Automatous Source Code Generation for Inductive Programming

Erick Miller
emill@stanford.edu

Abstract

Inductive programming is a unique area of Artificial Intelligence focused on the task of Automatic Programming; it covers broad areas of Ai and software architecture to accomplish the creation of logical programs from incomplete specifications. Automatic Programming has been an elusive [0] dream since the founding days of Ai [1A] with many sub-fields emerging proposed solutions, including statistical optimization methods, evolution inspired methods, and grammar based methods. Inductive synthesis of finite-state automata started in the 1970's [1B] and many early innovative works in programming were fueled by the desire for Automatic Programming. Interpreted languages, the preprocessor, automatic compiler optimizations, object oriented programming, shared dynamic libraries and modern IDEs are all descendants of the desire to automate computer programming. In this regard we are already standing on the shoulders of giants. The audacious goal of this work is to explore "completing the loop" – to posit an intelligent system that, given a short description of input and desired output, can automatically author usable software functions in a high-level programming language.

1. Introduction

The model and algorithm proposed in this work is an early and experimental novel solution towards the Automatic Programming of computer software using Deep Learning. Advances in Deep Learning [16] have recently led to new models that achieve or surpass many state-of-the art results in the areas of automated object recognition, automated speech recognition, natural language processing and machine translation; but as far as we know, have not been recently applied to Automatic Programming [2] [3].

We propose a hybrid ensemble method consisting of a framework driven by several "expert" generative Deep Recurrent Neural Networks [9][10][11][12] trained to learn logical sequence predictions, combined with a generative combinatorial "trees and forests" method that distills the predictions output by the neural networks into Abstract Syntax Trees (AST). AST simultaneously represents the logical control flow graph while also disambiguating high-level language syntax. The Abstract Syntax Trees are then stochastically combined at compatible leaf and branch nodes using crossover and mutation with inspiration taken from Genetic Programming [4] and Random Forest [5].

2. Algorithm and Model Overview

At the onset of the generative process, the pre-trained neural networks are sampled to "imagine" and generate high-level source code predictions, given a short textual input description stating the desired function that should be produced. The neural network's sequence predictions are then passed through a feature decoding and minor grammar enforcing syntax repair process, and used to generate a variety of valid Abstract Syntax Trees. The Abstract Syntax Trees (ASTs) are stochastically combined into forests using crossover with "conform" mutations.

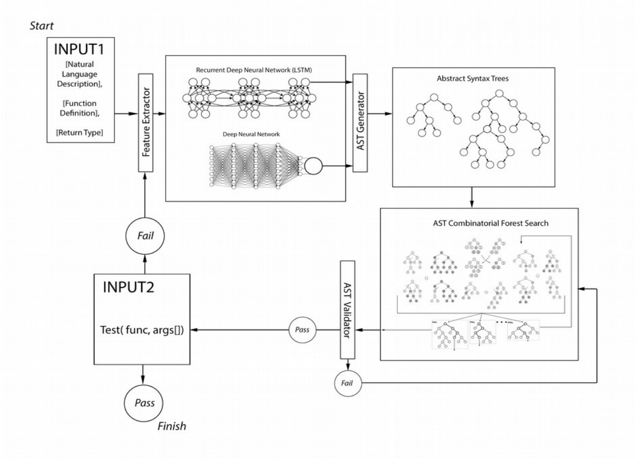


Figure 1a. Algorithm & Model (see Addendum 2[a])

Tree-health heuristics are applied, the forest is pruned, and then searched semi-exhaustively. Heuristics used include static analysis, function validation, compilation of object/machine code, and finally live logical execution testing using a generalized method that is a combination of a logic execution length timer (in milliseconds) and a sequence distance heuristic using a combination of the Levenshtein Distance [6] and fast *O(n)* Ratcliff-Obershelp [7] sequence matching method to test the generated output.

3. A Promising and Novel Method

The approach of building forests of trees, combining and mutating the trees in a variety of ways, then ranking tree health, and keeping the healthiest ones is directly analogous to techniques from Random Forest and Genetic Programming. Despite this, we believe the methods proposed in this work are novel as combined parts to form a whole; and have also yielded surprisingly encouraging experimental results in the automatic generation of small python [8] functions. The results of the experiments will be discussed in much more depth further; but the most notable result (see Addendum 6[a]) was able to successfully generate the square root function, generalizing logic that appears to be functionally equivalent to Newton's iterative method! This same model was also able to generalize new math functions such as $f(n)=n^2$ (see Addendum 7[a]).

4. Task Definition

The task of building the Automatic Programming system described in this paper can be concisely stated as follows:

> **Given a short textual description, build an intelligent system that can generate a working python function, using only the textual description as the system's input.**

For purposes of specificity, it is important to precisely describe the content of the textual description that the system will be expecting as it's input. This can be described as four independent (but related) statements:

```
Description    = 'very short description'

Definition     = 'def functionName( argument ):'

InputExample  = ['any', 'input', 'data']

CorrectAnswer = ['correct', 'output', 'data']
```

Given the above textual input, it is the role of the Automatic Programming system to generate a complete, usable python function using `Definition` that will take data of type `InputExample` as it's argument, and then transform `InputExample` using an unknown series of logical statements in python to perform computations that will generate output data that is exactly equal to `CorrectAnswer`.

Since the data described in this example is hypothetical and purely for explanatory purposes, there does not exist an interesting algorithm to transform this `InputExample` into this `CorrectAnswer`. Despite this fact, for descriptive completeness, a valid output of the system in this example would be:

```
def functionName( argument ):
        return ['correct', 'output', 'data']
```

While this example clearly explains the task, and is descriptive in regards to expected inputs and outputs, unfortunately the function generated contains no logic. A real-world running example would be much more useful, which follows.

4.1. Real World Running Example

While many strategies were attempted during the course of this research, and many variety of experiments were run, for the purpose of cohesive explanation, we will define a real-world running example that is included in our results, and will be referred to for the remainder of this paper.

The running example is a real experiment that our system ran to compute the square root of a number, thus will further be referred to as the Square Root Example and can be described using the previously stated four description statements as follows:

```
Description    = 'square root'

Definition     = 'def square_root( num ):'

InputExample   = 262144

CorrectAnswer  = 512
```

The real world results output by the system can be fully reviewed in Addendum 6a and Addendum 7a but for purposes of completeness, here is one example of a valid tested output function generated by the system:

```
def square_root(x):
    return math.sqrt(x)
```

4.2. Scope

It was immediately recognized that with this task, problems of tractability could quickly become an issue; specifically when all sequential variables are left unbounded. While the design of the algorithm is meant to help solve the problem of unbounded scope (while still generalizing), there was yet more to consider prior to starting the experiments.

Considering the potential that the input description given could be any unknown size or length, and contain any unknown number of words (presuming English language), and the potential that the data types and number of arguments could be any length, or any type – the problem of tractability became obvious, along with the need for limiting scope. See Addendum 1ª for further intuition, notes and informal analysis of this problem's tractability.

Fortunately, limiting the scope of the problem does not limit the interesting results of the experiments; and the scope limitations applied were simple in practice.

Intentional restriction of scope:

i. No function larger than 7 logic statements is mutated by the "combinatorial" forest portion of the algorithm (counting logic statements is achieved prior to compilation by counting variable assignments)

ii. No more than two words are used in the input description statement (for example "square root")

iii. The only functions considered are those that take one variable (of any type) as an input, and then return one variable (of any type) as an output

With these effective scope limitations in place, the remainder of the proposed methods remain unrestricted.

4.4. Dataset and Infrastructure

After some exploration, the initial strategy of using data acquired by GitHub was evaluated, and appeared to be the best possible approach for getting useful data. This of course required writing a robust and somewhat "tricky" time consuming (albeit simple) web scraper to avoid GitHub's many numerous rate limits and tricks to avoid getting the scraper's i.p. banned from their servers. The web scraping tool I wrote called ***scraper.py*** is included with the source code files. It is used to search GitHub for source code and then scrape all of the python files it found. Along those lines, the dataset is built is as follows:

The terms defined in `Description` which is defined as "square root" using our running Square Root Example, is sent into our GitHubScraper. GitHub is then searched for python files that contain the search terms. Before GitHub is searched, the GitHubScraper adds the terms "def" and "return" to the search terms. Then GitHub is searched only for python source files. This proved to be a highly effective way to gather a focused, yet relatively diverse set of source files guaranteed to include at least one function that might use the return statement, along with the search terms supplied by the user.

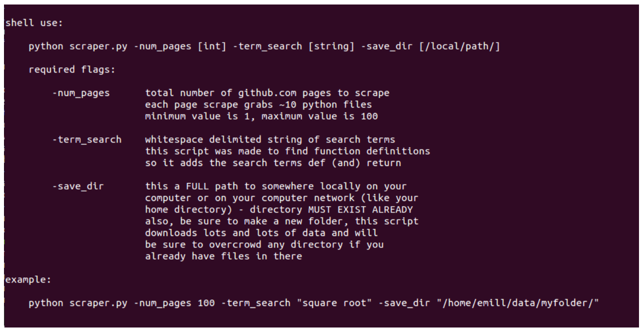
How to invoke and use GitHubScraper as a python Class:

```
import scraper
gs = GitHubScraper()
gs.scrapeGitHubForCode(searchQuery,
                                numPages,
                                saveDir )
```

The average source code scrape (after hitting the unavoidable GitHub limit of 1000 source files per query) results usually in well over ~1,000,000 raw tokens to parse through by the feature extractor, and in our running example for Square Root, resulted in extracting 559,887 sequential features used for training the Neural Network that produced the results in the Addendum. After feature extraction one single scrape results in a merged file size of approximately 1.2 megabytes, and over 4,000 unique expert focused variations of functions for training.

Both the scraper and the feature extractor take steps to minimize noise as well as reduce any unintended biasing of the model. For example, duplicate source files and duplicate function definitions are never included, and functions are extracted into a dataset that includes only Python functions. Classes, local variables, and all other random data outside top level python functions are handled elegantly by the feature extractor, converting Class methods into local functions in some cases, and discarding data in others.

One final note regarding infrastructure: the baseline algorithm ***baseline.py*** also uses the same GitHub scraper for implementing retrieval of baseline examples; to be explained in more depth later.



```
shell use:

    python scraper.py -num_pages [int] -term_search [string] -save_dir [/local/path/]

    required flags:

        -num_pages      total number of github.com pages to scrape
                        each page scrape grabs ~10 python files
                        minimum value is 1, maximum value is 100

        -term_search    whitespace delimited string of search terms
                        this script was made to find function definitions
                        so it adds the search terms def (and) return

        -save_dir       this a FULL path to somewhere locally on your
                        computer or on your computer network (like your
                        home directory) - directory MUST EXIST ALREADY
                        also, be sure to make a new folder, this script
                        downloads lots and lots of data and will
                        be sure to overcrowd any directory if you
                        already have files in there

example:

    python scraper.py -num_pages 100 -term_search "square root" -save_dir "/home/emill/data/myfolder/"
```

Figure 3a. Git Hub Scraper Shell Options

4.3. Datasets Explored / Considered

As described, the initial intuition for gathering the datasets used in these experiments was to utilize source code scraped from Git Hub, and train expert models that learn from source code already written by (hopefully) expert humans for a particular task. Despite this initial intuition, prior to writing a robust scraper for scraping GitHub several other more immediate and seemingly robust dataset options were downloaded and explored during the exploratory phase of this research.

First, a 38.8GB dump of all forum posts from Stack Overflow. See Addendum 3a for more notes regarding the Stack Overflow knowledge representation using EM and LDA clustering experimentation. This initial strategy of mapping knowledge representation to source code was ultimately discarded (despite it being a very promising and viable idea I'd like to explore further). There was just too much noise / ambiguity in the results of the experiments (and huge processing time); thus will not discussed further.

Another dataset considered was Python's own source repo (including all user installed packages) which on a Linux system is an incredibly simple task, with a short bash script to simply merge into one massive training file. The problem here was that, while this code is very well written, it is not an expert at any one task, thus complicating the process of feature extraction. Also, searching through this code-base proved less fruitful than expected as many functions are relatively low level and many make system modifications that could be dangerous for system stability; regardless of the ability to easily restrict permissions granted to the executing thread.

Finally, the source code dataset acquired by downloading, installing and searching through all available pip packages found by running the command **pip search *** was also considered and immediately discarded after seeing numerous questionable package names such as: "*crazy-ball*", "*artifacts*", and "*blackhole*", just to name a few.

4.5. Evaluation

An important part of the algorithm is the evaluation and scoring of the AST tree health after mutating tree variations and building a collection of trees (which we will refer to as a forest).

First, AST mutations that will not compile are immediately pruned from the forest. Since the AST are direct representations of the functional programming logic generated by the system, the scoring of the AST directly maps to scoring of the program code generated, as the algorithm applies transformations between these representations numerous times while evaluating the output.

While the most logical and natural evaluation metric is accuracy, running time is also an important factor. The following evaluation methods are used by function scoring method to evaluate, score and sort the AST trees. First, the Levenshtein distance as an approximation of error e (defined in Eq. 1a)

(Eq. 1a)

$$e_{tru,pred}(t,p)=\begin{cases} max(t,p) & if\ min(t,p)=0, \\ min\begin{cases} e_{tru,pred}(t-1,p)+1 \\ e_{tru,pred}(t,p-1)+1 \\ e_{tru,pred}(t-1,p-1)+1_{(tru_t \neq pred_p)} \end{cases} & else. \end{cases}$$

Next, Ratcliff-Obershelp similarity (defined in Eq. 1b)

(Eq. 1b)

$$e_{ro}(tru,pred)=\frac{LCS(tru,pred)\times 2}{|tru|\ +\ |pred|}$$

Where LCS is known as the Longest Common Subsequence problem, defined as the following recurrence (defined in Eq 1c)

(Eq. 1c)

$$LCS(t_i,p_j)=\begin{cases} 0 & if(i=0)\vee(j=0) \\ LCS(t_{i-1},p_{y-1}) & if\ t_i=p_j \\ longest\ (LCS(t_i,p_{y-1}),LCS(t_{i-1},p_y)) & if\ t_i\neq p_j \end{cases}$$

Finally, a measure of the evaluation heuristic is calculated using the sum of the Levenshtein distance with the normalized Ratcliff-Obershelp similarity such that: (defined in Eq 1e)

(Eq. 1e)

$$error\ =\ e_{tru,pred}(|tru|\ ,\ |pred|)\ +\ \left(1.0-\frac{e_{ro}(tru,pred)}{|tru|}\right)$$

Lastly, an approximation of the cpu clock cycles required to execute the function's logic is stored using python's built-in Timer mechanism with a carefully scoped call to python's literal **eval()** function to evaluate the current run-time result in milliseconds, which is also added to the Error heuristic for each function being evaluated. If the prior error was zero (prior to adding the execution time) then an additional bit is set to *True* on an associative array index, signaling this was a perfect function, so the forward passing results can still be sorted and prioritized by minimizing for the *fastest perfect* function.

It should also be noted the evaluation mechanism that runs **eval()** is designed to timeout after a certain number of seconds. Any AST that takes longer than this time (set to 5 seconds in these tests) is permanently discarded, and so are all mutated variations of that function. This effectively removes ASTs that contain flawed logic such as infinite recursion or infinite loops.

Examples of numerical output described by the equations in this section can be seen in the results shown in Addendum 5[a], 6[a], 7[a] , 7[b] , 7[c], 7[d] which is listed in column two of each Addendum page, and is labeled as **ERROR**.

5. Approach

Besides the advanced approach, which has been described thus far, there were also two other methods used during the evaluation of experiments: a baseline, and an oracle.

5.1. Baseline

The purpose of the baseline is to replicate a simple solution to this problem that, while not being perfect, already exists and is fast and simple to run.

The baseline algorithm in **baseline.py** used for these experiments is meant to replicate the manual human search process. It therefore simply runs a search on Git Hub, scrapes the first page's results, and randomly returns one of functions contained from the first page's search results. It is very decidedly intelligent than a human, but mirrors the manual process of randomly searching for code on the internet in an acceptably accurate manner.

When executed three times, the baseline algorithm for our Square Root experiment returned the following three random results.

Example 1a. Three baseline square root functions:

```
def square_root(s):
  i = 1
  j = s
  while (abs(j - i) > 0.001):
   print i, j
   j = (i + j) / 2.0
   i = s * 1.0 / j
  return i

def square_root(x):
   return np.sqrt(x)

def square_root(a, x):
    y = (x + a / x) / 2
    return y
```

As we can see above, unfortunately all three of our baseline results are deceptively not square root functions that would work for our defined problem, despite their name, as is often the case when randomly searching for source code on Git Hub – it almost never just works out-of-the-box. Thus, our baseline is acceptably modeling a real-world experience.

Problems with the three example baselines:

The first baseline example returns only approximate results.

The second baseline presumptuously appears to require the installation of `numpy` while using an abbreviated naming convention for `numpy` that would not even work out-of-the-box if `numpy` was installed and imported.

The third appears to intended as a utility function that needs to be called iteratively.

The first baseline is particularly interesting, though, not only because it is an approximate solution, but specifically because of it's near-similarity to several other square root functions found in the training data, as well as it's logical relationship to the function learned in Addendum 6[a].

5.2. Oracle

As mentioned, an oracle was also used to judge our results. The purpose of the oracle is to know the exact best case perfect answer. In the case of the Square Root running example, the oracle was simply:

```
math.sqrt( InputExample )
```

One of the best things about these experiments is that the oracle for all (and any) experiment is actually very simple to determine. Since the goal of our system is to generate python functions, we can actually run a massively large variety of experiments, always knowing an oracle, by simply choosing to run experiments for python scripts or functions that already exist and produce known results.

In other words, the oracle for learning any arbitrary preexisting function is simply the function itself! This is a very encouraging attribute of this problem that I'm hoping to exploit in a Reinforcement Learning framework for future work.

5.3. Advanced Method

As already introduced, the proposed model and algorithm is a hybrid ensemble method consisting of a framework driven by several pre-trained "expert" generative deep/recurrent neural networks trained on libraries of source code. The trained neural networks are sampled to "imagine" and generate data (for direct source code predictions) which are sent through a feature extraction and minor syntax repair process, and then used to form a variety of valid Abstract Syntax Trees. The Abstract Syntax Trees (AST) are stochastically combined into "forests" using crossover and evaluated in a semi-exhaustive manner using evaluation heuristics to sort and prune unhealthy trees.

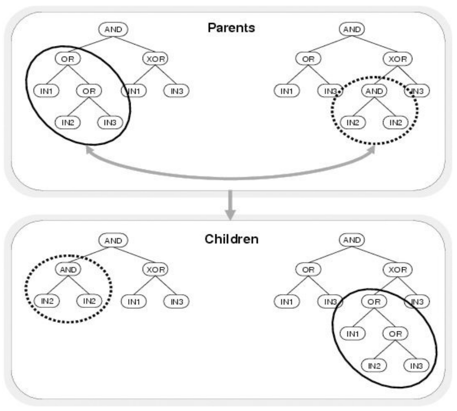


Figure 4a. Reference, Tree Crossover

The remaining healthy trees are transformed back into python source code, round-tripping them one final time as a checkpoint, and then passed into the compilation process, and executed using the user supplied data.

The output of the logic execution process is evaluated using the methods described in section: **4.5 Evaluation**, and then scores are updated and the trees are re-sorted based on their new scores – this process also inherently results in a pruning process of a variety of trees that contain logic flaws but trick the static analysis heuristics. If any tree scores a zero error, it is marked as a successful answer, and after evaluation is finished, is returned to the user.

See Addendum 8[a] for a representative example the trees explored using our running Square Root example.

The delightful and surprising result of this process, when neural networks are trained as experts on a domain specific code base, is a model and algorithm that generate a multitude of valid, exact functions that accurately reflect the chosen oracle, and can also generalize (within their domain knowledge) to predict new oracles.

5.4. Challenges, Unused Experiments

5.4.1 Challenges

There were many challenges encountered during this research, one of the most challenging was simply time! The following real-world challenges were anticipated early on, and were in fact directly encountered, and then overcome during the implementation of the methods described. If these challenges had not been solved to acceptable levels, the results accomplished here would not have been possible:

Scraping and cleaning sufficient amounts of stable open source software.

Effective feature extraction of the source code.

Use of additional models and algorithms to reason with this data.

Effectively constraining the complexity and search-space so as to generate usable general solutions within a reasonable execution time.

Generating source code candidates that can be safely compiled and are syntactically correct within the logical programming language constraints, and are also human readable and usable for practical software engineering purposes.

Creating a system that generalizes functional syntactically correct, logically correct source code.

Language features that posed specific challenges which were overcome in a variety of general ways during the feature extraction and/or generative prediction process:

- Untyped variables
- Dynamic variable type casting
- No return type in function definition
- Class methods
- Nested functions (functions inside of functions)
- Infinite Recursion and Infinite Loops

5.4.1 Unused Experiments

Numerous experiments were performed that resulted in less than desirable results, some of which have already discussed. For purposes of completeness, the following non-exhaustive list is supplied:

LDA knowledge clustering from Stack Overflow Post data (documented in Addendum 3[a])

Using Nervana's neon deep learning library with random sampling to numerous additional generative RNN models including both LSTM and GRU models trained on source code data data

Recurrent networks with 4, 5 and 6 layers including with and without dropout, as well as numerous ratios of test:training data during model training.

5.4.3 Specific Phenomena

To discuss specific phenomena, several key Python language features are discussed and analyzed.

Consider a list of Python's built-in complex keywords, functions and operators as primary differentiable language features that exist within a function definition:

```
python_keywords = ['and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del' 'elif', 'else', 'except', 'exec',
'finally', 'for', 'from', 'global'… 'vars', 'zip']
```

```
multi_char_operators = [ "==", ">=", "<=", "<>" ... "**" ]
```

Now, consider the following single character ASCII symbol representations as a means to compress the python logic sequence during an encoding process in the feature vector, prior to deep learning:

ü é â ä à å Ä Å É æ Æ ô ö ò û ù ÿ Ö º ┤ Á Â À © ╣ ║ ╗ ╝ ¢ ¥ ┐ └ ┴ ┬ ├ ─ ┼ ã Ã ╚ ╔ ╩ ╦ ╠ ═ ╬ ¤ ð Ê Ë È

As an example, mappings are such that encoding and decoding can easily be done using a dictionary such as:

```
keyword_map = {'and':u'Æ', 'as':u'É' ... 'class':u'Ã' }
```

Thus, a large bulk of Python 2.7's vocabulary (besides variables, logic flow and single-character operators) can be encoded, expressed and learned in ~100 dimensions of symbolic logic space.

There are several goals for feature extraction and learning based on the observed phenomena outlined. For one, we wish to clearly differentiate symbolic language features which represent logical flow, from things like language variables and comments, both which need to be differentiable as an important aspect for accurately learning sequence data.

The goal is to exploit these facts as much as possible to extract features which can be trained quickly, effectively and as accurately as possible for code prediction.

5.4.4 A Note On Specific Phenomena of RNN Model

One-Hot vectors are used in the RNN LSTM model and the learned language vocabulary is the vector space for each layer. Here is an example of a One-Hot, using python operators as the features for purposes of clear explanation. In this example, for simplicity, we show an eight dimensional vector space to represent the one-hot:

Operator	One-Hot
=	00000001
-	00000010
*	00000100
+	00001000
>	00010000
<	00100000
.	01000000
/	10000000

In practice, the actual vector space was much much larger, dependent on the size of the total language vocabulary used and the encoding methods used in the feature extractor.

5.4.5 A Note On The Specific Phenomena of Grammar

Python has a very well defined language definition; like most modern languages, Python uses Backus–Naur Form grammar (BNF). The BNF grammar definition is used by Python's compiler to enforce source code syntax and grammar constructs. Considering the language grammar is so clearly well defined, and can be directly validated through the compilation process (which is performed numerous times in our algorithm) it should be possible, given enough training data, for a model to learn all of the grammar rules perfectly. We will leave this for future work, but it is interesting phenomena to note.

6. Data and Experiments

After a variety of tests and model / algorithm refinements, the following real-world experiments were performed. Each experiment is shown here using the established conventions to describe the inputs given to the generative algorithm, as previously outlined in **Section 4 Task Definition**. Results listed for each experiment should be reviewed in greater detail in Addendum 5^{a}, 6^{a}, 7^{a} , 7^{b} , 7^{c}, 7^{d}

Addendum 5^{a}:

```
Description   =   'sort'

Definition    =  '"def sort( seq ):"'

InputExample = [ 'll', 'gg', 'aa', 'jj', 'cc', 'ee',
                 'hh', 'dd', 'p', 'o', 'm', 'bb',
                 'kk', 'ff', 'n', 'ii', 'z', 'qq' ]

CorrectAnswer = ['aa', 'bb', 'cc', 'dd', 'ee', 'ff',
                 'gg', 'hh', 'ii', 'jj', 'kk', 'll',
                  'm', 'n', 'o', 'p', 'qq', 'z']
```

Addendum 6^{a}, 7^{a}:

```
Description    = 'square root'

Definition='"def square_root( num ):"'

InputExample= 262144

CorrectAnswer = 512
```

The following three results contained in Addendum 7b 7c and 7d generalized their results using the square root training data with no additional model or algorithm tuning:

Addendum 7^{b}:

```
Definition = '"def squared( num ):"'

InputExample= 512

CorrectAnswer = 262144
```

Addendum 7^{c}:

```
Definition = '"def half( num ):"'

InputExample= 512

CorrectAnswer = 256
```

Addendum 7^{d}:

```
Definition = '"def double( num ):"'

InputExample= 512

CorrectAnswer = 1024
```

7. Analysis

7.1. Interpreting Results

To review and derive meaning from the results achieved, browse the Addendum data included. The first two most interesting and encouraging results accomplished are that first, in all cases the model was able to predict an answer that is equivalent to the Oracle and second the model was able to generalize to create new, related functions without needing new training data (ie the model for "Square Root" was used to generalize squared, divide by two, and a doubling function).

While this all seems almost unbelievably impressive at first, this behavior is certainly only possible due to two important attributes:

1. The scope limitations previously described were effectively enforced (disallowing us to generalize the current method to larger functions)
2. The "unreasonable effectiveness" of the deep recurrent lstm neural network was made even more effective through targeted feature extraction combined with highly focused "expert" training on the specific topic that is wished to be learned (for example "square root").

While on the surface this may appear to be nothing more than a simple search, it is very promising to see the generalization possible even in these early preliminary examples.

For the remainder of this section, we will use our running example of Square Root, since it is the most interesting result achieved. For additional results, please refer to the Addendum. The interpretation of results for the Square Root example should generalize to all other results currently possible with our system.

By far, the most impressive result can be seen in Addendum 6a – the neural network appears to have learned Newton's method for finding the square root of any number:

```
def square_root(a):
  x = (a / 2.0)
  epsilon = 0.001
  while True:
    print x
    y = ((x + (a / x)) / 2)
    if (abs((y - x)) < epsilon):
      break
    x = y
  return x
```

At first, my assumption was that this was due to over-fitting; but after searching through all of the training data, I realized it was definitely more of an interesting phenomena caused by a combination of many variations of the same functional logic, creating a statistical bias towards predicting these unique sequences – combined with the absolute magical properties of deep neural networks. My best explanation of these results, after spending much time analyzing the data and re-running several tests can be best explained by the multiple re-occurrence of several variations of the same, or a very similar variation of this algorithm:

```
def square_root(a):
  x = a / 2.0
  y = x
  while True:
    y = (x + a / x) / 2.0
    if abs(y - x) < sys.float_info.epsilon:
      break
    x = y
  return x

def square_root(a):
  x = 10
  while True:

    y = (x + a / x) / 2
    if abs(y - x) < 0.000001:
      break
    x = y
  return x

def square_root(a):
  x = a / 2.0
  epsilon = 0.001
  while True:
    print x
    y = (x + a / x) / 2
    if approx_equal(y, x, epsilon):
      break
    x = y
  return x

def approx_equal(a, b, limit):
  if abs(a - b) < limit:
    return True
```

Presumably this phenomenon was caused by human beings copy and pasting the function and then modifying it? Or perhaps just inherent logic contained within a function that can only contain a restricted number of combined logic sequences, which are represented enough in the data in order for the neural network to learn them? Now *that* would be really cool – and I think that's what has happened here based on observation of the data.

Several near variations existed in the training data, but there was nothing that represented an exact duplicate of this function line for line! There wasn't even a function that had two lines in a row that were exact duplicates.

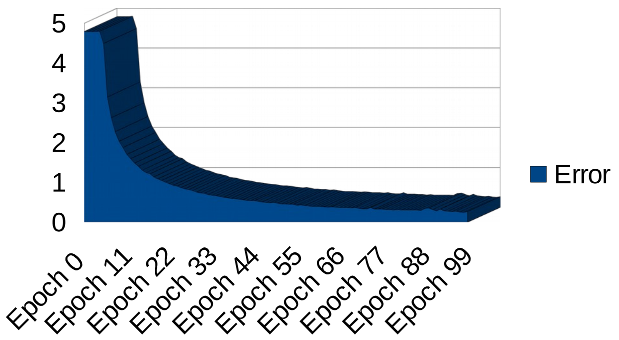
It appears the learning algorithm **generalized** syntax and logic sequences to generate a generalization of this complex function. *This was, in all honesty, a completely unexpected and delightful result.*

Based on the results reported elaborated here, as well as in great detail throughout this paper and further in the Addendum, I can say with some level of confidence that we can preliminarily conclude – yes, it appears it is possible for a Deep Recurrent Neural Network using Long Short Term memory gates to learn the sequence data needed for keyword prediction of logical source code – at least effectively enough for the output to be sent to additional search algorithms in order to achieve exciting and seemingly generalizable source code predictions.

7.2. Error Analysis

Since this is a multi-faceted system with several distinct stages, several distinct stages of error analysis were also performed.

First, test error was measured and attention was paid to avoid over fitting during the refinement, tuning, and development of the Deep Recurrent LSTM Neural Network model during the pre-training process. More detail can be seen in Addendum 4[a]



Next, Error measures of source code predictions at runtime can be estimated using the previously described Section 4.5 Evaluation – but first, an interesting error can be observed in the log files (supplied with the code and data sets as part of the project).

During the feature extraction used to decode the generative neural network data at runtime, a certain percentage of code is generated that is total garbage and can not pass a simple static analysis step. The way to identify the percentage of error is to notice the percentage of data skipped, as output by the log, during the feature extraction process:

```
Percent: [                     ] 0%
Percent: [##                    ] 3%
Percent: [###                    ] 7%
Percent: [#####                   ] 10%
Percent: [#######                  ] 13%
Percent: [########                  ] 17%
Percent: [##########                 ] 20%
Percent: [############                ] 23%
Percent: [#############                ] 27%
Percent: [###############               ] 30%
Percent: [#################              ] 33%
Percent: [##################              ] 37%
Percent: [####################             ] 40%
Percent: [######################            ] 43%
Percent: [#######################            ] 47%
Percent: [#########################           ] 50%
Percent: [###########################          ] 53%
Percent: [############################          ] 57%
Percent: [##############################         ] 60%
Percent: [################################        ] 63%
Percent: [#################################        ] 67%
Percent: [###################################       ] 70%
Percent: [#####################################      ] 73%
Percent: [######################################      ] 77%
Percent: [########################################     ] 80%
Percent: [##########################################    ] 83%
Percent: [###########################################    ] 87%
Percent: [#############################################   ] 90%
Percent: [###############################################  ] 93%
Percent: [################################################ ] 97%
```

This is ascii data is an interesting artifact of the feature extraction progress reporting mechanism after redirecting standard output into a text file for documentation purposes – regardless, it is a very interesting error rate to notice. This is theoretically an error that represents total noise in the model, since it is outputting data that was previously successfully learned using the feature extractor but when sampled, can not be processed by the feature extractor. This error range is reasonably low, ranging between 2% to 4% depending on the data and samples requested – but could be an area to explore as a window to improving overall predictions. *Finally*, Error measures using the Section 4.5 Evaluation heuristics are analyzed. Since approximate execution time is inherently included in these scores, it is important to understand the meaning of these approximate Errors scores – these are scores representing the errors of syntactically valid functions that contain some logic flaw causing them to return imperfect data.

Addendum 5[a]	**Max Sampled Error: ~110**
Addendum 6[a], 7[a]	**Max Sampled Error: ~15**
Addendum 7[b]	**Max Sampled Error: ~20**
Addendum 7[c]	**Max Sampled Error: ~12**
Addendum 7[d]	**Max Sampled Error: ~15**

It's interesting to notice that the maximum sampled error decreases with the size complexity of the required output. This makes sense, even though the error metrics can not be directly compared as they are not normalized – the lower error rates represent shorter sequence data being compared, and in practice, with our data used, does represent less actual error despite the lack of normalization.

Also note that this is an estimation of error and not exact error – thus meaning that there is some measure of error contained in the method used to measure error. The primary problem with the current sequence edit-distance and sequence matching based error metric is that it serializes all data into string prior to comparison. This is robust to many cases that occur in real world circumstances, such as comparison of sequential data of different data type (thus effectively handling dynamic variable typing and variable type mutations). This method is flawed in the very simple example of measuring the distance between a two floating point numbers such as comparing the number `1.0` with `1.0001563477677462437631`. While this is a very small true distance, the error measured by the current method results in an error approximately greater than 20! This would definitely result in the wrong optimization results in several circumstances – but of course this problem is easily correctable; yet it is worth mentioning because it currently exists in this experimental, imperfect implementation of our proposed method.

8. Literature and Attributions Review

While much interesting literature on sequence learning, automated programming, modern software architectures, and a variety of other topics in artificial intelligence and other fields that create software designed to create software, nothing was found that solves the specific problem as proposed – i.e. using Deep Learning for Inductive Programming.

There are many pieces of literature that could be discussed and elaborated upon in this section (see section: **References**), there are three primary areas worth mentioning, or further discussing in detail. One area is Genetic Programming, the next is Inductive Programming, and the last are two specific papers [11] and [13] along with a blog post and open source library that are definitely of high importance, and are discussed last.

First, while researching methods, many papers on the topics of Genetic Programming were reviewed (far too many to cite, so Koza's pioneering work is cited for this domain [4]). As mentioned numerous times, Genetic Programming was a source of reference for the project and implementing more features from Genetic Programming paradigm is something planned for future work; it is worth mentioning in this section but probably worth discussing in further detail here.

Second, a large number of papers on Inductive Programming were reviewed (the best one is a survey and is cited [2]). It is interested in to note that in [2] the author contrasts Inductive Programming with Neural Networks models in a competing context, rather than considering them as a generative method for Inductive Programming. This is most likely due to the major strides in progress that Neural Network models have made recently, as this paper was from 2010 – regardless, this was an interesting thing to note, especially since there is literature from the Inductive Programming sub-field [3] “Learning Logic Programs with Neural Networks” from 2001, where the authors leave the first-order theory refinement using neural networks as an open problem.

Lastly, and most importantly – there were two papers that were published approximately within the last year that were particularly inspiring for this work:

> Zaremba, Wojciech, and Ilya Sutskever. "Learning to execute." *arXiv preprint arXiv:1410.4615* (2014).

The learning to execute paper was a very large motivator for approaching this problem. The LSTM neural network model used in these experiments is exactly the same model as was used in the learning to execute paper. The primary difference between the learning to execute approach, and our approach, is that Learning to execute tries to solve the problem of predicting the result of executing a function, given sequences of logic – in other words, it is trying to predict the results of that compiled execution will generate, given source code. Where as our approach trains a model to predict the source code, this paper trains a model to predict the source code's compiled results. Of course the two have much in common, and could be looked at interchangeably, the utility of the results differs dramatically. Further, our model and process includes the additional AST mutation process as described whereas Learning to execute attempts to solve the problem with only Deep Learning.

It has been widely speculated and discussed in recent literature and within the deep learning community (and is informally accepted as true) that a finite-sized RNNs is Turing complete – despite there also being an apparent lack of formal proof that this is true. In fact this idea was one of motivations behind the research that has been presented in this paper. The following literature was very exciting in this regard:

> Graves, Alex, Greg Wayne, and Ivo Danihelka. "Neural Turing Machines." *arXiv preprint arXiv:1410.5401* (2014).

Further, there is an incredibly great blog post by Andrej Karpathy titled “The Unreasonable Effectiveness of Recurrent Neural Networks” where he uses the Linux kernel source (amongst many other examples) to train an LSTM:

> Karpathy, Andrej. "The Unreasonable Effectiveness of Recurrent Neural Networks." *Hacker's Guide to Neural Networks*. N.p., 21 May (2015). Web.

Andrej Karpathy deserves extra mention here, because not only was his blog post excellent, his lua / torch based LSTM RNN library produced the best results in my experiments, and his library was ultimately used to build, train and create the generative neural network used in this paper, which implemented the RNN model used in the Learning to execute paper. If you see Andrej, tell him thank you for his excellent open source contributions!

The following paper has not yet been published but it looks incredibly interesting and valid – I discovered it while working on these experiments, and have not yet had a chance to fully read and digest it but am planning to as soon as possible:

[15] Zaremba, Wojciech, and Ilya Sutskever. "Reinforcement Learning Neural Turing Machines." *arXiv preprint arXiv:1505.00521* (2015).

9. Limitations and Future Work

Despite these isolated successful results, it is the strong feeling of the author that the approaches proposed herein, albeit powerful and exciting, are in their most infant form; and there is yet much work to be done to explore, experiment and develop a fully realized system with this approach

The current implementation of the algorithm is lacking several planned capabilities (such as ϵ-greedy search using source code created from random generative BNF grammar) – these capabilities were initially excluded during practical implementation for the purposes of controlled experimentation; and while this capability would still be a great feature to add, in the experiments presented here, random search outside of the knowledge learned by the deep neural network does not appear to be needed! Only the leaves and branches the functions predicted by the neural network were used to create the AST forest. There is much refinement that could be added to this part of the model, including modeling the steps as a Markov Decision Process for Reinforcement Learning of generalizing successful code mutation decisions to further reduce the number of mutations created in the tree to forest process.

It would also be desirable to experiment with extending the forest model with a full Genetic Programming implementation that can run as a stop-gap mechanism if no solution is found, similar to the idea of using an epsilon greedy search method to introduce random branches into the search process when needed.

Finally, the most important planned work includes exploring more advanced RNN models such as attention based deep RNN models that are currently achieving state-of-the-art in Machine Translation, and apply them to the sequence prediction problem for more robust code prediction. Ultimately, the goal is to build the entire framework as a Markov Decision Process and use Reinforcement Learning with a hybrid of Deep Q Learning and the Tree / Forest and Genetic Programming techniques already discussed. Once that is stable, the plan is to attempt to train a very large ensemble of fuzzy experts that can accurately learn all known available Python functions, language grammar, and syntax as the end game. That would be a truly incredible accomplishment.

10. References

[0] Balzer, Robert. "Why haven't we automated programming." *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010.

[1A] Barstow, David R. "An experiment in knowledge-based automatic programming." *Artificial Intelligence* 12.2 (1979): 73-119.

[1B] Summers, Phillip D. "A methodology for LISP program construction from examples." *Journal of the ACM (JACM)* 24.1 (1977): 161-175.

[2] Kitzelmann, Emanuel. "Inductive programming: A survey of program synthesis techniques." *Approaches and Applications of Inductive Programming*. Springer Berlin Heidelberg, 2010. 50-73.

[3] Rodrigo Basilio Gerson Zaverucha Valmir C. Barbosa, “Learning Logic Programs with Neural Networks” *Inductive Logic Programming, 11th International Conference*, ILP (2001)

[4] Koza, John R. *Genetic programming II: automatic discovery of reusable programs*. MIT press, (1994)

[5] Liaw, Andy, and Matthew Wiener. "Classification and regression by randomForest." *R news* 2.3 (2002): 18-22.

[6] Okuda, Teruo, Eiichi Tanaka, and Tamotsu Kasai. "A method for the correction of garbled words based on the Levenshtein metric." *Computers, IEEE Transactions on* 100.2 (1976)

[7] Ratcliff, John W., and David E. Metzener. "Pattern-matching-the gestalt approach." *Dr Dobbs Journal* 13.7 (1988): 46.

[8] Oliphant, Travis E. "Python for scientific computing." *Computing in Science & Engineering* 9.3 (2007): 10-20.

[9] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.

[10] Graves, Alex. "Generating sequences with recurrent neural networks." *arXiv preprint arXiv:1308.0850* (2013).

[11] Zaremba, Wojciech, and Ilya Sutskever. "Learning to execute." *arXiv preprint arXiv:1410.4615* (2014).

[12] Karpathy, Andrej. "The Unreasonable Effectiveness of Recurrent Neural Networks." *Hacker's Guide to Neural Networks*. N.p., 21 May (2015). Web.

[13] Graves, Alex, Greg Wayne, and Ivo Danihelka. "Neural Turing Machines." *arXiv preprint arXiv:1410.5401* (2014).

[14] Kaiser, Łukasz, and Ilya Sutskever. "Neural GPUs learn algorithms." *arXiv preprint arXiv:1511.08228* (2015).

[15] Zaremba, Wojciech, and Ilya Sutskever. "Reinforcement Learning Neural Turing Machines." *arXiv preprint arXiv:1505.00521* (2015).

[16] Hinton, Geoffrey E., Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets." *Neural computation* 18.7 (2006): 1527-1554.

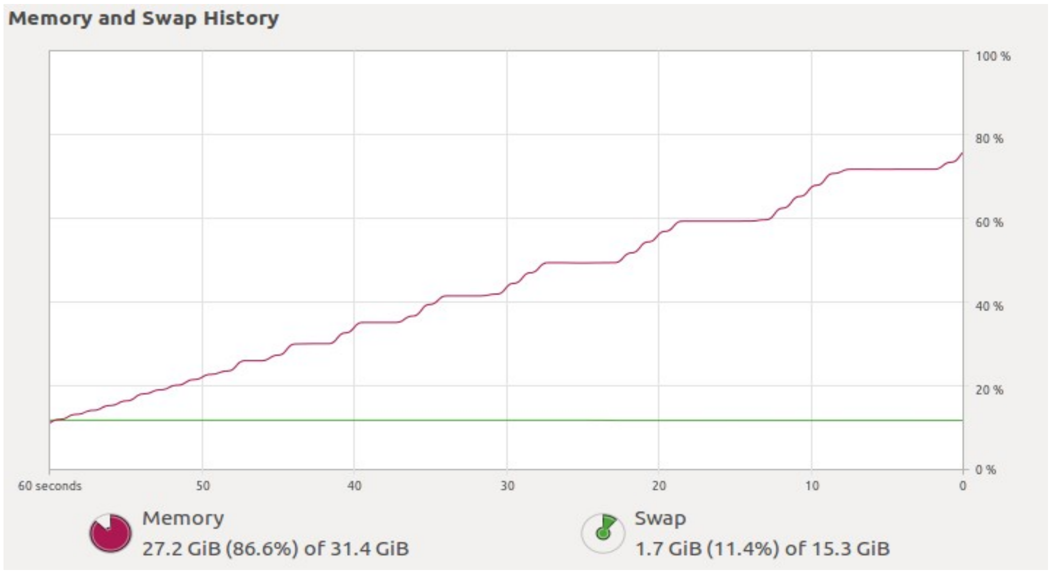
ADDENDUM 1[a]
Notes on:
Computational Tractability for a Theoretically NP Hard Problem

The general problem of finding an arbitrarily large function in an arbitrarily large sequential symbolic space with little to no knowledge of which operations to perform is undoubtedly an NP hard problem. If the symbols used to build the function are restricted to some known dimensionality, this reduces the problem to an knowable NP problem as this can be veritably computed in $O(n^s)$ time complexity where N are the total number of operations the function performs and s is the total number of all arbitrary operations that can be performed sequentially (*i.e.* total number of all possible sequential logic combinations existing, such as: variable assignments, condition statements, function calls, etc).

The algorithm implemented herein, which was used to search and find the functions as noted in the results (sort, square root, etc) had the space constrained in a variety of ways. First, it was limited through focused learning and training with a neural network, trained on a data set that had a very high probability of containing an answer based on rudimentary keyword search of the user's intent. Second, the algorithm used a variety of semi-exhaustive but not fully complete random / shuffled permutations and restricted Cartesian product combinations to find (and eliminate) variable and function combinations (effectively limiting k) as well as added heuristics for scoring functions so that functions with less error are searched first, all to make finding a solution tractable.

Without employing such methods, an exhaustive unbounded search of logic sequences even within the constraints of 11 logical sequences results in a worst case brute force search expansion of $O(n^n)$ where n is the number of logical operations expanded to 11^{11} combinations; over 285.3 billion sequential search operations. The resulting computational resources required for this exhaustive naive approach appear to be sub-quadratic in a real-world test case as can be seen below, yet still uses an unacceptable amount of compute resources. This experiment was killed just before physical memory limits of 32GB hit, at peak CPU usage (64 bit Linux, 8 processors @ 4.00Ghz) after only about ~1 minute of computation (as shown in diagram below, a real screen capture of computational resources utilized while running this first test).

Thus, an approach to circumvent and begin to solve this issue of intractability was devised herein.

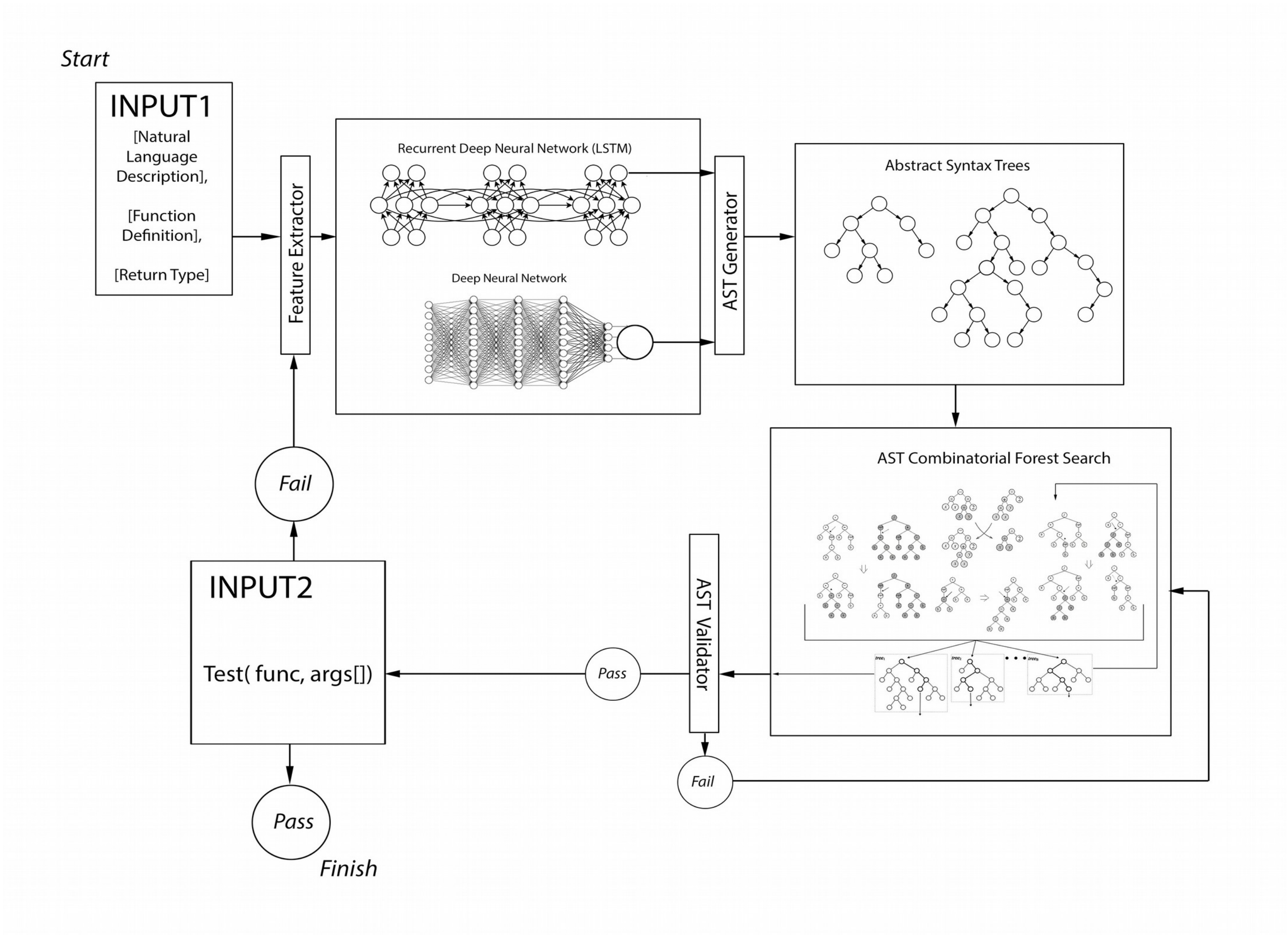


ADDENDUM 2[a]

Algorithm and Model Overview (larger view format)

The pre-trained Neural Networks are serving a dual purpose: both as intelligent symbolic logic sequence predictors, as well as effectively limiting the dimensionality of the AST's potentially otherwise very large unknown possible logical tree powerset search space.

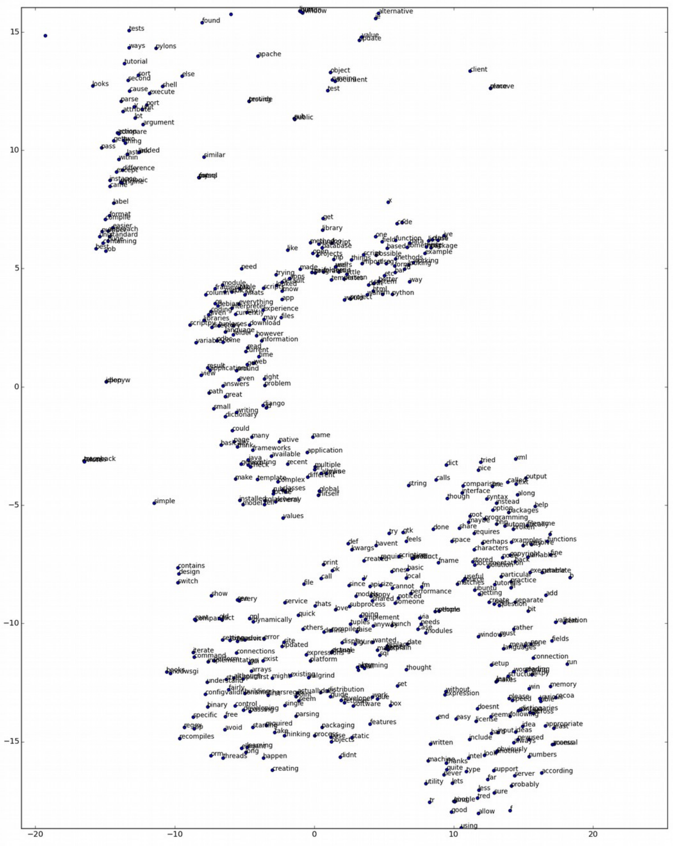
See diagram below for overview of the proposed run-time (after pre-training) generative process:



ADDENDUM 3[a]

Exploration: Stack Overflow for "Expert Code Topic" Knowledge Clusters Using EM based clustering with Latent Dirichlet Allocation

An initial idea to build the learning model from data was to map clusters of knowledge onto clusters of code and then use those mapped clusters to train a fuzzy ensemble of RNNs. Thus, LDA clustering of domain expert knowledge was explored for potential identification of concentrated expert topics / domain knowledge that could be used to search, scrape and train clustered ensembles expert "coder" generative deep neural networks. I was looking for a way to automatically map the vast knowledge of the Stack Overflow domain to raw source code. The S.O. domain data (+38.8GB of posts) was parsed (to remove xml tags and sematic stop words) and was then tokenized and statistically explored. It proved quite interesting but too noisey and very time consuming. *Below, a visualization of a batch of semantic knowledge clusters I trained can be seen (**zoom in**):*



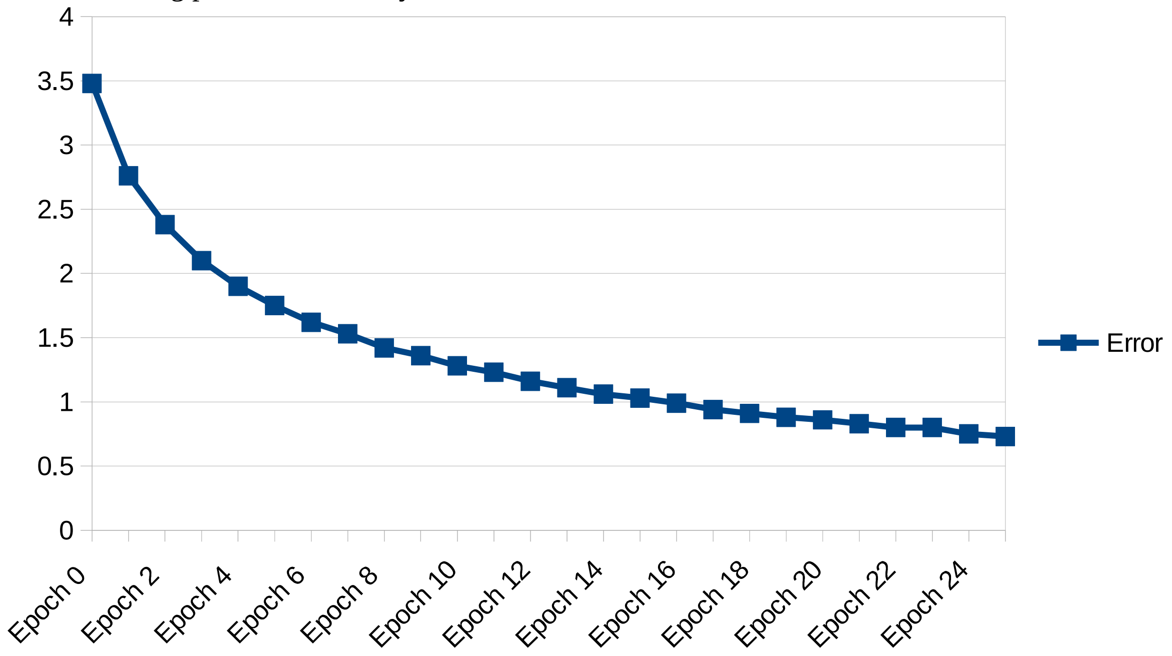
Note on above image: I used python and the **gensim** library with **sent2vec** and **doc2vec** along with **nltk**, **sklearn** and **matplotlib** to produce the clustering and visualization. After building a custom corpus and training the LDA model, I followed suggestion in the nltk library to use PCA first to reduce the multidimensional clusters to ~50 dimensions, then used TSNE to further reduce it to 2 dimensions before then showing the scatter plot as plt data.

Conclusion: While this exploration process proved very informative, the semantic data space was very huge, and quite noisey and thus implementation of an automatically trained ensemble of deep domain expert generative neural networks derived from this data will be left for exploration in future work once the knowledge space to domain space mapping is more well understood. I do believe there is a viable solution to be found here.

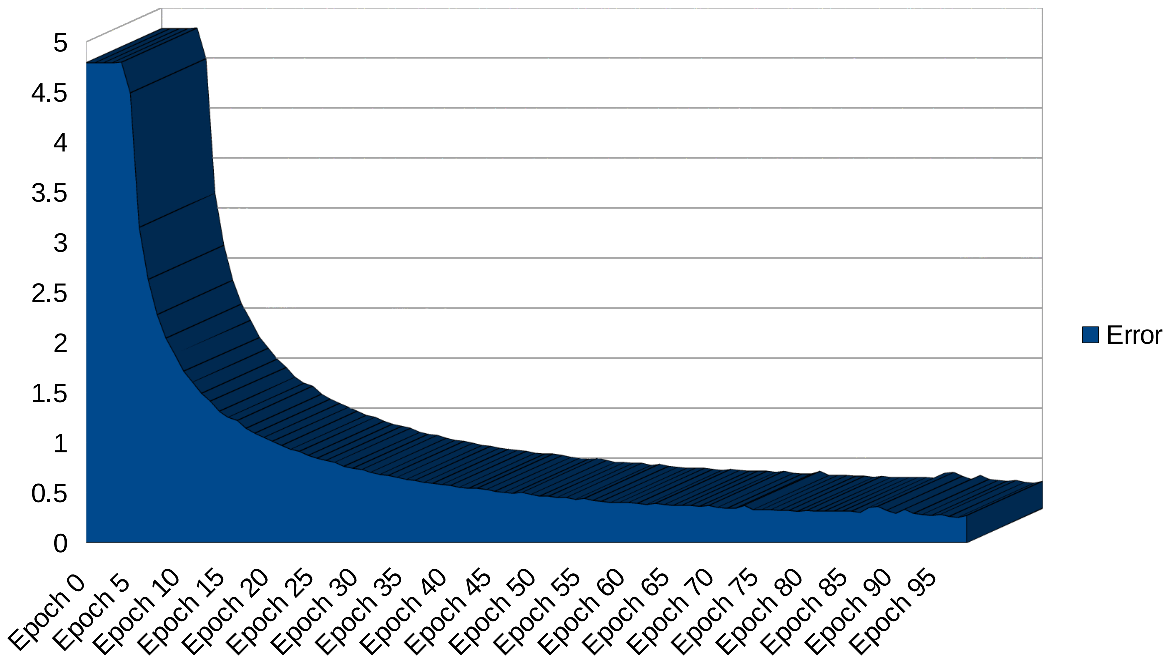
ADDENDUM 4ª

Training: Deep LSTM Neural Network Test Prediction Error per Training Epoch
(using 10% of training data as test data)

Prior to tuning parameters and layers of RNN model:



After experimentation, adding more data, tuning parameters of the model & increasing number of layers:
Final acceptable test error of deep recurrent models fell between ~0.2 and ~0.4 for different datasets



ADDENDUM 5ª Results: sort (part1)

Successful
(0 error, passed user supplied test data)
Oracle= `[].sort()`
Interesting *valid functions constructed successfully* performing the sort operation

```
def sort(a):
  return sorted(a)

def sort(alist):
  return sorted(alist)

def sort_by_name(my_class):
  a = sorted(my_class)
  return a

def __str__(sequence):
  return sorted(sequence)

def toSort(alist):
  return sorted(alist)

def sort_by_mark(my_class):
  return sorted(my_class)

def sort_files(seq):
  return sorted(seq)

def sort(array):
  '\n    Argument ant listTon
sorted countting ore'
  return sorted(array)
```

Unsuccessful
(error >0, failed user supplied test data)

Some interesting syntactically valid (but unsuccessful / logically flawed) functions constructed

```
def sort(lod):
  return sorted(lod, key=(lambda x: lod[1]),
reverse=True)
```
[AN ERROR OF: 40.9223300971]

```
def bubble_sort(sorted_list):
  for i in range(1, len(sorted_list)):
    sorted_list = sorted_list
    return sorted_list
```
[AN ERROR OF: 40.9223300971]

```
def sort_by_name(l):
  return sorted(l, reverse=True)
```
[AN ERROR OF: 44.640776699]

```
def pop(self):
  return self.pop()
```
[AN ERROR OF: 108.252336449]

```
def __repr__(self):
  return 'Sorted'
```
[AN ERROR OF: 109.459459459]

```
def merge(right):
  SortedList = []
  (-1)
  if (len(right) <= 1):
    return i[0]
```
[AN ERROR OF: 110.626168224]

```
def sort_by_name(k):
  return
```
[AN ERROR OF: 110.626168224]

ADDENDUM 5[b] Results: sort (part 2)

Examples of non-mutated (and) discarded functions

Examples of valid functions the algorithm constructs, validates, then parses, forms into an Abstract Syntax Tree, deconstruct into independent variables, and then recognizes the presence of "long" logic sequences (a variable length currently set to no greater than 7) during the combination process, and skips combining / mutating for these functions in order to achieve tractability. These long logic sequence functions are not discarded entirely, just not combined:

Example one contains multiple functions inside of functions (and logic errors):

```
def sort_by_mark(my_class):
  alist = sorted(items, key=(lambda l: lea((lambda : x))))

  def sort(self, sorted_items):
    return sorted_items.heappop()
```

Example two, contains loops with nested conditional branching (and logic errors):

```
def insertionSort(array):
  for i in range(1, right):
    if ((len(left) > 0) and (len(right) > 0)):
      if (left[i] < right[0]):
        sorted_stack.append(c)
      else:
        return False
```

ADDENDUM 6ª Results: square root (part 1) *(***wow!)*

Here's one of the coolest examples of a valid formed function (with low error) that failed the exactness of the final test, but resulted in a truly surprising result: ***The deep recurrent lstm neural network appears to have learned Newton's method for finding the square root of any number!***

```
def square_root(a):
  x = (a / 2.0)
  epsilon = 0.001
  while True:
    print x
    y = ((x + (a / x)) / 2)
    if (abs((y - x)) < epsilon):
      break
    x = y
  return x
```

[AN ERROR OF: 21.0001101494]

When this function is tested, it actually prints it's results along the way, to my *astonishment*:

```
square_root( 262144 )
65537.0
32770.4999695
16389.2496796
8202.6222773
4117.29041888
2090.47973814
1107.93935264
672.272180594
531.104741066
512.343615019
512.000115227
512.0001152266542
```

Note: *At first, my assumption was that this was rote learned; but after searching through all of the training data, while several near variations existed, it wasn't possible to find an exact duplicate of this function line for line! It appears the learning algorithm **generalized** syntax and logic sequences here!*

Newton's method?! The above code appears to be something equivalent to or very similar to Netwton's iterative method for solving the square root of any number via recurrence. While the learned function is not recursive, it's logic is equivalent to a recurrent function. *Wolfram Alpha* defines Newton's iteration as an application of *Newton's method* using recurrence that converges quadratically as $\lim_{k \to \infty} x_k$ and is defined as follows:

$$x_k + 1 = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right)$$

This very closely mirrors the function learned by the neural network!

ADDENDUM 7ª Results*: square root (part 2)

Successful !
(0 error, passed user supplied test data)
Oracle = `math.sqrt`
Interesting valid functions constructed successfully performing square root.
**** Read the doc-string of that last one! The Ai is trying to tell us something very deep :-)***

```
def cube_root_improve(a):
    return math.sqrt(a)

def square_root(x):
    return math.sqrt(x)

def root_mean_square(a):
    return math.sqrt(a)

def find_sqrt(x):
    "\n    Avime square root is the positive
integers, whine float an the square root is to
be computed.\n    Returns:\n        The square
root of x.\n    '''\n    assert x >= 0, 'x must
be non-negative, not' + str(x)\n    assert
epsilon > 0, 'epsilon must be positive, not' +
str(epsilon)\n    low = 0\n    high = max(x,
1.0)\n    guess = (low + high) / 2.0\n    ctr =
1\n    while abs(guess ** 2 - x) > epsilon and
ctr <= 100:\n        if guess ** 2 < x:\n
low = guess\n        else:\n            high =
guess\n        guess = (low + high) / 2.0\n
ctr += 1\n    assert ctr <= 100, ''\n    print
'The square root excain boof tepsicance the
square root of the first matrinuge hen
eloghacis:\n        The square root of x.\n
'''\n    assert x >= 0, 'x must be non-negative,
not' + str(x)\n    assert epsilon > 0, 'epsilon
must be positive, not' + str(epsilon)\n    low =
0\n    high = max(x, 1.0)\n    guess = (low +
high) / 2.0\n    ctr = 1\n    while abs(guess **
2 - x) > epsilon and ctr <= 100:\n        if
guess ** 2 < x:\n            low = guess\n
else:\n            high = guess\n        guess =
(low + high) / 2.0\n        ctr += 1\n    assert
ctr <= 100, 'Iteration count exceeded'\n
print 'Bi method format.\n    >>> from
pyromaths.classes.SquareRoot import SquareRoot\n
>>> SquareRoot([5, 8], [1,
45]).EstReductible()\n    False\n    >>>
SquareRoot([5, 8], [1, 7]).EstDecomposable()\n
False\n    :rtype: int\n    "
    return math.sqrt(x)
```

Unsuccessful
(error >0, failed user supplied test data)

Some interesting syntactically valid (but unsuccessful / logically flawed) functions constructed. Look at the one that raises an exception (it is valid logic!):

```
def subtract(a):
    return (a - a)
[AN ERROR OF:  13.0000779629 ]


def dist(s):
    x = (s ** s)
    return 0
[AN ERROR OF:  13.0116910934 ]


def square_root(a):
    x = (True / 2.0)
    while a:
        return
[AN ERROR OF:  14.0000739098 ]


def is_square(n):
    root = int((n ** 0.5))
    return (n == int(root))
[AN ERROR OF:  15.0001130104 ]


def fib(n):
    if (n < 0):
        raise ValueError('square root not
defined for negative numbers')
    n = int(n)
    if (n == 0):
        return 1
[AN ERROR OF:  14.0001008511 ]


def convergents(sequence):
    if int(True):
        return True
[AN ERROR OF:  14.0000879765 ]
```

***See all results that were echo'ed in the shell in the file supplied with code:**
square_root_ouput_log.txt

ADDENDUM 7^{b} Results*: squared *(generalized from square root training data!)*

Successful

(0 error, passed user supplied test data)

Oracle = `**2`

All valid functions successfully generalizing from square root training data to create “squared” functions!

Again, cool yet strange doc-string!

```
def multiply(a):
    return (a * a)
```

```
def df(x):
    return (x ** 2)
```

```
def dor_(x):
    return (x ** 2)
```

```
def cubeRoot(n):
    'Determines if n is arrayyyicce
digital number'
    return (n * n)
```

Unsuccessful

(error >0, failed user supplied test data)

Some interesting syntactically valid (but unsuccessful / logically flawed) functions constructed

```
def cube_root_iter(guess):
    return average(guess, (guess - guess))
[AN ERROR OF:  11.3637112271 ]
```

```
def improve(guess):
    return average(guess, (float(guess) /
guess))
[AN ERROR OF:  11.3637133728 ]
```

```
def square(number):
    return (number ** 1)
[AN ERROR OF:  12.7778776752 ]
```

```
def isSquare(n):
    root = int(math.sqrt(n))
    if ((root * root) == n):
        return True
[AN ERROR OF:  16.0000619888 ]
```

```
def square_root(x):
    return math.sqrt(x)
[AN ERROR OF:  20.6667386691 ]
```

***See all results that were echo'ed in the shell in the file supplied with code:**
squared_ouput_log.txt

ADDENDUM 7^c Results*: double ***(generalized from square root training data!)***

Successful
(0 error, passed user supplied test data)
***Oracle* = *2**

This time three functions were created. Interestingly they all perform the add operation and none multiply by two, which I had (arbitrarily) designateed as the *Oracle*.

Again, this was generalized from the square root training data!

```
def add(x):
    return (x + x)


def df(x):
    return (x + x)


def multiply(x):
    return (x + x)
```

Unsuccessful
(error >0, failed user supplied test data)

Again, many very interesting syntactically valid (logically flawed) but really intriguing functions constructed:

```
def triangle_rms(a):
    '\n    Returns L2 norm of the given matrix (while
petate-1.\n    >>> square_root = int(4 ** 2\n    '
    if (a < 0):
        raise ValueError('square root not defined
for negative numbers')
    r = int(a)
    return a
[AN ERROR OF:  7.28584708486 ]
```

```
def in_thas_inverse(can):
    for can in xrange(1, 1001):
        can += 1
    return (can ** 0)
[AN ERROR OF:  9.00008010864 ]
```

```
def iterate(x):
    return average(x, (float(x) / x))
[AN ERROR OF:  12.7778478728 ]
```

***See all results that were echo'ed in the shell in the file supplied with code:**
double_ouput_log.txt

ADDENDUM 7^{d} Results*: half *(generalized from square root training data!)*

Successful

(0 error, passed user supplied test data)

Oracle = `/2`

This time only two functions were created, *(clearly mutations of the same imagined function)* but again, this was generalized from the square root training data!

Syntax is a bit unusual but is valid and is also equivalent to the oracle!

```
def newpondix(y):
    return ((+ y) / 2)
```

```
def newpondix(x):
    return ((+ x) / 2)
```

Unsuccessful

(error >0, failed user supplied test data)

Again, many very interesting syntactically valid (logically flawed) but really intriguing functions constructed:

```
def add(a):
    return (+ a)
[AN ERROR OF:  9.66671983401 ]
```

```
def half(True):
    if is_prime(True):
        return True
[AN ERROR OF:  14.0000519753 ]
```

```
def is_prime(n):
    'Finds a primitian une prime or 
close nimger\n    :param errors: a 
number '
    if (n == 0):
        x = (n + 1.5)
    return (+ 1)
[AN ERROR OF:  13.0000519753 ]
```

```
def squareRoot(file):
    return (None is None)
[AN ERROR OF:  14.0000519753 ]
```

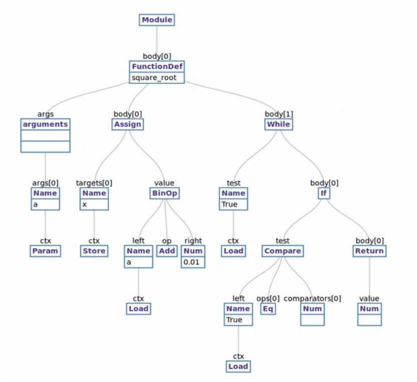
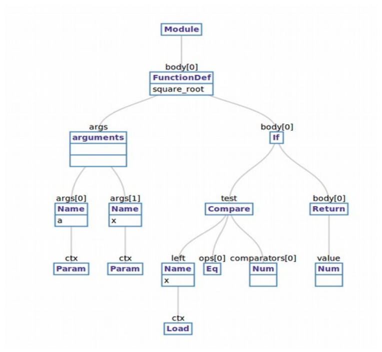
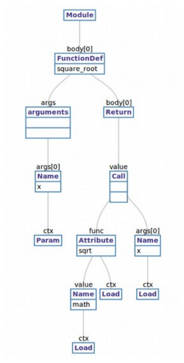
```
def improve(up):
    return (math.factorial(up) == 
math)
[AN ERROR OF:  15.00009799 ]
```

<u>*See all results that were echo'ed in the shell in the file supplied with code:</u>
half_ouput_log.txt

ADDENDUM 8ª

Representative examples of ASTs during the “Combinatorial Forest” portion of the generative algorithm. A large variety of Abstract Syntax Trees are created indiscriminately for the initial AST “forest” exploration process. Varying degrees of health are assigned to the trees during the combination, mutation and selection process.

Note that the current algorithm supports the creation of combinations and mutations of many multiples of functions all with the simultaneously with the same function name declaration. This was achieved through careful scoping during compilation process as well as targeted creation / deletion of code objects:

Infinitely unhealthy, this tree was eliminated completely by algorithm's infinite loop detection and by the execution time scoring component: 	***Midly unhealthy***, this tree is down-scored early during scoring and later permanently discarded: 	**Very healthy, this tree very efficiently computes the desired result, and actually generalizes by using the *Oracle* to do so!** 

`def square_root(a):`
`    x = (a + 0.01)`
`    while True:`
`        if (True == 0):`
`            return 0`
	`def square_root(a, x):`
	`    if (x == 0):`
	`        return 0`
		`def square_root(x):`
		`  return math.sqrt(x)`