Month 5 Report: Final Phase 1 Report

The AutoMATES Team

2019-04-01

Contents

Note: This PDF has been automatically generated from a web version, available here: `https://ml4ai.github.io/automates/documentation/deliverable_reports/m5_final_phase1_report` *Please visit the web version for the best experience.*

Link to the PDF version of this report.

1 Overview

This report describes the Phase 1 AutoMATES Prototype release. The overall architecture of the Prototype remains the same as described in the Month 1 Report Architecture Overview. Since that report, significant progress has been made in all six core components, and now significant portions of the pipeline can run end-to-end, as demonstrated in the AutoMATES CodeExplorer (described

in the next section). The remaining components are executable in stand-alone modules and will soon to be integrated.

Each section in this report generally includes (where appropriate) the following information:

- Architecture: description of component software architecture and functionality details.
- Instructions: ...for running the component(s).
- Data set(s): when applicable, a description of any data sets used and how to obtain them.
- Updates: summary of changes over the past two months, since the previous Month 3 Report.

All code comprising the AutoMATES Prototype is open-source and available at the AutoMATES and Delphi Github repositories.

To clone the tagged version of automates that corresponds to the Month 5 Phase 1 Final Prototype release, run the following commands from a *nix command-line (with git installed):

```
git clone https://github.com/ml4ai/automates
git fetch && git fetch --tags
git checkout v0.1.0
```

2 CodeExplorer

The UA team has significantly extended the functionality of the original AutoMATES demo webapp to create the AutoMATES CodeExplorer, the main HMI for the AutoMATES Prototype. The CodeExplorer demonstrates a processing pipeline that includes:

- Performing Program Analysis on input Fortran source code to produce an intermediate Grounded Function Network (GrFN) representation.
- Inspection of the GrFN representation.
- Summary of the GrFN as a Causal Analysis Graph.
- Inspection of the functionally equivalent Python representation generated by Program Analysis.

- An example of model Sensitivity Analysis (currently only for the Priestley-Taylor model, as described below).
- View of the results of Model Comparison.

This section describes how to access a remotely deployed instance of the webapp, as well as run it locally, followed by a description of its features.

2.1 Instructions for running

The webapp has currently been tested using the Safari and Chrome browsers. While it may run in other browsers, behavior is not guaranteed (it is known that some functionality breaks in Firefox). We note below an additional known issue with the web interface.

Accessing the webapp online

The easiest way to try out CodeExplorer, without having to install it and run locally, is by visiting the version deployed at `http://vanga.sista.arizona.edu/automates/`. This has all of the same functionality in the current code release.

Running the webapp locally

To run the webapp locally instead, you will need to install Delphi. For *nix systems, assuming you have the prerequisites (Python 3.6+, pip, and graphviz), you can install Delphi with the following commands:

```
pip install https://github.com/ml4ai/delphi/archive/master.zip
```

If you want to get the tagged version corresponding to this milestone, you can run the following command instead:

```
pip install git+https://github.com/ml4ai/delphi.git@4.0.1
```

(Note: to use the AnalysisGraph class to create and parameterize models from text reading, you'll need to download the Delphi SQLite database and set the `DELPHI_DB` environment variable as described here.) This will also install a

command line hook to launch the CodeExplorer app, so you can just execute the following from the command-line

```
codex
```

and navigate to `http://127.0.0.1:5000/` in your browser to view the app. The CodeExplorer webapp is built using Flask. All code and data used for the CodeExplorer can be found in the Delphi Github repository

2.2 CodeExplorer Features

The CodeExplorer has two main interfaces:

- The 'Home' page is for 'live' **Code Analysis**.
- The **Model Comparison** page provides a more 'curated' experience demonstrating the comparison of models extracted from two code sources.

Links to either interface are located adjacent to the CodeExplorer title bar (as shown in Figure 1).

Figure 1 shows a screenshot of the Code Analysis ('Home') interface. As with the Month 3 demo, Fortran source code is input in the pane to the left and submitted for analysis. The CodeExplorer now provides three source files through the `Choose Model` button:

- Crop Yield: A toy program that demonstrates basic Fortran code idioms (main, subroutine, loops, conditionals)
- Priestley-Taylor model of Evapotranspiration (DSSAT): This is the unaltered Priestley-Taylor model as found in the DSSAT Github repository.
- ASCE model of Evapotranspiration (DSSAT): This contains the complete functionality of the ASCE model, but has been altered by combining content from several files into one in order to make a single-file program.

Once one of these models is selected (or other Fortran is input), the user can click the `Submit` button to send the code for Program Analysis. The Program Analysis module is still under active development, and does not cover all Fortran features yet, so we encourage the user to try small modifications to one of the three example models provided, rather than trying arbitrary Fortran code.

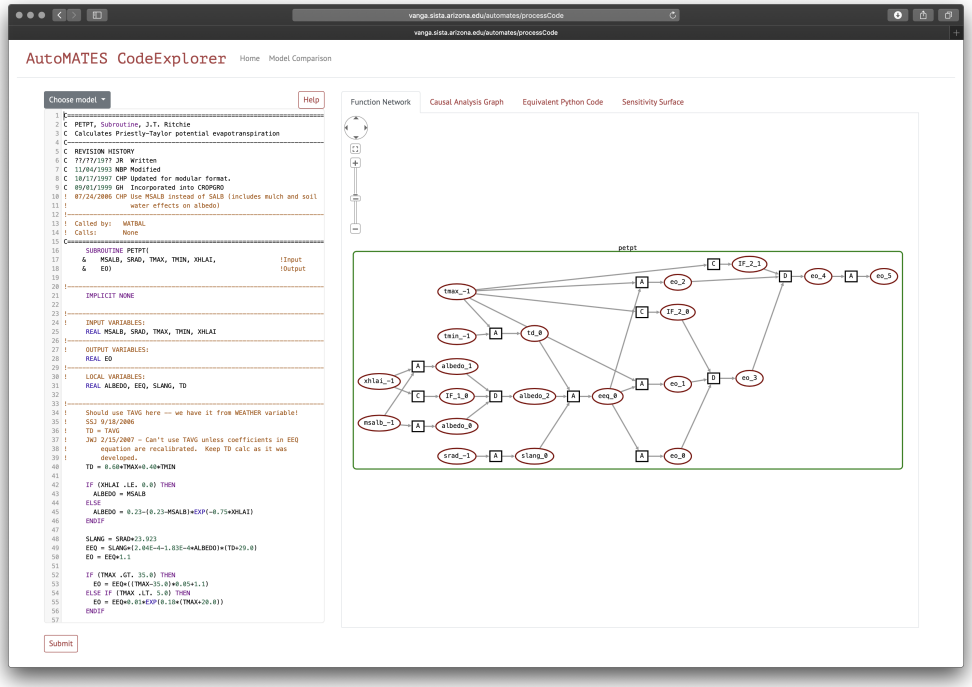

Figure 1: Screenshot of AutoMATES CodeExplorer interface

Function Network

Once the source code has been analyzed, a graphical representation of the Grounded Function Network is displayed in the Function Network pane to the right of the editor (as in Figure 1). Oval-shaped nodes in the graph represent Variables, while the small squares represent Functions. Four kinds of functions are labeled:

- `L`: Literal value assignment (a variable is assigned a literal value; no inputs to these functions)
- `A`: Assignment function (assigns a variable value as a function of one or more other variables)
- `C`: Condition function (assigns a conditional variable based on the condition of a conditional statement)
- `D`: Decision function (assigns a variable to the value based on the outcome

of a conditional)

Two types of colored **scope boxes** may appear around portions of the graph:

- Green boxes represent "code containers", generally corresponding to source code program scope.
- Blue boxes represent "loop plates", indicating the program scope that is iterated according to a loop. Everything within a loop plate (including other scope boxes) may be "iterated" in execution of the loop.

To see an example of both types of scope boxes, submit the `Crop Yield` Fortran source code. Both of the other code examples only involve a single code container scope box. When the mouse hovers over a box, the upper-left will show a "-" sign that can be clicked to collapse the box; mousing over a collapsed box will reveal a "+" that can be clicked to expand the box.

All of the nodes in the graph can be selected (by clicking) to toggle a view of the metadata associated with the node, as described in the following two subsections. (NOTE: once you select a node, it will stay selected, so you can view the metadata for several nodes at a time. However, if you change the view pane or collapse a scope box, the metadata views will remain open. This is a known issue that will be addressed in the future.)

Text Reading-derived variable metadata Clicking on the variable nodes (maroon ovals) brings up automatically-associated metadata and provenance for the variables extracted by Text Reading from (1) code comments and (2) scientific texts. Figure 2 shows examples of metadata from each of these sources.

Function lambda code and equations Clicking on a function node brings up the automatically generated Python lambda function and the equation it represents. Figure 3 shows examples of the equation and lambda function from two functions.

Causal Analysis Graph

Clicking on the "Causal Analysis Graph" tab shows a simplified view of the model, in which the function nodes are elided, and the edges between the variable

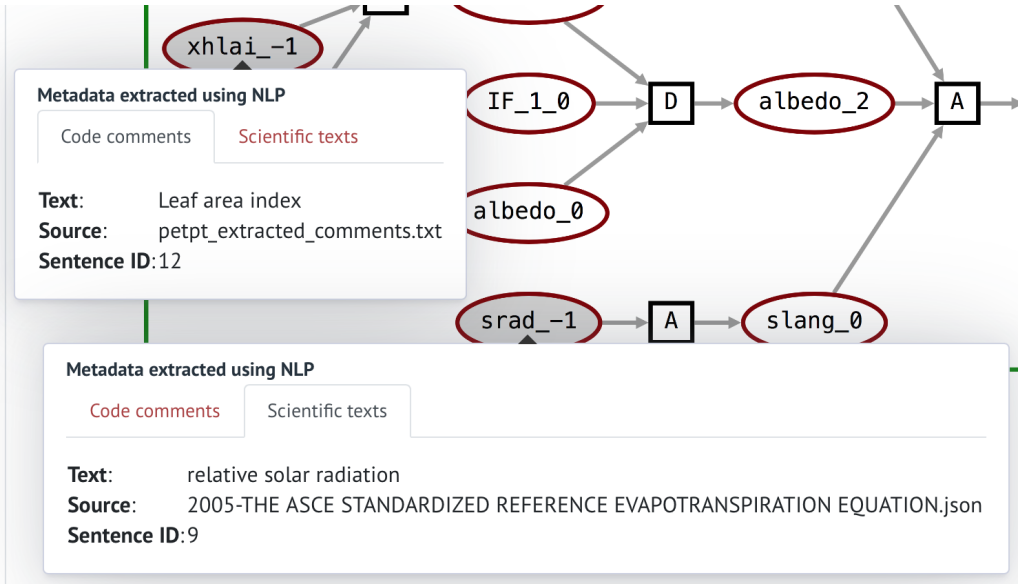

Figure 2: Variable metadata from code comments and scientific text.

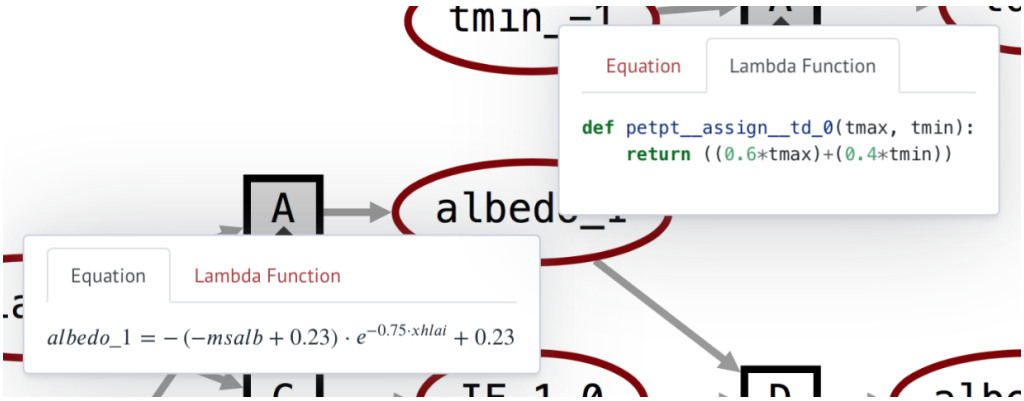

Figure 3: Function equation representation and lambda function.

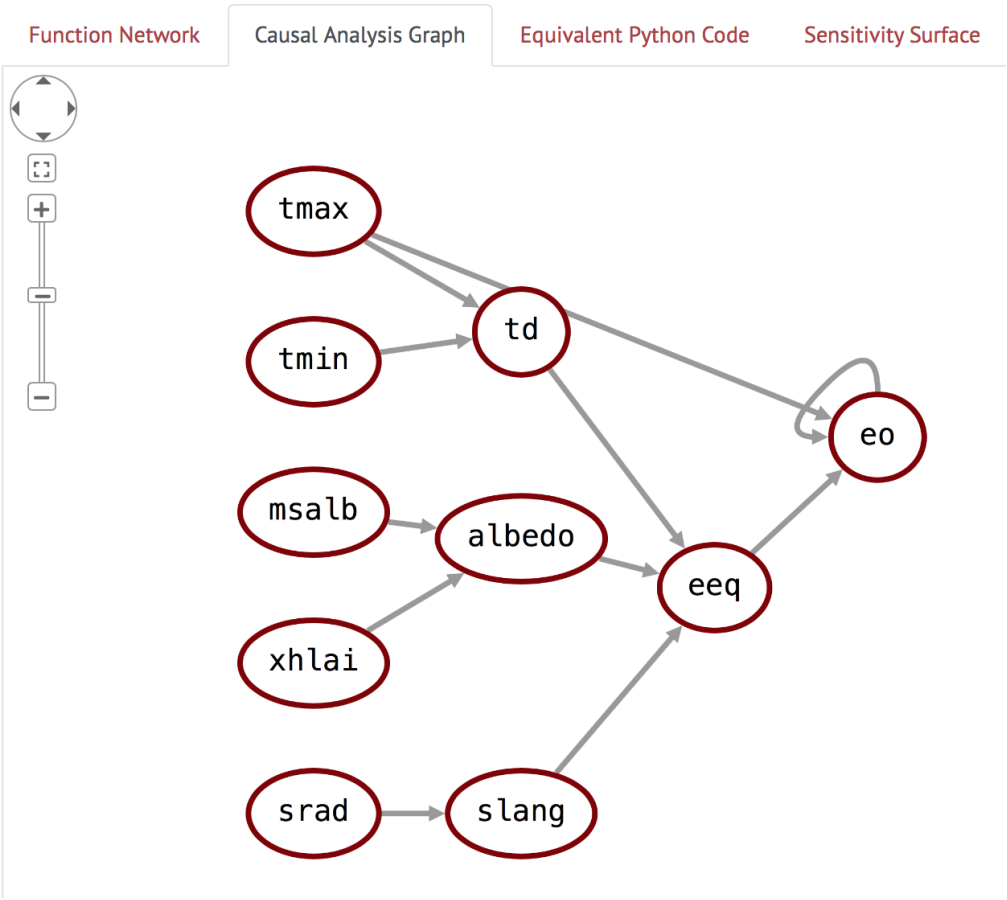

Figure 4: Causal Analysis Graph view

nodes denote causal influence relations. Clicking on the nodes brings up variable descriptions, similarly to the Function Network graph view.

(NOTE: although you can select Variable nodes, these do not currently populate their metadata; This is only currently possible in the Function Network graph pane.)

Equivalent Python Code

Clicking on the "Equivalent Python Code" tab displays the python code that was generated by Program Analysis and has the same functionality as the input For-

tran code. This is helpful for debugging the Program Analysis pipeline.

Sensitivity Surface

The "Sensitivity Surface" tab shows a surface plot of the output variable of the model with respect to the two input nodes that it is most sensitive to, as measured by the S_2 (second order) Sobol sensitivity index. The creation of this surface requires knowing the bounds of the variables - right now, we use a hard-coded dictionary of preset bounds for variables in the Priestley-Taylor model, but in the future we hope to automatically extract domain information for the variables using machine reading.

Currently, this tab only works for the Priestley-Taylor model. For the other two source code examples, the notice shown in Figure 6 is displayed below the `Submit` button.

2.3 Model Comparison

The **Model Comparison** page provides a more 'curated' experience demonstrating the outcome of comparing two models extracted from two code sources, in this case the source code for the Priestley-Taylor and ASCE models from the main page. As shown in Figure 7, the Causal Analysis Graphs for the Priestley-Taylor and ASCE models are displayed in the left and right columns, respectively.

In the middle column is a representation of the subset of the ASCE model is contained in the 'Forward Influence Blanket' (covered in the model analysis section) identified by the nodes shared between Priestley-Taylor and ASCE. The blue nodes indicate the variables shared between the models, and the green nodes the variables in ASCE that do not appear in Priestley-Taylor but are direct parents of any nodes along a path between the blue nodes; these are the ASCE variables that directly affect any functions between the overlapping variables.

In the following six sections we describe the current state of implementation and instructions for use for each of the core AutoMATES Prototype components.

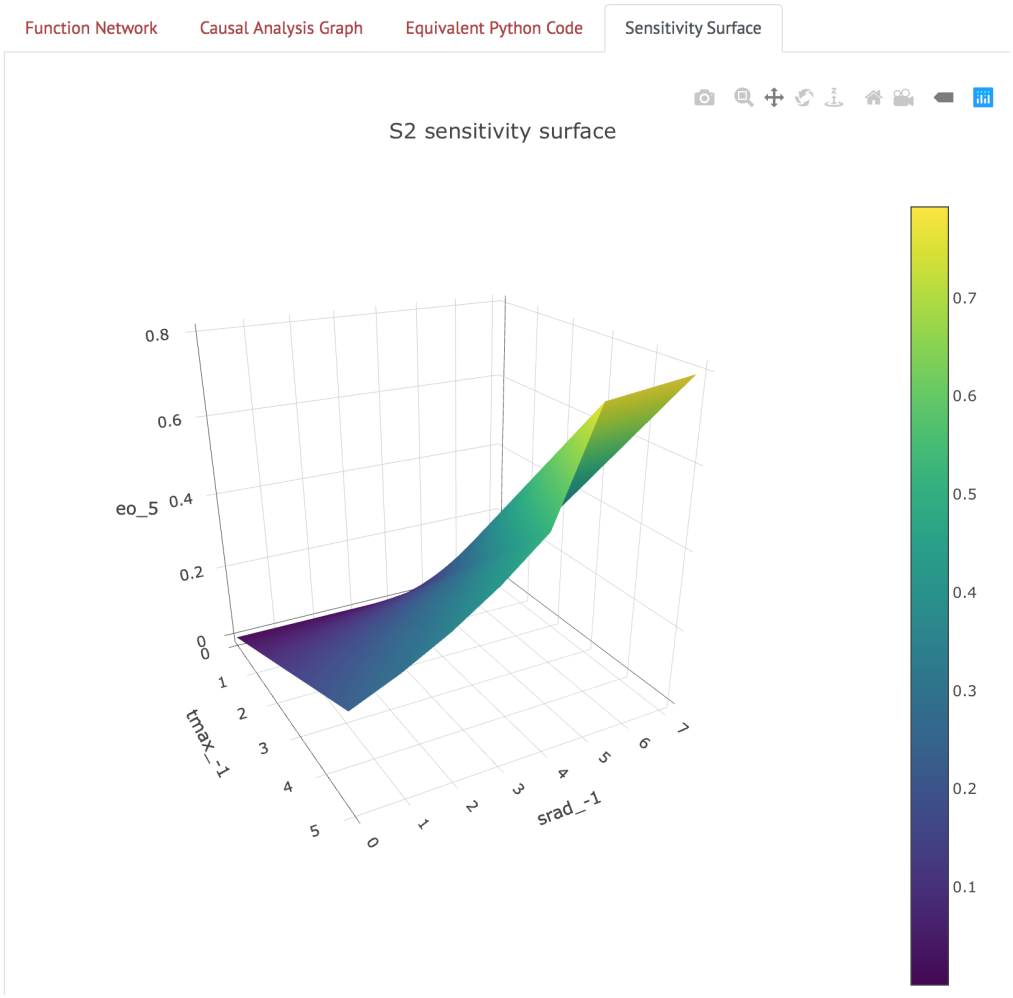

Figure 5: Sensitivity surface for pre-selected Priestley-Taylor model.

Bounds information was not found for some variables,so the S2 sensitivity surface cannot be produced for this code example.

Figure 6: Warning about missing bounds information.

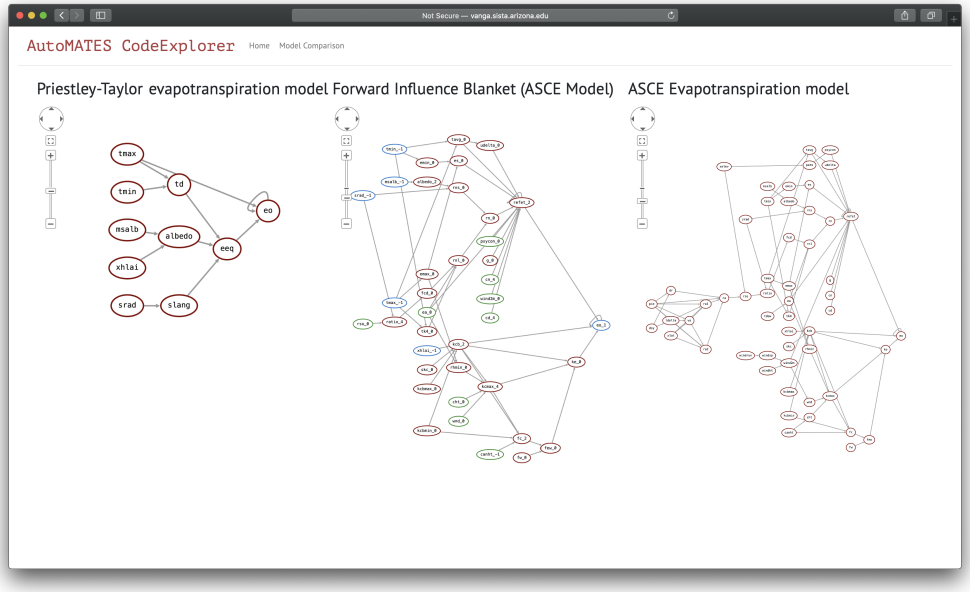

Figure 7: The model 'intersection'.

3 GrFN Specification

A Grounded Function Network (GrFN) serves as the central representation that integrates the extracted Function Network representation of source code (the result of Program Analysis) and associated extracted comments, links to natural language text (the result of Text Reading) and (soon) links to equations (the result of Equation Reading).

3.1 Specification Links

The Month 5 Phase 1 release version of the GrFN Specification as of this report is linked here GrFN Specification Version 0.1.m5. For the latest development version, see the Delphi docs.

3.2 Updates

Since the Month 3 Report on GrFN, we have made the following updates to the GrFN Specification:

- Added a "mutable" attribute to `<variable_spec>` to indicate whether a variable value can be changed (mutable) or is constant (immutable). While mutability is, strictly speaking, enforced implicitly by the wiring of functions to variables (as determined by Program Analysis), associating mutability with the variable is useful information to guide Sensitivity Analysis.
- Added a "variables" attribute to top-level `<grfn_spec>`, which contains the list of all `<variable_spec>`s. This change means that `<function_spec>`s no longer house `<variable_spec>`s, but instead just use `<variable_names>`s (which themselves are `<identifier_string>`s).
- Clarified the distinction between `<source_code_reference>`s (linking identifiers to where they are used in Program Analysis-generated lambdas file source code); previously these two concepts were ambiguous.
- Removed `<identifier_spec>` "aliases" attribute. This will be handled later as part of pointer/reference analysis.
- Did considerable rewriting of many parts of the specification to improve readability, and added links to help with topic navigation: now all `<...>`

tags are linked to the sections in which they are defined (unless already within that section).

4 Program Analysis: for2py

The core functionality of `for2py` consists of extracting the syntactic structure of the input code (Fortran) in the form of an *abstract syntax tree* (AST), which is then written out in a form suitable for subsequent analyses, eventually producing a GrFN representation and associated lambda representations of component functions. Idiosyncracies of the Fortran language make it non-trivial and time-consuming to construct a Fortran parser from scratch. Additionally, most available Fortran parsers are part of compilers and closely integrated with them, making the extraction of the AST difficult. The only suitable Fortran parser we have found, and which we use in `for2py`, is the open-source tool Open Fortran Parser, `OFP`. While its functionality as a Fortran parser fits our needs, `OFP` suffers from some shortcomings, e.g., an inability to handle some legacy constructs and handling a few other constructs in unexpected ways, that require additional effort to work around.

4.1 Architecture

The architecture of the `for2py` Fortran-to-GrFN translator is shown in Figure 8. It is organized as a pipeline that reads in the Fortran source code, transforms it through a succession of intermediate representations, as described below, and writes out the GrFN representation of the code along with associated Python code ("lambdas").

The processing of the input source code proceeds as follows:

- **Preprocessing:** This step processes the input file to work around some legacy Fortran features (Fortran-77) that `OFP` has trouble handling, e.g., continuation lines and comments in certain contexts.
- **Comment processing:** Like most other programming language parsers, `OFP` discards comments. Since comments play a vital role in AutoMATES, we use a separate processing step to extract comments from the input code for later processing. For comments associated with particular lines of code,

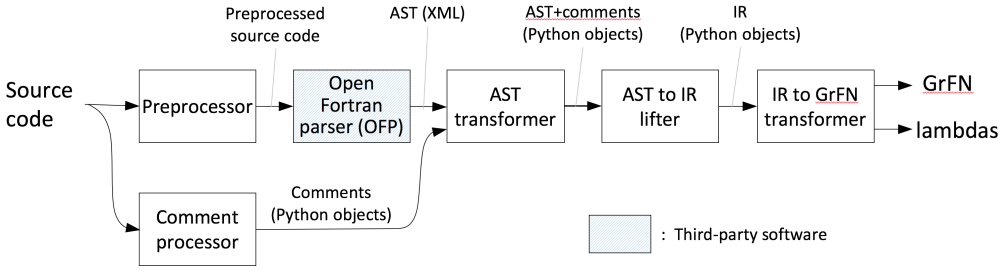

Figure 8: for2py architecture.

we add special marker statements into the source code that force `OFP` to identify them and embedded in the AST. This allows us to subsequently associate the extracted comments with the corresponding points in the code.

- **AST Extraction:** This step uses the `OFP` Fortran parser to extract an AST of the input program in XML form.
- **AST transformation:** This step takes the AST obtained from `OFP` and simplifies it by removing irrelevant Fortran-specific details. It then combines the resulting data structure with the comments extracted from the program so that comments can be correlated with the appropriate program points. The result is written out as a pickled Python data structure.
- **AST to IR lifter:** The simplified AST resulting from the previous step is then lifted into `for2py`'s internal representation (IR). As an intermediate step of this process, `for2py` generates a Python program that is behaviorally equivalent to the input code and which can be used to validate the translation.
- **GrFN generation:** The IR constructed in the previous step is mapped to the GrFN specification of the input code together with associated Python functions ("lambdas")

4.2 Instructions for running components

Many of the components in the pipeline described above can be run as standalone programs as described below.

- **Preprocessing:** `python preprocessor.py` *in_file out_file*
- **Comment processing:** `python get_comments.py` *src_file*
- **OFP parser:** `java fortran.ofp.FrontEnd --class fortran.ofp.XMLPrinter --verbosity 0` *src_file* > *ast_file*
- **AST transformation:** `python translate.py -f` *ast_file* `-g` *pickle_file* `-i` *src_file*
- **AST to IR lifter:** `python pyTranslate.py -f` *pickle_file* `-g` *py_file* `-o` *out_file*
- **GrFN generation:** `python genPGM.py -f` *py_file* `-p` *grfn.json* `-l` *lambdas.py* `-o` *out_file*

4.3 Updates

Our implementation effort since the last report has focused on the following:

1. **Modules.** At the time of our previous report, we were able to read in Fortran modules into the `for2py` IR. Since then we have focused on translating this IR into the corresponding GrFN representation. This led to a reexamination and generalization of the treatment of scopes and namespaces in GrFN, now represented within GrFN using `<identifiers>`. We have now incorporated this generalization into the translation of module constructs in `for2py`.
2. **Derived types.** These are the Fortran equivalent of `structs` in the C programming language; `for2py` translates these into Python `dataclasses` . The translation of derived types is complicated by `OFP`'s handling of this construct. We are currently working on (*a*) refactoring and cleaning up our initial code to handle derived types to make it easier to extend, debug, and maintain; and (*b*) extending the implementation to handle nested derived types. This work is in progress.
3. **Comment handling.** We have extended our handling of comments to include both function-level and intra-function comments. This necessitated developing a way to insert semantics-preserving markers into the ASTs generated by `OFP` and associating each such marker with the corresponding intra-function comment. This strategy has been implemented.
4. **Testing and debugging.** We have been working on testing and debugging the end-to-end translation pipeline to improve its usability and robustness.

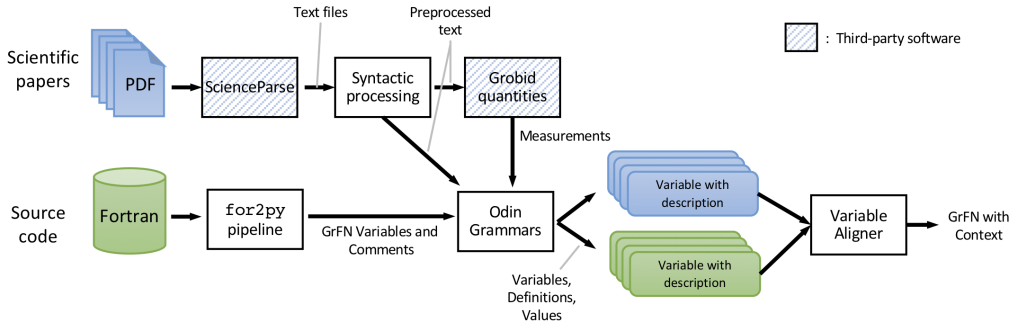

Figure 9: Architecture of the text reading pipeline

5 Text Reading

5.1 Architecture

To contextualize the models lifted from source code, we have implemented a machine reading system that extracts model information from two sources: (*a*) the scientific papers and technical documents that describe a model of interest (from which we extract the variables, the definitions for those variables, and the parameter settings for the variables); and (*b*) the comments from the Fortran source code (from which we can extract the variables, variable definitions, and potentially the units).

For paper text extraction, we use a pre-processing pipeline that converts PDFs to text, parses the text, and extracts measurements. A set of rule grammars then extract variables, their descriptions, and their values. For the source code comment extraction, we also use rule grammars to extract the variables and descriptions. The text variables are aligned with their corresponding comment variables using lexical semantics to provide richer context for the CodeExplorer webapp as well as to inform analyses performed in downstream components (e.g., model Sensitivity Analysis).

Detailed instructions on how to run this pipeline are provided below.

5.2 Natural language data preprocessing

The first stage in the text reading pipeline consists of processing the source documents into a format that can be processed automatically. As the documents are primarily in PDF format, this requires a PDF-to-text conversion. The team evaluated several tools on the specific types of documents that are used here (i.e., scientific papers) and in the end chose to use Science Parse for both its quick processing of texts as well as the fact that it does a good job handling section divisions and Greek letters. Science Parse was then integrated into the text reading pipeline via a Docker container such that it can be run offline (as a preprocessing step) or online during the extraction.

As the PDF-to-text conversion process is always noisy, the text is then filtered to remove excessively noisy text (e.g., poorly converted tables) using common sense heuristics (e.g., sentence length). This filtering is modular, and can be developed further or replaced with more complex approaches.

After filtering, the text is syntactically parsed and processed with grobid-quantities, an open-source tool which finds and normalizes quantities and their units, and even detects the type of the quantities, e.g., *mass*. The tool finds both single quantities and intervals, with differing degrees of accuracy. The grobid-quantities server is run through Docker and the AutoMATES extraction system converts the extractions into mentions for use in later rules (i.e., the team's extraction rules can look for a previously found quantity and attach it to a variable). While grobid-quantities has allowed the team to begin extracting model information more quickly, there are limitations to the tool, such as unreliable handling of Unicode and inconsistent intervals. For this reason, the extraction of intervals has been ported into Odin (rule-based extraction), where the team is using syntax to identify intervals and attach them to variables.

5.3 Rule-based extraction frameworks

For extracting model information (e.g., variables, their descriptions, values, etc.) from free text and comments, the team implemented a light-weight information extraction framework for use in the ASKE program. The system incorporates elements of the machine reader developed for the World Modelers program, Eidos (e.g., the development webapp for visualizing extractions, entity finders

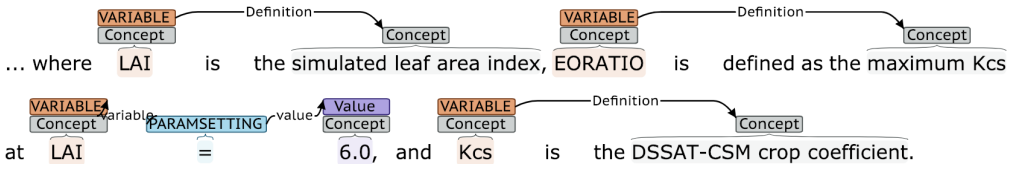

Figure 10: Screenshot of ASKE text reading webapp for rapid rule development

based on syntax and the results of grobid-quantities, and the expansion of entities (Hahn-Powell et al., 2017) that participate in relevant events) along with new Odin grammars (Valenzuela-Escárcega et al., 2016) for identifying, quantifying, and defining variables, as shown in the screenshot of the development webapp in Figure 10.

Odin grammars have proven to be reliable, robust, and efficient for diverse reading at scale in both the DARPA Big Mechanism program (with the Reach system) and the DARPA World Modelers program (with the Eidos system). The flexibility of Odin's extraction engine allows it to easily ingest the normalized measurements from grobid-quantities along with the surface form and the dependency syntax of the text, such that all representations can be used in the rule grammars during extraction.

To promote rapid grammar development, the team is using test-driven development. That is, the team has created a framework for writing unit tests representing ideal extraction coverage and is adding rules to continue to increase the number of tests that are passed. This approach allows for quickly increasing rule coverage (i.e., we are writing more rules as we continue to analyze the data and the outputs from the text reading system) while ensuring that previous results are maintained. Currently, there are 83 tests written to test the extraction of variables, definitions, and setting of parameter values from text and comments, of which 45 pass.

Summary of extraction unit tests

Type of Extraction	Number of Tests	Number Passing
Comment defintions	6	6

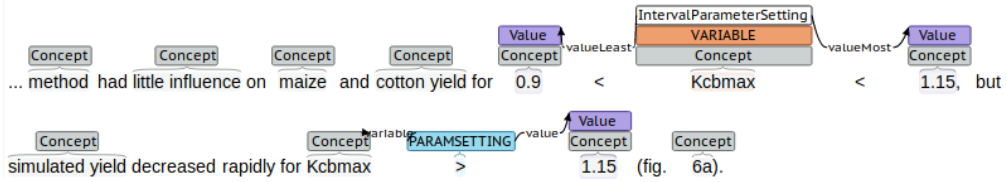

Figure 11: Example of extraction of parameter values and ranges from free text.

Type of Extraction	Number of Tests	Number Passing
Parameter Setting	12	6
Variables	30	17
Free text defintions	35	16
TOTAL	83	45

Unsurprisingly, fewer of the tests for free text (i.e., the scientific papers) pass, as there is much more linguistic variety in the way variables and their definitions or descriptions are written. Moving forward, the team will continue to develop the grammars to increase recall, as well as performing a more formal evaluation of the precision.

For paper reading, we currently have several small sets of rules written for extracting entities (eight rules), definitions (four rules), and parameter settings (eight rules). The rules for parameter settings extract both values and value intervals/ranges, as shown in Figure 11.

Extraction from comments

For comment extraction, we currently consider all of the comments from a given source file, though in the future we will implement the ability to query ever-widening program scopes to retrieve sections of comments which are most relevant to a given GrFN variable. We then use regular expressions to parse the comment lines, which cannot be straightforwardly be processed using standard tools because the sentence boundaries are not easily determinable, and even if segmented, the comments are not true sentences. Then, we locate instances of

the model variables which are retrieved from the GrFN JSON by using string matching. Finally, we use surface rules to extract the corresponding descriptions.

Since there is less variability in the way comments are written than text from scientific papers, there were only two rules necessary for comment reading—one for extracting the variable and one for extracting the definition for the variable.

Future work

We plan to work towards increasing the *recall* of free text extractions by writing rules to extract the target concepts from a wider variety of both syntactic and semantic contexts, increasing the *precision* by constraining the rules in such a way as to avoid capturing irrelevant information, and templatizing the rules wherever possible to make them easier to maintain. Additionally, we will gradually extract additional relation types (e.g., *precondition*) that are determined to be useful by the downstream consumers — that is, we will continue to develop the extractions to better facilitate Model Comparison and Sensitivity Analysis.

5.4 Alignment

We have implemented an initial variable mention alignment module, based on the lexical semantic approach used to ground concepts in the Eidos system. Variables are first retrieved from the GrFN JSON and then their comment descriptions are extracted as described above. Then, each of these variables is aligned with variables (and their defintions) from free text. The alignment is based on comparing *word embeddings* from the words in the defintions. Formally, each of the vector embeddings for the words in the description of a GrFN variable, v_G, are compared (using cosine similarity) with each of the embeddings for the words in the definition of a text variable, v_t. The score for the alignment between the variables, $S(v_G, v_t)$, is the sum of the average vector similarity and the maximum vector similarity, for a score which ranges between $[-2.0, 2.0]$. The intuition is that the alignment is based on the overall similarity as well as the single best semantic match. This approach is fairly naive, but turns out to be robust. The team will continue to develop this approach as the program continues, ideally by making use of external taxonomies which contain additional information about the variables and also the variable type (i.e., int, string, etc.) and units found in

the text and comments. The final alignments are output to the GrFN JSON for downstream use.

5.5 Instructions for running text reading components

We have separate README files that provide instructions for running the individual components of the text reading pipeline:

- Development webapp for visualizing the extractions
- ExtractAndAlign entrypoint, for extracting model information from free text and comments and generating and exporting alignments. This is the primary pipeline for the text reading module.

5.6 Updates

The team has made progess in several areas since the last report. Here we summarize the new additions, which are described in much more detail in the sections above.

- **Alignment**: Since the last report, the team has added the Alignment component descibed above to align the variables from the source, the mentions of these variables in the comments, and the corresponding variables in the free text model descriptions.
- **Import and export in GrFN JSON format**: To facilitate the extraction of variables and descriptions from comments as well as to provide the aligment information to downstream components, the team has added code to parse GrFN JSON files and generate new versions with the text-extracted context associated with GrFN variables.
- **Reading of comments**: The team added code to select relevant lines from the source code comments and tokenize them. Additionally, a small number of new rules were developed to extract variables and descriptions from the comment text.

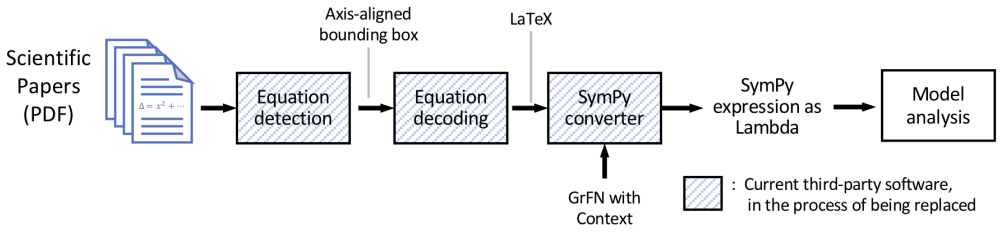

Figure 12: The overall architecture for equation detection and reading

6 Equation Detection and Parsing

Models are often represented concisely as equations, at a level of abstraction that can supplement both the natural language description as well as the source code implementation. Accordingly, the AutoMATES team is implementing a module for automatically reading equations found in scientific papers. This section details the approaches for (*a*) data acquisition, (*b*) detecting the location of equations, (*c*) encoding the image of the equation and then decoding it into a formal representation, and (*d*) converting the formal representation into an executable form that can be used in Program Analysis. Here we discuss current progress as well as planned next steps. To make rapid progress, the team has extensively explored available state-of-the-art (SOA) open-source tools and resources, and so additionally we discuss the limitations of these tools as well as our plans for addressing these limitations.

All code for the Equation Reading pipeine is implemented within the AutoMATES equation_extraction repository directory. There is additionally a section below devoted to explaining the steps needed to run each component of the architecture.

6.1 Architecture

Figure 12 shows the Equation Detection and Reading architecture. Blue-stripe-filled boxes indicate SOA open source components that are currently used. These components currently run stand-alone and are not yet integrated into an end-to-end pipeline, but will soon. The grey-label lines between the SOA components

indicate the representation that is to be passed between them. In the following sections we describe the discovered limitations of these third-party components for our use case and our plans for replacing/extending each.

6.2 Data collection

We constructed several datasets in order to train and evaluate the neural machine learning components used in the detection and decoding of equations found in text. For this work, the team is making use of papers written in LaTeX (TeX), downloaded in bulk from arXiv, an open-access preprint database for scientific publications. The team has downloaded the complete set of arXiv PDFs and their corresponding source files from Amazon S3 (as described here). Similar datasets have been constructed previously, but they are very limited in scope. For example, a sample of source files from the `hep-th` (theoretical high-energy physics) section of arXiv was collected in 2003 for the KDD cup competition (see equation decoding section for examples of the consequence of this limited scope). By downloading the full set of arXiv, the team has extended this dataset to both increase the number of training examples and to include a variety of AutoMATES-relevant domains (including agriculture, biology, and computer science).

Dataset preprocessing pipeline

The team has put together a preprocessing pipeline with the purpose of preparing the downloaded arXiv data for use in equation detection and decoding.

First, the paper source files are organized in a directory structure that can be processed efficiently. Then, for each paper, the TeX file that has the `\documentclass` directive is selected as the main TeX file (see here for more information). Once a main TeX file has been selected, the TeX source is tokenized using plasTeX and the content of certain environments are collected together with the environment itself (e.g., the tokens inside the `\begin{equation}` and `\end{equation}` directives, together with the label `equation`). User-defined macros are expanded using code developed by the team to normalize the input to the neural decoder.

Based on an analysis of 1600 arXiv papers, the most commonly used math environments (in order) are: `equation`, `align`, and `\[ \]`. While the Prototype

currently only handles the `equation` environment (40% of the equations found), the pipeline will be extended to accomodate the other two types in the future.

The extracted code for each equation is rendered into a standalone equation image. The paired standalone image and source tokens form the training data for the equation decoder. Additionally, the PDF file for the entire paper is scanned for the standalone equation image using template matching. The resulting axis-aligned bounding box (AABB) is stored for the subsequent training of an equation detector.

While much of the preprocessing pipeline is complete, there are a few remaining elements that need to be implemented. The planned work includes additional normalization of the extracted TeX tokens to provide the equation decoder a more consistent input. At minimum, this will include the removal of superfluous code such as `\label{}` directives, the normalization of certain LaTeX expressions (e.g., arbitrary ordering of super- and sub-script annotations in equations), and the augmentation of the rendered standalone equations.

6.3 Equation detection

Before equations can be decoded, they first need to be located within the scientific papers encoded as PDF files. For this, the team evaluated standard machine vision techniques. The SOA Mask-RCNN (He et al., 2017) was selected both for its robust performance across several detection tasks as well as its ease of use. Here, as the desired output of the model is the page and AABB of the detected equations, we ignore the mask (i.e., the precise set of pixels which compose the object), and as such the model is essentially an easy to use Faster R-CNN (Ren et al., 2015).

The Faster R-CNN model uses a base network consisting of a series of convolutional and pooling layers as feature extractors for subsequent steps. This network is typically a ResNet backbone trained over ImageNet or COCO.

Next, a region proposal network (RPN) uses the features found in the previous step to propose a predefined number of bounding boxes that may contain equations. For this purpose, fixed bounding boxes of different sizes are placed throughout the image. Then the RPN predicts two values: the probability that the bounding box contains an object of interest, and a correction to the bounding box to make it better fit the object.

At this point, the Faster R-CNN uses a second step to classify the type of object, using a traditional R-CNN. Since there is only one type of object of interest (equations), the output of the RPN can be used directly, simplifying training and speeding up inference. However, one potential disadvantage of only having a single label is that the model could be confused by similar page components (e.g., section titles and tables). Since we have access to the TeX source code, in the future we can include these other objects and their labels and will train the model to differentiate between them explicitly. Currently, the team has been able to train the Mask-RCNN on the equation images and AABBs from a subset of the arXiv data.

6.4 Equation decoding

Once detected, the rendered equations need to be automatically converted into LaTeX code. For this task we employ a variant of an encoder-decoder architecture that encodes the equation image into a dense embedding and then decodes it into LaTeX code capable of being compiled back into an image. LaTeX was selected as the intermediate representation between input image and the eventual target executable model of the equation because of the availability of a large amoutn of training data (arXiv) and because LaTeX preserves both typographic information about how equations are rendered (e.g., bolding, italics, subscript, etc.) while also preserving the components of the notation that will be used for the successful interpretation of the equation semantics.

Encoder-decoder architectures have been successfully applied to image caption generation (e.g., Vinyals et al., 2017), a task that is similar to our task of mapping equation images to equation code. In order to make rapid progress, we began with an existing SOA model previously trained for the purpose of converting images to markup (Deng et al., 2017). The model was trained with the 2003 KDD cup competition, which itself consists of a subset of arXiv physics papers.

We evaluated the pre-trained Deng et al. model for our use case by taking a sample of 20 domain-relevant equations from a scientific paper describing the ASCE evapotranspiration model (Walter et al., 2000) (which is also implemented in DSSAT and has been made available for analysis in the Prototype CodeExplorer webapp).

Based on this, we have learned that the Deng et al. model is sensitive to the source image equation font as well as the size of the image. For this reason, we present here the results of performance of the model based on two versions of the source equation images, as we have learned important lessons by considering both.

For the first version of the equation images, we attempted to create a version of the equations that matches as closely as possible the equation font and rendered image size as was used in the origina training data used by Deng et al. This was done by manually transcribing the equations into a LaTeX that generates stylistically close equation renderings. We refer to this as the *transcribed* version of the equations.

For the second version of the equation images, we modeled how we expect to get the equations from the equation extraction pipeline. Here, the equation image was constructed by taking a screenshot of the equation from the original source paper, and then cropping the image to the size of the AABB returned from the equation detection process described above. We found that in this case, the pre-trained Deng et al. model was unable to successfully process these images until they were rescaled to 50% of their original size. After rescaling, we then ran the images through the Deng et al. pipeline. We refer to these images as the *cropped* version of the equations.

When comparing the results of the model on the *transcribed* and *cropped* equation images, we found several patterns. First, there are several cases in which the pre-trained model is able to completely correctly decode the *transcribed* image (12 out of 20, or 60% of the images). This correct decoding accuracy drops to 5% when using the *cropped* images (strictly speaking, it is 0%, but on one image the only mistake is superficial). The proportion of character-level mistakes (e.g., wrong characters or incorrect super/sub-script, etc.) also increases from 2.7% to 31.2% when comparing the results of the *transcribed* and *cropped* images (numbers reflect only images which are renderable). This highlights the brittleness of the pre-trained model: when used on images that do not conform to the conditions of the original training data used by Deng et al., the model is unable to produce usable results.

We also found that when the model is given the *cropped* images, because of the font difference the model over-predicts characters as being in a bold typeface.

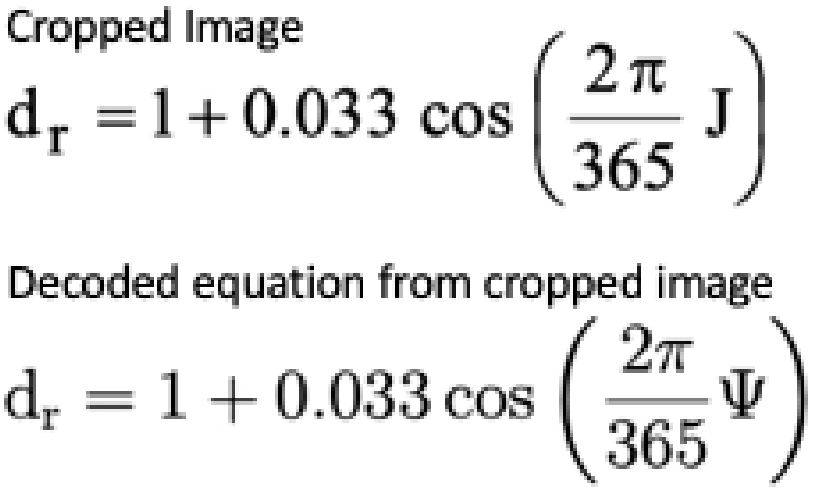

Figure 13: Incorrect decoding of 'J' to 'Ψ'.

While this difference is minor for general text, bold typeface is semantically meaningful in mathematical notation as it signals that the variable is likely a vector rather than a scalar.

We also observed that in multiple cases, a 'J' in the original equation was decoded it as a Ψ, as shown in the comparsion in Figure 13.

A likely explanation for the confusion between 'J' and 'Ψ' is that the Deng et al. model was pre-trained on a subset of arXiv that contains only articles on particle physics. In this domain, there is a specific subatomic particle that is referred to as J/Ψ. The model is likely over-fitted to the specific domain it was trained on.

To address these issues, our next steps will use the following strategies:

1. First, we will retrain the model using a much larger portion of arXiv. This represents a much wider set of equation styles (from different disciplines) and has orders of magnitude more data. We anticipate that this will help address the issue of overfitting.
2. Second, in order to make the model more robust to differences in image size, font, typesetting, etc., we will augment the training data by taking

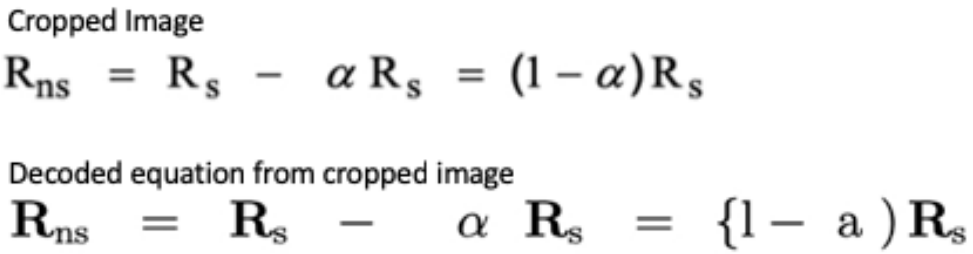

Figure 14: Example with mistmatched brace and parenthesis.

the source LaTeX and rendering the equations under different conditions, such as keepin the same base equation but changing the font, image size, rotating, adding blurring, etc. This kind of data augmentation is a standard technique for improving model generalization in machine vision systems (e.g., Baird, 1993; Wong et al., 2016; Wang & Perez, 2017, *inter alia*).

3. And third, we will use methods for post-processing the sequent-to-sequence decoder output. This will address several kinds of errors that appear to be a result of the unconstrained nature of the sequence decoder. Decoder errors are manifested in several ways. In some cases, the pre-trained model generates a sequence of tokens which cannot be compiled in LaTeX (10% of the *transcribed* images and 35% of the *cropped* images). For example, the generated LaTeX cannot be compiled because the decoder produced a left bracket without producing the corresponding right bracket. This mismatch may occur even when the generated LaTeX *can* be compiled, as in the example in Figure 14, where an open brace is generated followed by a close parenthesis.

   Figure 14: Example with mistmatched brace and parenthesis.

   Another example of decoding that may benefit from post-processing is when multiple mentions of a single variable in an equation are decoded differently, as shown in Figure 15.

   Figure 15: Multiple variable mentions with inconsistent decoding.

   In the left-most example (a) of Fig 15, the problem is minor, as the wrongly decoded variable is where the equation is being *stored*. In the right-most equation (b), the semantics of the formula are completely lost when 'T'

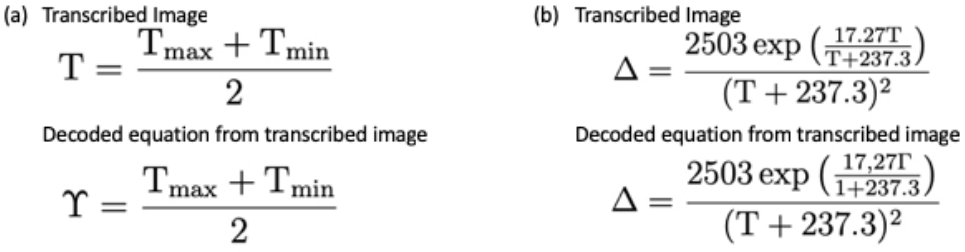

Figure 15: Multiple variable mentions with inconsistent decoding.

is incorrectly decoded as 'Γ'. In both cases, the problem will be exacerbated when converting the equations to executable code and especially when the extracted information needs to be assembled for model analysis.

To address decoding errors like these, the team will explore methods for enforcing syntactic constraints on the decoded sequence. For example, one solution is to shift from making local decoding decisions to finding the global best decoding for the image. Currently, at a given point in the sequence during decoding, the decision about what token should be produced next is made by greedily choosing the output token with the highest likelihood, given the input encoding and the previously decoded tokens. Instead, when the decisions are made to find the best *global* sequence, then the model is unlikely to produce certain token combinations that never occured in training (e.g., a left bracket without a matching right bracket). We will explore several strategies, including using a conditional random field layer on top of the decoder as well as using the Viterbi algorithm with domain-specific constraints. Additionally, a grammar can be used with the decoder (as was done by Krishnamurthy et al. (2017) for generating well-formed logical forms for use in querying a knowledge base) to ensure valid LaTeX is generated.

6.5 Conversion to executable representation

The final stage in the pipeline is the conversion of the equation to an executable representation. We chose to use SymPy for two reasons. First, and primarily,

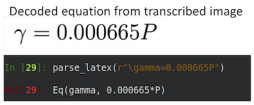

Figure 16: Example of LaTeX to SymPy conversion.

(a) SymPy created from direct output of model (spaces removed) using `latex2sympy`

```
In [32]: parse_latex(r"\mathrm{e}_{\mathrm{a}}=\mathrm{e}^{\mathrm{o}}\left(\mathrm{T}_
{\mathrm{max}}\right)\frac{\mathrm{RH}_{\mathrm{min}}}{100}")
Out[32]: e*mathrm
```

(b) After removing font typesetting

```
In [33]: parse_latex(r"e_a=e^o\left(T_{max}\right)\frac{RH_{min}}{100}")

Out[33]: Eq(e_{a}, e**o*(((H_{m*(i*n)}*R)/100)*left(T_{m*(a*x)}*right)))
```

(c) After removing specialized grouping symbols

```
In [36]: parse_latex(r"e_a=e^o(T_{max})\frac{RH_{min}}{100}")

Out[36]: Eq(e_{a}, e**o*(T_{m*(a*x)}*((H_{m*(i*n)}*R)/100)))
```

Figure 17: Improving SymPy conversion by removing additional LaTeX notation.

SymPy provides a symbolic representation of the equation so that while it is executable, variables can remain unassigned. Second, the Program and Model Analysis uses python as the intermediate language. There is an available open-source library called latex2sympy (which has been experimentally incorporated into sympy) for converting LaTeX expressions into SymPy expressions. The conversion makes use of a manually crafted antlr4 grammar.

After some small post-processing (e.g., removing spaces introduced between characters in the generated LaTeX), we found that simple expressions were correctly converted, such as in Figure 16:

However, more complex equations may not be handled due to additional LaTeX font specification and specialized grouping symbols, as demonstrated in Figure 18. Removing these additional annotations improves equation SymPy conversion.

That said, even after taking these steps, it is clear that we will need to extend the antlr4 grammar in order to handle the decided equations. In particular, we need to inform the splitting of characters into distinct variables (e.g., subscripts such as *max* should not be considered as three variables multiplied together, *eo* should not be considered as an exponential if we have previously seen it defined as a variable, etc.). Also, equations that contain other equations need to be represented with function calls, rather than multipication (e.g., *eo(T)* is a reference to an equation so needs to be interpreted as a single symbol, but latex2sympy converts it as `e**o*(T)`). Moving forward, our strategy is to expand the latex2sympy grammar and also consider expanding the plasTeX library that we are using for LaTeX tokenizing LaTeX (which will improve LaTeX code handling, such as spacing, etc.).

6.6 Instructions for running components

We have separate README files for the individual components of the equation reading pipeline:

- Downloading and processing arXiv
- Detecting equations in a pdf
- Decoding equation images into LaTeX (requires a GPU)

6.7 Updates

Since the last report, progress has been made in the following four areas, with many of the details described in the sections above.

- **Data collection**:
  - Since the last report, the team added LaTeX macro expansion, accomplished through a recursively applied lookup table. This allows for the normalization of tokens for training the equation decoder.
  - The team also incorporated template rescaling to better match the rendered equation against the original PDF image. This resulted in significantly more accurate axis-aligned bounding boxes.
- **Equation detection**:

- The team installed the current SOA Mask-RCNN model, processed the training data to fit the required model format, and evaluated the model with our corpus.

- **Equation decoding**:
  - The team succesfully reproduced equation decoding results from Deng et al.
  (2017) paper using their pre-trained model and the provided evaluation data. We have additionally successfully run the training procedure with a small toy dataset. We are in the process of reimplementing the model to allow for greater control of the inputs, architecture, and computation resource requirements (CPU in additional to GPU) to address the limitations found in the original Deng et al. implementation.

- **Conversion to executable representation**:
  - The team has chosen a library for converting the generated LaTeX to SymPy and evaluated the output. Based on the our evaluation, the team is working on expanding the antlr4 grammar and also looking into extending the plasTeX library.

7 Model Analysis

The outputs of the Program Analysis, Text Reading, and Equation Reading pipelines form the inputs to the Model Analysis pipeline. The Program Analysis module provides a wiring specification and a set of associated lambda functions for the GrFN computation graph. Together, these fully describe the structure of a computational model implemented in source code. The outputs from Text and Equation Reading are combined in the GrFN specification in order to link the variables used in computation. This information is critical for the model comparison phase of Model Analysis. The Model Analysis pipeline utilizes all of these inputs, as well as similar sets of inputs for other scientific models, to generate a model report. The eventual goal of the report is to contain all information recovered and inferred during Model Analysis about the sensitivity of the model to various inputs over various ranges and, if requested, information about comparative

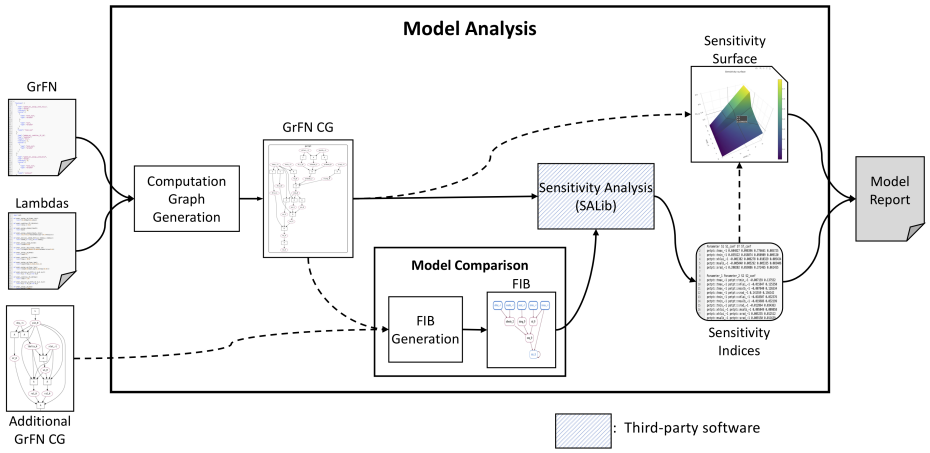

Figure 18: Model Analysis architecture

model fitness to other models of the same output. As of this Phase 1 release of the AutoMATES Prototype, individual components of a model report are generated during Model Analysis, but the integration and format of the final report is still under development. In the following, we describe the current state of the Model Analysis components.

7.1 Architecture

The architecture for the Model Analysis module is shown in Figure 18. The overall structure of the module is a pipeline that transforms extracted information from text, equations, and software into a report about a scientific model. However, this pipeline has optional additional components (represented by dashed input arrows) that can be included in the final model report based upon the amount of information requested by the user. In this section we will describe the intermediate objects created during model analysis as well as the algorithms used during their generation.

Computation graph generation

Generating an actual computation graph from a GrFN spec and lambdas requires two phases, a wiring phase and a planning phase. The output from these two phases will be an executable computation graph.

- **Wiring phase**: This phase utilizes the GrFN specification of how variables are associated with function inputs and outputs in order to "wire together" the computation graph.
- **Planning phase**: In this phase, a partial order is imposed over the lambda functions in the call graph of the GrFN, to determine an efficient order of computation and discover sets of functions that can potentially be executed in parallel. The ordering is recovered using `HeapSort` on the function nodes in the computation graph, where the heap invariant is the distance from the function node to an output node. The output of this algorithm is a call stack that can be used to execute the `GrFN` computation graph. After this phase is completed the computation graph can be executed as many times as needed without requiring a call-stack creation per computation, allowing for more efficient sampling and sensitivity analysis.

An example of the `GrFN` computation graph for `PETASCE` (the ASCE Evapotranspiration model) is shown in Figure 19.

In the figure, variables are represented by ovals and functions by boxes. The function nodes are labeled by a single-letter code denoting the type of function computed - these correspond to the same labels in the CodeExplorer Function Network description.

Model comparison

Computation graphs can be used as the basis to compare models. First, we identify what variables are shared by the two models. We can then explore how the models make similar (or different) assertions about the functional relationships between the variables.

Of course, the functions between the variables may be different, and other variables that are not shared might directly affect the functional relationships between the shared variables.

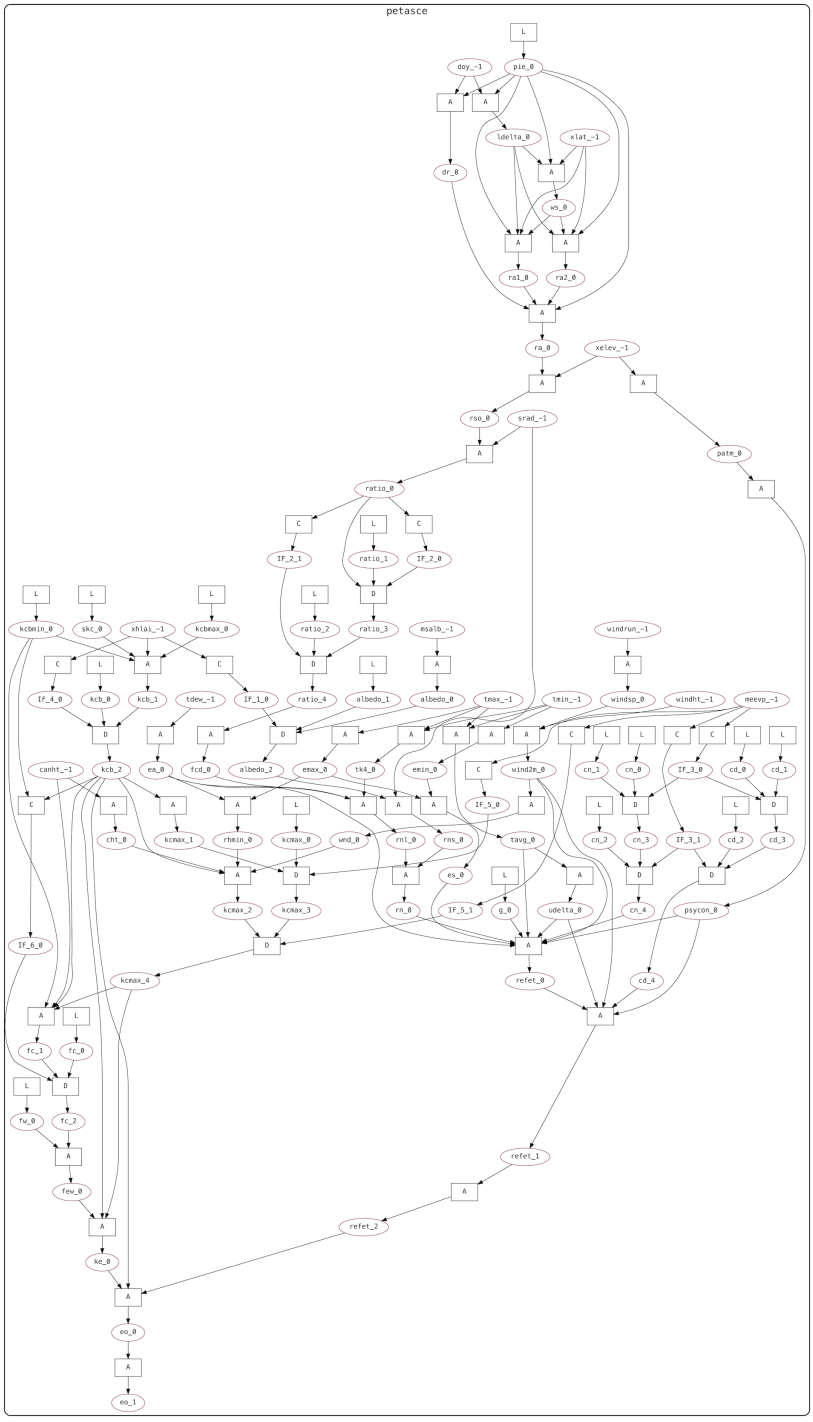

Figure 19: PETASCE GEN computation graph.

Given the computation graphs for two models, we first identify the overlap in their variable nodes. In the examples we consider here, this is done using variable base-name string matching, since we are comparing two models from DSSAT, and the variables have standardized names in the code; more generally, we will rely on variable linking using information recovered from Text and Equation Reading. Once we have a set of shared variables, we then identify the subgraphs of each model that contain all directed paths from input shared variables to output shared variables. As part of this, we also identify any other variables in the model that serve as direct inputs to the functions along the paths between input and output shared variables. These additional, direct input variables that are not shared are referred to as the *cover set* of the subgraph. These are variables whose state will affect how the shared variables relate.

As a final step, we look to see if any variables in the cover set are assigned values by a literal function node. We remove these variable nodes from the cover set, and add them and their literal assignment function node directly to the subgraph. The combination of shared variable input nodes, cover set variable nodes, and additional wiring forms a `Forward Influence Blanket` or `FIB` for short. An example of the `FIB` for the `PETASCE` model is shown in Figure 20. Here we can see the shared input variables colored in blue, while the *cover set* variables are colored green.

The identified `FIB` of each model then forms the basis for directly comparing the components shared between the models. An open challenge is to automate identification of *cover set* variables value ranges that make the functional relations between shared variables achieve the same input-to-output functional relationships between the two models (or as close the same as possible). As of this report, we assume these value ranges are identified. Given these value ranges, we can then contrast the two model `FIB`s by performing sensitivity analysis over the shared variables.

Sensitivity analysis and model reporting

Sensitivity analysis identifies the relative contribution of uncertainty in input variables (or variable pairs, in second order analysis) to uncertainty in model output. In the current AutoMATES Prototype, we use use the SALib python package to perform several types of sensitivity analysis. The package provides

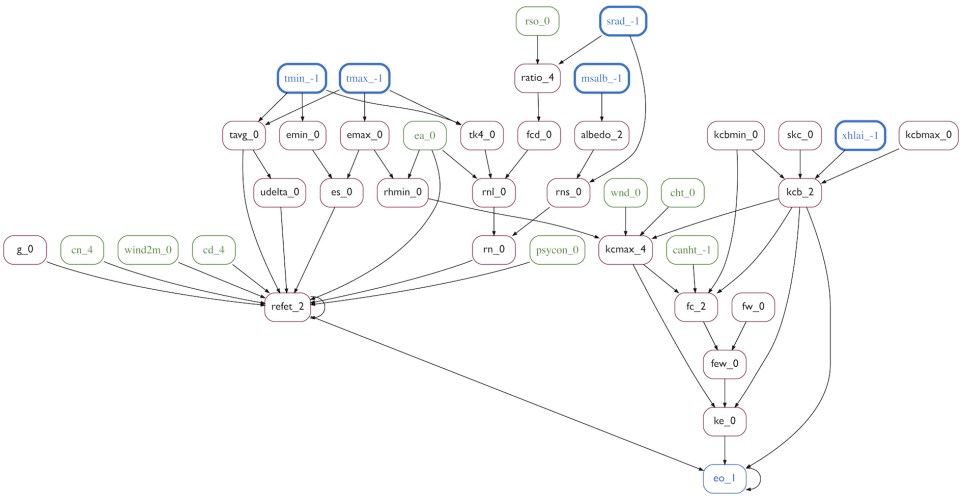

Figure 20: PETASCE GrFN Forward Influence Blanket.

the functionality to sample from our input variable space(s) and estimate output sensitivity over an input space. We have created a simple API that takes a `GrFN` computation graph along with a set of search bounds for all model inputs and computes the sensitivity indices associated with each input or pairs of inputs.

The sensitivity indices that are returned from a call to the sensitivity analysis API can be used to determine what information about a model is worth including in the model report. For a report about a single model, we will likely be interested in which variables (or pairs of variables) contribute the most to output uncertainty. Once we have recovered the indices we can plot the response of a target output variable as a function of the input (or pair of inputs) that contributes the most to the uncertainty in the output variable.

7.2 Instructions for running components

The model analysis pipeline has been integrated into the CodeExplorer web application. Users can visualize the outputs from model analysis by loading one of the provided Fortran programs and inspecting the GrFN Computation Graph along with its associated Function Call Graph and Causal Analysis Graph. In

addition, for the PETPT (Priestley-Taylor) model, users can visualize the sensitivity surface generated by the S2 index of highest degree: the effects of changes in srad and t–max on the output variable eo; this is plotted in the sensitivity surface tab.

This functionality can also be programmatically accessed using the S2_surface method of the GroundedFunctionNetwork class.

7.3 Current caveats

Currently we are using the PETPT and PETASCE modules for the evaluation of our model analysis module. This comes with certain advantages as well as challenges.

One advantage of these modules is that variables that represent the same real-world object have the exact same names. This enables us to easily conduct model comparison experiments on these models without relying on potentially noisy variable linking from the Text Reading module. In future iterations, our model comparison component will rely on variable link hypotheses from Text Reading.

One obstacle we needed to overcome in order to create an executable computation graph for the PETASCE model was the identification of the range of possible values variables may take on. The PETASCE model includes calls to several functions that have restricted value domains, such as log and acos. The PETASCE model itself contains no explicit guards against input values that are "out-of-domain" (mathematically undefined) for these functions, thus, domains for each input variable must be discovered via analysis. We are currently developing a component of the Model Analysis module that will infer input variable domain value constraints. Currently, the GrFN computation graph catches value exceptions to gracefully alert the sampling component of failure to execute, but in the near future we hope to infer bound information directly.

7.4 Updates

- **End-to-end generation**: Our previous results from Model Analysis were based on running analysis methods over translated Python programs from original Fortran source code or hand-constructed CAGs for model compari-

son. We are now able to conduct all the steps of Model Analysis directly on the GrFN representation generated by the Program Analysis team.

- **Vectorized model evaluation**: We have begun experimenting with vectorizing the execution of the computation graph using the PyTorch tensor computation framework, which provides a useful general interface to GPU resources, speeding up sampling. Using PyTorch requires a few changes to the GrFN specification and generated lambda functions. Preliminary experiments suggest that vectorization using PyTorch increases the speed of sampling on a CPU by a factor of $\sim 10^3$, and this increases to $\sim 10^6$ when using GPU resources. This evaluation is ongoing.
- **Additional SA methods**: Due to the computational cost of `Sobol` sensitivity analysis, we have explored two new methods of sensitivity analysis - the Fourier Amplitude Sensitivity Testing (FAST) and the variant `RBD-FAST` sensitivity analysis methods - to our suite of analysis tools. While these methods can only be used for computing first-order sensitivity indices, their compute time scales constantly with number of samples (for comparison, Sobol analysis scales linearly). Figure 21 demonstrates these runtime differences; these results were generated by running all of our sensitivity methods on the `PETPT` model.

8 Code Summarization

The goal of the Code Summarization module is to provide one or more methods to automatically generate natural language descriptions of components of source code. Work in Phase 1 has been devoted to evaluating and make an initial adaptation in application of the state of the art `Sequence2Sequence` neural machine translation (NMT) model described in *Effective Approaches to Attention-based Neural Machine Translation*, an encoder-decoder method for machine translation that utilizes Bi-LSTM recurrent networks over each of the input sequences to map sequences in one language to another. In the Month 3 Report on Code Summarization we presented the initial empirical results of training the NMT model on a training/evaluation corpus we developed, starting with training the code and language embedding models (the initial encodings of the source and target domains). We evaluated the capacity of the embeddings to perform a

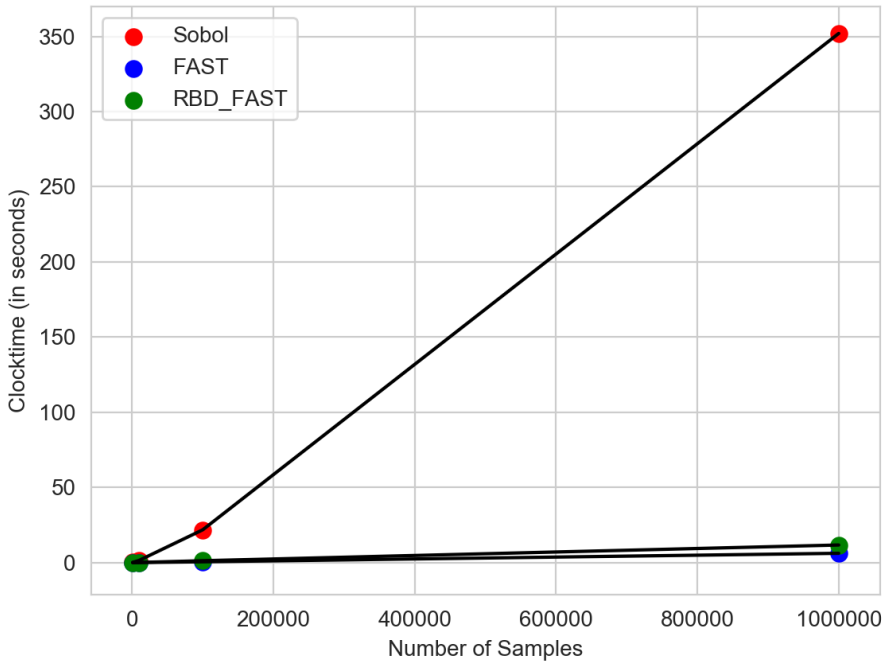

Figure 21: Sensitivity analysis compute times.

simpler "relevant/not-relevant" document classification task, which helps us assess whether the embeddings are learning information relevant to code summarization. Since the Month 3 report, we have continued evaluating the model and here summarize lessons learned so far. So far, our results in raw generation are low quality, but this is still very early and we have several different directions to take. This has also led us to consider some alternative approaches to code summarization that we will explore in the next phase of the program. In this section we review the experiments we have performed, discuss the limitations of the current neural methods, and end with description of how to obtain and process the data and train the models in our initial Prototype release.

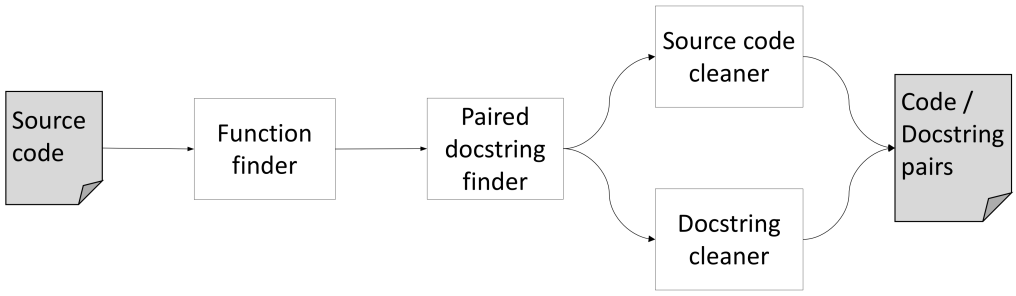

Figure 22: Code Crawler architecture

8.1 Dataset overview

The NMT model was originally developed and evaluated on data where the code sequences to be summarized are relative small: on the order of under 100 source code tokens. Source code tokens are the result of the pre-processing used in NLP before embedding: identifying what constitutes a "word", and to the extent possible, resolving variant spellings or expressions of the same concept as a single token. In our application, functions tend to have an order of magnitude more source code tokens: around 1000. For this reason, we need to identify a suitable corpus of larger source code segments with associate docstring summaries, and the corpus needs to be large. We extracted a very large corpus of Python function / docstring pairs by scraping the entirety of the awesome-python list of Python packages, extracting function source code and their associated docstrings. In order to automate the function / docstring extraction while attempting to find higher quality docstring descriptions paired with source code, we created a tool called CodeCrawler. The processing pipeline for CodeCrawler is shown in Figure 22.

The pipeline consists of these components:

- **Function finder**: This component uses Python's built-in `inspect` module to find all function signatures in a Python module. The `inspect` module also has the ability to grab the docstring associated with a function using the `__doc__` variable.
- **Paired docstring finder**: This component determines whether the doc-

string returned from the `__doc__` attribute qualifies as a descriptive function comment. To do this, we check the docstring to see if it conforms to the PEP 257 standards for docstrings, in an attempt to generally find higher quality comments, under the assumption that docstrings formatted in this way will tend to be higher quality. If the docstring conforms, we then use the function description section as our descriptive docstring.

- **Source code cleaner**: Code summarization methods generally try to learn to map from source code vocabulary tokens (what terms are used in code) to a natural language description. In general, this task is easier when the source code vocabulary is observed repeatedly under different conditions in the corpus. In order to reduce the number of unique source code vocabulary terms (and thereby increase the frequency with which each term is observered), this component "cleans" the function code by the following rules:
  - All identifiers are split based on `snake_case` and `camelCase` rules.
  - All numbers are separated into digits (e.g., 103 becomes 1, 0, 3).
  - All string literals were replaced with the token `<STRING>`.
- **Docstring cleaner**: Following the same intuition as the source code cleaner, the docstring cleaner is responsible for removing all parts of the docstring that are not specified as descriptive, according to PEP 257 standards. This is accomplished with simple pattern matching on the docstring.

Using CodeCrawler, we found a total of 76686 Python functions with well-formed docstrings.

8.2 Experimental results

In the Month 3 Report on Code Summarization, we presented the initial results of training and evaluating the embedding layer to the NMT model using the corpus extracted by the CodeCrawler. To evaluate whether the trained embeddings appear to be capturing information about the relationship of source code features to natural language summaries, we first created a simple classification task that evaluates the ability of the embeddings to inform a decision about whether a provided natural language description is "relevant" to input source code.

As reported in the Month 3 Report on Code Summarization, we created two datasets from our corpus for this classification task. In both cases, we use the 76686 *positive* association examples found using CodeCrawler, but the differences are in how *negative* function/docstring pairs are constructed:

- In the first dataset, which we call the *random-draw* dataset, for each source code function to be summarized, we uniformly randomly sampled a docstring from the corpus and associated it with the function. We expect learning to distinguish positive from negative function/docstring pairings to be relatively easy given the general varaince in docstring and code. It is possible that some randomly paired docstrings may actually be accurate or at least related summaries of the function. But in general, we estimate this probability to be low. Verifying this is still a work in progress.
- For the second, *challenge*, dataset, we used a Lucene index over the docstrings to then search for docstrings with the highest lexical overlap to the true docstring originally associated with the function. This classification task is significantly more difficult as now the associated "negative" example shares much in common with the original description, but must still be distinguished.

For both datasets, we generated 10x more negative examples than the positive examples: 766,860.

By the Month 3 report, we had only just constructed the two datasets and only had very preliminary results for the *random-draw* dataset. This phase we completed this initial study, training the neural network model using both data sets, in both cases performing a grid search to optimize the hyperparameters. The results of the two models are summarized in the following table:

Dataset	Accuracy	Precision	Recall	F1
`random-draw`	89%	48%	82%	60%
`challenge`	64%	14%	61%	23%

Not surprisingly, the accuracy is generally much higher than the F1 scores due to the general (and intended) imbalance in the data (10x more negative than positive examples).

The model achieves fairly high accuracy on the *random-draw* dataset, and this is expected given that random code strings will very likely not be relevant to the code the function has been randomly paired with. The challenge data set, however, is much more difficult, as reflected in the results. Precision, in particular, takes a very big hit.

Next, we evaluated the NMT model by training and evaluating the model on our overall corpus, itself containing soruce code tokens that are generally an order of magnitude longer than those used in the original NMT evaluation. Here, we use the BLEU-4 score as an estimate of how close the generated natural language summarization is to the original docsctring associated with the source code. Here we have found that the BLEU-4 score is so far very low: 1.88. It is hard at this point to assess by this score alone how we're doing, as to date this task has not been attempted, but it does provide a baseline for us to improve on.

In the next section we discuss the lessons learned so far.

8.3 Lessons and next steps

We manually inspected a randomly sampled subset of 100 code/docstring pairs from our corpus to better understand what might be limiting the current model's performance and identify how we can adapt the method. Based on these samples, we now have a few conjectures about the nature our code/docstring corpus.

An assumption we have been making about the corpus is that the function / docstring pairs it contains are fairly high quality, meaning that the docstrings do indeed do a good job of summarizing the function. When we initially inspected this data, we assumed that that given that the code bases used were of production quality, the quality of the documentation should be high in general. Below are two examples from our corpus that do meet these standards. The docstrings are descriptive of the actual code in the functions and the identifiers in the functions can be used to deduce the proper docstrings.

```
def __ne__(self, other):
    """Returns true if both objects are not equal"""
    return not self == other

def printSchema(self):
```

```
    """Prints out the schema in the tree format."""
    print(self._jdf.schema().treeString())
```

Unfortunately, after gathering the CodeCrawler corpus and taking an unbiased random sample, we are now finding that many of the functions found in our corpus had "associative", rather than "descriptive", docstrings, meaning that the portion of the dosctring that we are recovering, which is intended to be descriptive according to the PEP 257 standards, merely associates the function either with some other module-level construct or some real-world phenomena, rather than summarizing the code itself. Some examples of such functions are included below. It is easy to see how creating these docstrings would require outside context and cannot be recovered from the actual source code itself. While these functions may be ideal for a different task, they are not as useful, and are far more frequent than we had expected.

```
def __repr__(self):
    """For `print` and `pprint`"""
    return self.to_str()

def get_A1_hom(s, scalarKP=False):
    """Builds A1 for the spatial error GM estimation
    with homoscedasticity as in Drukker et al."""
    n = float(s.shape[0])
    wpw = s.T * s
    twpw = np.sum(wpw.diagonal())
    e = SP.eye(n, n, format='csr')
    e.data = np.ones(int(n)) * (twpw / n)
    num = wpw - e
    if not scalarKP:
        return num
    else:
        den = 1. + (twpw / n) ** 2.
        return num / den

def is_hom(self, allele=None):
```

```
    """Find genotype calls that are homozygous."""

    if allele is None:
        allele1 = self.values[..., 0, np.newaxis]
        other_alleles = self.values[..., 1:]
        tmp = (allele1 >= 0) & (allele1 == other_alleles)
        out = np.all(tmp, axis=-1)
    else:
        out = np.all(self.values == allele, axis=-1)

    if self.mask is not None:
        out &= ~self.mask

    return out
```

Additionally, we have identified a second class of problematic functions in our corpus, which have docstrings that are "descriptive", but composed mainly of references to items created in the module hierarchy of the Python module that the function resides in. This presents two challenges for docstring generation. First, it is expected to include information when generating the description of a function that is not contained in the function. This is illustrated in the first of the two examples below, where a docstring includes references to the `pandas` and `scipy` libraries. Such information would never be included in the source code of a Python function. The second challenge for docstring generation is in dealing with module level information. Python is an object-oriented language, and thus functions written in Python deal with objects, and it is expected that programmers inspecting and utilizing the functions will be aware of those objects. This additional information is again outside of the actual function itself and our current docstring generator will not have access to this information; a good example of this is shown in the second example below, where the `Point` and `imageItem` class are both objects at the module level that are referenced by the function.

```
def sparseDfToCsc(self, df):
    """convert a pandas sparse dataframe to a scipy sparse array"""
```

```
    columns = df.columns
    dat, rows = map(list, zip(
        *[(df[col].sp_values - df[col].fill_value,
          df[col].sp_index.to_int_index().indices)
          for col in columns]
    ))
    cols = [np.ones_like(a) * i for (i, a) in enumerate(dat)]
    datF, rowsF, colsF = (
      np.concatenate(dat),
      np.concatenate(rows),
      np.concatenate(cols)
    )
    arr = sparse.coo_matrix(
      (datF, (rowsF, colsF)),
      df.shape,
      dtype=np.float64
    )
    return arr.tocsc()

def getArrayRegion(self, data, img, axes=(0,1), order=1, **kwds):
    """Use the position of this ROI relative to an imageItem to pull a slice
    from an array."""

    imgPts = [self.mapToItem(img, h['item'].pos()) for h in self.handles]
    rgns = []
    for i in range(len(imgPts)-1):
        d = Point(imgPts[i+1] - imgPts[i])
        o = Point(imgPts[i])
        r = fn.affineSlice(
          data,
          shape=(int(d.length()),),
          vectors=[
            Point(d.norm())],
          origin=o,
```

```
            axes=axes,
            order=order,
            **kwds
          )
        rgns.append(r)

    return np.concatenate(rgns, axis=axes[0])
```

Alternative strategies for code summarization

In summary, we found that many of the code/docstring pairs we have found in our extracted corpus involve docstrings describing code that includes information that also relies on at least one form of additional information not present in the code. The current NMT model assumes that all of the information needed for summarizing is contained within the input code itself. Our lambda functions generated by the AutoMATES pipeline certainly fit this definition, and thus our code/docstring corpus is not a good model for our target generation task. Based on these lessons, we are considering two new approaches to improving our corpus:

1. We are exploring possible methods to incorporate more of the missing context into the code/docstring corpus. This is not an easy task, but if successful would have high payoff for extending `Sequence2Sequence` models to the much more challenging task of summarizing large code segments, as found more naturally in mature code bases.
2. We are also now exploring rule-based methods for generating function descriptions. In this approach, we aim to take advantage of the highly structured, simplistic, and relative self-contained nature of our generated lambda functions in order to programmatically summarize of each lambda functions, in such a way that rule-based summarizations may be composed to form larger descriptions of the original subroutines, with the option to collapse and expand summary details associated with code structure.

8.4 Acquiring the corpus and running the training scripts

The data necessary to replicate our experiments can be downloaded using the following commands (for *nix systems):

```
curl -O http://vision.cs.arizona.edu/adarsh/automates/code-summ-corpus.zip
```

Unzip the file to expose the contained folder `code-summ-corpus/`, and then set the `CODE_CORPUS` environment variable to point to the data folder:

```
export CODE_CORPUS="/path/to/code-summ-corpus"
```

The code summarization tools require the following Python modules: `numpy`, `torch`, `torchtext`, and `tqdm`. These can all easily be installed using `pip` or `conda`.

Running the CodeCrawler tool To run `CodeCrawler`, you will need to install `nltk` as well. The `CodeCrawler` tool can be run using this command (assuming you have cloned the automates Github repo):

```
python automates/code_summarization/code-crawler/main.py
```

Running the classification experiment scripts The classification experiments can be run with the following command:

```
python automates/code_summarization/src/main.py [-e <amt>] [-b <amt>] [-
c <name>] [-g] [-d] [-t]
```

The flags associated with this command are:

- `-e`: The number of epochs to train the model. Increasing the number of epochs will increase the training time, but result in higher accuracy.
- `-b`: The size of a batch for training and testing. Larger batch sizes decrease the training time for a model, but can decrease accuracy.
- `-c`: The classification corpus to use, either `rand_draw` or `challenge`
- `-g`: GPU availability. Include this flag if your system has an Nvidia GPU

- `-d`: Development set evaluation. Include this flag to evaluate the model on the development set.
- `-t`: Test set evaluation. Include this flag to evaluate the model on the test set.

In addition to running the code, our pre-computed results can be observed by running:

```
python automates/code_summarization/src/scripts/dataset_scores.py <score-file>
```

where `score-file` is one of the pickle files stored at `/path/to/code-summ-corpus/results/`.

Running the summary generation experiment The natural language summary generation experiment can be run with the following command:

```
python automates/code_summarization/src/generation.py [-e <amt>] [-b <amt> [-g] [-d] [-t]
```

All of the flags mentioned in the command above have the same definitions as the commands outlined in the above section dealing with our classification experiments.

Our generation results can also be verified using our pre-computed results with the following script:

```
python automates/code_summarization/src/utils/bleu.py <truth-file> <trans-file>
```

where `truth-file` and `trans-file` are text files located at stored at `/path/to/code-summ-corpus/results/`. Note that the two files should have the same prefix, the `truth-file` should end with `_truth.txt`, and `trans-file` should end with `_trans.txt`.

9 References

Hahn-Powell, G., Valenzuela-Escarcega, M. A., & Surdeanu, M. (2017). Swanson linking revisited: Accelerating literature-based discovery across domains using a conceptual influence graph. *Proceedings of ACL 2017, System Demonstrations*, 103-108.

Valenzuela-Escárcega, M. A., Hahn-Powell, G., & Surdeanu, M. (2016, January). Odin's Runes: A rule language for information extraction. In *10th International Conference on Language Resources and Evaluation, LREC 2016*. European Language Resources Association (ELRA).

Baird, H. S. (1993, October). Document image defect models and their uses. In *Proceedings of 2nd International Conference on Document Analysis and Recognition (ICDAR'93)* (pp. 62-67). IEEE.

Deng, Y., Kanervisto, A., Ling, J., & Rush, A. M. (2017, August). Image-to-markup generation with coarse-to-fine attention. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (pp. 980-989). JMLR. org.

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision* (pp. 2961-2969).

Krishnamurthy, J., Dasigi, P., & Gardner, M. (2017). Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing* (pp. 1516-1526).

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems (pp. 91-99).

Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2017). Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. *IEEE transactions on pattern analysis and machine intelligence*, *39*(4), 652-663.

Walter, I.A., Allen, R.G., Elliott, R., Jensen, M.E., Itenfisu, D., Mecham, B., Howell, T.A., Snyder, R., Brown, P., Echings, S. & Spofford, T. (2000). ASCE's standardized reference evapotranspiration equation. In Watershed management and operations management 2000 (pp. 1-11).

Wang, J., & Perez, L. (2017). The effectiveness of data augmentation in image classification using deep learning. *Convolutional Neural Networks Vis. Recognit*.

Wong, S. C., Gatt, A., Stamatescu, V., & McDonnell, M. D. (2016, November).

Understanding data augmentation for classification: when to warp?. In *2016 international conference on digital image computing: techniques and applications (DICTA)* (pp. 1-6). IEEE.