Gregory George (gkg26), Hahnbee Lee (hl985)
Professor Kevin Ellis
CS 6172
23 November, 2021

ReadableCoder

Abstract

One of the core questions that arise when code is being generated by a machine is whether it's readable or not. While the readability of software is not something strictly defined or measurable, there are certainly guidelines and practices that allow beginners and experienced programmers alike to better understand a block of code. Furthermore, humans usually look at the output of program synthesis applications such as FlashFill in hopes of seeing the program created to fulfill the problem they posed it with. In situations such as this, it would be useful for a program synthesizer to provide code that can be easily understood in addition to being functionally correct. It is very common for synthesizers to output solutions that are time and space efficient and that consist of the fewest number of instructions possible. While efficient from the perspective of the computer, such programs can be unintuitive or tedious to parse through for a human viewer. ReadableCoder explores how a program synthesizer can create programs optimized for "human readability".

To this end, we were able to apply the Codex classification API to train a model of code readability. This model was used to extract likelihood probabilities of a given code block's readability level using similar examples from its training set. We then used these probabilities in a cost function that could be used by a synthesizer. Our desired domain specific language is a programming language similar to that of python, but since that is too ambitious we created features of both an imperative language and of lambda calculus onto the grammar from assignment 1. Our plans for the future is to use the expanded grammar and our new cost function to synthesize code optimized for readability.

Current Progress

The Codex classification API provides the functionality of defining training data in a standard (datapoint, label) format and using it to create a model. For our readability model, we decided to begin with a simple four-label output space. In order from worst to best, these labels are Unreadable, Difficult, Acceptable, and Readable and are intended to reflect a code block's rating across characteristics including program length, complexity of syntax and notation, and use of complicated or obscure methods and functions. To create and fine tune a cost function, we generated a model using a hand-made training set. The examples in this training set were created by focusing on a set of small programming problems and writing multiple programs with different approaches and syntax to solve them. These programs were then assigned a label based on how we perceived their rating in the previously mentioned criteria. Table 1 below shows some of the example programs used.

Programming Problem	**Example Program**	**Assigned Label**

Contains - Finding if an item is in an array	`def contains(item, arr):`
	`    return item in arr`
		Readable - Program is short, uses basic Python syntax and does not use obscure methods.

	`def contains(item, arr):`
	`    for elem in arr:`
	`        if elem == item:`
	`            return True`
	`    return False`
		Acceptable - Program is longer than necessary since it implicitly iterates through the list. However, the syntax is still clear and understandable.

	`def contains(item, arr):`
	`  for ind in range(len(arr)):`
	`    if arr[ind] == item:`
	`      return True`
	`  return False`
		Difficult - Program is verbose and requires lots of setup for implicit iteration. Uses nested Python methods and array indexing in a way that is not immediately clear.

	`def contains(item, arr):`
	`    return next(filter(lambda arr_item: arr_item == item, arr), None) != None`
		Unreadable - Program is one line and extremely dense. Uses advanced Python syntax such as lambda functions in tandem with complex nested methods. Careful examination is needed to understand how the program works.

Table 1: Subset of example programs used in training set

The model would output probabilities associated with each label, giving the most likely label as the output for the queried program. To create an associated cost from this information, we decided to apply a cost to each label and take the sum across all the labels weighted by their likelihood. This would mean that the cost would be near the associated label cost if it had a high likelihood whereas it would deviate if there were multiple labels with near equal probabilities. Worse readability labels had a higher associated cost so that unreadable code would cost more for the synthesizer.

Initially we wanted to test our cost function on assignment 1, but then we realized that an ‘if’ statement was the most complex expression. Thus we decided to expand upon the grammar from assignment 1 and added lambda expressions, function applications, and let expressions and plan on synthesizing using this expanded grammar.

Related Works

Most of the previous work related to producing readable code is concentrated in creating readable code for data scientists as seen by Wrex (Drosos et al 2020) and cogram. Cogram is applicable to us in the way that they are powered by Codex. And Wrex is applicable because it produces Python code. However, it's different in that it focuses on code that formats and wrangles data which is a different context from ours.

One interesting aspect about readable code that was pointed out in the Wrex paper is that some features of acceptable, readable synthesized code that data scientists wanted were better indentation, line breaks, naming conventions, and meaningful comments in the synthesized code. But other data scientists desired synthesized code that was brief and easy to follow. In our context when we were creating test cases we also came to this realization that something short could be considered readable, but something that is more verbose with more variable instantiations could also be classified as readable. We believe that the difference in preferences between the two varies with experience: a new programmer might prefer something more verbose while a more experienced software engineer might prefer a brief program.

Future Plans and Concerns

For the next week, we plan on creating more test cases - possibly training and classifying on our versions of assignment 3 and figuring out who has the more readable code.

We also plan on fully integrating the cost function into the synthesizer from assignment 1. We weren't able to finish up integrating the logic for the expanded grammar because we realized quite late that we'd need to expand upon the language. We are concerned whether the synthesizer can handle our expanded grammar and whether adding lambdas is complicated enough to determine readability. We are especially concerned about the capabilities of the synthesizer from assignment 1 after finding this paper about synthesizing Java code which seems quite complicated (Grigorenko, P.).

Any advice would be appreciated - our main problem is that Codex and ourselves are best at classifying the readability of more complex programs than the ones we've been synthesizing in our assignments. Are there any openly available synthesizers that synthesize Java, Python, or any relatively similar code?

Alternatively, we are also considering using the probabilities for our classification as a feature vector for an expanded version of the PCFG from assignment 3. After the work we've done thus far and applying what we learned from assignment 3, we're considering scrapping our work regarding the codex classification and expanding upon the grammar in assignment 3, defining our own feature vector based on readable features and running the synthesizer.

Citations

Drosos, Ian, et al. "Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists." *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, https://doi.org/10.1145/3313831.3376442.

Grigorenko, P.. "PROGRAM SYNTHESIS IN JAVA ENVIRONMENT." (2004).