A FRAMEWORK FOR PROBABILISTIC ANALYSIS OF PROGRAMS

by

Paul Douglas Hein

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

2019

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Master's Committee, we certify that we have read the thesis prepared by Paul Douglas Hein, titled A Framework for Probabilistic Analysis of Programs and recommend that it be accepted as fulfilling the thesis requirement for the Master's Degree.

Mihai Surdeanu — Date: 3 May 2019

Clayton Morrison — Date: 3 May 2019

Richard Snodgrass — Date: 3 May 2019

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to the Graduate College.

I hereby certify that I have read this thesis prepared under my direction and recommend that it be accepted as fulfilling the Master's requirement.

Mihai Surdeanu — Date: 3 May 2019
Associate Professor
Department of Computer Science

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The pervasiveness of the use of scientific models in all fields of research has led to model selection being a critical and challenging component of the scientific process for modern day research. Given a phenomena of interest, the model selection problem includes the following challenges: identifying which models exist, accessing the models in a form that allows for observation and experimentation, and comparing the models by some metric in order to determine which model is best suited for a given set of experimental criterion. Modern day researchers are required to overcome all of these challenges with little computational aides if they wish to be certain that the models they are using to observe phenomena do represent the best that the state-of-the-art research in their field of study has to offer. In this thesis I will present a computational tool that automates the process of extracting scientific models that are present in scientific source code, grounding the extracted models to ancillary documents, and efficiently analyzing the models both individually and comparatively to facilitate automatic model selection.

CHAPTER 1

INTRODUCTION

When studying a given natural phenomena, many researchers turn to modeling as a method to formalize their reasoning about the phenomena. Scientific models are a way of studying how phenomena in the natural world affect one another. Using models to study phenomena also comes with many advantages for researchers, including:

- Models allow researchers to clearly communicate the relations between variables observed to be associated with a phenomena.
- Models explicitly represent the uncertainty present when studying a phenomena, and allow the uncertainty to be combined with what is known about the phenomena
- Models are exportable, comparable, and updatable. One researcher can use the model of another, competing models of the same phenomena can be compared, and under-performing components of a model can be updated upon discovering new information about the phenomena under study.

These advantages have been the driving force that has pushed the scientific community towards models as a method of communication of scientific research. To get a better of sense of what these models are like we can examine the model below that is being used to study how the yield of a particular crop is affected by changes in the amount of rain and absorption of the soil over a range of days.

In this model we see a set of input values, shaded to show that they are observed, and an output value. Upon observing the inputs through measurement, a value for the output can be recovered. We can also see in the model above that there is an arbitrary amount of wiring that can occur to transform a set of model inputs

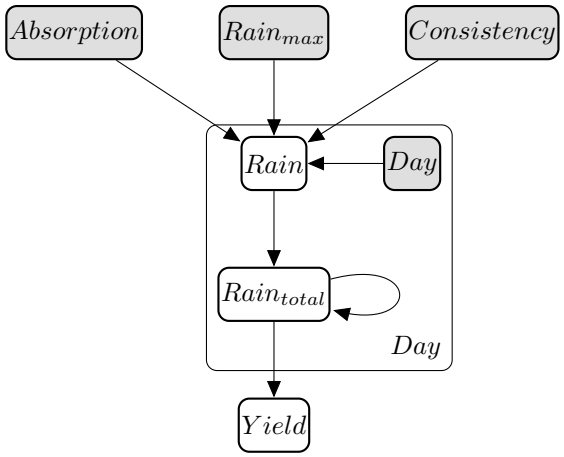

Figure 1.1: A model depicting the affects of rain on the yield of a crop over a span of days given observed values for absorption and consistency constants.

into the model output. Along with this arbitrary wiring, it is also worth noting that, given an output phenomena to study, the set of chosen observable inputs is also arbitrary. From these two facts we arrive at one of the modern challenges associated with modeling, models have become ubiquitous. In many fields there are so many competing models for the same phenomena that modelers have entered an age marked by *The Paradox of Choice*, where modelers must now spend a non-trivial amount of time deciding which model to use for their experiments. This task is formally known as the task of *Model Selection*.

Not only do modelers have to explore many competing models when deciding which to use to model a particular phenomena, this exploration is also expensive. In the information age, many of these models exist as source code with associated grounding documents. However, with the extreme prevalence of programming languages many competing models for the same phenomena are likely written in different programming languages. Asking modelers to learn a single new programming language is already a large drain on research time, but the prospect of needing to learn multiple new languages represents a barrier to entering the realm of model selection for most researchers. Given the enormous amount of items competing for the limited time of modelers the task of model selection is commonly side-lined.

This entails that in many fields a single or small set of models are used and hardly ever challenged. Not only that but this means that many researchers who create competing models will see their research go unused by the members of the field that the new model was intended to benefit.

1.1 Problem Scope

Creating a tool that largely automates the task of model selection is the main goal of this thesis; however, the task of model selection is not the only problem that this thesis seeks to address. The larger problem addressed by this thesis is the problem of creating a framework that allows the task of model selection to be performed autonomously.

Therefore the scope of the problem addressed by this thesis has four components:

1. The system must be able to extract models from scientific source code. This method must be generalizable to all programming languages, and it must produce models that are executable.

2. The system must be able to compare competing models of the same phenomena to determine how they are similar and how they differ.

3. The system must be able to analyze the extracted models to develop data that can be used to differentiate one model from the other.

4. The system must have a defined set of outputs that can, at the very least, be presented to the user that allow the user to easily choose which model to use between competing models of the same phenomena.

With all four of these components in place the system will have automated the tedious portions of model selection, and will allow modelers to avoid many of the hurdles outlined in the section above that prevent them from effectively conducting model selection. Such a system, given additional information about the qualities of a model that would make one model more desirable than another, should also then

be able to select a best-fit model for an experiment. Therefore the final product from the system should be a selected model from a set of extracted models, accompanied with the information gained by the system during the analysis phase in order to make the selection. A modeler would then be able to either utilize the model selected by the system or use the systems analysis output to inform their own decision.

1.2 Prior Work

For modelers in many disciplines the current state-of-the art for addressing the problem of model selection is not to use an intelligent tool or service that performs the selection, but to dig into the literature and find examples of comparisons that have already been done amongst competing models for a given phenomena. Unsurprisingly, this approach has many disadvantages. The obvious initial cost of this approach is that it requires a group of scientists to devote a substantial portion of time to conducting this analysis by hand. A second disadvantage of this method is that the analysis performed and documented by research groups in scientific publications are difficult to extend. For example the analysis conducted by Camagaro et al. contains information on the sensitivity of the compared models to certain inputs; however, modelers are left without any discussion or investigation into the pairwise affect of inputs on model sensitivity. In order for modelers to obtain this information there best option is to request access to the software used by the authors so that they can manually extend the software to extend the analysis. This is a less-than-ideal solution as not only will additional time be required to extend the analysis, but modelers must now rely upon the generosity of the authors to release the software they used to conduct their analysis, otherwise extending the analysis presented would require conducting all of the software design necessary to produce the initial analysis. One final disadvantage of this approach is that relying upon model analysis reports found in the literature places an artificial restriction upon the phenomena that modelers are likely to study and the models they are likely to use. This limitation entails that fields where less time and fewer research funds have

been spent on the task of model analysis will have a marginalized ability to conduct model selection. Automating the process of model extraction, analysis, and selection therefore not only increases the extensibility of an analysis but also increases the diversity of scientific research and discovery by placing disparate fields on a more equal footing.

Current tools such as the DAKOTA system allow modelers access to analysis tools that can be used to study and compare models. DAKOTA has been an excellent resource for modelers because of the extent of evaluation methods that are provided to modelers for model study. However DAKOTA has some disadvantages. The most notable disadvantage of the DAKOTA system is that modelers are still required to undergo the labor-intensive task of translating their models by hand into the pre-defined DAKOTA format. Not only does the labor and time present a barrier to entry for modelers wishing to use the DAKOTA system, but this system comes with an additional barrier for modelers who wish to use models from other researchers. While the modelers will likely be familiar with the structure of their own models, allowing for a simple translation task, they will likely require additional time and mental labor to become well-acquainted with other groups model specifications in order to create a faithful translation of the groups model to the required DAKOTA format. Not only is this an additional cost and barrier to entry for modelers, but it also introduces an unnecessary potential source of error. Another shortcoming of the DAKOTA system is that it does not provide any service to compare models. Modelers may be interested in what variables overlap between competing models, and may wish to augment the existing models to allow for analysis upon the overlapped portions of two competing models. DAKOTA does not offer any framework to handle this task. This means that while DAKOTA is a very useful tool for modelers to analyze existing models that they are well-acquainted enough to translate, it does not solve the problem of automating the task the model selection, as modelers are still required to do large amounts of setup, reconstruction, and investigation of analysis methods in order to gather useful information from DAKOTA that they can then use to select a model.

New methods have been created in recent years that are beginning to automate the process of extracted scientific models from free text. One such method is presented in the Eidos, INDRA, and Delphi software tool that seeks to extract models from textual documents, such as scientific publications and technical reports. This software has seen great success at the task of model extraction and assembly. Indeed many models are presented in the scientific literature and sometimes only in the scientific literature without being present in the form of source code in a software repository. However modelers will undoubtedly benefit from the reliability guarantees of models that are extracted from source code. Firstly, due to the nature of free text, identifying and extracting models from free text is inherently a probabilistic task. Thus models that are extracted from text are prone to include errors. On the other hand source code is explicit, and thus, given the proper framework, it should be possible to extract models from source code that are entirely faithful to the initial representation of the model in the source code. The explicit nature of source code also guarantees that the models extracted from source code will be fully specified, whereas it is possible for a model extracted from free text to be fully faithful to the model as described in the text and to still be underspecified. A simple example of this phenomena would be the lack of data type information for numerical values. This information is commonly absent from the text descriptions of a model, but is vital in many cases when creating an executable version of the model. However, extracting models from source code does share one of the challenges associated with extracting models from free text. This challenge is the challenge of grounding the variables present in the source code to the natural phenomena they represent. Thus the approach presented in this thesis can be viewed as an extension of the approach of the Eidos, INDRA, and Delphi system. In fact the system presented here utilizes the work of the Eidos, INDRA, and Delphi system in order to tackle the aforementioned task of variable grounding.

1.3 This Work

In this thesis I will present the Scientific Model Selection (SMS) pipeline; an automated system that seeks the select the appropriate scientific model for a given experiment from a selection of models that have been extracted from source code. To accomplish this task the system must be able to extract scientific models from source code, ground the real-world variables contained in the models using information gained from associated texts, and finally perform the model selection task using information gained from sensitivity analysis. Accomplishing this task will further unlock the potential of models to revolutionize the objective study of naturally occurring phenomena. The SMS pipeline is a component of the larger software system designed as part of the DARPA funded Automated Model Assembly from Text Equations and Software (AutoMATES) project.

As a component of the AutoMATES project, this pipeline focuses on part of the overall goal of model extraction, grounding, and analysis. The AutoMATES project includes three additional modules that all provide input to the SMS pipeline. These modules are the Program Analysis (PA) pipeline, the Text Reading (TR) module, and the Equation Reading (ER) module. The inputs provided by these modules will assist the SMS pipeline in extracting abstract representations of source code from the original source code languages, as well as grounding the variables found in that source code to real-world concepts. Any inputs to the SMS pipeline from these modules will be documented in the sections of the thesis where they are used.

The remainder of this thesis is organized to present the components of the pipeline that will solve the problem presented in the problem scope section. During the course of this thesis I will use two models from the DSSAT Cropping Systems model collection to demonstrate the capabilities of the SMS pipeline. The models used in this study will be targeting the natural phenomena of Potential Evapo-Transpiration (PET). The specific models I will be comparing are the Priestly-Taylor model of Potential Evapo-Transpiration (PETPT) and the ASCE model of Potential Evapo-Transpiration (PETASCE). Chapter II will introduce the algorithms used to

extract models from source code and transform them into a form that is both executable and comparable across competing models. Chapter III will introduce the analysis methods used by the SMS pipeline to derive information about the models under comparison. Chapter IV will document how the derived information from the analysis phase can be used to perform automated model selection, as well as how the information will be presented to users of the SMS pipeline to allow them to make their own final model selection decisions. This thesis will then conclude with Chapter V that will be a discussion of the results and implications of the pipeline and will introduce possible extensions for continuing this research.

CHAPTER 2

MODEL EXTRACTION FROM SOURCE CODE

In order to perform analysis on scientific models found in software we must first build a system that is able to detect source code that corresponds to scientific models, extract that necessary associated source code, and then represent the model in a form that allows analysis to take place. The technique developed must be generalizable to allow model extraction from source code of the various programming languages that are used to define scientific models. While large variations in syntax exist among the various programming languages used to encode scientific models, they can all be abstracted to a unifying abstract syntax tree (AST) representation. Constructing an AST representation of a program requires a large number of design decisions that are largely dependent on the intended use-case of the information contained in the AST. For the purposes of model selection we are keenly interested in identifying all potential models contained in the AST, as well as being able to easily inspect data-flow through the AST.

This chapter will begin with a discussion of the AST representation design decisions made to facilitate model extraction from source code. Following this discussion will be an explanation on the conversion process from the AST representation of a model to an executable computation graph. This chapter will conclude with an analysis of the computational cost of running the computation graph for a model and means of reducing this cost by leveraging vectorized computation and GPU computing resources.

At the time of writing this thesis the PA pipeline of the AutoMATES system is capable of handling source code from the Fortran programming language. The AutoMATES team plans to support additional widely used languages in the near future and for the purposes of model analysis, and the extent of this thesis, the subtle language differences that will be introduced from different programming languages

will be handled by the PA pipeline, before being generated into AST form.

2.1 Grounded Function Network Extraction

When translating a source code program, the SMS pipeline will receive a JSON specification and a set of associated lambda functions as input. The JSON specification will fully represent the control-flow of variables through the original source code. The lambda functions file will contain all of the individual computations necessary to execute a model that is faithful to the models found in the original source code. From these two files the SMS pipeline will construct a scientific model that will be executable and can be used for comparison and analysis.

This executable model is called a Grounded Function Network (GrFN). As the name suggests, this model will be a network. A network is a directed acyclic graph that contains a set of source nodes and a set of sink nodes. A node is considered a source node if the in-degree of the node, the number of directed edges arriving at the node, is zero. Similarly a node is considered a sink node if the out-degree of the node, the number of directed edges leaving the node, is zero. The name GrFN also implies that this network will be a network with functions. Indeed this network will include two types of nodes, variable nodes and function nodes. This network will be bipartite such that no function node has an edge incident to another function node, and no variable node has an edge incident to another variable node. Since our GrFN is supposed to represent a scientific model, the source variable nodes of the GrFN will be the inputs of the model represented by the GrFN, and the sink node, which will be a variable node, will be the model output. It is possible for the model to have some function nodes as sources to the network. These will be literal assignment functions and thus are not included in the set of scientific model inputs.

The last element left in the name has to deal with the naming of the variable nodes. GrFNs must be comparable upon their variable nodes, since the variables track the actual observed and calculated values. This is what connects the networks to the real-world and to one another. As defined in code, two variables that

represent the same phenomena can have different names, and two variables that represent different phenomena can have the same name. To circumvent this problem, the names of variable nodes in a GrFN will not come directly from the source code, but they will instead be shared names that are derived from grounding the variables found in source code with associated publications and other free text that can provide descriptive names for the variables contained in them. The grounding process ensures that variable node names are identical across various GrFNs, even if extracted from separate code repositories. This will allow for graph matching upon named nodes during downstream analysis.

2.2 Computation Graph Generation

2.2.1 Containers and Function Calls

At the top level the JSON has a list of containers. These containers represent different scope levels in the source program. A container can be a function or subroutine found in the source code or a loop. The branches of a conditional will not be considered containers due to the nature of how the GrFN will be wired to deal with conditional evaluation. a container will contain a body that will be in the form of an ordered list of statements. These statements will correspond to statements from the original source code that was present in the container. Computation graph construction will begin at a top-level container and will process through the statements in the container, constructing the computation graph as it goes. When a statement that references a new container, such as a function call, is reached the body for that statement will be processed and connected to the computation graph that is being generated. Once this task is completed, the rest of the body for the original container will be processed. This recursive process works well for traversing all containers and constructing a computation graph based upon the statements contained in each container as long as no container has a call in it's body to another container that also has a call to the original container. Unfortunately this is precisely the case with recursive functions. They act as a special case for this processing pipeline and

thus they are handled separately as will be discussed in subsequent sections.

Landing on a function call when parsing the body of a container can be thought of as an indicator to process the body of the child container referenced by the function call. The only intricacy here deals with the correct wiring of variable inputs into the containers plate, and the correct wiring of outputs from the container to the current position in the computation graph.

2.2.2 Assignment Statements

During the processing of a container, when an assignment statement is reached it will include the following items:

- A list of source variables
- The name of a lambda function
- A target variable node

Using these items the wiring for an assignment statement performs the following tasks to fully incorporate the information contained in the assignment statement into the computation graph:

- Create a new function node for the lambda function
- Create a new variable node for target variable
- Create new variable nodes as needed for the input variables
- Create a directed edge from each input variable node to the function node
- Create a directed edge from the function node to the target variable node.

Assignment statements can be handled by loading the assignment statement found in the source code into the function node so that values can propagate from the input variable nodes to the output variable node during computation graph execution.

2.2.3 Conditional Statements

Conditional statements are handled via a set of two lambda evaluations. The first evaluation is known as a `conditional` function node, that will actually evaluate the conditional property. The second is a `decision` function node that takes as input the evaluation from the `condition` function as well as the two possible assignments for an output variable. The `decision` function will be responsible for assigning the appropriate value to the output variable node based upon the conditional input.

Both the `decision` function node and the `conditional` function nodes output variable node are artifacts that did not exist in the original source program that have been added to the computation graph. Thus these will not be displayed when rendering the function node or variable node views of the computation graph.

2.2.4 Indexed and Open-ended Loops

Indexed loops require a loop plate and have a specific index variable as well as a number of iterations through the loop. They can easily be handled like containers as mentioned above, but require additional storage to handle information about the number of executions needed to satisfy the plate during compuation.

Open ended loops are hard. Here we can have conditional exit cases defined at a start or end point of a loop, which presents a much larger challenge than loops with an index and pre-defined amount of iterations.

An extra challenge is added when dealing with open-ended loops that can include multiple exit points (introduced either by `break` statements or `goto`s) as well conditional skip points where parts of the loop are skipped on an iteration (introduced either by `continue` statements or `goto`s).

The usage of the `goto` statement has been hotly debated by computer scientists for nearly half a century. In most modern programming languages the usage of `goto` or other such statements that allow for unstructured branching is prohibited. However, the AutoMATES system seeks to handle source code inputs from languages that do allow for unstructured branching, and thus this paradigm must be handled

during the wiring phase of a GrFN computation graph.

Recursion is a commonly used software practice that must be handled for our computation graphs. Most importantly, recursion must be identified and recursive edges that would create loops in the computation graph must be pruned.

Possibly the most difficult challenge for our graph wiring is the identification and handling of indirect recursion.

2.3 Data Type Assignment

So far I have discussed the wiring needed to create the computation graph that will allow a GrFN to be executed, and I have shown the methods necessary to make the GrFN executable. However, one more crucial component for creating a GrFN that we can perform inference upon is a discussion of how we will handle the actual data being processed from the GrFN. At the time of writing this thesis only basic data types are allowable in a GrFN. This includes numerical, string, and boolean values. These primitive data types are all singular values that represent a single phenomena, thus they match perfectly with the definition of variable nodes in the GrFN CG. The specification for each variable in a GrFN CG contains a type annotation that declares the data type of the variable. These values are derived from the JSON specification during the wiring stage of the GrFN. At execution time, these type annotations are used to validate a set of inputs and ensure proper storage format of computed variable values.

The infrastructure to represent complex data types such as Arrays, user-defined types, and unions in the AST form is still being developed by the program analysis team, and thus they will not be included in this thesis. The main challenge with representing these data types is that they are not singular variables, but are instead collections of variables. To properly perform inference over a GrFN CG all variable nodes must be singular variables, thus we cannot allow any of the above collections of variables to be represented by a variable node. This presents a problem of representation that will be studied and resolved in future work that extends this

thesis.

2.4 GrFN Computation Graph Execution

Once the wiring stage for a GrFN has been completed, the GrFN CG must be made executable. The execution of a GrFN CG requires the execution of all functions stored at the function nodes of the computation graph. Therefore the problem of executing a GrFN CG can be simplified to the problem of determining an ordering of execution of the function nodes that allows for all of the function nodes to be executed without any failures. Of course a GrFN CG also needs to save the state of variables during execution in a way that allows the function nodes to access the information stored for each variable they require. In this section I will present the methods developed for the GrFN CG to handle the function evaluation ordering problem and the variable storage access problem.

2.4.1 Call Stack Creation

Creating a computation graph from a GrFN specification allows us to formally represent an extracted scientific model as a graph data structure. However, if we wish to analyze the extracted model, then we will need the ability to compute information over this data structure. To accomplish this we introduce the idea of execution over a computation graph. The computation graph contains a set of function nodes. Computing the lambda function stored at each function node is analogous to executing the computation graph from the set of inputs to the output. However, the function nodes rely upon having values populated at each of their input variable nodes in order to perform their computation. Therefore the task of executing a GrFN CG can be rephrased as determining how to order and execute the functions nodes contained in the computation graph.

A Naïve first-pass solution to accomplish this goal would be to use a graph traversal from the output to the inputs where at each function node, the node will determine whether values for each input variable node have been populated. For any

input variable nodes that have not been populated, the function node will call the parent function node responsible for computing the value of the input variable node. Once all such calls have returned, the function itself will evaluate. This recursive calling procedure is very similar to message-passing, a method for inference on factor graphs. While this will ensure correct model execution, this method of handling execution is not as efficient as possible. To start the recursive call structure adds additional function setup and calls to the execution, on the order of the number of functions included in the computation graph. The second obvious inefficiency with this execution is that the recursive operations must be done for each execution, and they require all function nodes to be executed sequentially.

These two concerns can be addressed by creating a call stack comprised of all the function nodes contained in the GrFN CG. The order of computation between function nodes implies that the call stack is equivalent to a partially ordered set (poset) upon the function nodes. Once this poset is recovered, execution of the GrFN CG can occur by executing all functions at the first level in the poset, then moving to the next level in the poset, and then repeating the sequence until reaching the end of the call stack. This allows all function nodes at the same level in the poset to be executed concurrently, and the computation required to create this call stack only needs to occur once.

Computing the call stack requires a series of separate computations. First a control flow graph (CFG) must be extracted from the GrFN CG. As defined above, the GrFN CG is bipartite with respect to the variable and function nodes, such that no variable node is adjacent to another variable node and vice-versa for a function node. Therefore a GrFN CFG can be extracted from the GrFN CG by simply squashing any variable node between a pair of function nodes into a singular edge connecting the two function nodes. Since the GrFN CFG is derived from the GrFN CG it will have a set of input nodes. Starting from this set of input nodes an index is assigned to each node, beginning at zero. Each time an edge is traversed the index increases. If a function node is reached and it already has an index the index is updated to the maximum of the current index and the newly calculated

index. Once the full graph traversal is complete we have an index for the poset. Function node sets are created according to the index values, and the index values also provide the ordering of the function node sets. Smaller index values correspond to the function node sets that are to be computed first. Function sets can then be placed into the call stack by pushing the sets with the largest index value onto the stack first. The GrFN CG now has access to a call stack that can be used to execute an input set. During execution, when the call stack is being used, it can be maintained by pushing popped values from the stack onto a different empty stack. After execution, the values can be popped from the temporary stack and pushed back on to the original call stack. This ensures that the call stack order will be maintained for the next execution.

2.4.2 Singular Input Execution

During execution, a GrFN utilizes a value storage tag at each variable node. During computation, function nodes pull their input data from the value tag of each parent variable node of the function node. The output from the function node is stored in the value tag of the variable node that is the child of the function node.

2.4.3 Vectorized Input Execution

While a GrFN CG is perfectly capable of executing one set of inputs at a time, the CG can also handle executing over multiple sets of inputs at once. The SMS pipeline accomplishes this by making use of the PyTorch tensor computation framework. It is advantageous to compute multiple inputs at once using vectorized computations because pooling like computations lowers overall compute time.

CHAPTER 3

ANALYSIS OF MODEL UNCERTAINTY

The greatest benefit researchers receive from modeling is being able to reason about the uncertainty involved in observing a phenomena of choice. From the modeling perspective, explicit statements about the uncertainty of a phenomena can be made by adding inputs to the model of the phenomena that represent a source of variance upon the phenomena.

3.1 Global Sensitivity Analysis

A powerful tool used by modelers to quantify the uncertainty present in models is global sensitivity analysis. Broadly speaking, global sensitivity analysis is the study of how variance in the inputs of a model affect the variance, or uncertainty in the output of the model. Since the goal of the SMS pipeline is to provide modelers with a tool select a model that lowers the amount of uncertainty in estimation of a given phenomena we have chosen to use sensitivity analysis in order to evaluate the extracted models.

For this thesis we employ variance-based methods of sensitivity analysis. Variance-based methods of sensitivity analysis focus on the computation of sensitivity indices. Sensitivity indices are numerical values assigned to each model input, or set of inputs, that denote how sensitive the output of the model is to that input, or set of inputs. The first order sensitivity index, commonly denoted by $S1$, notates the sensitivity of the model output from each individual input. Similarly the second order sensitivity index, commonly denoted $S2$, notates the sensitivity of the model output from each pair of model inputs. This pattern continues for any given model for all higher order sensitivity indices.

The final sensitivity index is the total sensitivity index, commonly denoted as

ST. This index has an entry that corresponds to each model input. Each entry tracks the total amount of sensitivity on the model output contributed by the input as a portion of the first, second, and all higher order sensitivity indices.

While other methods of sensitivity analysis exist, such as Variogram Analysis of Response Surfaces (VARS), the variance-based methods fit well within the scope of our problem domain as we are conducting sensitivity analysis over probabilisitic models. While the VARS method does claim a significant faster computation time, it does not reveal as information about the affects of model inputs upon the model output. This is due to the lack of pairwise-information that the variance-based methods are able to capture.

In the following subsections I will discuss three different variance-based methods of sensitivity analysis employed by the SMS pipeline.

3.1.1 Sobol's Method for Index Calculation

Sobol analysis is the most widely used version of variance-based sensitivity analysis. The Sobol analysis method gets its name from the sampling method used during analysis. Samples for Sobol analysis are drawn from the Sobol sequence. Defining the precise nature of the Sobol sequence is beyond the scope of this masters thesis; however, we can observe how the Sobol sequence aids sensitivity analysis by observing the differences between draws from the Sobol sequence and draws from a random sample. A visualization of the difference is shown below in figure 3.1.1.

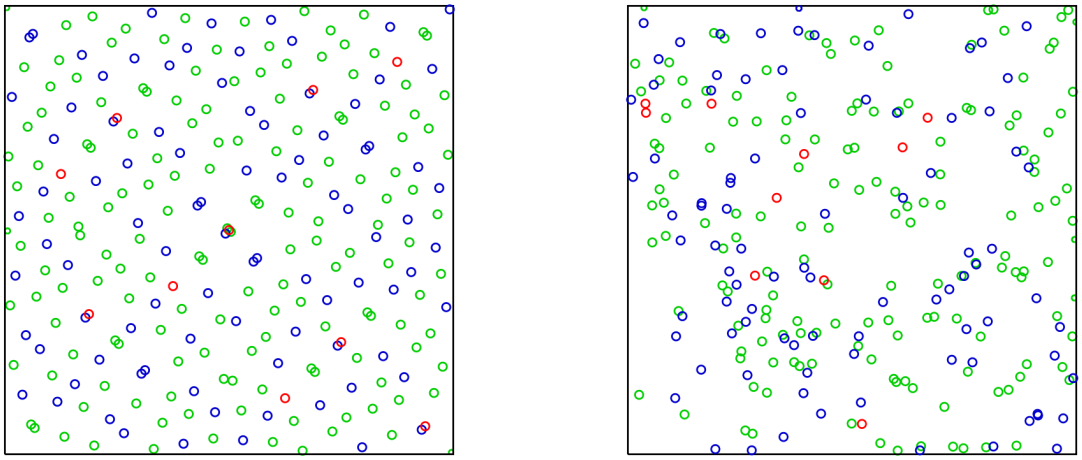

Figure 3.1: A visualization of 1000 draws from the Sobol Sequence (left) paired with 1000 randomly drawn points (right) in two dimensions. For both images the first 10 draws are colored red, the remaining first 100 draws are colored blue, and the remaining 1000 draws are colored green. Images created by James Heald, distributed under a CC BY-SA 3.0 license.

Notice that the red, blue, and green draws from the Sobol sequence are much more evenly dispersed across the sample space than the same draws from a uniform random sample. This allows the Sobol sampling method to achieve much better coverage of the search space, while still maintaining some variability in sample locations. The SMS pipeline employs a variant of the Sobol sequence introduced by Saltelli et al. that is designed to minimize the error when utilizing the samples to calculate the sensitivity indices of a function.

The sample matrices that must be computed for analysis are defined in algorithm 1. We can see that this sampling algorithm draws N samples of d dimension from the Sobol sequence and then creates $N(d+2)$ samples total to be evaluated by the model. The matrices returned from the sampling step can then be evaluated using the model computation graph. These evaluations will be represented as $f(A)$, $f(B)$, $f(\mathbf{A}_B)$, and $f(\mathbf{B}_A)$. Once the evaluation step is completed, the evaluations will be analyzed according to the Sobol analyzer. The Sobol analyzer computes three different sensitivity indices using approximation equations.

Algorithm 1 Sobol Sampling Algorithm

define $d \leftarrow$ amount of input variables

define $N \leftarrow$ amount of desired samples

define $R \leftarrow$ set of sample space ranges for all d input variables

1: $A \leftarrow N$ samples from R using the Saltelli corrected Sobol sequence

2: $B \leftarrow$ shuffled rows from A

3: $\mathbf{A}_B \leftarrow \forall i = 1 \ldots d \; A_B^i = \forall j < i \; A[j] \oplus B[i] \oplus \forall j > i \; A[j]$

4: $\mathbf{B}_A \leftarrow \forall i = 1 \ldots d \; B_A^i = \forall j < i \; B[j] \oplus A[i] \oplus \forall j > i \; B[j]$

5: **return** $A, B, \mathbf{A}_B, \mathbf{B}_A$

The i^{th} element of the first order sensitivity index, S_i, is computed using equation 3.1. Notice that this function will approximate the variance $w.r.t$ the i^{th} input of the expectation $w.r.t$ all other inputs.

$$\begin{aligned} S_i &= \frac{\mathbb{V}_{X_i}\left(\mathbb{E}_{X_{\sim i}}(Y|X_i)\right)}{\mathbb{V}(Y)} \\ &\approx \frac{1}{N * \mathbb{V}(Y)} \sum_{j=1}^{N} f(B)_j \left(f(A_B^i)_j - f(A)_j\right) \end{aligned} \tag{3.1}$$

Once all of the first order sensitivity indices have been computed, they can be used to calculate the second order sensitivity indices. This calculation is done using equation 3.2 as defined below. A second order sensitivity index, S_{ij}, can be described as the variance of the ij input pair without the variance of the separate i^{th} and j^{th} components.

$$\begin{aligned} S_{ij} &= \frac{\mathbb{V}_{ij}\left(\mathbb{E}_{X_{\sim ij}}(Y|X_iX_j)\right) - (S_i + S_j)}{\mathbb{V}(Y)} \\ &\approx \frac{\sum_{k=1}^{N}\left(f(B_A^i)_j * f(A_B^k)_j\right) - (f(A)_j * f(B)_j)}{N * \mathbb{V}(Y)} - (S_i + S_j) \end{aligned} \tag{3.2}$$

The total sensitivity indices, S^T, can also be calculated from the sampled matrices. The i^{th} total sensitivity index can be computed using equation 3.3 as defined

below. By inspecting the result from this equation we can see that the i^{th} total sensitivity index is an approximation of the expected variance caused by the i^{th} input upon all other inputs j such that $j \neq i$.

$$\begin{aligned} S_i^T &= \frac{\mathbb{E}_{\sim i}\left(\mathbb{V}_{X_i}(Y|X_{\sim i})\right)}{\mathbb{V}(Y)} \\ &\approx \frac{1}{2N * \mathbb{V}(Y)} \sum_{j=1}^{N} \left(f(A)_j - f(A_B^i)_j\right)^2 \end{aligned} \tag{3.3}$$

3.1.2 FAST S1 Index Calculation

Some text.

3.1.3 RBD-FAST S1 Index Calculation

The purpose of utilizing the RBD-FAST method of sensitivity analysis is to further lower the runtime of analysis as compared to the standard FAST method. It has been claimed that RBD-FAST can lower the runtime of sensitivity analysis for models with a large number of input parameters.

3.2 Model Output Surface

A Model Output Surface (MOS) plot is a 3D plot where the z-axis is the output variable of a model and the remaining axes are input variables. A MOS plot consists of the plotted outputs over a predefined range for each of the two inputs which creates a smooth surface object in three dimensions. Evaluation to create this surface is done using a mesh grid of finite points. extrapolation between point evaluations can then be conducted to create the smooth surface visualization. As expected, a higher number of input samples over the same input space range will lead to a smoother surface that is a more accurate representation of the true output surface for the input pair. In this section I will cover the process of selecting which input variables to study with MOS plots, how to set values for the additional inputs not

under study, and how to efficiently perform the evaluations necessary to generate a MOS plot.

3.2.1 Input Variable Selection

Most models will have more than two input variables and thus when creating a MOS plot a decision must be made about which two input variables should be varied during output surface creation. Generally speaking, for a model with N input variables, a user could generate $\frac{N(N+1)}{2}$ plots to expose all possible pairs of variable interactions and examine how those interactions affect the output surface. For most sizes of N this study is likely to not be computationally feasible, and modelers are unlikely to desire to view such a large number of output plots when performing model selection. A better use-case for MOS plots would be to showcase the pairs of variables that have the greatest combined affect on the model output.

In order to accomplish this task we need a way to rank the pairs of inputs in order of how greatly they affect model output. Fortunately we can attain this ranking directly from the second order sensitivity index, $S2$. Utilizing the $S2$ index as our ranking schema we can either create MOS plots for the top k variable pairs, or we can use a threshold cutoff and create MOS plots for all pairs on input variables that exceed this cutoff. An appropriate cutoff could be a pre-defined value, or it could be based upon a difference between neighboring values of the sorted $S2$ indices. An example of such a cutoff would be to generate MOS plots for all input pairs, until seeing a drop in $S2$ index score of at least an order of magnitude.

It is important to note that utilizing the $S2$ index is not the only way to select which input pairs to generate a MOS plot for, and is likely not the only metric of interest to modelers. If we have additional information, such as which inputs can be most or least accurately measured by the modelers then we may want to generate MOS plots for the modelers based upon this criterion so they can observe how output varies for input variables that they can measure well, vs those that they can measure poorly. This will become even more important for the task of model selection when competing models have differing sets of inputs, some of which may

be easier or harder to measure than others.

3.2.2 Input parameter estimation

Once the input variables for a given MOS plot are selected, the remaining inputs must have values assigned to them to allow for the model to be evaluated for MOS plot generation. To avoid confusion, I will refer to this set of input variables that will no longer be varied as the input parameters for the model.

There are many options present for assigning values to each of the input parameters. Unfortunately the nature of models also demands that either one or few values be chosen for each of the input parameters due to the high number of input parameters that are likely to be present. Even if we wished to allow for only two values for each input parameter, and wanted to study all possible combinations of them then for a set of N input parameters we would generate 2^N MOS plots. Expecting modelers to review this many MOS plots goes against the idea of summarizing model behavior. This entails that a single set of values for the input parameters should be selected.

The task of input parameter estimation is now the task of finding the best-fit set of input parameters for a MOS plot. For each parameter we know that we have a range of values that were used for the parameter during analysis. From this information we can define that the best-fit for an input parameter is the Maximum Likelihood Estimate (MLE) over the range. The MLE is a useful estimate for each input parameter since the ranges are included for each parameter. However, if observational data were present, then it would be possible to use the Maximum A Priori (MAP) estimate as the setting for our input parameters.

3.2.3 Surface Generation and Evaluation

Some text.

CHAPTER 4

METHODS FOR INFORMED MODEL SELECTION

Some intro about the task of selecting a model given the information from model analysis

4.1 Model Overlap Identification

While analytical methods can provide a large amount of useful information to modelers about a single model, the real benefit of the AutoMATES system comes from the ability to automate comparisons among competing models. Competing models can be identified from the output variables of their computation graphs. After identifying a selection of competing models the comparison phase can begin.

For any two competing models of the same phenomena the comparison phase consists of the following:

1.) Identify the overlap between the variables in each models computation graphs. This corresponds to overlap in observable real-world phenomena.

2.) Extract the sub computation graphs for each model based on the variable node overlap.

3.) Perform analysis on the overlapping computation graphs and compare the results with the analysis results from the models whole computation graphs.

In order to accomplish the tasks outlined above, I will introduce a new construct, known as a Forward Influence Blanket (FIB). A FIB is a specific instance of a Markov blanket, derived from a GrFN computation graph, that can be used for forward analysis. After the completion of these tasks the information gained from the comparative study of these models can be added to the final model report, or used for automatic model selection. In this chapter I discuss model comparison in terms of two models compared directly with one another. At the end of the

chapter I will elucidate on the necessary steps to generalize this form of binary model comparison to a set of N models.

4.1.1 Forward Influence Blanket Creation

Imagine the structure of two computational models of the same phenomena in the most general sense. We can say with certainty that both models will have the same output variable, namely the variable that represents the phenomena of interest. From this there are three options for how the set of input variables between the two computational graphs can overlap. The least interesting option is that the two models could share no inputs variables. This would mean that the computations involved in each model are wholly independent and could be combined if necessary in a trivial manner, at least at the input level. The more interesting option is that a subset of the inputs are shared between the two models. This entails that the models will make use of the same data, though the computations used to transform that data into a model output will almost certainly differ. It is possible that the set of input variables will overlap exactly between the two models; however, it is much more likely that there will be some input variables that are not contained in both. In the following subsections I will discuss how to build a computation graph that represents the computation present in a GrFN that corresponds to utilization of shared variable nodes with another model.

Some text.

The key aspect of a FIB that distinguishes it from a GrFN computation graph is that the portions of the original computation graph that are not shared between the two models under comparison are pruned. In order to ensure that the resulting models are still executable, variable nodes representing new inputs to the FIB computation graph must be retained. We have identified this set of variables as the cover variable set.

Identifying a variable as being a member of the cover set stems from the initial shared graph structure extracted from the original GrFN computation graphs. From the

4.1.2 Forward Influence Blanket Execution

In order to execute a FIB the user must provide values for the input variable nodes and values for the cover variable nodes. At execution time, both sets of variable nodes will be populated before beginning to compute the function nodes in the partial order of functions provided by the GrFN computation graph. This is the only difference between computing on a FIB computation graph and computing on a GrFN computation graph.

Execution of a FIB computation graph can be done either with singular preset values for all of the cover variables, or with ranges for each cover variable. FIB computation graph supports Torch-aided vectorized computation similar to the GrFN computation graph structure, and no additional memory constraints are imposed on execution by the FIB class.

4.2 Model report generation

Some text

4.2.1 Comparative Sensitivity Index Assessment

Some text.

4.2.2 Cross-model Sensitivity Surface Examination

Some text.

4.3 Input space partitioning

Some text

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

Some intro.

5.1 Conclusions

Some text.

5.2 Future Work

As discussed above, AutoMATES project has created an excellent framework for the extraction and comparison of scientific models found in source code. While many methods for analysis already exist in the AutoMATES system, as well as information necessary to facilitate automated model selection, there are still plenty of directions for future work. Many of the opportunities for extending the AutoMATES system build upon one another, and all are focused on expanding the scope of modeler questions that AutoMATES is able to handle without additional input from the modeler. Below I catalog some of the immediately visible extensions to the AutoMATES program improve the power of the AutoMATES model selection capabilities.

5.2.1 Alternative Sampling Methods for Analysis

Some text.

5.3 Variable Domain and Range Detection

Some text.

5.3.1 Model Selection via Uncertainty Analysis

The current analysis methods employed by AutoMATES allow for automated model selection based upon behavior of model inputs or sets of model inputs. While this advanced capability is likely desired by modelers in many situations, it only allows for indirect comparison between models. A method for direct comparison such as error propagation that includes an estimate of metrics such as variance in model output allows for stronger comparison statements that will likely be more acceptable metrics for automated model selection.

5.3.2 Iterative Model Improvement

After gathering enough information about various competing models as well as error information of model improvements,AutoMATES should be able to begin learning how to update models to lower uncertainty in model outputs.

A key component to this enhancement would be to identify similar function nodes or series of function nodes in a computation graph that correspond to the same overall computation. This, along with the grounding of variable nodes, which has been assumed, will enable modular computation components for variables to be added, removed, or mutated in order to improve model accuracy, efficiency, or other metrics of choice to modelers.

5.3.3 Input Space Division

Modelers would likely benefit from the AutoMATES being able to answer more general questions about what models to use to study a certain phenomena. For instance, modelers may not be able to provide bound information for the variables of interest to them when gathering data to study a particular phenomena. AutoMATES could assist modelers in this regard by discovering the furthest possible extent of all possible variables for each competing model of a phenomena and then partition the input variable space based upon peak model performance, such that each separate partition has an identified ideal model for studying the phenomena

given inputs contained in that range. As previously stated the partition criterion would be some aspect of model fitness.

5.3.4 Data Space Discovery

An extension of the idea of automatically discovering the input bounds of a set of input variables for a model is the idea of discovering the total possible data space for a phenomena of interest. Modelers would benefit from future versions of AutoMATES being able to identify potential additional variables for a given phenomena, other than just those identified by the modeler. A potential example would be the identification of a combination of variables that can be used to model a given model input with higher precision.

APPENDIX A

Sample Appendix

Stuff.....

REFERENCES