Probabilistic Programming

Carlos Diaz, Dan Hassin, Charles Lehner, Dan Viterise

Department of Computer Science
University of Rochester

April 21, 2014

Abstract

We present Probabilistic Programming, a system for generating JavaScript programs and code snippets based off a corpus of JavaScript source files. In this paper, we will demonstrate storing statistical data from non-linear structures (abstract syntax trees of programs) in a Markov model, and then using the model to generate new trees. We present a proof of concept for a web-based program editor that utilizes this technology to guide programmers through constructing JavaScript programs.

1 Introduction

This paper will begin by discussing Markov chains and how we set out to use them for our goal of creating an assistive program editor. We will then describe in detail how we implemented a Markov chain to house program syntax trees, how we were able to generate code using it, and finally we'll discuss our results and address the limitations of our product.

1.1 Markov chain

A Markov chain is a mathematical system that undergoes a stochastic process whereby the current state and a set of probabilities are used to determine what state to transition to next. In the most simple case of a Markov chain, the next action is based only upon the current state and all other previous actions are forgotten. Thus, this process is essentially "memoryless" because all past states can be forgotten once a new state is reached [2]. In this way, Markov chains can be likened to deterministic finite automata. So while they cannot remember a limitless amount of their history, calling them "memoryless" may be misleading, as they can indeed be implemented to remember a fixed size of their state history.

Markov chains find application in many different fields. Already familiar with Markov chains used for generating random dialogue (by training a model based on bigrams and trigrams of common speech), we were interested in more novel domains. We chose to use a Markov chain to drive code generation, because, like text, we feel that code can be personal and expressive, and the idea of generating code possibilities from a training set seemed like exciting research. It was also challenging. Since randomly generating code tokens would most likely produce syntactically invalid programs, we looked at the abstract syntax trees of the programs and had to fit them into a Markov chain model. And as our research developed, we found ourselves with a new vision of an exciting project.

1.2 Emerging goal

For many novice programmers, programming can seem like a confusing and tedious activity. It can be unclear for many new programmers how they should begin and what code should come next. They may conceptually understand what the next step should be, and yet be unable to express it with code. Or, they could be unsure of what to do next, and perhaps giving code suggestions could inspire them and help them to understand it conceptually.

Even as one becomes more adept at programming, he or she may notice typing the same or similar lines of code over and over again. This repetitiveness slows down the user and can make programming more tiresome a task.

So, the prospect of using a Markov chain to generate code snippets in an assistive, predictive integrated development environment (IDE) for JavaScript seemed to be a natural progression from our research. Our final product, a Probabilistic Programming web application, allows a user to code in JavaScript by way of selecting from a list of suggestions for the user's next line of code. We imagine that this won't only be helpful for new programmers, but that this system would allow for expert programmers to write code with more ease and efficiency.

2 Tree-based Markov chain

2.1 Abstract syntax trees

An abstract syntax tree (AST) is a tree-based, rather than text-based, representation of a program. Such a tree is generated using the grammar rules of a programming language and text input. In the tree, *nodes* are language control structures such as `for` loops, `if` statements, variable and function declarations, and so on. The *edges* connecting a parent node to its children represent the different *features* of that node. For instance, a `for` loop would have the features *init* (e.g. `var i = 0;`), *test* (e.g. `i < array.length;`), *update* (e.g. `i++`), and a *body* (a list of statements that should be executed per loop iteration). To implement lists, we include the additional *follow* edge as a feature to all statements, which would connect a statement node to the next statement in the list. If the statement is last in its container list, we connect its *follow* edge to a special `_end` node. (See figure 3 for an example of a AST.)

Our system deals with programs at the level of AST's as they provide more structure and modularity than text. We use the syntax tree format of Mozilla's Parser API [5], and *esprima* [1], an ECMAScript (the language standard of which JavaScript is an implementation) parser, to transform program source code into an AST of this format. We chose JavaScript because it is (a) dynamic and interpreted, (b) runnable in a web browser, and (c) because JavaScript AST tools already existed.

2.2 Our Markov chain

To build our Markov chain structure, we parse a corpus of JavaScript files into their respective AST's, and then collect statistics from these trees to establish probability distributions of nodes for a given AST context.

An *AST path* is a list of items, alternating between node types and features, that encodes the unique position of a specific node in the AST. The *probability distribution* for an AST path is the set of possible node types or values that would appear in the syntax tree following the given path, and the respective probabilities of their occurrence at that point. The model maps AST paths to the probability distributions of node types (or values, as we'll see later) for that given path.

More formally, given a depth d (indicating how many nodes we should backtrace) for a given $node_d$ and feature $edge_d$, the generation of $node_d$'s child node given by $edge_d$ can be expressed by the following function f:

$$f(node_1, edge_1, ..., node_d, edge_d) = \begin{cases} newnode_1, & P(newnode_1) \\ newnode_2, & P(newnode_2) \\ ... & \\ newnode_n, & P(newnode_n) \end{cases},$$

where

$$\sum_{i=0}^{n} P(newnode_i) = 1,$$

and $Path(node_1, edge_1) = node_2$, $Path(node_2, edge_2) = node_3$, ..., $Path(node_d, edge_d) = newnode_k$, where $newnode_k$ is randomly selected from $newnode_1, ..., newnode_n$ given the probabilities $P(newnode_1), ..., P(newnode_n)$. From this it can be seen that generating programs is simply a repetitive application of this function, where for the chosen $newnode_k$, we can generate each child of $newnode_k$ with the following formula:

$$\forall edge \in Features(newnode_k),$$
$$Path(newnode_k, edge) = f(node_2, edge_2, ..., node_d, edge_d, newnode_k, edge).$$

This process must eventually terminate as the probability of generating `_end` nodes for a *follow* edge will always be non-zero (since the programs we parse into our corpus are surely finite) and such nodes have no features to be "expanded."

2.3 Implementation

2.3.1 Data structure

Building the Markov chain model means being able to provide the probability distribution of new nodes for arbitrary input, as described by the function in Section 2.2. To do this, we construct a data structure using JavaScript Object Notation (JSON), a structural format useful for ease of specification and integration with JavaScript programs.

The data structure is a hash table (also commonly called a map, dictionary, or associative array), that, on a given input, returns another hash table for paths that begin with that input. This behavior is repeated until the number of inputs reaches the desired depth, at which point a final hash table, mapping the node type to its probability distribution, is returned. This, in effect, is modularizing the function f described above, so that we can reach the probability distribution with minimal code and data duplication. This is demonstrated by following example, given $depth = 2$.

$$lookup(lookup(lookup(lookup(Model, node_1), edge_1), node_2), edge_2) = \mathbf{P}(newnodes).$$

2.3.2 Building the model

We first obtain collection of JavaScript programs in AST form. We then run the following algorithm to construct our model:

```
procedure addToModel(node, path):
    model = CorpusModel
    for each elem in path[path.length-depth*2 .. path.length]:
        model = model[elem]
    model[node.type] += 1
    model[_total] += 1
    for each feature of node:
        temp_path = path + [node.type, feature]
        if node.feature is list of nodes:
            for each item in node.feature:
                addToModel(item, temp_path)
                temp_path += [item.type, "FOLLOW"]
            addToModel(END_NODE, temp_path)
        else:
            addToModel(node.feature, temp_path)

for each parsed AST:
    addToModel(AST.root, [])
```

For each AST, we walk through every node in the tree by recursively exploring each feature[1] of each node. In each call to explore a node's feature, we maintain the path up until that node by appending to it the node type and the name of the feature to reach the wanted child. We trim the beginning of the path off so that the length of the path is appropriate with respect to the chosen depth (i.e. so that $|path| = 2 * d$).[2] We traverse through the layers of hash tables, creating the path elements if they don't already exist, add the node type to the probability distribution if it isn't already included, and then increment a counter associated with that node type. We also increment a "`_total`" counter for the convenience of the generator program.

When this process is complete, the JSON object is saved to a file and can be augmented by more code samples thereafter. (An example of such a model is shown in figure 4 in the Appendix. In this model we see, for instance, that the path [*Program, body, VariableDeclaration, declarations*] leads to *VariableDeclarator* with probability 1. This indicates that a *Program* node whose *body* is a *VariableDeclaration* always has a statement that is of type *VariableDeclarator* as its first statement.)

2.4 Generating code

Code can be generated rather easily using the approach explained in Section 2.2. The program tree is built up one node at a time, each one generated based on its unique AST path. The following algorithm demonstrates this process.

[1] While we use *follow* as if it were a feature edge, programs encoded in the AST's we parse use arrays of nodes (rather than linked-lists as we represent them) for a collection of statements. The given pseudocode demonstrates how we can transform an array into a linear sequence of edges.

[2] Note that this requires that the path is at least of length $2 * depth$. If the path is shorter than this, it represents not having at least d levels of context. This can occur, say, for the first statement in the program, where there are no preceding nodes in its AST path. For these cases, we add the "`_null`" feature until we reach the desired depth (omitted in the pseudocode.)

```
procedure generateNode(model, path):
    for each elem in path[path.length-depth*2 .. path.length]:
        model = model[elem]
    nodeType = pickRandomNodeType(model)
    node = {'type':nodeType}
    for each feature of nodeType:
        new_path = path + [node.type, feature]
        if feature is a list feature:
            nodes = []
            while (newNode = generateNode(model, new_path) != END_NODE:
                nodes += [nodeNode]
                new_path += [newNode.type, "FOLLOW"]
            node.feature = nodes
        else:
            node.feature = generateNode(model, new_path)
    return node

generateNode(CorpusModel, [])
```

The algorithm looks very similar to the parse algorithm, as it is almost the same process, simply inverted. Rather than examine nodes, determine a path by building it up, and then increment a count in the model, this algorithm first finds the counts in the model for a *given* path, generates a node, and generates nodes for all of that node's features recursively. (Since the final hash table is not a proper probability distribution, and is rather a tally of each node type, here the `_total` is useful for determining ratios.) The result is a generated AST for our program, which is then fed into *escodegen* [4], an ECMAScript code generator to output program source code that we can display and run.

To generate random programs, we always begin with a "Program" node, and then add nodes to its *body* feature (an array.) However, we quickly found that while entertaining, generating random programs had little potential for real application. Instead, we turned to generating code snippets (to be placed in specific contexts of the syntax tree.) Since our algorithm was general enough for this purpose, no more had to be done besides knowing the path for the node we wanted to generate.

2.4.1 Identifiers and literals

Naturally, identifiers and literals are a very large part of programming. Without them, code would be entirely unspecific, and would only consist of syntactic symbols and keywords. Yet they pose an issue with code generation, as we found generating identifiers or literals that are likely to be what the user is expecting an especially difficult problem to solve, let alone model. Struggling to come up with a simple, intelligent, and robust solution, and short on time, we fell back to using our existing Markov approach.

In the current implementation of this system, following the paths [*Identifier*, *Name*], or [*Literal*, *Raw*] result not in node types, but rather in the text of the identifier names or literal values. This way, generating identifiers and literals is simply a probabilistic process like the rest of the code generation. Of course, this means that sometimes identifiers are added that are not in scope, that have mismatched types, or that simply make no semantic sense. (But adjusting *depth* did seem to make programs more meaningful, as expected. This can be seen experimentally by modifying depth on our web IDE, which is discussed in Section 4.1.)

3 Applications

3.1 *vim* autocomplete script

We integrated our model into the text editor vim to offer autocompletion suggestions of programs upon the user's request, using vim's `completefunc` feature. When the user is editing a JavaScript file in vim with this feature, they may press a keyboard command to get a list of program snippets to choose from. The snippets are generated in the context of the program path at cursor's position. The user may pick one of the snippet suggestions to have it inserted into their code at their cursor. The autocomplete snippets are generated as follows:

1. Parse the contents of the current file into an AST.
2. Find the node path corresponding to the cursor by traversing the AST.
3. Generate programs starting from the node path of the cursor.
4. Present the programs to the user as code completion snippets.

(a) Inserting a `for` loop

(b) Inserting an object property

Figure 1: Screenshots of vim using autocomplete of programs

The vim autocomplete feature is limited to inserting program snippets on a single line, and requires the user to explicitly trigger the `completefunc` feature to be prompted with code snippets. To further explore using our model for editing, we created a web-based editor that more deeply integrates Probabilistic Programming.

3.2 Guided IDE

The Guided IDE offers users an environment for building a program entirely using our model, using program snippets generated at the cursor's context as building blocks. The editor presents a window for the program text, and a list of suggestion code snippets. The user can move a cursor through the code window at positions corresponding to boundaries of program nodes. As the user moves the cursor, the snippet suggestions are regenerated using the cursor's node path, to continuously provide suggestions relevant to the current context of the program. When the user selects a suggestion snippet, its AST is inserted into the program's AST at the cursor's position in the tree, and the code window is redrawn. The user can move the cursor within newly inserted blocks, so that the program can grow deeper. The initial corpus for the IDE's model consists of the JavaScript files that comprise our project.

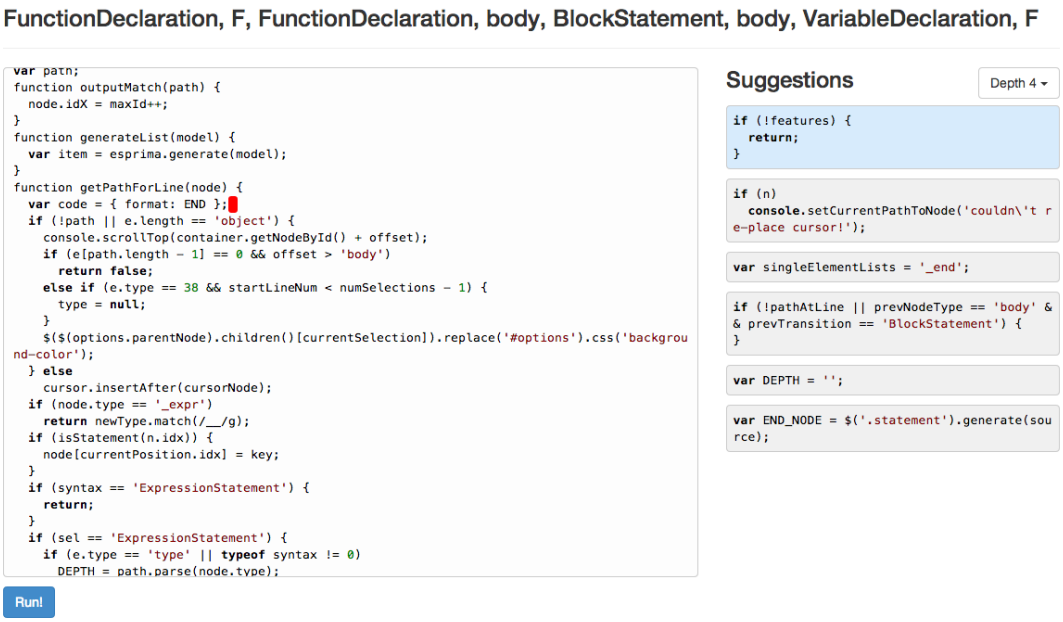

Figure 2: Screenshot of the web program editor GUI. The header along the top indicates the path (trimmed to the selected depth) to the current position of the cursor. Code snippets on the left can be navigated using the arrow keys.

4 Results

4.1 Using the IDE

We can use the Guided IDE to generate programs. The programs look like actual programs but do not actually function unless the user is very careful about which snippets they pick. Crucially, the variable and identifier names that our system generates are not in general meaningful in the scope of the generated program. Whereas with the vim autocompletions the identifiers would often be valid because because the corpus was the current file, with the larger corpus of the Guided IDE and writing a program from scratch, the user is less likely to find program snippets that use valid identifiers.

4.2 Limitations of the model

It is a limitation of the current model that identifier names are chosen probabilistically based on their position in the code, without regard to whether they are in scope and without refering to anything in another part of the program. Because of this limitation, the Guided IDE is not useful for practical purposes. We consider some alternative approaches for the Guided IDE using a similar model.

4.3 Alternative approaches for the IDE

4.3.1 Prompt for identifiers

Rather than picking identifier names from the model, use placeholder identifiers and prompt the user to enter actual identifiers, with suggestions based on the current variables in scope and previously used identifiers. This would give the user the opportunity to create semantically valid programs, but the entering of identifier names could be tedious.

4.3.2 Stubs

Instead of having a movable cursor in the code window, consider stubs in the program. Each stub would be a marker in place of an AST node. The user would select a stub and be given choices of snippets to replace it. Selecting a snippet would move the current snippet into the program in place of the selected stub. The generated snippets could themselves have stubs in them, so that the user may continue to complete stubs and expand their program. At the level of variable identifiers, the snippet suggestions would be taken from the computed list of in-scope variables at the position of the selected stub, unless the completion was for a variable declaration, in which case the user would be prompted to name a new variable.

This approach, while offering less places to insert code then the current approach, would allow for programs to more closely be generated from our model, since the stub completions would always be at leaf nodes of the AST, whereas in the current approach the user may choose to insert a snippet in the middle of a list, which means that the transition from the inserted snippet to the following node in the list does not in general follow from the model.

While the stub approach would be more restrictive for the user as it requires them to build the program in a certain way, with proper handling of identifiers, it may provide for an interesting or useful paradigm for interacting with a program source.

4.4 Intelligent handling of identifiers and literals

We explored other approaches to solve the issue of generating meaningful identifiers and literals described before.

4.4.1 Secondary models

When generating identifiers, we were interested in generating meaningful ones rather than random ones (albeit from a weighted distribution.) We wanted to design a model that could keep track of when the same variables were used in the same (or similar) place.

One idea was to, during the parse, keep track of the path at which each variable identifier was used, grouped by the identifier name. When an identifier needs to be generated, we can find that path in our secondary model, and if so, bind to a random (but weighted) identifier that has been used at that path, with the weight being based on the frequency of that identifier appearing at that path. For subsequent statements that need identifiers, the same process could occur, and, if the same variable group was chosen, it could generate the variable name that was bound to the first statement.

Another idea also involved constructing a secondary model during the parse, that kept track of statements that used the same identifier. When such statements are found, it would compute the path between those statements and increment a counter associated that path. This would represent the likelihood of two nodes connected by that path using the same identifier. Then, during code generation, whenever an identifier is needed, the generator would make a list of all variables in scope at that point, and compute the path from the variable declaration (and/or

last use) to the node being generated. It would look up each path in the secondary model and assign each of the in-scope variables a weight. A variable could then be chosen based on this probability.

Unfortunately, we were unable to implement these ideas or explore them further due to time constraints.

4.4.2 Using a higher-order model

Just as our approach of modeling a program with a tree-based Markov chain is a higher order model than a linear Markov chain of the program source code tokens, there may be applications of probabilistic modeling to higher-order representations of programs than the AST, such as a *higher-order abstract syntax*, which promises to "incorporate name binding information in a uniform and language generic way" [3]. Such an application may model the relationship between different parts of an AST, and handle scoping of variables to model identifiers abstractly.

With a more sophisticated model of identifiers that could generate programs with identifiers in meaningful relation to each other, Probabilistic Programming could be used to generate semantically valid programs, for greater assistive, expressive, and programmatic power.

5 Conclusion

We consider the Probabilistic Programming web IDE a strong proof of concept for context-based code snippet generation. We are aware of some of its serious structural limitations, but we speculate that related concepts, such as genetic algorithms, might enjoy more success. Specifically, we are interested in the possibility of creating genetic algorithms that take advantage of this platform to build and "evolve" code, swapping parts of AST's in a Markovian-like process. Perhaps the evolution could even be given a goal; for instance, to satisfy a set of given test cases. Regardless, we believe that our research opens new possibilities with program generation.

Source code & demo

The source code for Probabilistic Programming is available online under an open source license: `https://github.com/RocHack/jschain`.

The web IDE can be found at `https://csug.rochester.edu/u/clehner/jschain/web/`.

References

[1] Ariya Hidayat *Esprima*
`http://esprima.org/`

[2] Takis Konstantopoulos *Introductory lecture notes on Markov Chains and Random Walks.* Uppsala University,
`http://www2.math.uu.se/~takis/L/McRw/mcrw.pdf`

[3] Frank Pfenning, Conal Elliott *Higher-Order Abstract Syntax.* ACM SIGPLAN Notices. Vol. 23. No. 7. ACM, 1988.
`http://www.cs.cmu.edu/~fp/papers/pldi88.pdf`

[4] Yusuke Suzuki *Escodegen*
`https://github.com/Constellation/escodegen`

[5] Mozilla Developer Network *SpiderMonkey Parser API*
`https://developer.mozilla.org/en-US/docs/SpiderMonkey/Parser_API`

Appendix

Figure 3: JSON AST sample
An AST generated from the program "`var x = 34;`"


```
{
    "type": "Program",
    "body": [
        {
            "type": "VariableDeclaration",
            "declarations": [
                {
                    "type": "VariableDeclarator",
                    "id": {
                        "type": "Identifier",
                        "name": "x"
                    },
                    "init": {
                        "type": "Literal",
                        "value": 34,
                        "raw": "34"
                    }
                }
            ],
            "kind": "var"
        }
    ]
}
```

Figure 4: JSON Markov model sample
A Markov chain model generated from the program "`var x = 34;`"


```
{
    "_null": {
        "_null": {
            "_total": 1,
            "Program": 1
        }
    },
    "Program": {
        "body": {
            "_null": {
                "_total": 1,
                "VariableDeclaration": 1
            },
            "VariableDeclaration": {
                "declarations": {
                    "_total": 1,
                    "VariableDeclarator": 1
                },
                "F": {
                    "_total": 1,
                    "_end": 1
                }
            }
        }
    },
    "VariableDeclaration": {
        "declarations": {
            "VariableDeclarator": {
                "init": {
                    "_total": 1,
                    "Literal": 1
                },
                "id": {
                    "_total": 1,
                    "Identifier": 1
                },
                "F": {
                    "_total": 1,
                    "_end": 1
                }
            }
        }
    },
    "VariableDeclarator": {
        "init": {
            "Literal": {
                "raw": {
                    "34": 1,
                    "_total": 1
                }
            }
        },
        "id": {
            "Identifier": {
                "name": {
                    "_total": 1,
                    "x": 1
                }
            }
        }
    }
}
```