 3, 4\}$ to obtain the probabilities of the Gaussian mixture. Next $\mu_1, \mu_2, \ldots \mu_4$ are each generated from a Uniform$(-100, 100)$ distribution, and centered to make the expected value of the

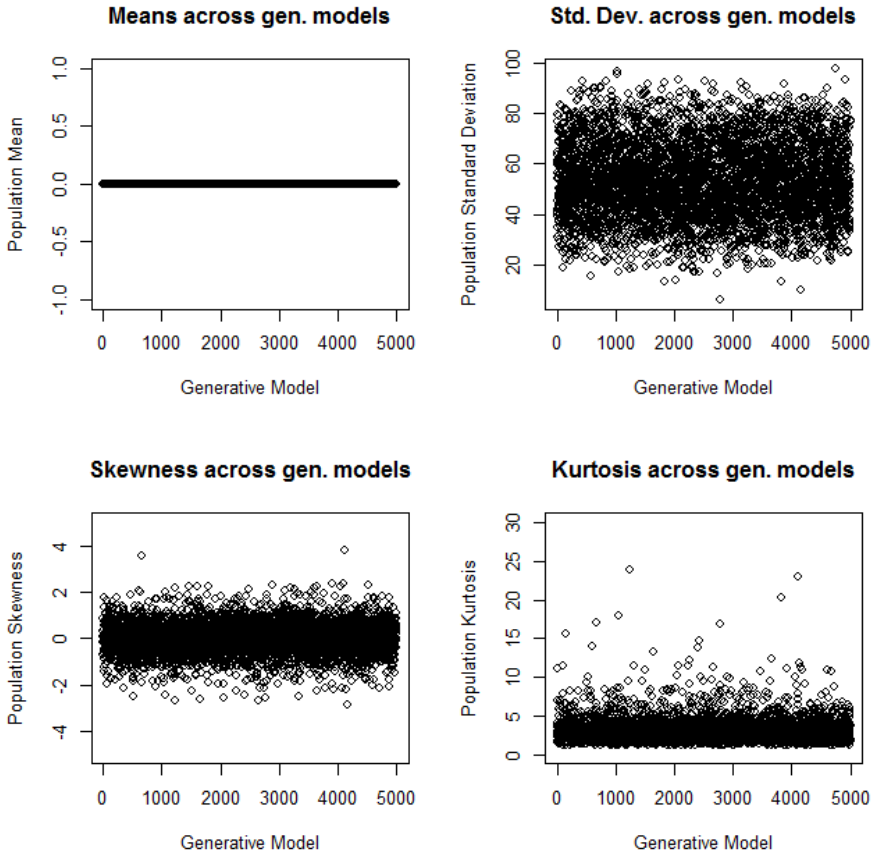


Figure 2.1: Coverage achieved for randomly chosen parameters of the Gaussian mixture with 5000 generative models

Gaussian mixture distribution, $\sum_{i=1}^{4} p_i \mu_i = 0$. Then $\sigma_1, \sigma_2, \ldots \sigma_4$ are each generated using a Uniform$(0, 50)$ distribution. Thus $\theta = (p_1, p_2, \ldots, p_4, \mu_1, \mu_2, \ldots, \mu_4, \sigma_1, \sigma_2, \ldots, \sigma_4)$ is specified, and $f(\theta) = \{\sigma_X, \tau_X, \kappa_X\}$ is calculated for each generative model using Equations (2.1.3), (2.1.4), and (2.1.5). Figures 2.1 and 2.2 show the coverage achieved of $\mathcal{S}^*$ using this naive approach. Considering a wide range of parameters values in $\theta$ using a large number of generative models, does not achieve good coverage in the space of features of interest $\mathcal{S}^*$. Generally uniform coverage of $\theta_1, \theta_2, \ldots, \theta_k$ values, does not imply a

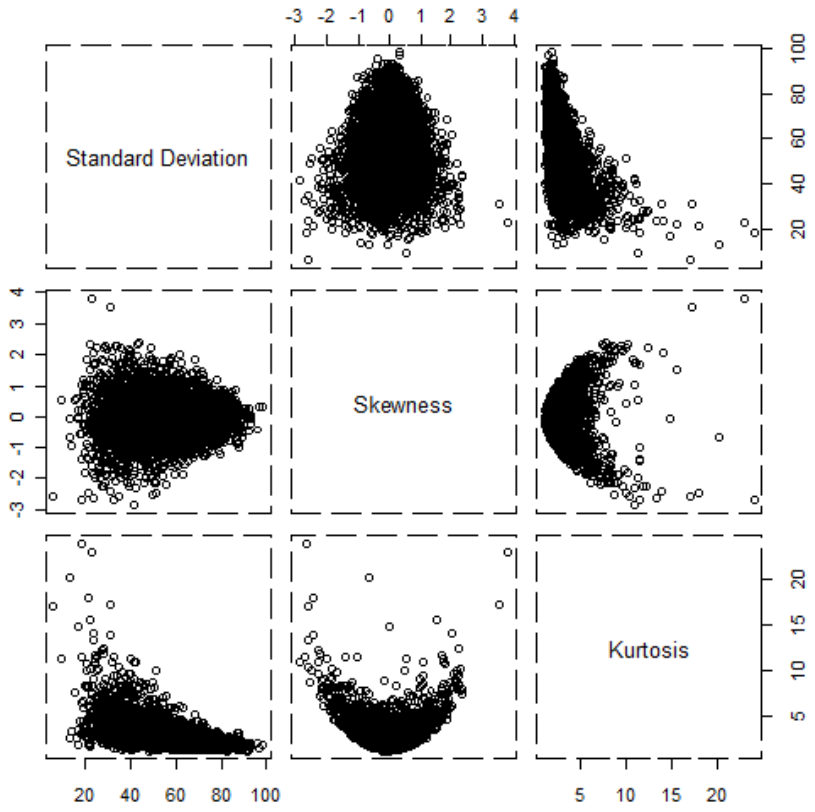


Figure 2.2: Coverage achieved for randomly chosen parameters of the Gaussian mixture with 5000 generative models

similar coverage for the values of $f(\theta)$.

Since the naive approach does not achieve the desired uniform coverage of $\mathcal{S}^*$, we consider a more systematic approach to choosing generative models using optimization algorithms. Solving the following optimization problem:

$$\begin{aligned} \text{Minimize:} \quad & (\sigma_X - \sigma_{\text{target}})^2 + (\tau_X - \tau_{\text{target}})^2 + (\kappa_X - \kappa_{\text{target}})^2 \\ \text{subject to:} \quad & \theta \in \Theta, \end{aligned} \tag{2.1.7}$$

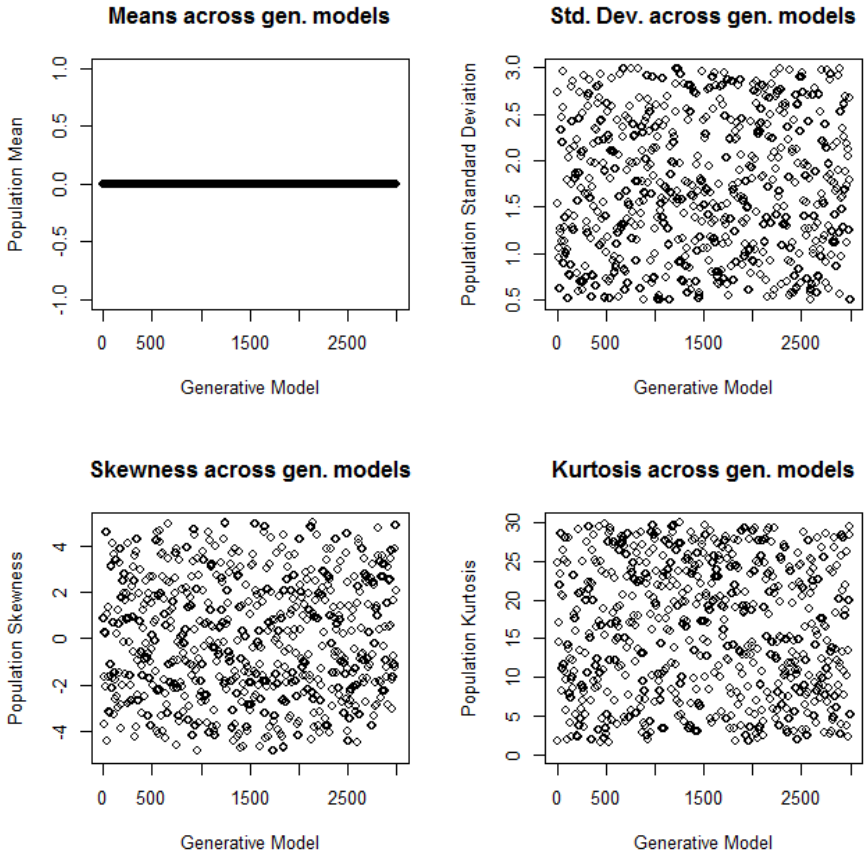


Figure 2.3: Uniform coverage achieved by SEED with 3000 generative models

gives us values of $\theta$, for a target value of $f(\theta)_{\text{target}} = (\sigma_{\text{target}}, \tau_{\text{target}}, \kappa_{\text{target}})$. In this example, the L-BFGS-B algorithm (Byrd et al., 1995) is used to obtain values of $\theta = \{p_1, p_2 \ldots, p_4, \mu_1, \mu_2, \ldots, \mu_4, \sigma_1, \sigma_2, \ldots, \sigma_4\}$ such that the corresponding generative model has $\sigma_X, \tau_X, \kappa_X$ values arbitrarily close to their respective target values $\sigma_{\text{target}}, \tau_{\text{target}}, \kappa_{\text{target}}$. Since the solution to the optimization problem (2.1.7) is not unique, and random starting values for $\theta$ are used, repetitions of the same target $f(\theta)$ value may correspond to different generative models. We choose 600 target feature values $\sigma_{\text{target}}, \tau_{\text{target}}$ and $\kappa_{\text{target}}$ at random from Uniform$(0, 3)$, Uniform$(-5, 5)$, and Uniform$(0, 30)$ distributions respec-

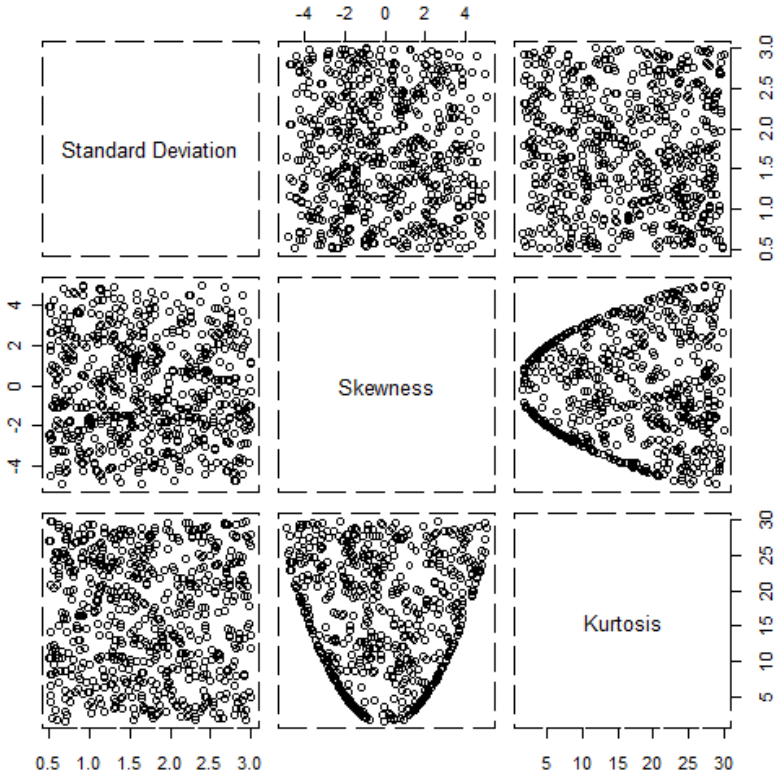


Figure 2.4: Uniform coverage achieved by SEED with 3000 generative models

tively. Each triplet of target values $\sigma_{\text{target}}, \tau_{\text{target}}, \kappa_{\text{target}}$ is repeated 5 times.

The constraints on $\theta$ are taken into account automatically by the optimization problem (2.1.7) since using the formulae given in Equations (2.1.3), (2.1.4), and (2.1.5), no admissible values of $\theta$ exist in $\mathcal{S}^*$ where Equation (2.1.6) is not satisfied. The space of admissible values of $\theta = \{p_1, p_2 \ldots, p_4, \mu_1, \mu_2, \ldots, \mu_4, \sigma_1, \sigma_2, \ldots, \sigma_4\}$ is given by $\Theta \equiv [0, \infty]^4 \times [-\infty, \infty]^4 \times [0, \infty]^4$, specified as constraints in the optimization problem (2.1.7). When Equation (2.1.6) is not satisfied for the target values of skewness ($\tau$) and kurtosis ($\kappa$), the closest possible points on the boundary are obtained. The coverage of

$\mathcal{S}^*$ achieved using the optimization algorithm approach is shown in Figures 2.3 and 2.4.

The performance measure $P = \widehat{\text{MSE}}(\overline{X})/\widehat{\text{MSE}}(\overline{X}_{tr})$ is obtained for each of the 3000 generative models. Define $\text{MSE}(\overline{X}) = \mathbb{E}\{\overline{X} - \mu\}^2$, and $\text{MSE}(\overline{X}_{tr}) = \mathbb{E}\{\overline{X}_{tr} - \mu\}^2$. The expectation is estimated using 10000 MC simulations, and the estimates are denoted by $\widehat{\text{MSE}}(\overline{X})$ and $\widehat{\text{MSE}}(\overline{X}_{tr})$. At noted earlier, in this toy example $\mu = 0$. In the next section we will analyze the relationship between the features $\sigma_X, \tau_X, \kappa_X$ and performance of the two estimators $\overline{X}$ and $\overline{X}_{tr}$ using a statistical model.

### 2.1.3 Performance analysis

Figure 2.5 is a first look at the performance of the two estimators $\overline{X}$ and $\overline{X}_{tr}$ from the SEED experiment. The MSE ratio $P = \widehat{\text{MSE}}(\overline{X})/\widehat{\text{MSE}}(\overline{X}_{tr})$, does not show a specific pattern across different generative models and different values of standard deviation. The performance graph looks symmetric around a skewness of zero. Values of $P$ increase as kurtosis increases. For our reference a standard normal variate has a skewness of 0 and kurtosis of 3 according to the definitions used here. By definition $P > 1$ implies that are $\overline{X}_{tr}$ has a lower estimated MSE than $\overline{X}$. Hence for generative models where the absolute values of skewness and the values of kurtosis are high, $\overline{X}_{tr}$ performs better than $\overline{X}$. Figure 2.6 presents a box plot of $P$, $\sigma_x$, $\tau_x$, and $\kappa_x$. At the median performance value of 1.2, $\overline{X}_{tr}$ has a lower estimated MSE than $\overline{X}$.

Next we choose an appropriate metamodel for the data. We use the term metamodel to refer to statistical models where the response variable is the performance of a methodology (or comparative performance of multiple methodologies), and the explanatory variables are the features of the generative model under which the performance was calculated. The appropriate choice of metamodel is up to the researcher, and depends

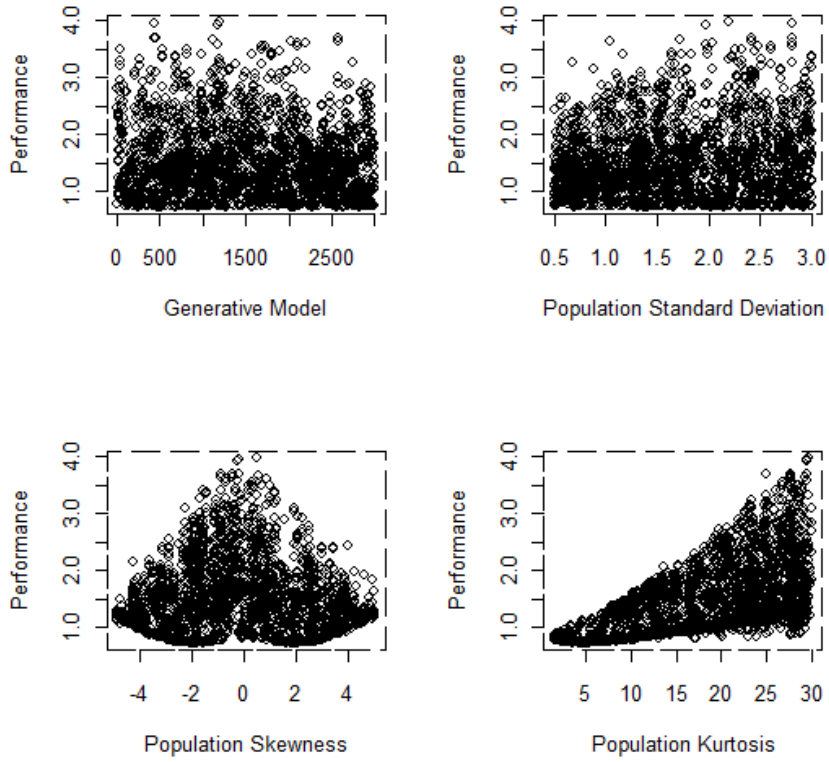


Figure 2.5: Performance defined as $\widehat{\text{MSE}}(\overline{X})/\widehat{\text{MSE}}(\overline{X}_{tr})$in the toy example across 3000 generative models. The MSE values are estimated using 10000 Monte Carlo simulations. Higher performance values are favorable for $\overline{X}_{tr}$.

on the methods being studied. We use a classification tree as the metamodel in this toy example.

A classification tree takes as input one response variable, and one or more predictor variables. In this case the response variable

$$Y_j = \begin{cases} 1 & \text{if } P_j < 1 \\ 0 & \text{otherwise} \end{cases},$$

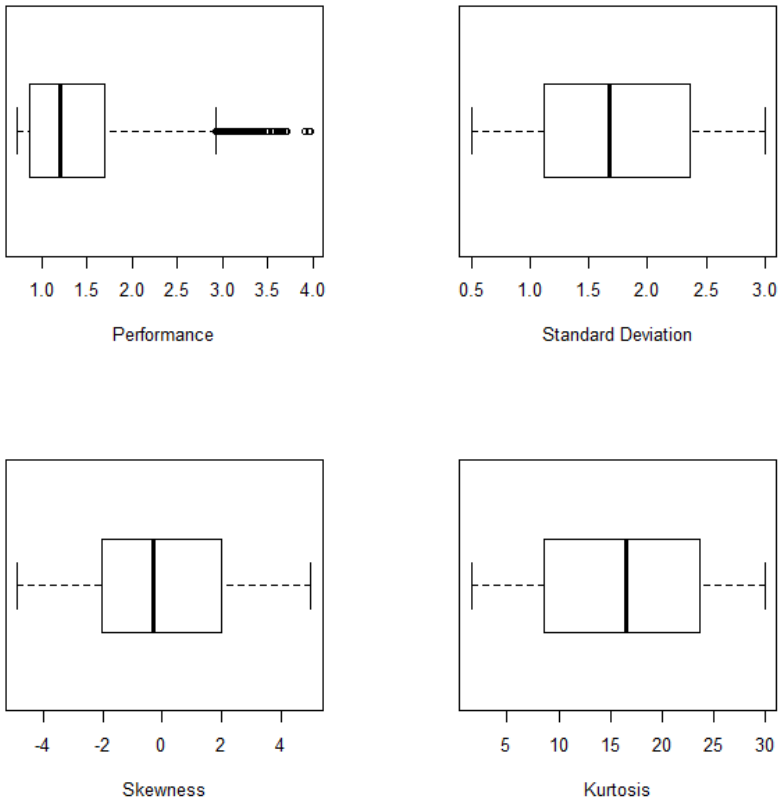


Figure 2.6: Box plots of performance, standard deviation, skewness, and kurtosis in the toy example across 3000 generative models. Performance defined as $\widehat{\text{MSE}}(\overline{X})/\widehat{\text{MSE}}(\overline{X}_{tr})$ is estimated using 10000 Monte Carlo simulations. Higher performance values are favorable for $\overline{X}_{tr}$.

where $P_j$ is the ratio of the estimated mean squared errors from the $j^{th}$ generative model, $j = 1, 2, \ldots, 3000$. By definition, $P_j < 1$ implies $\widehat{\text{MSE}}(\overline{X}) < \widehat{\text{MSE}}(\overline{X}_{tr})$, hence $Y = 1$ implies better performance of $\overline{X}$ compared to $\overline{X}_{tr}$ in the $j^{th}$ generative model ($j = 1, 2, \ldots, 3000$).

Figure 2.7 is a plot of the fitted classification tree. The predictor variables in this classification tree are $\{\sigma_x, |\tau_x|, \kappa_x\}$, with $|.|$ denoting the absolute value function. In

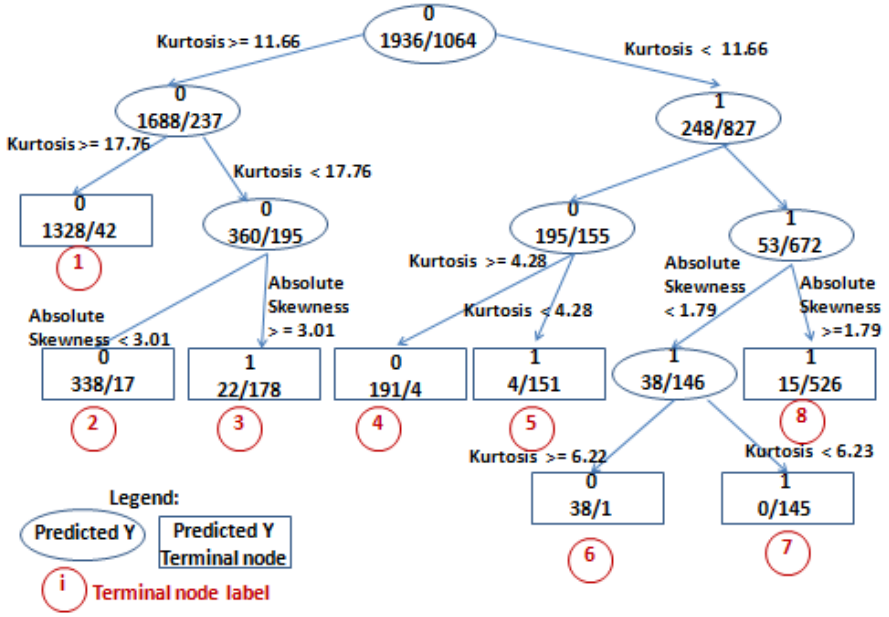


Figure 2.7: Classification tree in the toy example using 3000 generative models. The response variable is defined as $Y = I(\widehat{\text{MSE}}(\overline{X}) < \widehat{\text{MSE}}(\overline{X}_{tr}))$, where $I(.)$ is the indicator function and the predictor variables are $\{\sigma_x, |\tau_x|, \kappa_x\}$, with $|.|$ denoting the absolute value function. For $Y = 1$, $\overline{X}$ performs better than $\overline{X}_{tr}$. The MSE values are estimated using 10000 Monte Carlo simulations.

Figure 2.5 it is found that $P$ is symmetric about a skewness of zero, hence we use $|\tau_x|$ in place of $\tau_x$ as one of the predictors in this model. Using $|\tau_x|$ in place of $\tau_x$ also results in a simpler tree and smaller misclassification rates. We find that values of $\sigma_X$ does not effect $P$ for the class of generative models considered here as the fitted tree does not use $\sigma_X$ as a classification variable in any node. From the terminal nodes we see that our metamodel predicts that $\overline{X}_{tr}$ performs better than $\overline{X}$ for datasets with high kurtosis and high absolute value of skewness. In terminal nodes $3, 5, 7$, and $8$, $\overline{X}$ performs better than $\overline{X}_{tr}$. The number of nodes in the fitted tree corresponds to the smallest ten-fold cross validation error. It is recommended to consider multiple metamodels. Polynomial regression models and regression trees fit to these data, resulted in similar conclusions. However, we finally

choose a classification tree, because of its easy interpretability, low misclassification rate, and low cross validation error rate in the fitted data. The misclassification rate for a node is the proportion of responses out of the total number of responses in that node, where the predicted value is not equal to the observed value. In Figure 2.7, the lowest misclassification rate is 0% for node 4, the highest misclassification rate is 12.36% in node 3 and the overall misclassification rate is 3.63%.

### 2.1.4 Factorial experiments in SEED

A special case of space-filling experiments in SEED is a factorial experiment where features of interest are the factors, and generative models that correspond to different factors levels are considered in the SEED experiment. In a factorial experiment using SEED, the target feature values are factor levels whereas in a general SEED space-filling experiment target feature values are chosen using a Uniform distribution over the ranges considered for each feature of interest. Thus the factor levels of interest are the target features values $f_i(\theta), i = 1, 2, \ldots, d$ for $d$ factors; and generative models are obtained by solving the following optimization problem:

$$\begin{aligned} \text{minimize:} \quad & \sum_{i=1}^{d} (f_i(\theta) - f_{i(\text{target})}(\theta))^2; \\ \text{subject to:} \quad & \theta \in \Theta. \end{aligned} \tag{2.1.8}$$

Statistical theory on design of experiments (DOE) can guide the choice of factor levels or $f_{i(\text{target})}(\theta); i = 1, 2, \ldots, d$. Kleijnen et al. (2005) study the problem of DOE for simulation experiments, the choice of $d$ and levels of each factor $f_{i(\text{target})}(\theta); i = 1, 2, \ldots, d$. Once the levels are chosen, the optimization problem in (2.1.8) is solved to obtain generative models and performance measures are calculated at each generative model using MC

simulations. Performance of methods is analyzed using appropriate metamodels.

This concludes the discussion on space-filling experiments using SEED. Note that in the toy example used in this section, the features of interest exist in closed form as derived in Section 2.1. When $f(\theta)$ is not expressible in closed form, simulation optimization is used to obtain generative models that fill the space of interest $\mathcal{S}^*$. Section 2.2 describes the simulation optimization algorithm used in SEED.

## 2.2 Simulation optimization

Consider the problem of minimizing $l(\theta)$, a function of the parameters of the generative model; subject to constraints on $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$:

$$\begin{aligned} \text{minimize:} \quad & l(\theta) \\ \text{subject to:} \quad & \theta \in \Theta. \end{aligned} \tag{2.2.1}$$

Simulation optimization is one approach to solving the optimizing problem in (2.2.1) when the function $l(\theta)$ is not expressible in closed form, but can be estimated using MC simulations (Gosavi, 2003). In SEED we use simulation optimization in the following two cases.

- Obtaining generative models for the MC study by solving the optimization problem (2.1.8), when the features of interest $f(\theta)$ are not expressible in closed form, but can be estimated using MC simulations. Thus for the optimization problem in (2.2.1), in this case $l(\theta) = \parallel f(\theta) - f_{\text{target}}(\theta) \parallel$, where $\parallel . \parallel$ denotes the Euclidean norm. An example for this is presented in Section 3.2.

- Finding feature values for which methods perform poorly or outstandingly well,

is referred to as stress testing of methods in SEED. In MC studies we estimate a method's performance using simulations, hence the objective function in a stress testing scenario in SEED requires applying simulation optimization. In stress testing $l(\theta) = (P_\theta - P_{\text{Target}})^2$, for the optimization problem in (2.2.1). Here $P_\theta$ is a performance measure of the method under study, obtained using a generative model with parameters $\theta$; and $P_{\text{Target}}$ is the target performance measure. This case is presented in Section 2.2.1.

Since $l(\theta)$ is estimated using MC simulations, it is a function of the data used in the MC simulations. The simulation optimization algorithm used in SEED implements a mechanism such that for equal values of $\theta$, the estimate of $l(\theta)$ found using MC simulations is equal. Let us consider this detail. Assume that $l(\theta)$ is estimated using $M$ simulations using the following steps:

1. For $b = 1, 2, \ldots, M$ repeat the next three steps:
   (a) draw a pseudo random sample $R(n)$, of size $n$, for the generative model under consideration;
   (b) obtain the dataset $D(n) \sim g_D(\theta; R(n))$ required to estimate $l(\theta)$;
   (c) calculate the statistics $T_b(D(n))$ required to estimate $l(\theta)$.
2. Calculate $l(\theta)$ as a function of $T_1(D(n)), T_2(D(n)), \ldots, T_m(D(n))$.

Here $R(n)$ is a sample of size $n$ drawn from certain standard distributions for the corresponding generative model with density $g(.)$; such that for fixed $R(n)$, $D(n)$ is equal for equal values of $\theta$. The idea behind generating the draws $R(n)$ using standard distributions corresponding to a generative model instead of generating data directly using

the generative model is that $R(n)$ does not depend on the values of $\theta$. A simple example is as follows: if the class of generative models is a univariate Gaussian distribution, $\theta = (\mu, \sigma)$; $R(n)$ is a vector of $n$ draws from a $N(0,1)$ distribution; and $D(n)$ is obtained using $D(n) = \mu + \sigma \times R(n)$. Thus instead of generating $D(n)$ directly from a $N(\mu, \sigma)$ distribution, we first generate $R(n)$ from a $N(0,1)$ distribution, and then obtain $D(n)$. Estimates of $l(\theta)$ at different values of $\theta$, but using a fixed value of $R(n)$ (large $n$) depend only on the value of $\theta$. The SEED simulation optimization algorithm is as follows:

1. Obtain $R^*$, a set of $M$ pseudo random draws each of size $n$ for the generative model under consideration. Let us assume without loss of generality that the values in $R^*$ can be ordered $1, 2, \ldots, M$.

2. For a particular value of $\theta$, the objective function $l(\theta)$ is estimated as follows:

   (a) For $b = 1, 2, \ldots, M$ repeat the next three steps:

      i. extract $R(n)$ the $b^{th}$ draws from the set $R^*$;

      ii. obtain the dataset $D(n) \sim g_D(\theta; R(n))$ required to estimate $l(\theta)$;

      iii. calculate the statistics $T_b(D(n))$ required to estimate $l(\theta)$.

   (b) Calculate $l(\theta)$ as a function of $T_1(D(n)), T_2(D(n)), \ldots, T_m(D(n))$.

3. Thus the objective function $l(\theta)$ is estimated by $\widehat{l^{R^*}(\theta)}$, which given $R^*$ is expressed as a deterministic function of $\theta$. Now use numerical optimization algorithms to minimize $\widehat{l^{R^*}(\theta)}$, subject to the constraints on $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$, as given in (2.2.1). In Section 2.2.1, we use the numerical optimization algorithm L-BFGS-B (Byrd et al., 1995) to minimize $\widehat{l^{R^*}(\theta)}$ subject to the constraints on $\theta$.

This ends the SEED simulation optimization algorithm and the algorithm is repeated for few different values of $R^*$. Stress testing in the toy example is described next, that further

illustrates the SEED simulation optimization algorithm. Let us recall the toy example described in Section 2.1. Two estimators $\overline{X}$ and $\overline{X}_{tr}$, (see Equations 2.1.1 and 2.1.2) are used to estimate the unknown population mean given a sample $X_1, X_2, \ldots, X_n$. To compare the performance of these two estimators, we use the ratio of their mean squared errors (MSE), $\text{MSE}(\overline{X})/\text{MSE}(\overline{X}_{tr}) = P$. A Gaussian mixture with four components is used as the flexible class of generative models. Hence the parameters features of the generative model $\theta$ is given by $\theta = \{p_1, p_2, \ldots, p_t, \mu_1, \mu_2, \ldots, \mu_t, \sigma_1, \sigma_2, \ldots, \sigma_t\}$ where $p_i, \mu_i$, and $\sigma_i$ $(i = 1, 2, \ldots, 4)$ are the probabilities, means, and standard deviations respectively of the individual Gaussian distributions in the mixture. Features of interest or factors that might potentially effect the performance of the two estimators $\overline{X}$ and $\overline{X}_{tr}$, are standard deviation $(\sigma_X)$, skewness $(\tau_X)$, and $(\kappa_X)$. Hence the set of features denoted by $f(\theta) = \sigma_X, \tau_X, \kappa_X$.

### 2.2.1 Stress testing of methods

Finding feature values where methods under study in a MC experiment outperform or break down is referred to as stress testing in SEED. Stress testing leads to more general conclusions about a method's performance using a simulation study. Typically MC studies involve showing where a proposed method works well, however information regarding conditions where a method is likely to not work well can be equally useful, especially in real data applications. Stress testing involves solving the following optimization problem:

$$\begin{aligned} \text{Minimize:} \quad & (P_\theta - P_{\text{Target}})^2 \\ \text{subject to:} \quad & \theta \in \Theta. \end{aligned} \tag{2.2.2}$$

Here $P_\theta$ is the performance measure obtained using a generative model with parameters $\theta$, for the method under study in the MC experiment. $P_{\text{Target}}$ is the target value of performance measure, that is chosen depending on (i) whether we are searching for outstandingly good performance or poor performance; and (ii) nature of $P_\theta$, the range of values it can take, whether higher $P_\theta$ values imply favorable performance, etc.

In the toy example used in this chapter, $P_\theta$ is the MSE ratio of the sample mean versus the sample trimmed mean, corresponding to a generative model specified by $\theta$. We are interested to find where the $\overline{X}_{tr}$ outperforms $\overline{X}$, hence we specify $P_{\text{Target}} = 10$. Here 10 is an arbitrarily large value of the MSE ratio, and it signifies that we are looking for feature values where MSE($\overline{X}$) is ten times MSE($\overline{X}_{tr}$). We use $M = 5000$ MC iterations to estimate $P_\theta$. The estimators $\overline{X}$ and $\overline{X}_{tr}$ are calculated on a dataset of size 20, that is $n = 20$. At $\tau_X = -2.67$ and $\kappa_X = 162.53$, the objective value in the optimization problem (2.2.2) is less than 0.000.

In this toy example the class of generative models is a Gaussian mixture with 4 components. Here $R^*$ contains two elements $Z$ and $U$. Each of $Z$ and $U$ are $5000 \times 20$ dimensional matrices, containing pseudo random draws from a standard Normal and standard Uniform distribution respectively. The values in $Z$ and $U$ remain the same across functional evaluations by the search algorithm. For the $i^{th}$ MC iteration ($i = 1, 2, \ldots 5000$), and $j^{th}$ data point ($j = 1, 2, \ldots 20$), a draw from the Gaussian mixture for $\theta = \{p_1, p_2 \ldots p_4, \mu_1, \mu_2 \ldots \mu_4, \sigma_1, \sigma_2 \ldots \sigma_4\}$ is given by

$$\mu_k + (\sigma_k \times Z_{ij}),$$

for $k = min\{q : \sum_{l=1}^{q} p_l \geq U_{ij}; q \in \{1, 2, 3, 4\}\}$. Here $Z_{ij}$ and $U_{ij}$ denote the elements in the $i^{th}$ row and $j^{th}$ column of $Z$ and $U$ respectively.

We repeat the steps outlined above for 10 different starting values of $\theta, U$ and $Z$,

Table 2.1: Simulation optimization to search for population skewness and kurtosis in a Gaussian mixture such that MSE-ratio = MSE($\overline{X}$)/MSE($\overline{X}_{tr}$) = 10.

| Generative model | Standard Deviation | Skewness | Kurtosis | MSE-Ratio |
|---|---|---|---|---|
| 1 | 0.502 | -2.663 | 162.527 | 10.000 |
| 2 | 1.317 | -0.937 | 83.364 | 10.523 |
| 3 | 3.750 | -0.074 | 9.038 | 18.542 |
| 4 | 0.904 | -2.051 | 116.154 | 10.000 |
| 5 | 0.820 | -4.761 | 74.120 | 16.794 |
| 6 | 0.081 | -52.256 | 12665.217 | 10.015 |
| 7 | 0.570 | 1.371 | 81.861 | 10.017 |
| 8 | 2.658 | -0.028 | 14.367 | 17.950 |
| 9 | 0.956 | 2.014 | 85.242 | 10.019 |
| 10 | 0.534 | -0.370 | 50.733 | 14.248 |

since the L-BFGS-B algorithm finds the local minima. The results are given in Table 2.1. Notice that in the cases where $P_\theta = P_{\text{Target}} = 10.0$ is reached, the results correspond to the metamodel obtained in Section 2.1.3. $P_\theta = 10$ implies that $\overline{X}_{tr}$ performs much better than $\overline{X}$, and the corresponding values of $\theta$ are in node 1 in Figure 2.7.

### 2.2.2 Parallel computing

"'Statistical simulations are embarrassingly parallel" - Alfons (2011). Performance measures of methods in a simulation study are obtained using a large number of MC iterations, and evaluations in an iteration is independent of evaluations in all other iterations. For instance in Section 2.1.3, for every generative model we estimate the MSE of $\overline{X}$ and $\overline{X}_{tr}$ using 5000 iterations. In this case we use Python libraries `multiprocessing` and `joblib` for multi-core parallel computing, to parallelize the section of code used to esti-

mate the MSE of $\overline{X}$ and $\overline{X}_{tr}$ over multiple cores.

In this chapter we discussed space-filling experiments and stress testing of methods to arrive at more general conclusions from a simulation experiment using SEED. In a space-filling SEED experiment we aim to cover the space of features of interest using a flexible class of generative models. The features of interest in the experiment, $f(\theta)$ may be either (i) expressed in closed form in terms of $\theta$; or (ii) require estimation using simulations. The former case is presented in Section 2.1.1 and a simulation optimization algorithm to deal with problems belonging to the latter case has been discussed in Section 2.2. Using adaptive search algorithms, optimization algorithms, and a flexible class of generative models, desired coverage of the features of space of interest is achieved. We evaluate the methods under study across generative models, and formally analyze the results (Section 2.1.3). In Section 2.2.1 we search for feature values corresponding to excellent and poor performance, and the results found conform with the analysis in Section 2.1.3. In the next two chapters we consider the internal framework of SEED in detail and present a case study.

# Chapter 3

# Case Study: $Q$-learning and $IQ$-learning

In this chapter we use SEED to compare two methodologies from personalized medicine given in Laber, Linn and Stefanski (2014). Deciding the best available treatment for an individual given his or her health data is the key idea behind personalized medicine. In many cases treatment effects vary across individuals, hence using information about a specific patient and available treatment options, to optimize a desired outcome for each patient can aid clinical decision making. In chronic illnesses, a patient might receive treatment involving multiple stages. A dynamic treatment regime (DTR) specifies the decision rule for assigning treatments in the different stages, and is referred to as the optimal regime if it optimizes the outcome of interest (Laber, Linn and Stefanski, 2014). $Q$-learning and interactive $Q$-learning ($IQ$-learning) are two methods used to estimate the optimal DTR.

We consider a sequential multiple assignment randomized trial (SMART) design with two stages and two treatments, to randomly assign treatments at each stage (Laber, Linn

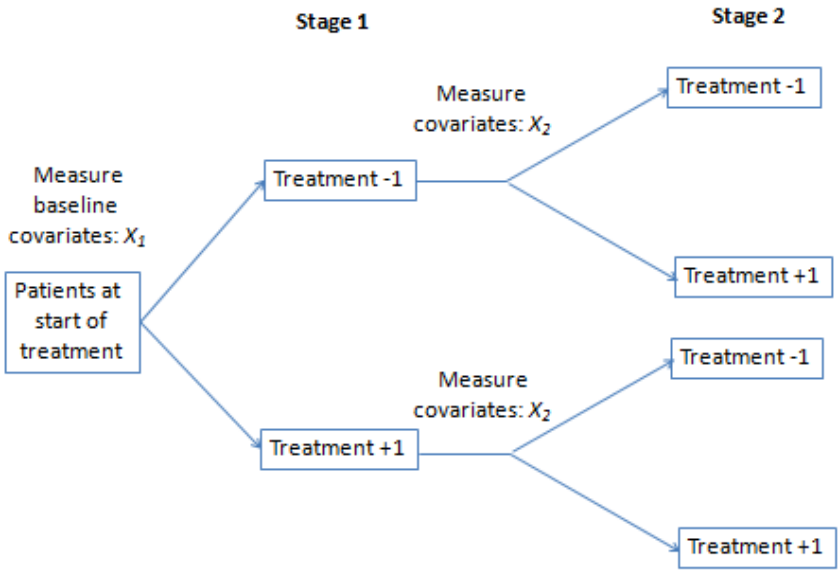


Figure 3.1: Design in a two-stage two-treatment SMART trial. Baseline covariates of patients are measured at the beginning of each stage, and patients are then randomized to treatments 1 or -1 at each of the two stages.

and Stefanski, 2014). SMARTs are described in Collins, Murphy and Strecher (2007) as "an innovative research design especially suited for building time-varying adaptive interventions". Figure 3.1 shows the design in a two-stage two-treatment SMART trial. The two possible medical treatments are coded as -1 and 1. Let $X_t$ denote the vector of of observed covariates at stage $t$, and $A_t \in \{-1, 1\}$ denote the treatment assigned at stage $t$, $t = 1, 2$. Let $Y$ denote the observed scalar outcome at the end of the second stage, with higher values implying better results. Let $\pi$ denote a regime defined as a set of decision rules to map the space of baseline covariates to the set of treatments (Laber, Linn and Stefanski, 2014). Value of a fixed regime $\pi$ denoted by $V^\pi$ is the expected value of the outcome, $Y$, when treatments to all patients are assigned according to that regime (Laber,

Linn and Stefanski, 2014). An optimal regime denoted by $\pi^{\text{opt}}$ is a regime that maximizes value (Laber, Linn and Stefanski, 2014). Given data for $n$ patients, $(X_1, A_1, X_2, A_2, Y)_{i=1}^n$, $Q$-learning and $IQ$-learning are two methods to estimate the optimal regime denoted by $\pi^{Q-\text{opt}}$ and $\pi^{IQ-\text{opt}}$ respectively. Next we briefly describe the $Q$-learning and $IQ$-learning algorithms as given in Laber, Linn and Stefanski (2014).

As the names suggest, both methods involve $Q$-functions which are defined as follows:

$$\begin{aligned} Q_2(h_2, a_2) &\triangleq \mathbb{E}\{Y \mid H_2 = h_2, A_2 = a_2\}, \\ Q_1(h_1, a_1) &\triangleq \mathbb{E}\{\max_{a_2 \in \{-1,1\}} Q_2(H_2, a_2) \mid H_1 = h_1, A_1 = a_1\} \end{aligned}$$

(Laber, Linn and Stefanski, 2014). Here $H_t$ refers to the history of a patient before treatment at stage $t$ is assigned ($t = 1, 2$). The $Q$-functions are unknown and are commonly estimated using linear models:

$$Q_t(h_t, a_t; \beta_t) = h_{t,0}^T \beta_{t,0} + a_t h_{t,1}^T \beta_{t,1}, \quad t = 1, 2, \tag{3.0.1}$$

(Laber, Linn and Stefanski, 2014). We assume $h_{t,0} = h_{t,1} = h_t$. The estimated optimal regime $\widehat{\pi}^{Q-\text{opt}}(h_t) = \arg\max_{a_t} \widehat{Q_t}(h_t, a_t)$, where $\widehat{Q_t}(h_t, a_t)$ are the estimates of $Q$-functions at stage $t$ ($t = 1, 2$). Using the $Q$-learning algorithm, $\widehat{Q_t}(h_t, a_t)$ ($t = 1, 2$) are obtained as follows (Laber, Linn and Stefanski, 2014):

- Obtain the least squares estimates of $\beta_{2,0}$ and $\beta_{2,1}$ by regressing $Y$ on $H_2$ and $A_2$ as given in Equation (3.0.1). Denote these estimates by $\widehat{\beta}_{2,0}$ and $\widehat{\beta}_{2,1}$. Thus $\widehat{Q_2}(H_2, A_2) = H_{2,0}^T \widehat{\beta}_{2,0} + A_2 H_{2,1}^T \widehat{\beta}_{2,1}$. Notice that $A_2 = (-1)^{I(H_2^T \widehat{\beta}_{2,1}) < 0}$ where $I$ is the indicator function, maximizes $\widehat{Q_2}(H_2, A_2)$ at $H_2^T \widehat{\beta}_{2,0} + \left| H_2^T \widehat{\beta}_{2,1} \right|$. Hence $\widehat{A_2}^{Q_{\text{opt}}}(H_2) = (-1)^{I(H_2^T \widehat{\beta}_{2,1}) < 0}$.

- Define $\widetilde{Y} = \max_{a_2 \in \{-1,1\}} Q_2(H_2, a_2; \widehat{\beta}_2) = H_2^T \widehat{\beta}_{2,0} + \left| H_2^T \widehat{\beta}_{2,1} \right|$. $\widetilde{Y}$ is the predicted outcome under the estimated optimal treatment at stage 2.

- Regress $\widetilde{Y}$ on $H_1$ and $A_1$ using Equation (3.0.1) to obtain the least squares estimates $\widehat{\beta}_{1,0}$ and $\widehat{\beta}_{1,1}$. Thus $\widehat{Q_1}(H_1, A_1) = H_{1,0}^T \widehat{\beta}_{1,0} + A_1 H_{1,1}^T \widehat{\beta}_{1,1}$. As in stage two, $A_1 = (-1)^{I(H_1^T \widehat{\beta}_{1,1}) < 0}$, maximizes $\widehat{Q_1}(H_1, A_1)$ at $H_1^T \widehat{\beta}_{1,0} + \left| H_1^T \widehat{\beta}_{1,1} \right|$. Hence $\widehat{A_1}^{Q_{\text{opt}}}(H_1) = (-1)^{I(H_1^T \widehat{\beta}_{1,1}) < 0}$.

- Using the estimates $\widehat{\beta}_t = (\widehat{\beta}_{t,0}, \widehat{\beta}_{t,1})$, the estimated optimal treatment regime under $Q$-learning is given by: $\widehat{\pi}^{Q-\text{opt}} = \max_{a_t \in \{-1,1\}} \widehat{Q}_t^Q(h_t, a_t; \widehat{\beta}_t)$, $(t = 1, 2)$.

In $IQ$-learning, the $Q_1$ function is estimated in two parts, instead of one as is the the case of $Q$-learning.

$$\begin{aligned} Q_1(h_1, a_1) &= \mathbb{E}(\max_{a_2 \in \{-1,1\}} Q_2(H_2, a_2) \mid H_1 = h_1, A_1 = a_1) \\ &= \mathbb{E}(H_2^T \beta_{2,0} + \left| H_2^T \beta_{2,1} \right| \mid H_1 = h_1, A_1 = a_1) \\ &= \mathbb{E}(H_2^T \beta_{2,0} \mid H_1 = h_1, A_1 = a_1) + \int |z| \, g(z \mid h_1, a_1) dz. \end{aligned}$$

Here $g(z \mid h_1, a_1)$ is the conditional density of $h_2^T \widehat{\beta}_{2,1}$. The main steps in the $IQ$-learning algorithm are as follows (Laber, Linn and Stefanski, 2014):

- Obtain the least squares estimates of $\beta_{2,0}$ and $\beta_{2,1}$ by regressing $Y$ on $H_2$ and $A_2$ as given in Equation (3.0.1). Denote these estimates by $\widehat{\beta}_{2,0}$ and $\widehat{\beta}_{2,1}$. Hence $\widehat{Q}_2^{IQ}(H_2, A_2) = H_2^T \widehat{\beta}_{2,0} + A_2 H_2^T \widehat{\beta}_{2,1}$. This step is the same as in $Q$-learning.

- Regress $H_2^T \widehat{\beta}_{2,0}$ on $H_1$ and $A_1$ to obtain an estimate $\widehat{L}(H_1, A_1)$ of $\mathbb{E}(H_2^T \widehat{\beta}_{2,0} \mid H_1, A_1)$.

- Estimate the conditional distribution $g(. \mid H_1, A_1)$ as $\widehat{g}(. \mid H_1, A_1)$ using $\{H_{1,i}, A_{1,i}, \widehat{\beta}_{2,1}, H_{2,i}\}_{i=1}^{n_{\text{trn}}}$.

- The estimated $Q_1$ is given by $\widehat{Q}_1^{IQ}(H_1, A_1) = \widehat{L}(H_1, A_1) + \int |z|\, \widehat{g}(z \mid H_1, A_1) dz$.

- Thus the estimated optimal treatment regime under $IQ$-learning is given by: $\widehat{\pi}^{IQ-\text{opt}} = \max_{a_t \in \{-1,1\}} \widehat{Q}_t^{IQ}(h_t, a_t; \widehat{\beta}_t)$, $(t = 1, 2)$.

The conditional distribution $g(. \mid h_1, a_1)$ has been estimated using a Normal location scale model in this paper as outlined in (Laber, Linn and Stefanski, 2014). Define the main effects and contrast functions as follows:

$$\begin{aligned} \mu(H_2) &\triangleq \frac{1}{2}\{\widehat{Q_2}(H_2, 1) + \widehat{Q_2}(H_2, -1)\}, \\ \Delta(H_2) &\triangleq \frac{1}{2}\{\widehat{Q_2}(H_2, 1) - \widehat{Q_2}(H_2, -1)\}. \end{aligned}$$

Their respective estimates $\widehat{\mu}(H_2)$ and $\widehat{\Delta}(H_2)$ are obtained using Equation (3.0.1) and $\widehat{\beta}_2$. We obtain estimates of $m(h_1, a_1) \triangleq \mathbb{E}\{\Delta(H_2) \mid H_1 = h_1, A_1 = a_1\}$ and $\sigma^2(h_1, a_1) \triangleq \mathbb{E}\{(\Delta(H_2) - m(h_1, a_1))^2 \mid H_1 = h_1, A_1 = a_1\}$ denoted by $\widehat{m}(h_1, a_1)$ and $\widehat{\sigma}^2(h_1, a_1)$ respectively (Laber, Linn and Stefanski, 2014). Then $g(. \mid h_1, a_1)$ the conditional distribution of the contrast $\Delta(H_2)$ given $H_1 = h_1$ and $A_1 = a_1$ is estimated by

$$\widehat{g}(z \mid h_1, a_1) = \frac{1}{\widehat{\sigma}(h_1, a_1)} \phi\left(\frac{z - \widehat{m}(h_1, a_1)}{\widehat{\sigma}(h_1, a_1)}\right)$$

(Laber, Linn and Stefanski, 2014).

In the next two sections we conduct a space filling SEED experiment to study the performance of $Q$-learning and $IQ$-learning. Let $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$ denote the set of parameters specifying a generative model. Denote by $\mathcal{S}$, the space corresponding to

the domain of the parameters in $\theta$. Define $f(.) : \mathcal{S} \mapsto \mathcal{S}^*$, $\mathcal{S}^* \subseteq \mathcal{R}^d, d = 1, 2 \ldots, k$; $f(\theta) = (f_1(\theta), f_2(\theta), \ldots, f_d(\theta))$ as the set of all features of interest in the simulation experiment.

Average value under the regime $\pi$ is defined as $\mathbb{E}\mathbb{E}^{\widehat{\pi}}(Y)$, where the inner expectation is taken over a test set of size $n_{test} = 10000$, and the outer expectation is taken over 1000 MC iterations. In each MC iteration, a training set of size $n_{trn} = 200$ is generated and used to estimate $\widehat{\beta}_t$ $(t = 1, 2)$; and $\mathbb{E}^{\widehat{\pi}(Y)}$ is estimated using a test set of size 10000. We define the odds ratio as $\mathbb{E}\{\mathbb{E}^{\widehat{\pi}^{IQ}}(Y) > \mathbb{E}^{\widehat{\pi}^{Q}}(Y)\}/\mathbb{E}\{\mathbb{E}^{\widehat{\pi}^{IQ}}(Y) \leq \mathbb{E}^{\widehat{\pi}^{Q}}(Y)\}$; and it indicates the odds of $IQ$-learning leading to a higher expected outcome than $Q$-learning. Define integrated mean squared error (IMSE) of $Q_1(H_1, A_1)$ as $\mathbb{E}\{Q_1(H_1, A_1) - Q_1(H_1, A_1)\}^2$ (Laber, Linn and Stefanski, 2014). The IMSE ratio reported here is given by $\frac{\widehat{IMSE}^{Q}}{\widehat{IMSE}^{IQ}}$. Thus we estimate three performance measures to compare $Q$-learning and $IQ$-learning: (i) ratio of average values, (ii) odds ratio, and (iii) IMSE ratio. For all three performance measures higher values imply better performance of $IQ$-learning compared to $Q$-learning.

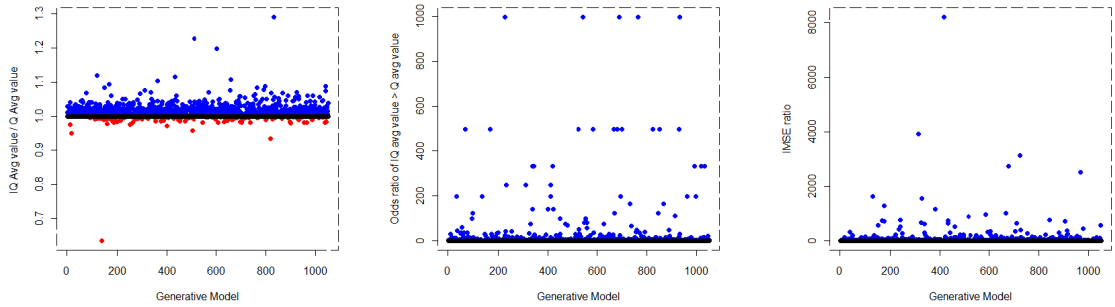


Figure 3.2: Overall performance for the generative model given in Section 3.1 using $n_{\text{test}} = 10000$, $n_{\text{trn}} = 200$ and 1000 training sets. Higher values are favorable to $IQ$-learning.

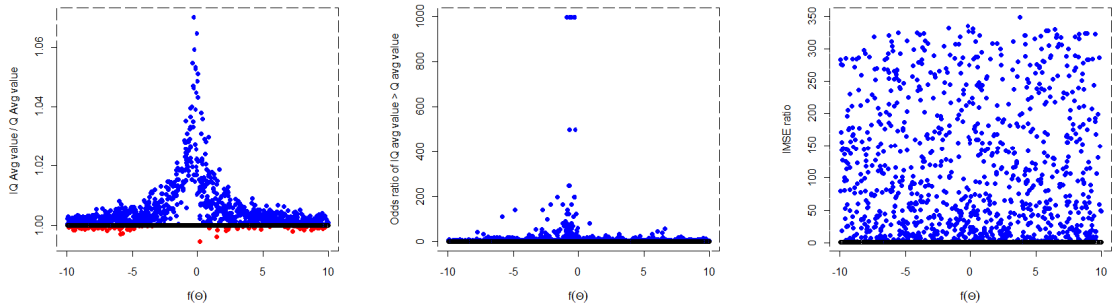


Figure 3.3: Performance versus $f(\theta) = \beta_{2,0,3}$, the effect size of $A_1X_1$ for the generative model given in Section 3.1 using $n_{\text{test}} = 10000$, $n_{\text{trn}} = 200$ and 1000 training sets. Higher values are favorable to $IQ$-learning.

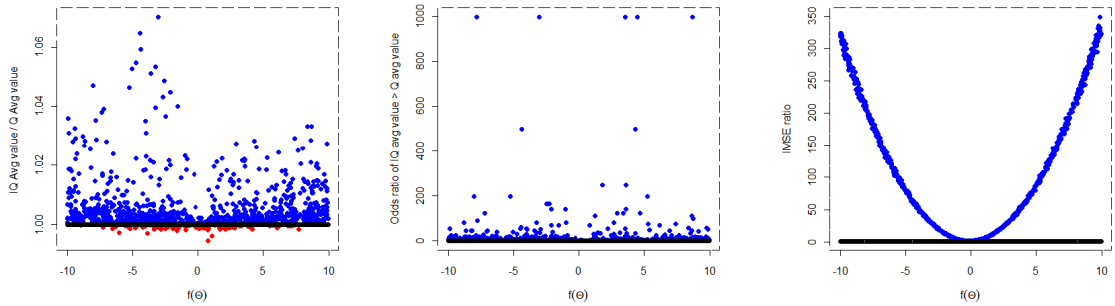


Figure 3.4: Performance versus $f(\theta) = \beta_{2,1,4}$, the effect size of $A_2X_2$ for the generative model given in Section 3.1 using $n_{\text{test}} = 10000$, $n_{\text{trn}} = 200$ and 1000 training sets. Higher values are favorable to $IQ$-learning.

## 3.1 Space-filling over directly controllable $f(\theta)$

The class of generative models we use in this section is as follows.

$$
\begin{aligned}
X_1 &\sim N_2(0_2, \Sigma_2) \\
\Sigma_2 &= \begin{pmatrix} \sigma_{X1} & \rho_{X1} \\ \rho_{X1} & \sigma_{X1} \end{pmatrix} \\
A_t &\sim \text{Discrete Uniform}(-1, 1), t = 1, 2 \\
X_2 &= \beta_1 X_1 + \xi \\
\xi &\sim \text{Normal}(0, \sigma_\xi) \\
H_2 &= (1, A_1, A_1 X_1, X_2)^T \\
Y &= H_2^T \beta_{2,0} + A_2 H_2^T \beta_{2,1} + \eta \\
\eta &\sim \text{Normal}(0, \sigma_\eta)
\end{aligned}
$$

The dimensions of $X_1, X_2, \beta_1, \xi, H_2, \beta_{2,0}, \beta_{2,1}, \eta,$ and $Y$ are $(1 \times 2), (1 \times 2), (1 \times 1), (1 \times 2), (6 \times 1), (6 \times 1), (6 \times 1), (1 \times 1)$ and $(1 \times 1)$ respectively. 1000 generative models are used. Here $\theta \triangleq (\sigma_{X1}, \rho_{X1}, \beta_1, \sigma_\xi, \beta_{2,0}, \beta_{2,1}, \sigma_\eta)$.

We start by considering $f(\theta) = \theta$. In each generative model, the values of the elements in $\theta$ are randomized from a Uniform distribution over a range of interest. Here uniform coverage is achieved directly without using optimization algorithms since $f(\theta)$ can be directly controlled by changing the values in $\theta$. The space of interest we cover uniformly is $\mathcal{S}^* \equiv (0.5, 5) \times (0.1, 9) \times (-10, 10) \times (0, 2) \times (-10, 10) \times (-10, 10) \times (1, 50)$. Figure 3.2 shows the overall comparative performance of the two methods. $IQ$-learning does better than $Q$-learning for most generative models with respect to all three performance measures. However, we did not find a discernible trend in the performance for different values of

$f(\theta)$.

Next we consider $f(\theta) = (\beta_{2,0,3}, \beta_{2,1,4})$, a subset of $\theta$. These coefficients determine the effect of the terms $A_1X_1$ and $A_2X_2$ on $Y$. The space of features of interest covered uniformly is $\mathcal{S}^* \equiv (-10, 10) \times (-10, 10)$. The remaining elements in $\theta$ are fixed at $\sigma_{X1} = 2, \rho_{X1} = 0.5, \beta_1 = 0.5, \sigma_\xi = 1, \beta_{2,0,-3} = 0.5, \beta_{2,1,-4} = 0.5$ and $\sigma_\eta = 2$. Figures 3.3 and 3.4 show the results for performance versus $A_1X_1$ and $A_2X_2$ effects respectively. For small values of $\beta_{2,0,3}$, $IQ$-learning does noticeably better. Large values of $\beta_{2,1,4}$ correspond to IMSE ratios favorable to $IQ$-learning.

## 3.2 Space-filling over $f(\theta)$ using simulation optimization

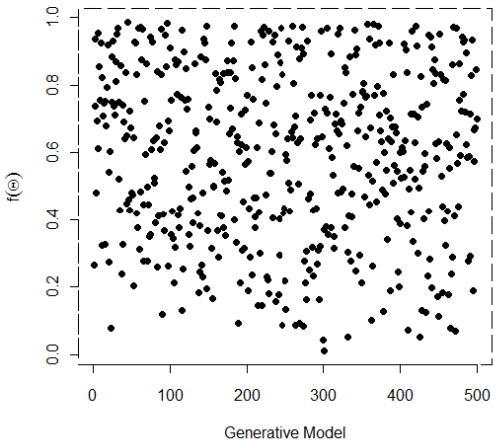


Figure 3.5: Coverage of $f(\theta)$ using SEED with 500 generative models

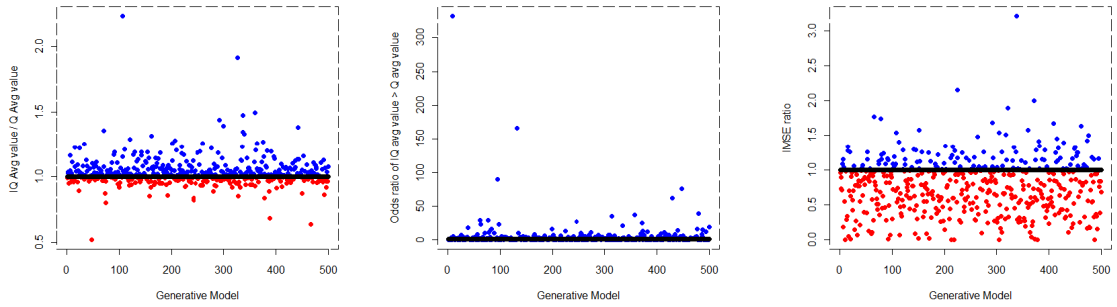


Figure 3.6: Overall performance for model given in Section 3.2 using $n_{\text{test}} = 10000$, $n_{\text{trn}} = 200$ and 1000 training sets.

We use a more flexible generative model to achieve space-filling for $f(\theta) \triangleq Pr(\left|H_2^T \beta_{2,1}\right| > 2\sigma_\eta/\sqrt{n_{\text{trn}}})$. Here $\theta \triangleq (p_{1X_1}, \ldots p_{4X_1}, \mu_{1X_1}, \ldots \mu_{4X_1}, \sigma_{1X_1}, \ldots \sigma_{4X_1}, B_{1,0}, B_{1,1}, \gamma_0, \gamma_1, \beta_{2,0}, \beta_{2,1}, \sigma_\eta)$. The data for this section has been generated using the following equations.

$$
\begin{aligned}
X_1 &\sim \text{Gaussian Mixture}(p_{1X_1}, \ldots p_{4X_1}, \mu_{1X_1}, \ldots \mu_{4X_1}, \sigma_{1X_1}, \ldots \sigma_{4X_1}) \\
A_t &\sim \text{Discrete Uniform}(-1, 1), \quad t = 1, 2 \\
H_1 &= (1, X_1)^T \\
X_2 &= H_1^T B_{1,0} + A_1 H_1^T B_{1,1} + s(H_1, A_1 : \gamma)\xi \\
\xi &\sim \text{Normal}(0, 1) \\
s(H_1, A_1 : \gamma) &= exp\{(H_1^T \gamma_0 + A_1 H_1^T \gamma_1)/2\} \\
H_2 &= (1, X_2)^T \\
Y &= H_2^T \beta_{2,0} + A_2 H_2^T \beta_{2,1} + \eta \\
\eta &\sim \text{Normal}(0, \sigma_\eta)
\end{aligned}
$$

For each patient the dimensions of $X_1, H_1, X_2, B_{10}, B_{11}, s(H_1, A_1 : \gamma), \xi, H_2, \beta_{20}, \beta_{21}, \eta$

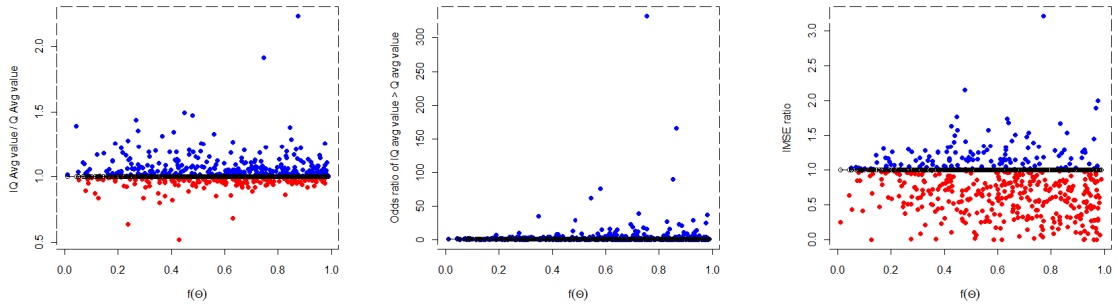


Figure 3.7: Performance across values of $f(\theta) = Pr(\mid H_2^T \beta_{2,1} \mid > 2\sigma_\eta/\sqrt{n_{\text{trn}}})$ for the model given in Section 3.2 using $n_{\text{test}} = 10000$, $n_{\text{trn}} = 200$ and 1000 training sets. Higher values are favorable to $IQ$-learning.

and $Y$ are $(1\times 2), (3\times 1), (1\times 2), (3\times 2), (3\times 2), (1\times 1), (1\times 2), (3\times 1), (3\times 1), (3\times 1), (1\times 1)$ and and $(1 \times 1)$ respectively. The learning algorithms are applied to data tuples of $(X_1, A_1, X_2, A_2, Y)$ of size $n_{\text{trn}}$, and the estimates used to estimate the optimal treatment regime on the test data consisting of the tuple $(X_1, X_2)$ of size $n_{\text{test}}$.

The simulation optimization algorithm outlined in Section 2.2 has been used to achieve coverage of $f(\theta)$ over $(0, 1)$. Figure 3.5 shows the coverage achieved on the test data. The overall performance of $Q$ and $IQ$-learning in given in Figure 3.6. Figure 3.7 shows the performance of the two methods versus $f(\theta)$. We see that $IQ$-learning performs better than $Q$-learning in most cases with respect to ratio of average outcome values. However there does not seem to be a strong trend in the performance measures for different values of $f(\theta)$.

In this chapter we considered a case study for space filling experiments in SEED to compare $Q$-learning and $IQ$-learning. We used principles of space-filling designs and simulation optimization presented in Chapter two to achieve desired coverage in the space

of features for these two methods. In the next chapter we provide details of the SEED implementation and framework, and present an R package for SEED.

# Chapter 4

# Implementation of the SEED Framework

Monte Carlo (MC) simulation experiments are an invaluable tool to study and compare statistical methodologies. MC simulation experiments help us study finite sample properties of methods, validate asymptotic approximations for finite sample inference, and test robustness of methods to violation of assumptions. In Chapter one, we discussed problems that affect typical MC experiments including: (i) considering a small number of generative models in the study leading to results that are hard to generalize; (ii) lack of clarity in the description of the simulation experiment and methods, making it difficult to reproduce results; (iii) lack of transparency due to simulation code not being available. With Space-filling Exploratory Experimental Design (SEED) we attempt to provide a way to help solve these problems. SEED is a framework that facilitates rigorous, generalizable statistical simulation experiments by evaluating methods on a large number of generative models. The generative models are chosen to ensure that the space of features of interest is covered either uniformly or conform to a factorial design. The

performance of proposed methods across these models can be used to build statistical models to characterize feature values where the methods under study perform well and where they do not.

Results from a simulation study are generalizable if qualitative conclusions from the study can be extended to the space of features of interest or to other generative models, based on the study design, leading to more general conclusions about the methods under consideration. In addition, examining where methods perform both well and poorly make simulation results more generalizable. SEED uses a large number of generative models that are systematically chosen to uniformly cover a space of features of interest. Performance measures of methods are obtained for each generative model. In addition, SEED applies stress testing to look for feature values where a method breaks down or outperforms. Thus, at the end of a SEED experiment a more informative assessment of a method's properties and performance is obtained. In Chapter two we described how to obtain general conclusions from a simulation experiment using space-filling designs and stress testing of methods.

In addition to being generalizable, a SEED experiment aims to be reproducible and repeatable. A repeatable simulation study is one where exact reported results can be regenerated using the information made available by the researcher, like the simulation experiment code, seeds for pseudo random number generation, software versions, random number generators, etc. A simulation experiment is called reproducible, if another researcher can reconduct the experiment using the description of the methodologies and generative models provided, and reach the same qualitative conclusions. We define reproducibility to also include exploring a method's performance for different feature values and generative models, within the class of generative models considered originally. A SEED experiment records the following details for all generative models used in the ex-

periment: (i) seeds used for pseudo random number generation; (ii) feature values; (iii) parameter values; and (iv) performance measures of the methods being studied. Sharing these details along with the SEED code helps make the simulation experiment repeatable. The SEED framework uses a modular code structure that makes changes easy to implement. For example, a new method may be compared to already existing methods in a SEED experiment by making changes to only one part of the SEED framework. Methods can be evaluated on a new set of features and generative models with relative ease in SEED, facilitating reproducibility. The modular SEED framework makes it simpler to understand the structure of the code, which also facilitates repeatability and reproducibility. In the next section we introduce and discuss the components of the SEED framework in detail.

## 4.1 Components of the SEED framework

Figures 4.1 and 4.2 describe the main steps in a SEED experiment. Each step in the SEED experiment consists of one or more specific tasks. Each task of a SEED experiment is implemented using a class, which is a structure combining the inputs and the functions used for a task. The functions in a class are accessed using an instance of the class, also called an object of the class. Object oriented programming (OOP) is a powerful method of programming to deal with complexity (Lafore, 1997). Instead of viewing a program as a series of steps to be carried out, OOP views it as a group of class objects that can perform certain tasks using variables and functions belonging to the class (Lafore, 1997). SEED uses an OOP structure where a different class is used to implement each of the main steps of a simulation experiment.

Before starting a SEED experiment a researcher much specify the following attributes

of the experiment: (i) features of interest and their corresponding space or levels of interest; (ii) class of distributional families in the generative model; (iii) mechanism for covering the space of interest such as a uniform space-filling experiment or a factorial experiment; (iv) methods to be studied; (iv) performance measures to be used to evaluate methods such as mean squared error (MSE), mean absolute deviation (MAD); etc., (v) seeds used for pseudo random number generation; (vi) number of generative models; and (vii) sample sizes of datasets generated from the generative model in the simulation experiment; etc. The steps in a space-filling SEED experiment and the corresponding classes are:

- Choose target feature values for a generative model. This choice is made depending on the type of SEED experiment: uniform space-filling or factorial experiment. This is implemented in the class `ITERATOR`.

- Search for a generative model with feature values equal or close to the target values of the features. In the SEED framework this step is implemented in the class `GENMODEL`. If the target values are not admissible, admissible feature values that are closest to the target feature values are chosen. A target feature value may be inadmissible due to constraints in the space of features. For example in Section 2.1 we compared the performance of two estimators $\overline{X}$ and $\overline{X}_{tr}$ of the population mean, for different values of standard deviation ($\sigma_X$), skewness ($\tau_X$), and kurtosis ($\kappa_X$). Constraints on the space of features in this toy example was given in Equation (2.1.6). Details on searching for generative models in SEED is discussed in Chapter two.

- Next, the following two steps are implemented using the class `SIMULATOR` for each MC iteration:

Table 4.1: Description of classes used in the SEED framework

| class | description |
|---|---|
| ITERATOR | Obtains target feature values for a generative model. |
| GENMODEL | Obtains generative model with feature values arbitrarily close to the target feature values |
| SIMULATOR | Obtains MC estimates of performance measures by applying methods under study to datasets from the generative model. |
| DISTRIBUTIONAL FAMILIES | For specific probability distributions generate random numbers from it, specify lower and upper bounds of its parameters, and generate random starting values of the parameters. |
| DATASETS | Using draws from the generative model, obtains datasets that are used by the methods under study. |
| METHODS | Apply methods under study to a given dataset. |
| STRESSTEST | Find feature values where the methods under study break down or outperform. |
| RECORDER | Record results and settings of the experiment. |

  - Generate datasets from the generative model, implemented in the classes DATASETS and DISTRIBUTIONAL FAMILIES.
  - Apply methodologies under study, implemented in the class METHODS.

  After the MC iterations are over, SIMULATOR obtains estimates of performance measures for the methods under consideration.

- Record parameters, feature values, and performance of methods of the generative model. This is implemented in the class RECORDER.

Results obtained from the SEED experiment are then analyzed to explore relationships between features and method performance measures.

Figures 4.1 and 4.2 describe the main steps in a space-filling and stress testing SEED experiment respectively, along with the key user inputs required in each step. The flow

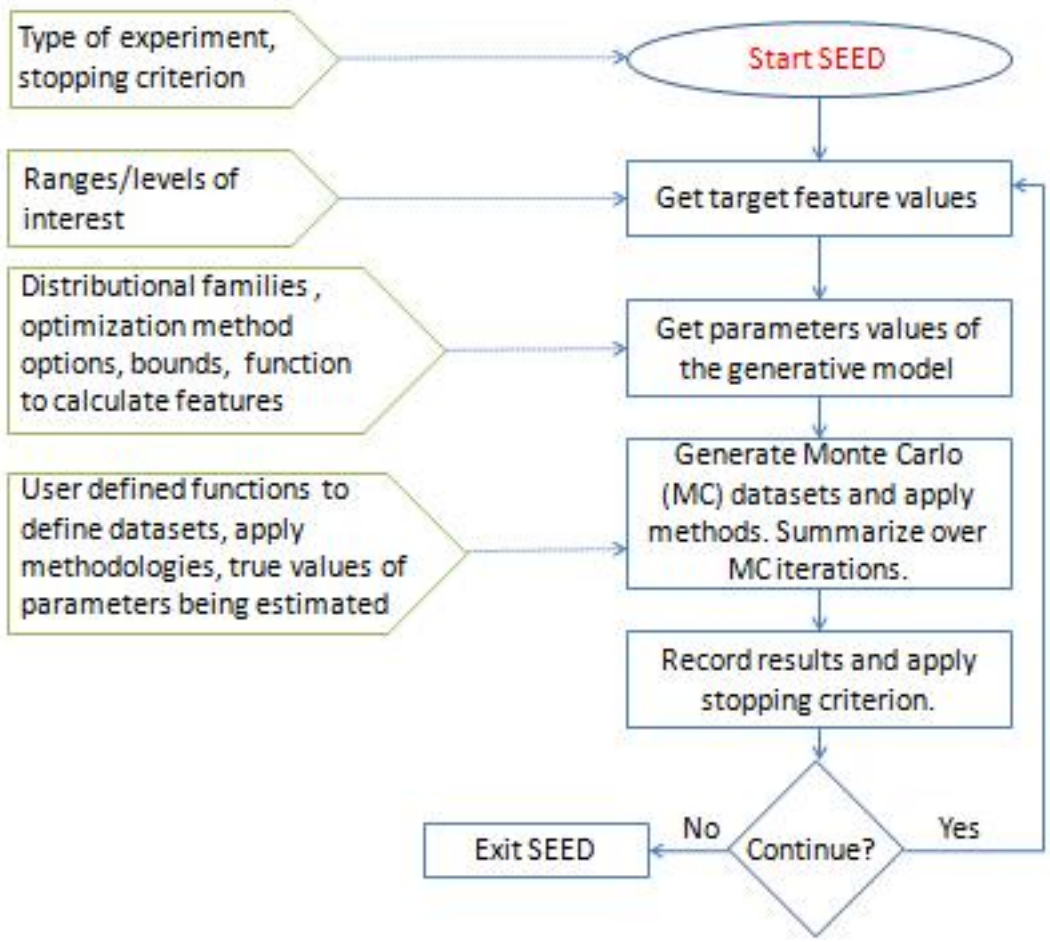


Figure 4.1: Main steps in a SEED space-filling experiment

of control between the classes in a SEED experiment is shown in Figure 4.3, and class descriptions are given in Table 4.1. In SEED, we use virtual classes to combine the inputs and the functions for a specific task in the simulation experiment. However, the functions are often not explicitly defined in the virtual classes. Classes that are based on the virtual classes, also called derived classes, contain the function definitions or extensions. To implement SEED these derived classes must be implemented. Using virtual and derived classes in this way makes SEED sufficiently flexible to be applied to

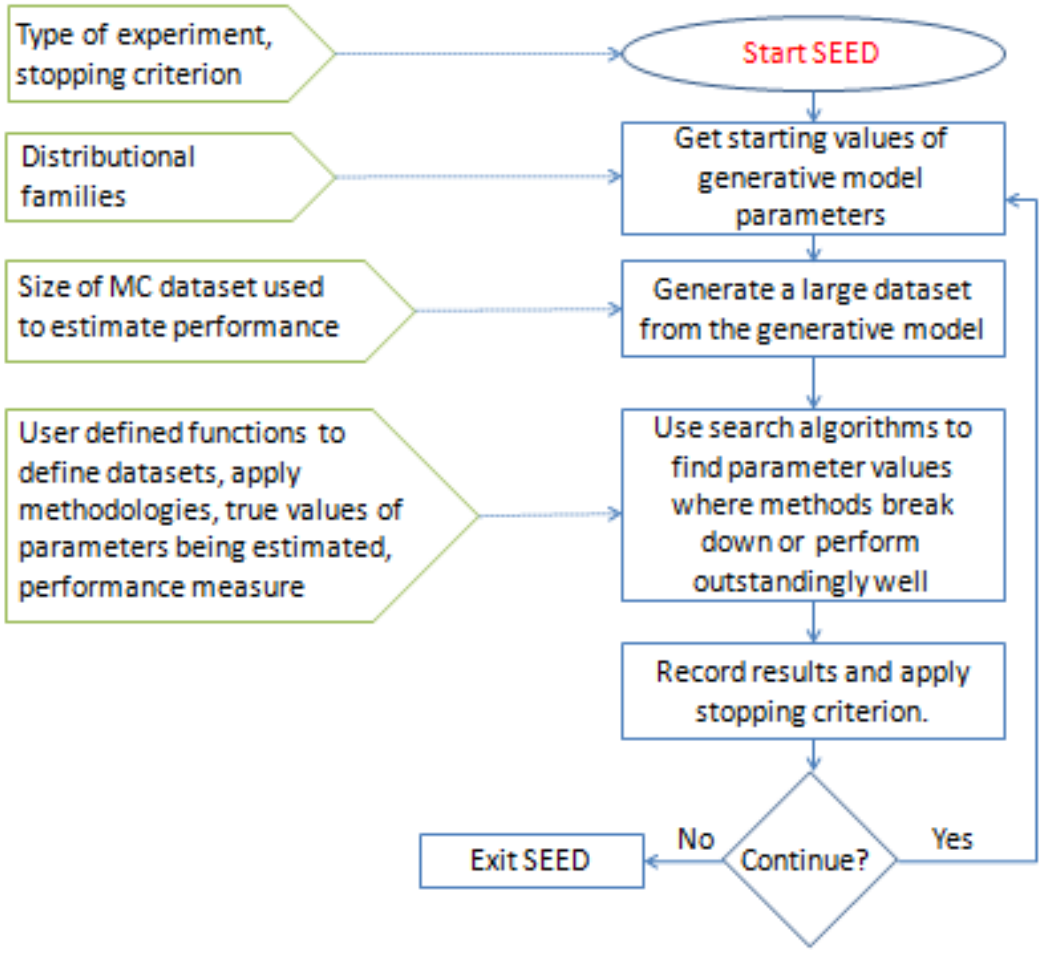


Figure 4.2: Main steps in a SEED stress testing experiment

a wide range of statistical simulation experiments. In Section 4.2.3, the classes in the SEED package are described in detail.

## 4.2 Implementing SEED using an R Package

The SEED R package is designed to help conduct a wide range of SEED experiments. The SEED framework has been implemented using an R package since R is: (i) widely

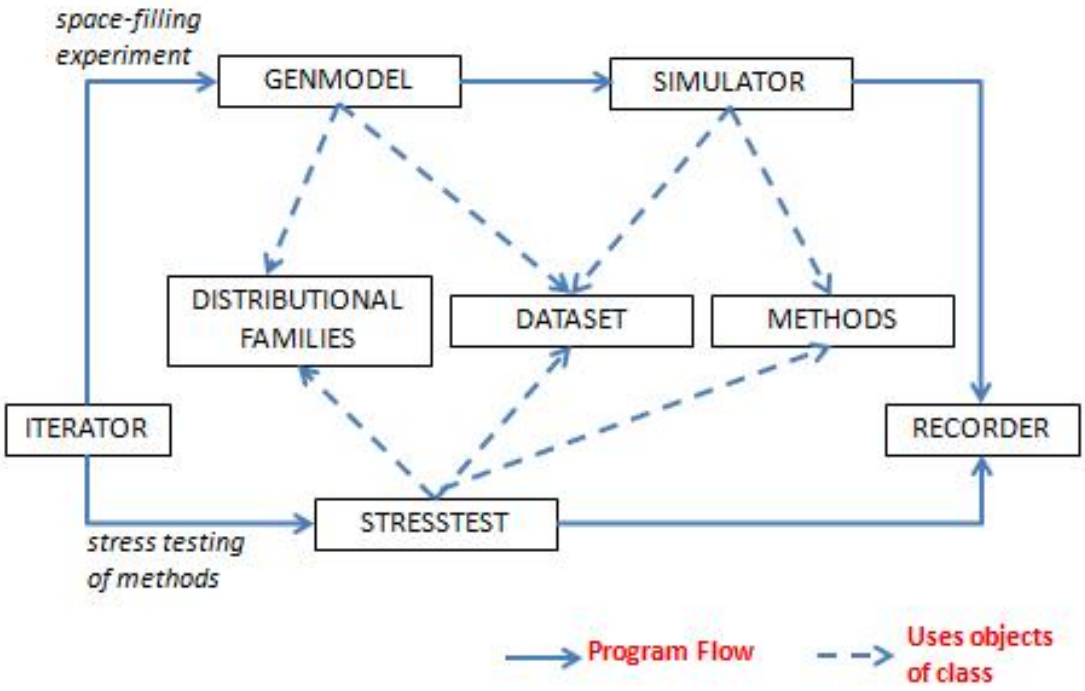


Figure 4.3: Flow of control between classes in SEED. Each box represents an abstract class that performs a specific task in a SEED experiment as described in Figures 4.1 and 4.2.

used in statistical research, especially to conduct simulation experiments, (ii) open source and, (iii) free available under the GNU license. In addition, R includes a rich collection of statistical packages. We have used `S4` classes in R to implement the object oriented structure in SEED. `S3` and `S4` are two available mechanisms for object oriented programming in R. The `S4` class system provides greater flexibility, and provision for formal definition of classes and their functions (Chambers, 2008). `S4` classes being more rigorous are considered more trustworthy for software development in R as compared to `S3` classes (Chambers, 2008). Functions of a class are defined using generic functions in R. A difference between `S4` classes in R from traditional OOPs in C++ or Python is that function definitions are not contained inside the class definitions in R (Wang and

Day, 2010). Functions with similar functionality may be defined under the same generic method, and so have the same name. These functions may be defined differently within different classes, and have different arguments. Thus, they are flexible yet modular in nature.

The SEED package code is written to conduct a SEED experiment for fairly general simulation experiments, hence a user provides detailed inputs about the specific simulation experiment that he or she wants to conduct. There are three kinds of user inputs for using the SEED package: (i) lists, (ii) functions and, (iii) derived classes (optional). We take a look at each of the user inputs in Sections 4.2.1, 4.2.2, and 4.2.3. We use the following simple linear regression example to illustrate the different user inputs described in Sections 4.2.1 and 4.2.2.

**Example 4.2.1. Linear regression**: Let us consider a experiment to study the performance of ordinary least squares (OLS) estimates for a simple linear regression model, for different error distributions. The model is

$$Y_i = \beta_0 + \beta_1 X_i + \xi_i, \tag{4.2.1}$$

where $i = 1, 2, \ldots, n$. $X_i$, $i = 1, 2, \ldots, n$ are independent draws from a Student's $t$ distribution. The error variable observations are denoted by $\xi_i$, $i = 1, 2, \ldots, n$ are independent draws from an univariate Gaussian mixture distribution with 4 components. The values of $\beta_0$ and $\beta_1$ are 1 and 5 respectively. We have used $n = 20$ in this example. We are interested to conduct a space filling experiment where the features of interest are the standard deviation, skewness and kurtosis of $\xi_i$'s ($i = 1, 2, \ldots, n$). Let $\widehat{\beta_0}$ and $\widehat{\beta_1}$ be the OLS estimates of the regression coefficients in Equation (4.2.1). Consider two test statistics: (i) sample coefficient of determination ($r^2$), calculated using the training data

used to fit the model, and (ii) Test Prediction Error (TPE) = $\frac{\sum_{t=1}^{T}((Y_t-\widehat{Y}_t)^2}{\sigma_\xi^2}$, where $Y_t$ are response values in the test data; $\widehat{Y}_t$ are the fitted values in the test data; and $\sigma_\xi^2$ is the variance of $\xi$.

Next we look at the various user defined lists specified for a SEED experiment.

### 4.2.1 SEED interface: user defined lists and variables

**Character string:** `typeOfExperiment`

`typeOfExperiment` contains a character string that indicates the mechanism for choosing feature values for the generative model. The value of `typeOfExperiment` is either `Uniform Space-Filling` or `Factorial Experiment`. In Example 4.2.1

```
typeOfExperiment = "Uniform Space-Filling".
```

**List:** `dataComponents`

`dataComponents` contains names of all the different components in the data that are generated using the generative model. In every MC simulation, data would be generated for each element is this list. The order of the elements in this list is important and must be preserved for related lists like `distributionNames` and `distributionParams`. Preserving the order in these lists help make the code general enough to be used by a wide range of simulation experiments involving different generative models. In Example 4.2.1 we will need to generate $Y_i, X_i$ and $\xi_i$ $(i = 1, 2, \ldots, n)$, in each simulation. Hence:

```
dataComponents = list("X","Error", "Y").
```

**List:** `distributionNames`

`distributionNames` is a list of names of classes of distribution families for every data

component generated using the generative model. For example if a element of the list `dataComponents` is `X` and it has a Gaussian Mixture distribution, the corresponding element in `distributionNames` is specified as `GAUSSMIX`. In case a new distribution class is added, its class name is specified for the corresponding element in `dataComponents`.

The two lists `dataComponents` and `distributionNames` are of equal length, since every data component's data generating mechanism needs to be specified. In case an element of `dataComponents` depends on the values of the other elements in the list, `conditional` is specified for it in `distributionNames`.

In Example 4.2.1 $X_i$ is drawn from a Student's $t$ distribution; $\xi_i$ is drawn from a univariate Gaussian mixture; and $Y_i$ is obtained using Equation (4.2.1) for $i = 1, 2, \ldots, n$. Thus using the names of the corresponding derived classes of `DISTRIBUTIONALFAMILIES`, for the first two different elements in `dataComponents`, and `conditional` for the third element in `dataComponents` we have,

`distributionNames = list("T_DIST","GAUSSMIX","conditional").`

**List:** `distributionParams`

`distributionParams` contains starting values for all parameters of the generative model. For each element in `dataComponents`, the distributional family is specified in `distributionNames`, and an appropriate vector or list initializing the class specified in `distributionNames` is specified in `distributionParams`. The order of all three lists should be preserved for the SEED code to work correctly. Although in most cases the actual parameters values are found by `GENMODEL`, specifying parameters in `distributionParams` is required for the dynamic definition of data generating objects for each element in `dataComponents`. If the `dataComponents` element is specified as `conditional` in `distributionNames`, `distributionParams` contains values for the arguments to the function

`dataForMethods`. The function `dataForMethods` is described in Section 4.2.2.

The starting parameters of the generative model in Example 4.2.1 are specified as follows. For the `dataComponents` element `"X"`, starting values of the degrees of freedom for a Student's $t$ distribution is given. For the `dataComponents` element `"Error"`, starting values of the 4 component Gaussian mixture distribution consisting of the weights, means, and standard deviations for the 4 univariate Normal components in the mixture, are given. For the `dataComponents` element `"Y"`, values for $\beta_0$ and $\beta_1$ are provided, which are passed to the function `methodsData`, to obtain the values of "`Y`" during code execution.

```
distributionParams = list(c(10),c(1:12),c(1,5)).
```

The next five user inputs provide information related to features of interest. These inputs are not needed for a stress testing experiment. These feature values determine the values of the parameters of the generative models in SEED. The variables are `fi1`,`fi2`,..., `fi5`, where `fi` stands for feature information. For Example 4.2.1 let the features of interest be the standard deviation, skewness and kurtosis of the $\xi$ distribution.

**List:** `fi1`

`fi1` is a list of the elements of `dataComponents` that are features of interest. The elements in this list are a subset of the element in `dataComponents`. This is a list of strings. In Example 4.2.1,

```
fi1 = list("Error").
```

**List:** `fi2`

For each element in the list `fi1`, `fi2` is used to specify the nature of the features, or how they are to be obtained. For each element in `fi1`, specify `analytic`, or `data driven`. For short, we refer to simulated data as data in this work. If `fi2` is `analytic`, the features for that element in `fi1` can be evaluated as a function of the parameters in the corresponding distribution in closed form. This implies that the required generative model can be obtained using stochastic search. If `fi2` is `data driven`, the features for that element in `fi1` are estimated using simulations. This implies that the required generative model is to be obtained using stochastic search and simulation optimization.

In Section 2.1.1 we derived standard deviation, skewness, and kurtosis for a univariate Gaussian Mixture. In Example 4.2.1 the features of interest can be evaluated analytically using the parameters of the generative model, hence:

```
fi2 = list("analytic").
```

**List:** `fi3`

For each element in the list `fi1`, `fi3` contains a list of the names of features. `fi3` defines the number of features of interest, and makes code easier to read and understand. Thus `fi3` is a list of lists. For Example 4.2.1,

```
fi3 = list(list("standard deviation","skewness","kurtosis")).
```

**List:** `fi4`

With `fi3` containing a list of the names of all features for every element in `fi1`, `fi4` contains the corresponding levels or ranges of interest for each of these features. If `typeOfExperiment` is `Uniform Space-Filling`, then `fi4` contains a list for each element

in `fi1`, that contains a list of ranges for each related feature. If `typeOfExperiment` is `Factorial Experiment`, then `fi4` contains a list for each element in `fi1`, that contains a complete list of factor levels of interest for each related feature. For Example 4.2.1 we specify the space of interest that we aim to cover uniformly using the generative models in SEED hence,

```
fi4 = list(list(list(0.5,5),list(-3,3),list(0.5,50))).
```

**List:** `fi5`

`fi5` is specified to be `combination` or `simOpt` if the features of interest are obtained analytically or estimated using simulation optimization respectively. In case there is only one element in `fi1`, and all features of interest can be obtained analytically as a function of the parameters, `fi5` is specified as `NULL`. In Example 4.2.1 since the features of interest are obtained analytically and we provide the function `featureCalcFuncCombi` (given in Section 4.2.2), we have:

```
fi5 = list("combination").
```

In this case `fi5 = NULL`, is also valid, if the function `featureCalcFunc` is provided. However, providing the function `featureCalcFuncCombi` will be useful if we decide to consider a new feature in the future, that involves the other elements in `dataComponents`.

**Lists:** `lowerBoundsFuncArgs` **and** `upperBoundsFuncArgs`

For the element of `dataComponents` corresponding to `distributionParams` specified as `conditional`, `lowerBoundsFuncArgs` and `upperBoundsFuncArgs` contain a list of lower and upper bounds respectively for each argument of the user defined function `methodsData`. The function `methodsData` is described in Section 4.2.2. For `dataComponents`

where `distributionParams` is not specified as `conditional`, `lowerBoundsFuncArgs` and `upperBoundsFuncArgs` are specified to be `NULL`. Order in `dataComponents` must be preserved in `lowerBoundsFuncArgs` and `upperBoundsFuncArgs`, for the SEED code to work correctly. In case there are no `conditional` elements in `distributionParams`, the lists `lowerBoundsFuncArgs` and `upperBoundsFuncArgs` need not be specified.

For Example 4.2.1 the lower and upper bounds of $\beta_0$ and $\beta_1$ is given in the following two lists. Note that for the other parameters in the generative model, the lower and upper bounds are provided by the function `getStartingParams` defined in the different derived classes of `DISTRIBUTIONALFAMILIES`. Hence,

```
lowerBoundsFuncArgs = list(NA,NA,c(rep(-Inf,-Inf))),
upperBoundsFuncArgs = list(NA,NA,c(rep(Inf,Inf))).
```

**Character string:** `optimMethod`

`optimMethod` specifies the method to be used by R's `optim` function for search of generative models corresponding to the target values of features. The options given in `optim` are `Nelder-Mead`, `BFGS`, `CG`, `L-BFGS-B`, `SANN`, and `Brent`. The default used by SEED is `L-BFGS-B`, as it is common for distributional families' parameters to have finite bounds, which is handled well by `L-BFGS-B`. In Example 4.2.1 the default value of `optimMethod` is used.

**List:** `optimList`

`optimList` is a list of options that are passed to R's `optim` function while searching for generative models corresponding to the target values of features. Providing this list is optional and the default values defined in R are used when `optimList` is not specified. In Example 4.2.1 the default value of `optimList` is used.

**Numeric:** `startingSeed`

A scalar which acts as a starting value for seeds used in random number generation in R. For each generative model, the value of the seed is incremented by one. For Example 4.2.1

`startingSeed = 1`.

**Numeric:** `nSampleLists`

A scalar sample size for each element in `dataComponents` used to generate data samples used by methods under study in each Monte Carlo (MC) iteration. Thus, when `dataComponents` has more than one element, `nSampleLists` is a vector with each of its elements denoting the sample size for the corresponding element in `dataComponents`. For Example 4.2.1 we use $n = 20$ hence,

`nSampleLists = c(rep(20,3))`.

**Numeric:** `testDataSize`

A scalar sample size for each element in `dataComponents` used to generate a test dataset used by methods under study. Thus, when `dataComponents` has more than one element, `testDataSize` is a vector with each of its elements denoting the sample size for the corresponding element in `dataComponents`. The test data is generated once for each generative model. For Example 4.2.1 we use a test data size of 5000 hence,

`testDataSize = c(rep(5000,3))`.

**Numeric:** `nSims`

Scalar number of Monte Carlo (MC) simulations used to estimate performance measures of the methods under study. Default value for `nSims` is 10000. In Example 4.2.1 the

default value of `nSims` is used.

**Numeric:** `optimTolerance`

Desired tolerance level to be used in the generative model search to look for a generative model corresponding to the target values of features. Default value for `optimTolerance` is 0.001. In Example 4.2.1 the default value of `optimTolerance` is used.

**Numeric:** `nStart`

Number of starting sets used in the generative model search until desired tolerance level is achieved. Default value for `nStart` is 10. In Example 4.2.1 the default value of `nStart` is used.

**Numeric:** `nGenModels`

Number of generative models to be considered in the simulation experiment. In Example 4.2.1 we use 2000 generative models hence,

```
nGenModels = 2000.
```

**Numeric:** `nMethods`

The total number of test statistics calculated using the methodologies under study in the SEED experiment. The function `methodsUserFunc` returns two test statistics in Example 4.2.1 hence

```
nMethods = 2.
```

**Character string:** `outputFilesDirectory`

`outputFilesDirectory` specifies the directory on the computer where the output files from SEED are stored.

Next we describe the input functions to be defined by the user. Although the user can define these functions in any file for his or her use, we recommend storing them in a separate file and sourcing it to the R file where SEED is used.

### 4.2.2 SEED interface: user defined functions

Let $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$ denote the parameters specifying a generative model in a SEED simulation study. And $f(\theta) = (f_1(\theta), f_2(\theta), \ldots, f_l(\Theta))$ denote the features of interest.

**Function:** `featureCalcFunc`, `featureCalcFuncCombi`, `featureCalcSimOpt`

These three functions calculate $f(\theta)$, given the value of $\theta$. `featureCalcFunc`, `featureCalcFuncCombi`, and `featureCalcFuncSimOpt` return a vector of length equal to the number of features of interest in the experiment. One of the three functions is used to calculate $f(\theta)$, the choice is made based on how $f(\theta)$ is calculated. This is described as follows:

- `featureCalcFunc` is used when the value of $f(\theta)$ depends on the distributional parameters of only one element in `dataComponents`, and $f(\theta)$ can be evaluated analytically given $\theta$. The input to `featureCalcFunc` is a vector containing the distributional parameter values of the element in `dataComponents` that corresponds to the feature value of interest.

- `featureCalcFuncCombi` is used in the case when `fi5` is specified as `combination`, and $f(\theta)$ can be evaluated analytically given $\theta$. The input to `featureCalcFuncCombi` is a vector containing all the parameters of the generative model.

- The function `featureCalcFuncSimOpt` is provided by the user if $f(\theta)$ cannot be evaluated analytically given $\theta$, and $f(\theta)$ is estimated using simulation optimization. Inputs to `featureCalcFuncSimOpt` are: (i) parameter values for all `dataComponents`; (ii) draws from the generative model which are used to implement the SEED simulation optimization algorithm as described in Chapter two; and (iii) test dataset sizes of each element is `dataComponents`, based on which feature values are estimated.

Based on the user input lists for Example 4.2.1, the function `featureCalcFuncCombi` is provided in this example. The function `featureCalcFuncCombi` is defined as follows.

```
featureCalcFuncCombi = function(theta)
{
  theta = theta[2:13]
  nMix = length(theta)/3
  pVec = theta[1:nMix]
  pVec = pVec/sum(pVec)#scale weights to probabilities
  muVec = theta[(nMix+1):(nMix*2)]
  sigVec = theta[((nMix*2)+1):(nMix*3)]
  popMixMu = sum(pVec*muVec)
  muVec = muVec-popMixMu
  popMixSd = sqrt(sum((sigVec^2)*pVec) +
                  sum((muVec^2)*pVec))
```

```
  pop3mt = sum((muVec^3)*pVec) +
    3*sum(muVec*(sigVec^2)*pVec)
  pop4mt = sum((muVec^4)*pVec) +
    6*sum((muVec^2)*(sigVec^2)*pVec)+
    3*sum((sigVec^4)*pVec)
  popMixSkew = pop3mt/(popMixSd^3)
  popMixKurt = pop4mt/(popMixSd^4)
  return(c(popMixSd,popMixSkew,popMixKurt))
}
```

**Function:** `makeParamsList`

This function returns a list of the form `distributionParams`, given a vector of parameter values of the generative model or $\theta$. This function needs to be defined if `fi5` is specified as `combination`. In Example 4.2.1, `makeParamsList` is defined as follows.

```
makeParamsList = function(theta)
{
  #distributionParams = list(c(10),c(1:12),c(1,5))
  listOfParams = list(theta[1],theta[2:13],theta[14:15])
  return(listOfParams)
}
```

**Function:** `methodsData`

This function returns a list containing a dataset, on which methodologies are applied during each MC run. The inputs to this function are draws of size `nSampleLists` for each `dataComponents`, and a vector of arguments that are used to obtain the elements

in `dataComponents`, which are specified as `conditional` in the user defined list `distributionNames`. This function uses the complete dataset drawn in an MC run and returns the data that is used as an input to the user defined function, `methodsUserFunc`.

In Example 4.2.1 we require a dataset containing $X_i$, and $Y_i$ $(i = 1, 2, \ldots n)$, to obtain least squares estimates of $\beta_0$ and $\beta_1$. In Example 4.2.1, `methodsData` is defined as follows.

```
methodsData = function(listOfdataValues,listOfArguments)
{
  b0=listOfArguments[1]
  b1=listOfArguments[2]
  data_x =listOfdataValues[[1]]
  err = listOfdataValues[[2]]
  data_y = b0 + (b1*data_x) + err
  return(list(data_x,data_y))
}
```

**Function:** `methodsUserFunc`

This function uses datasets obtained using draws in an MC run, applies all methodologies and returns a vector of test statistics in a vector of length `nMethods`. The input to `methodsUserFunc` are two lists containing the training and test datasets, each returned by `methodsData`. Since methodologies vary widely between experiments, input to this function depends on other user defined functions in order to provide flexibility to the user. At times methodologies are applied to a training dataset, but a test dataset is used to calculate a test statistic of interest. The function `methodsUserFunc` also inputs a test dataset for such cases.

In Example 4.2.1, training and test datasets are used to obtain the test statistics of

interest in this example. Here `methodsUserFunc` returns two test statistics: (i) sample $r^2$ using the training data, and (ii) $\sum_{t=1}^{T}((Y_t - \widehat{Y}_t))^2$, where $Y_t$ are observed response values in the test data, and $\widehat{Y}_t$ are the fitted values in the test data. A test data set size of 5000 is used. In Example 4.2.1 `methodsUserFunc` is defined as follows.

```
methodsUserFunc = function(inputData,testData)
{
  # From definition in function methodsData
  # Y is the second element in the list.
  responseVar = inputData[[2]]
  predictorVar = inputData[[1]]
  regData = data.frame(sapply(inputData,c))
  slrFit = lm(responseVar~predictorVar,data=regData)
  rawRsq = cor(responseVar, slrFit\$fitted.values)**2 # raw R2
  test_responseVar = testData[[2]]
  test_predictorVar = testData[[1]]
  b0hat = coefficients(slrFit)[1]
  b1hat = coefficients(slrFit)[2]
  testYhat = b0hat + b1hat*test_predictorVar
  predErrorTest = mean((test_responseVar-testYhat)^2)
  return(c(rawRsq,predErrorTest))
}
```

**Function:** `truthFunc`

This function returns the true values of the population quantities that are used along with test statistics calculated by the methods being studied, to obtain estimates of a

method's performance. A vector containing all parameters of the generative model, and a list of objects to generate data from the generative model are the arguments of this function. This function returns a vector of length `nMethods`. In Example 4.2.1, the required population quantity returned by `truthFunc` is $\sigma_\xi^2$. In Example 4.2.1 `truthFunc` is defined as follows.

```
truthFunc = function(dataCompParams,dataGenObjects)
{
  theta = dataCompParams[[2]]
  nMix = length(theta)/3
  pVec = theta[1:nMix]
  pVec = pVec/sum(pVec)#scale weights to probabilities
  muVec = theta[(nMix+1):(nMix*2)]
  sigVec = theta[((nMix*2)+1):(nMix*3)]
  popMixMu = sum(pVec*muVec)
  muVec = muVec-popMixMu
  popMixvar = sum((sigVec^2)*pVec) +
    sum((muVec^2)*pVec)
  truthVec = c(0,popMixvar)
  return(truthVec)
}
```

**Function:** `perfUserFunc`

This function returns the estimated performance measures for the methods under study. The inputs to this function are: (i) a matrix of test statistics calculated by the `methodsUserFunc` over all MC iterations for a generative model, and (ii) values returned by

truthFunc. Providing this function is optional. If perfUserFunc is not defined by the user, mean squared error, mean absolute deviation, expected value, and variance of the test statistics are used by default.

Using the values returned by methodsUserFunc and truthFunc, in Example 4.2.1, perfUserFunc is defined as follows.

```
perfUserFunc = function(estimatesByMethods,truthVals)
{
  nMethods = ncol(estimatesByMethods)
  estExpVal = c(rep(0,nMethods))
  estVar =  c(rep(0,nMethods))
  for (j in 1:nMethods)
  {
    estExpVal[j] = mean(estimatesByMethods[,j])
    estVar[j] = var(estimatesByMethods[,j])
  }
  estExpVal[2] = estExpVal[2]/truthVals[2]
  estVar[2] = estVar[2]/truthVals[2]
  PerfValue = c(estExpVal,estVar)
  return(PerfValue)
 }
```

Details on the classes in the SEED framework follows next. The descriptions refer to user inputs given in Sections 4.2.1 and 4.2.2.

### 4.2.3 SEED: in-built classes and functions

Figure 4.3 shows the framework in SEED and the flow of control from one component to the next. We use the `S4` object oriented structure available in R to make this framework modular, so that changes in one aspect of the the experiment has a minimal or no effect on the other parts. Each component is a class and can be accessed by defining an object of the class. In R a numeric vector is a collection of numbers, similarly a class may be thought of as a collection of variables and functions. There are eight virtual classes in the SEED structure. Each class is designed to perform a specific task of a generic statistical simulation experiment. Class descriptions in short is presented in Table 4.1. The main functions in a class, as shown in Figure 4.3, remain the same across different examples. However depending on the experiment, functions may be added to classes and existing function definitions may change. The flow of the SEED experiment remains the same.

**Class:** `ITERATOR`

`ITERATOR` provides a mechanism to specify the feature values of a generative model, that is the target values of $f(\theta)$. In case of a uniform space-filling experiment, target values are chosen from the user defined ranges of interest in `fi4`, with equal probability. The derived class `ITERATOR_DIRECT_UNIFORMSPACEFILLING` is used to obtain target values of $f(\theta)$ in this case. If the user wants to do a factorial experiment, a design matrix of feature levels is calculated. For a generative model, the corresponding row of the design matrix is used to obtain the target values of $f(\theta)$. In this case we used the derived class `ITERATOR_DIRECT_FACTORIALEXP`.

The main generic function in this class is `whichmodel` that returns a list of feature values that are the target $f(\theta)$ values for the next generative model. Inputs to `whichmodel`

include `fi3` and `fi4` (defined in Section 4.2.1).

**Class:** `GENMODEL`

`GENMODEL` is used to obtain the values of $\theta$ and specify the generative model given the target values of $f(\theta)$. `GENMODEL` is a virtual class with three main derived classes: `GenModelAnalytic`, `GenModelCombi`, and `GenModelSimOpt`. The derived class of `GENMODEL` used in a SEED experiment is related to the user defined functions `featureCalcFunc`, `featureCalcFuncCombi`, and `featureCalcSimOpt`.

`GenModelAnalytic` or `GenModelCombi` is used when the features of interest can be evaluated analytically from the parameters of the generative model, else `GenModelSimOpt` is used. The choice between `GenModelAnalytic` and `GenModelCombi` depends on the value of `fi5`. If `fi5` is `combination`, `GenModelCombi` is used to obtain the values of $\theta$ in the generative model. In this case the function `featureCalcFuncCombi` is provided by the user. If `fi5` = `simOpt`, `GenModelSimOpt` is used to obtain the values of $\theta$ for a generative model. In this case the function `featureCalcFuncSimOpt` is provided by the user. If `fi5` equals neither `combination` nor `simOpt`, `GenModelAnalytic` is used and the function `featureCalcFunc` is defined by the user.

The main generic function in `GENMODEL` is called `getModel`. Using user inputs such as `optimMethod`, `optimTolerance`, `nStart`, `truthFunc`, and one of `featureCalcFunc` or `featureCalcFuncCombi`, or `featureCalcSimOpt`; `getModel` is used to specify all details of the generative model, using stochastic search algorithms and simulation optimization when required. If $f_i^M(\theta)$ are the current generative model's feature values and $f_i^T(\theta)$ are the target values ($i = 1, 2, \ldots, l$), the objective function minimized by `getModel` is $\sum_{i=1}^{l} (f_i^M(\theta) - f_i^T(\theta))^2$. This function returns a list containing three lists:

- a list of data generating objects for each element in `dataComponents`;

- a list of all parameters values of the generative model corresponding to the target feature values, for each element in `dataComponents`; and
- a list of true population values of the generative model being estimated by the methodologies being compared in the simulation experiment.

**Class:** `SIMULATOR`

`SIMULATOR` is the class used to carry out MC simulations for the specified generative model, and estimate the performance of methods. `SIMULATOR` uses the classes `DISTRIBUTIONALFAMILIES`, `DATASETS`, and `METHODS`. The main generic function used by `SIMULATOR` is `simulate`. Using the list of data generating objects provided by `GENMODEL`, `simulate` does the following for `nIterations` number of MC runs:

- generates data for each element in `dataComponents` using `DISTRIBUTIONALFAMILIES`;
- uses `DATASETS` to obtain a list containing the dataset to be used by different methodologies being compared;
- passes the list returned by `DATASETS` to `METHODS` to obtain `nMethod` test statistics using methodologies under study in the SEED experiment.

`nIterations` test statistics are then compared to the true values of the population quantities using the outputs from `GENMODEL` and `truthFunc`, and performance measures are obtained using the function `perfUserFunc`.

**Class:** DISTRIBUTIONALFAMILIES

DISTRIBUTIONALFAMILIES is a virtual class, using two main generic functions: getSample and getStartingParams. The definitions of getSample and getStartingParams are different across the derived classes of DISTRIBUTIONALFAMILIES. For a specified sample size getSample returns draws from the distributional family of the derived class. getStartingParams returns the starting values, lower bounds, and upper bounds of the distributional family parameters corresponding to its derived class. We have five derived classes of DISTRIBUTIONALFAMILIES: (i) GAUSSMIX for a mixture of univariate Gaussian distributions; (ii) MULTIGAUSSMIX for a mixture of multivariate Gaussian distributions; (iii) NORMAL_DIST_UNI for a univariate Gaussian distribution; (iv) T_DIST for a univariate Student's t distribution; and (v) UNIFORM_DIST for a Uniform distribution. The names of these classes are specified in distributionNames for each element in dataComponents. A user can define a derived class of DISTRIBUTIONALFAMILIES in the R file userDefinedClasses.R, and define for getSample and getStartingParams for the new derived class. The new derived class can be then be used in SEED.

**Class:** DATASETS

This class is used to obtained datasets in a form that can be used by the methodologies being studied in the experiment. The main function used is getDataset, which calls the user defined function methodsData. The user defined function methodsData returns a dataset that can be used directly by the user defined function methodsUserFunc. See Section 4.2.2 for details about both these functions.

**Class:** `METHODS`

The `METHODS` class is used to apply the methodologies under study in the SEED experiment, to the MC datasets returned by `methodsData`. The generic function used in this class is `evaluateMethods` which uses the function `applyAllMethods` and returns test statistics of interest.

**Class:** `RECORDER`

`RECORDER` is used to write all information about the simulation experiment to files. The following details for all generative models used in the experiment are recorded: (i) seeds, (ii) feature values, (iii) parameter values, and (iv) estimated performance measures.

### 4.2.4 Examples

We consider two regression examples to explain use of the SEED R package for designing simulation experiments. These two simple examples further illustrate the different user inputs described in Sections 4.2.1, 4.2.2, and 4.2.3. User inputs for the Example 4.2.1 are given in Sections 4.2.1 and 4.2.2. To run SEED, load the SEED R package on the computer. The function `sfexp` is used to run a space-filling SEED experiment. All the user inputs described in Sections 4.2.1 and 4.2.2 are the arguments to this function. Hence for Example 4.2.1, we define the various user inputs as given in Sections 4.2.1 and 4.2.2 and call the `sfexp` function in the SEED R package.

```
simResults = sfexp(typeOfExperiment = typeOfExperiment,
                   dataComponents = dataComponents,
                   distributionNames = distributionNames,
                   distributionParams= distributionParams,
```

```
fi1=fi1,fi2=fi2,fi3=fi3,fi4=fi4,fi5=fi5,
lowerBoundsFuncArgs=lowerBoundsFuncArgs,
upperBoundsFuncArgs = upperBoundsFuncArgs,
nSampleLists=nSampleLists,
testDataSize=testDataSize,
nGenModels= nGenModels,
nMethods=nMethods,
featureCalcFuncCombi=featureCalcFuncCombi,
makeParamsList=makeParamsList,
methodsData= methodsData,
methodsUserFunc=methodsUserFunc,
truthFunc=truthFunc,
perfUserFunc=perfUserFunc,
outputFilesDirectory = outputFilesDirectory)
```

At the end of the SEED experiment, the results from the experiment are written in five files by the class `RECORDER`. For each generative model the following details are recorded:

- seeds are written to `allSeeds.csv`;
- feature values are written to `allGenmodelFeatures.csv`;
- parameter values are written to `allGenmodelParams.csv`;
- values returned by the function `truthFunc` are written to `allTruthVals.csv`; and
- estimated performance measures are written to `allGenmodelPerfs.csv`.

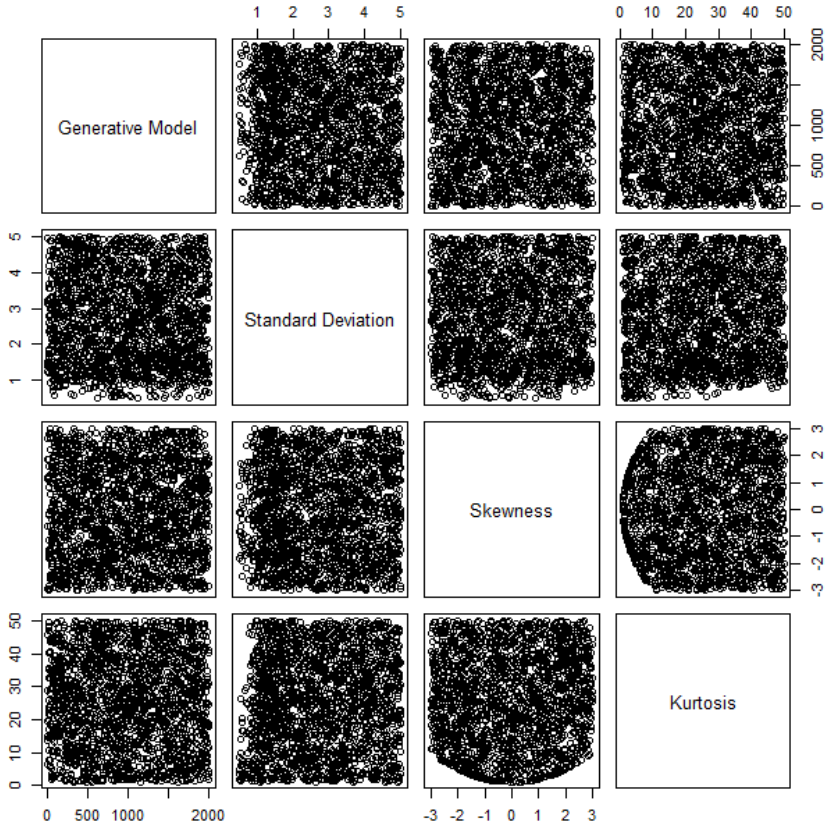


Figure 4.4: Coverage achieved in the feature space of interest in Example 4.2.1 using 2000 generative models.

The results for Example 4.2.1 are shown in Figures 4.4, 4.5, 4.6 and 4.7. In Figure 4.4, we find that the features space of interest is covered uniformly over the admissible region. In Chapter two, we saw that the admissible region in given by Equation (2.1.6). As expected performance of both the test statistics is random across the generative models (Figure 4.5). Figure 4.6 shows that the average sample $r^2$ decreases with increasing standard deviation. The average values of sample $r^2$ do not seem to be effected by the error skewness (Figure 4.6). For cases with higher values of kurtosis in the distribution

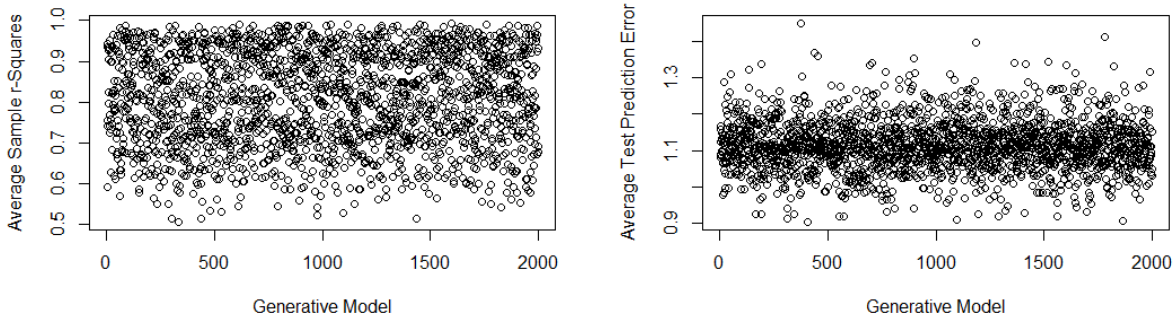


Figure 4.5: Performance of OLS estimators over 2000 generative models in Example 4.2.1. Test Prediction Error is defined as $\frac{\sum_{t=1}^{5000}(Y_t-\widehat{Y}_t)^2}{\sigma_\xi^2}$, where $Y_t$ and $\widehat{Y}_t$ are the observed and fitted response values in the test data respectively ($t = 1, 2, \ldots, 5000$); and $\sigma_\xi^2$ is the variance of $\xi$. Both performance measures are estimated using 10000 MC samples.

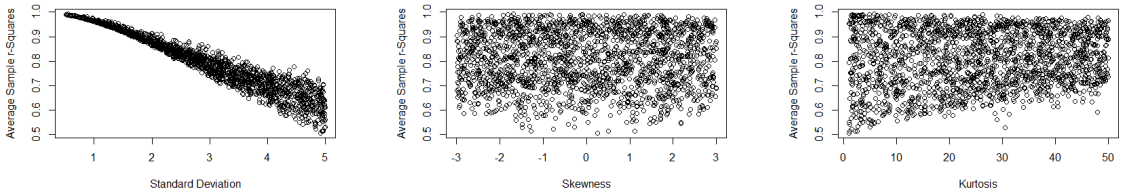


Figure 4.6: Performance of OLS estimators over different feature values in Example 4.2.1 using 2000 generative models. Performance measures are estimated using 10000 MC samples.

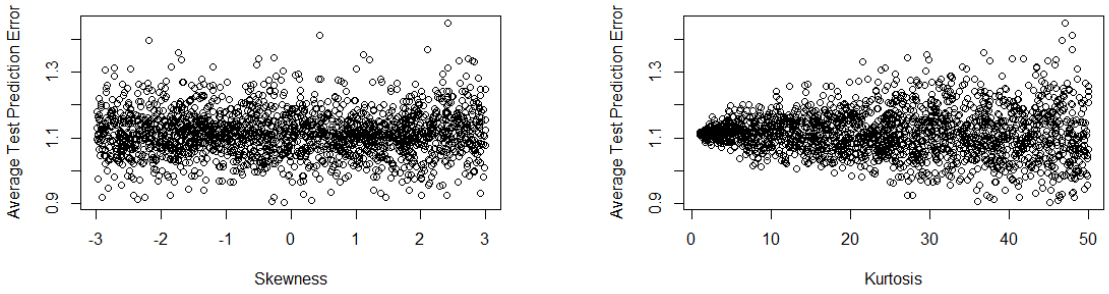


Figure 4.7: Performance of least squares estimates over different feature values in Example 4.2.1 using 2000 generative models. Test Prediction Error is defined as $\frac{\sum_{t=1}^{5000}((Y_t-\widehat{Y}_t)^2}{\sigma_\xi^2}$, where $Y_t$ and $\widehat{Y}_t$ are the observed and fitted response values in the test data respectively ($t = 1, 2, \ldots, 5000$); and $\sigma_\xi^2$ is the variance of $\xi$. Performance measures are estimated using 10000 MC samples.

of $\xi$, average sample $r^2$ values seem to be less varied (Figure 4.6). Figure 4.7 shows that the average TPE values have higher variance for higher kurtosis in in the distribution of $\xi$. The average TPE values do not seem to be effected by changes in the skewness in the distribution of $\xi$ (Figure 4.7).

### A logistic regression example

Consider an example to compare the performance of generalized linear model (GLM) estimates using the R package `glm` and ordinary least squares (OLS) estimates using the

R package `lm`, for a logistic regression model defined as follows:

$$\begin{aligned} X_i &\sim \text{Gaussian Mixture } (p_{1X}, \ldots, p_{4X}, \mu_{1X}, \ldots, \mu_{4X}, \sigma_{1X}, \ldots, \sigma_{4X}), \\ \xi_i &\sim \text{Normal } (0, 1), \\ p_{Y_i} &= \frac{\exp(\beta_0 + \beta_1 X_i + \xi_i)}{1 + \exp(\beta_0 + \beta_1 X_i + \xi_i)}, \\ Y_i &\sim \text{Bernoulli } (p_{Y_i}), \end{aligned} \tag{4.2.2}$$

for $i = 1, 2, \ldots, n$. The values of $\beta_0$ and $\beta_1$ are 0.5 and 1 respectively. We consider a space filling experiment with the features of interest as the standard deviation, skewness and kurtosis of $X_i$ ( $i = 1, 2, \ldots, n$.).

```
typeOfExperiment = "Uniform Space-Filling"
```

For each MC run we will need to generate $Y_i, X_i$ and $\xi_i$, $i = 1, 2, \ldots n$. Hence:

```
dataComponents = list("X","Error", "Y").
```

$X_i$, is drawn from a univariate Gaussian mixture; $\xi_i$ is drawn from a univariate Gaussian distribution; and $Y_i$ is obtained using Equation (4.2.2), for $i = 1, 2, \ldots, n$. In this example we use $n = 50$. Thus using the names of the corresponding derived classes of `DISTRIBUTIONALFAMILIES`, for the first two different elements in `dataComponents`, and `conditional` for the third element in `dataComponents` we have,

```
distributionNames = list("GAUSSMIX","NORMAL_DIST_UNI","conditional").
```

Next the starting parameters of the generative model are specified. For the `dataComponents` element `"X"`, starting values of the 4 component Gaussian mixture distribution consisting of the weights, means, and standard deviations for the 4 univariate Normal

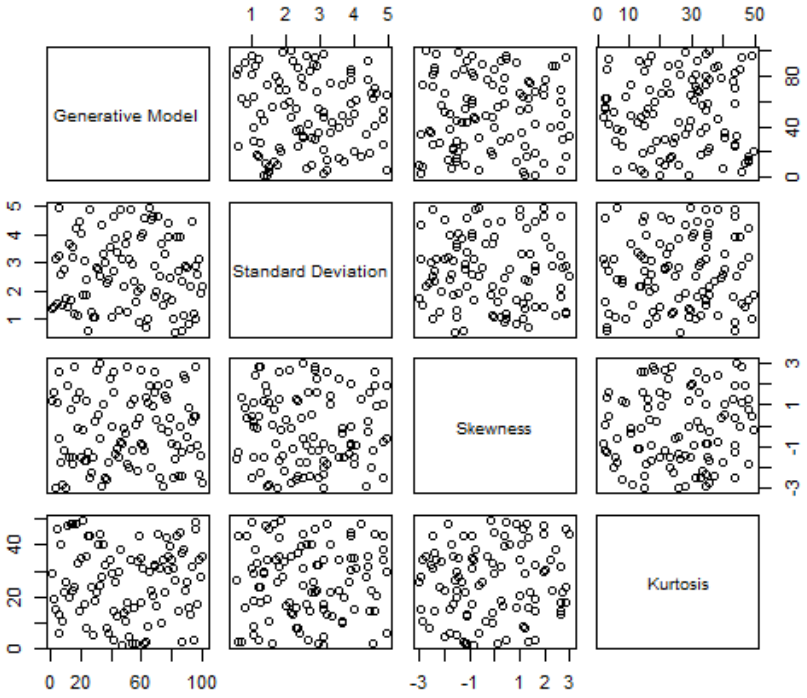


Figure 4.8: Coverage achieved in the feature space of interest in the logistic regression example given in Section 4.2.4 using 100 generative models.

components in the mixture, are given. For the `dataComponents` element `"Error"`, starting values of the mean and variance of a Normal distribution is provided. For the `dataComponents` element `"Y"`, values for $\beta_0$ and $\beta_1$ are provided, which are passed to the function `methodsData`, to obtain the values of "`Y`" during code execution.

```
distributionParams = list(c(1:12),c(0,1),c(1,0.5)).
```

Let the features of interest be the standard deviation, skewness and kurtosis of the $X$ distribution.

```
fi1 = list("X")
```

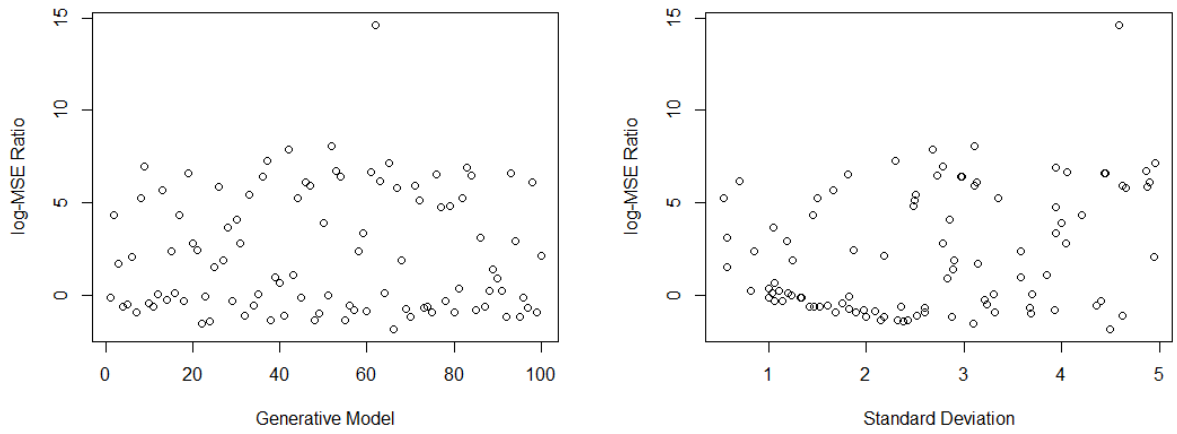


Figure 4.9: Estimated log-mean squared error ratio of GLM and OLS estimates of $\beta_1$, for 100 generative models plotted against generative model number and standard deviation of $X$ (see Equation (4.2.2)). Larger values are favorable to OLS. The log-mean squared error ratios are estimated using 10000 MC samples.

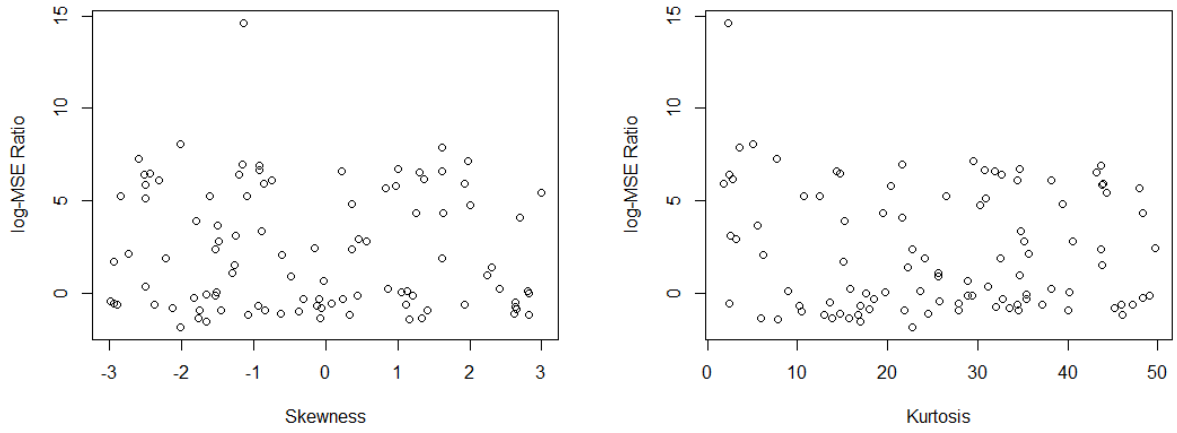


Figure 4.10: Estimated log-mean squared error ratio of GLM and OLS estimates of $\beta_1$, for 100 generative models plotted against skewness and kurtosis of $X$ (see Equation (4.2.2)). Larger values are favorable to OLS. The log-mean squared error ratios are estimated using 10000 MC samples.

In Section 2.1.1 we derive standard deviation, skewness, and kurtosis for a univariate Gaussian Mixture. The features of interest here can be evaluated analytically using the parameters of the generative model, hence:

```
fi2 = list("analytic").
```

The names of the features of interest are specified in `fi3`.

```
fi3 = list(list("standard deviation","skewness","kurtosis"))
```

Ranges of feature values for the space-filling experiment are specified next. This defines the space of interest that we aim to cover uniformly using the generative models in SEED.

```
fi4 = list(list(list(0.5,5),list(-3,3),list(0.5,50)))
```

Since we define the function `featureCalcFuncCombi`, we have:

```
fi5 = list("combination")
```

In this case `fi5 = NULL`, is also valid, if the function `featureCalcFunc` is provided. However, providing the function `featureCalcFuncCombi` will be useful if we decide to consider a new feature in the future, that involves the other elements in `dataComponents`. Next the lower and upper bounds of $\beta_0$ and $\beta_1$ is given in the following two lists. Note that for the other parameters in the generative model, the lower and upper bounds are provided by the function `getStartingParams` defined in the different derived classes of `DISTRIBUTIONALFAMILIES`.

```
lowerBoundsFuncArgs = list(NA,NA,c(rep(-Inf,-Inf)))
upperBoundsFuncArgs = list(NA,NA,c(rep(Inf,Inf)))
```

Default values of `optimMethod`, `optimList`, `optimTolerance`, `nIterations` and `nStart` are used. We use 100 generative models in this example. The other variables are:

```
nGenModels = 100
```

For this example we use $n = 50$, hence

```
nSample = c(rep(50,3)).
```

Also we use five cores for running the code in this example, hence

```
nCores = 5.
```

The function `methodsUserFunc` returns four test statistics, hence

```
nMethods = 4.
```

The user defined functions for this example are given next. The function `featureCalcFuncCombi` takes as input the vector $\theta$ and returns the value of $f(\theta)$.

```
featureCalcFuncCombi = function(theta)
{
  theta = theta[1:12]
  nMix = length(theta)/3
  pVec = theta[1:nMix]
  pVec = pVec/sum(pVec)#scale weights to probabilities
  muVec = theta[(nMix+1):(nMix*2)]
  sigVec = theta[((nMix*2)+1):(nMix*3)]
  popMixMu = sum(pVec*muVec)
  muVec = muVec-popMixMu
  popMixSd = sqrt(sum((sigVec^2)*pVec) +
                  sum((muVec^2)*pVec))
  pop3mt = sum((muVec^3)*pVec) +
```

```
    3*sum(muVec*(sigVec^2)*pVec)
  pop4mt = sum((muVec^4)*pVec) +
    6*sum((muVec^2)*(sigVec^2)*pVec)+
    3*sum((sigVec^4)*pVec)
  popMixSkew = pop3mt/(popMixSd^3)
  popMixKurt = pop4mt/(popMixSd^4)
  return(c(popMixSd,popMixSkew,popMixKurt))
}
```

`makeParamsList` returns a list of parameters in the format of the list `distributionParams`. In this example, the `makeParamsList` is as follows:

```
makeParamsList = function(theta)
{
  listOfParams = list(theta[1:12],theta[13:14],theta[15:16])
  return(listOfParams)
}
```

`methodsData` is used to manipulate the MC datasets and return a list directly usable by the methodologies under study. In this case we require a dataset containing $X_i$, and $Y_i$ $(i = 1, 2, \ldots n)$, to obtain least squares estimates of $\beta_0$ and $\beta_1$. Notice that here we did not use the error observations drawn using the generative model, rather we draw samples from a standard Normal distribution within the function `methodsData`. This is because distributional parameters of the elements in `dataComponents` which are not specified in `fi1`, are chosen at random for each generative model. In this case when a Normally distributed $\xi$ with a randomly chosen mean and variance was used, the GLM algorithm does not converge. Hence we fix a standard normal distribution for $\xi$, using the function

methodsData. A drawback of the current SEED package is highlighted here, although the function methodsData provides flexibility to overcome it in this example. For future improvement of this package we would like to have the option of fixing values of the distributional parameters of the elements in dataComponents which are not specified in fi1.

```
methodsData = function(listOfdataValues,listOfArguments)
{
  b0=listOfArguments[1]
  b1=listOfArguments[2]
  data_x =listOfdataValues[[1]]
  nSample = length(data_x)
  err = rnorm(nSample)
  probYis1 = exp(b0 + (b1*data_x) + err)/(1+exp(b0 + (b1*data_x) + err))
  data_y = unlist(lapply(probYis1,
                    function(x) rbinom(n=1, size=1, prob=x)))
  return(list(data_x,data_y))
}
```

In this example methodsUserFunc returns the estimated regression coefficients found using the R packages glm and lm, as the test statistics of interest.

```
methodsUserFunc = function(inputData,testData)
{
  #from definition in function methodsData the user
  #defined Y to be the second element in the list.
  responseVar = inputData[[2]]
```

```
  predictorVar = inputData[[1]]
  regData = data.frame(sapply(inputData,c))
  glmFit = glm(responseVar~predictorVar,data=regData,
             family=binomial())
  glmCoeffs = coef(glmFit)
  slrFit = lm(responseVar~predictorVar,data=regData)
  slrCoeffs = coefficients(slrFit)
  return(c(glmCoeffs,slrCoeffs))
}
```

`truthFunc` returns the true values of the regression coefficients in the model specified by the Equation (4.2.2).

```
truthFunc = function(dataCompParams,dataGenObjects)
{
  trueBetaVals = dataCompParams[[3]]
  truthVec = c(trueBetaVals,trueBetaVals)
  return(truthVec)
}
```

Using the values returned by `methodsUserFunc` and `truthFunc`, `perfUserFunc` returns estimates of mean squared errors of the regression coefficient estimates.

```
perfUserFunc = function(estimatesByMethods,truthVals,nMethods)
{
  estMSE = c(rep(0,nMethods))
  estExpVal = c(rep(0,nMethods))
  estVar =  c(rep(0,nMethods))
```

```
  for (j in 1:nMethods)
  {
    estMSE[j] = mean((estimatesByMethods[,j]-truthVals[j])^2)
    estExpVal[j] = mean(estimatesByMethods[,j])
    estVar[j] = var(estimatesByMethods[,j])
  }
  PerfValue = c(estMSE,estExpVal,estVar)
  return(PerfValue)
}
```

Note that inputs to SEED for the simple linear regression example and the logistic regression example are very similar. In fact changing the function `methodsData` only, would be sufficient to run SEED for the OLS case in the logistic regression example, provided all other factors like distributions of $X$ and $\xi$, performance measures, etc. are kept constant. For the logistic regression example, using a Gaussian mixture distribution for $\xi$ like in the simple linear regression example leads to convergence problems in the GLM method. Thus we consider different generative models and features in the two examples, which require changes to the other user input lists and functions.

At the end of the SEED experiment, the results from the experiment are written in five files by the class `RECORDER`. These files are stored in the directory specified by the user in `outputFilesDirectory`. These five files contain detailed information about the SEED experiment that can greatly aid reproducibility and repeatability of the experiment. For each generative model the following details are recorded:

- seeds used for pseudo random number generation are written to `allSeeds.csv`;
- feature values are written to `allGenmodelFeatures.csv`;

- parameter values are written to `allGenmodelParams.csv`;
- values returned by the function `truthFunc` are written to `allTruthVals.csv`; and
- estimated performance measures are written to `allGenmodelPerfs.csv`.

Results from this example are shown is Figures 4.8, 4.9, and 4.10. In Figure 4.8, we find that the features space of interest is covered uniformly using 100 generative models. As expected performance of both the test statistics is random across the generative models (see Figure 4.8). However the performance measure does not seem to be affected by the feature values, indicating the need for more generative models and different features (see Figures 4.9 and 4.10).

In this chapter the framework of a SEED experiment was presented in detail. An R package for SEED is discussed, giving details of its properties, classes, and functions. Using the SEED framework we facilitate reproducibility and repeatability in two ways. First, an extensive simulation experiment using SEED is broken into specific parts and implemented using classes and OOP, making the simulation code easier to understand and modify. Second, all details of a SEED experiment is recorded systematically. Using the recorded details of the experiment, along with a clear description of the methodology, a reader will find it easier to reproduce or repeat experimental results.

# Chapter 5

# Conclusions and Future Work

In SEED we present a framework to do rigorous statistical simulation experiments that lead to more general conclusions than conclusions from a typical Monte Carlo (MC) study. Table 5.1 shows a comparison of SEED with typical MC experiments. In SEED we consider different features of interest that potentially impact the methodologies under study. We then proceed to uniformly fill the space of features of interest by using a very large number of flexible generative models. The features of interest can be any function of the parameters of the generative model, that can be derived analytically or estimated using simulations. Mathematical optimization algorithms are used to obtain generative models corresponding to specific feature values. In the case when feature values are estimated using simulations, we present a simulation optimization algorithm to obtain generative models corresponding to a specific feature value. Using the SEED simulation optimization algorithm we also look for feature values where methods under study break down or perform outstandingly well. At the end of a SEED experiment, results are thoroughly analyzed using statistical models.

SEED is designed using object oriented programming (OOP) so that results from

a statistical simulation experiment can be reproduced and replicated more easily. The modular SEED code ensures that changes to one part of the code have little or no effect on the other parts of the code. For example a new feature can be considered by making changes to only one part of the code; and a new method can be studied in the experiment by making minimal changes. We present a SEED R package to implement a SEED experiment for a wide range of MC studies. The SEED package stores all details of the experiment such as seeds used for pseudo random number generation; parameter values for all the generative models; all feature values; and values of performance measures for each generative model. Sharing details on all settings of the simulation experiment aids transparency, and facilitates reproducibility and replicability of experimental results.

There is significant scope for further research in SEED. In this work we considered a uniform space-filling design. Different space-filling criterion for filling the space of features of interest can be explored for cases where each generative model requires a high amount of computing time. For problems where computing is expensive, optimal ways for parallel computing in SEED can be studied, in additional to parallelizing over MC simulations within each generative model, as done in this work. Implementation of SEED using in a dynamic document that combines SEED code and formal documentation of the methods under study would enable a reader to replicate simulation results with further ease. Constructing a response surface of the performance of methods on the space of features of interest including searching for areas of excellent and poor performance on the fitted response surface, may be another area of future work.

Table 5.1: Comparison between a SEED experiment and typical MC experiments.

| Typical MC experiments | SEED |
|---|---|
| Small number of generative models | Millions of generative models |
| Limited justification of choice of generative models | Generative models systematically chosen to cover a space of features of interest. |
| Simplistic choice of features | Features are any functions of the generative model that potentially affect performance of methods under study and can be derived analytically or estimated using simulations. |
| No formal analysis of simulation findings | Formally model the relation between features and outcomes in the experiment |
| Generative models where methods perform poorly typically not considered | Search for generative models where methods do well as well as poorly |
| Results are laborious to reproduce and replicate | An OOP approach used to aid reproducibility and replicability |

# REFERENCES


Alfons A. 2011. simframe: Simulation framework. R package version 0.4. 4.

Ange B, Symons J, Schwab M, Howell E, Geyh A. 2004. Generalizability in epidemiology: an investigation within the context of heart failure studies. Annals of Epidemiology. 14:600–601.

Ankenman B, Nelson BL, Staum J. 2010. Stochastic kriging for simulation metamodeling. Operations research. 58:371–382.

ASTM. 2014. What are repeatability and reproducibility? [Online; accessed 1-August-2014].

Atkinson AC, Donev AN, Tobias RD. 2007. Optimum experimental designs, with SAS. Oxford University Press Oxford.

Barr RS, Golden BL, Kelly JP, Resende MG, Stewart Jr WR. 1995. Designing and reporting on computational experiments with heuristic methods. Journal of Heuristics. 1:9–32.

Baumer B, Cetinkaya-Rundel M, Bray A, Loi L, Horton NJ. 2014. R markdown: integrating a reproducible analysis tool into introductory statistics. arXiv preprint arXiv:1402.1894. .

Burton A, Altman DG, Royston P, Holder RL. 2006a. The design of simulation studies in medical statistics. Statistics in medicine. 25:4279–4292.

Burton A, Altman DG, Royston P, Holder RL. 2006b. The design of simulation studies in medical statistics. Statistics in medicine. 25:4279–4292.

Byrd R, Lu P, Nocedal J, Zhu C. 1995. A limited memory algorithm for bound constrained optimization. SIAM Journal on Scientific Computing. 16:1190–1208.

Chambers J. 2008. Software for data analysis: programming with R. Springer.

Claerbou JF, Karrenfach M, et al. (3 co-authors). 1992. Electronic documents give reproducible research a new meaning. In: 1992 SEG Annual Meeting. Society of Exploration Geophysicists.

Collins LM, Murphy SA, Strecher V. 2007. The multiphase optimization strategy (most) and the sequential multiple assignment randomized trial (smart): new methods for more potent ehealth interventions. American journal of preventive medicine. 32:S112–S118.

Conlisk J. 1974. Optimal response surface design in monte carlo sampling experiments. In: Annals of Economic and Social Measurement, Volume 3, number 3, NBER, pp. 17–28.

Dalle O. 2013. Using computer simulations for producing scientific results: Are we there yet? In: NS3 Workshop.

Dave G. 1993. Replicability, repeatability, and reproducibility of embryo-larval toxicity tests with fish. Progress in standardization of aquatic toxicity tests. Boca RatonFlorida: Lewis Publishers. pp. 129–157.

De Leeuw J. 2001. Reproducible research. the bottom line. Department of Statistics, UCLA. .

Donoho DL. 2010. An invitation to reproducible computational research. Biostatistics. 11:385–388.

Donoho DL, Maleki A, Rahman IU, Shahram M, Stodden V. 2009. Reproducible research in computational harmonic analysis. Computing in Science & Engineering. 11:8–18.

Forster MR. 2000. Key concepts in model selection: Performance and generalizability. Journal of mathematical psychology. 44:205–231.

Gelman A, Jakulin A, Grazia Pittau M, Su YS. 2008. A weakly informative default prior distribution for logistic and other regression models. The Annals of Applied Statistics. 2:1360–1383.

Gentleman R, Lang DT. 2007. Statistical analyses and reproducible research. Journal of Computational and Graphical Statistics. 16.

Geyer C. 1992. Practical markov chain monte carlo. Statistical Science. 7:473–483.

Goodman AF. 2011. Analysis, biomedicine, collaboration, and determinism challenges and guidance: Wish list for biopharmaceuticals on the interface of computing and statistics. Journal of biopharmaceutical statistics. 21:1140–1157.

Gosavi A. 2003. Simulation-based optimization: parametric optimization techniques and reinforcement learning, volume 25. Springer.

Hastie T, Tibshirani R, Friedman JJH. 2001. The elements of statistical learning, volume 1. Springer New York.

Hernández MA, Stolfo SJ. 1998. Real-world data is dirty: Data cleansing and the merge/purge problem. Data mining and knowledge discovery. 2:9–37.

Horton NJ, Baumer BS, Wickham H. 2014. Teaching precursors to data science in introductory and second courses in statistics. arXiv preprint arXiv:1401.3269. .

http://reproducibleresearchnet/. ???? Reproducible research. [Online; accessed 7-August-2014].

Johnson NL. 1949. Systems of frequency curves generated by methods of translation. Biometrika. 36:pp. 149–176.

Joseph VR, Hung Y, Sudjianto A. 2008. Blind kriging: A new method for developing metamodels. Journal of mechanical design. 130:031102.

Kirkpatrick S, Gelatt CD, Vecchi MP. 1983. Optimization by simulated annealing. Science. 220:pp. 671–680.

Kleijnen JP. 2008. Design and analysis of simulation experiments, volume 111. Springer.

Kleijnen JP, Sanchez SM, Lucas TW, Cioppa TM. 2005. State-of-the-art review: a users guide to the brave new world of designing simulation experiments. INFORMS Journal on Computing. 17:263–289.

Knuth DE. 1984. Literate programming. The Computer Journal. 27:97–111.

Laber EB, Linn K, Stefanski LA. 2014. Interactive q-learning. Biometrika. .

Lafore R. 1997. Object-oriented programming in C++. Pearson Education.

Ledgerwood A. 2014. Introduction to the special section on advancing our methods and practices. Perspectives on Psychological Science. 9:275–277.

Leisch F. 2002. Sweave, part i: Mixing r and latex. R News. 2:28–31.

Lenth R, Hjsgaard S. 2011. Reproducible statistical analysis with multiple languages. Computational Statistics. 26:419/–426.

Li R, Sudjianto A. 2005. Analysis of computer experiments using penalized likelihood in gaussian kriging models. Technometrics. 47.

Liu J. 2008. Monte Carlo strategies in scientific computing. Springer.

Lovell DP. 2012. Commentary: statistics for biomarkers. Biomarkers. 17:193–200.

McLachlan G, Peel D. 2004. Finite mixture models. John Wiley & Sons.

Murdoch DJ, Carey VJ. 2001. On the edge: Statistics & computing: Literate statistical programming: Concepts and tools. Chance. 14:46–50.

Nolan D, Lang DT. 2010. Computing in the statistics curricula. The American Statistician. 64.

Nolan D, Temple Lang D. 2009. Comment. The American Statistician. 63:117/–121.

of Work I, Health. 2006. What researchers mean by .. generalizability. [Online; accessed 7-August-2014].

Orcutt GH, Winokur Jr HS. 1969. First order autoregression: inference, estimation, and prediction. Econometrica: Journal of the Econometric Society. pp. 1–14.

Peng RD. 2009. Reproducible research and biostatistics. Biostatistics. 10:405–408.

Peng RD. 2011. Reproducible research in computational science. Science (New York, Ny). 334:1226.

Peng RD, Dominici F, Zeger SL. 2006. Reproducible epidemiologic research. American journal of epidemiology. 163:783–789.

Porta MS, Greenland S, Hernán M, Silva IDS, Last JM. 2014. A dictionary of epidemiology. Oxford University Press.

Prinz F, Schlange T, Asadullah K. 2011. Believe it or not: how much can we rely on published data on potential drug targets? Nature reviews Drug discovery. 10:712–712.

Pronzato L, Müller WG. 2012. Design of computer experiments: space filling and beyond. Statistics and Computing. 22:681–701.

Rossini A, Lumley T, Leisch F. 2003. On the edge: Statistics & computing: Reproducible statistical research. Chance. 16:41–45.

Rupp M, Gini F, Pérez-Neira A, Pesquet-Popescu B, Pikrakis A, Sankur B, Vandewalle P, Zoubir A. ???? Reproducible research in signal processing. .

Sacks J, Welch WJ, Mitchell TJ, Wynn HP. 1989. Design and analysis of computer experiments. Statistical science. 4:409–423.

Shaffer JP. 1995. Multiple hypothesis testing. Annual review of psychology. 46:561–584.

Smith-Spangler CM. 2012. Transparency and reproducible research in modeling why we need it and how to get there. Medical Decision Making. 32:663–666.

Stigler SM. 1973. The asymptotic distribution of the trimmed mean. The Annals of Statistics. pp. 472–477.

Stodden VC. 2011. Trust your science? open your data and code. Amstat News. 409:21–22.

Van Laarhoven PJ, Aarts EH. 1987. Simulated annealing. Springer.

Wang Y, Day R. 2010. An r package for simulation experiments evaluating clinical trial designs. AMIA Summits on Translational Science Proceedings. 2010:61.

Writing@CSU. 2014. Generalizability and transferability. [Online; accessed 7-August-2014].

Yekutieli D, Benjamini Y. 1999. Resampling-based false discovery rate controlling multiple test procedures for correlated test statistics. Journal of Statistical Planning and Inference. 82:171–196.