Recognizing Handwritten Source Code

Qiyu Zhi* Ronald Metoyer†

University of Notre Dame

ABSTRACT

Supporting programming on touchscreen devices requires effective text input and editing methods. Unfortunately, the virtual keyboard can be inefficient and uses valuable screen space on already small devices. Recent advances in stylus input make handwriting a potentially viable text input solution for programming on touchscreen devices. The primary barrier, however, is that handwriting recognition systems are built to take advantage of the rules of natural language, not those of a programming language. In this paper, we explore this particular problem of handwriting recognition for source code. We collect and make publicly available a dataset of handwritten *Python* code samples from 15 participants and we characterize the typical recognition errors for this handwritten *Python* source code when using a state-of-the-art handwriting recognition tool. We present an approach to improve the recognition accuracy by augmenting a handwriting recognizer with the programming language grammar rules. Our experiment on the collected dataset shows an 8.6% word error rate and a 3.6% character error rate which outperforms standard handwriting recognition systems and compares favorably to typing source code on virtual keyboards.

Keywords: programming, handwriting recognition, touch screen, source code, python.

Index Terms: H.1.2. [User/Machine Systems]: Human information processing; H.5.2. [User Interfaces]: Input devices and strategies (e.g., mouse, touchscreen)

1 INTRODUCTION

With the rapid technology shift in current computing devices, high-quality low-cost mobile devices such as tablets and smartphones are being increasingly used in everyday activities. Many tasks that previously required a PC are now feasible on mobile devices. For example, tablets are typically equipped with powerful batteries, advanced graphic processors, high-resolution screens and fast processors, making writing and compiling code on them completely plausible. TouchDevelop, for example, is a novel programming environment, language and code editor for mobile devices [41]. Furthermore, Tillmann et al. predict that programming on mobile devices will be widely used for teaching programming [42]. However, mobile devices are also inherently restricted by their limitations such as small screens and the clumsy virtual keyboard. Entering and editing large amounts of text for programming tasks can quickly become difficult and time consuming with these virtual keyboards because they are notoriously difficult to use when compared to a physical keyboard and they consume valuable screen space [35].

While keyboards have been the primary input device for entering computer programs since the computer was invented [15], this predominant mechanism is not ideal for all programming situations. For example, software developers that suffer from repetitive strain injuries (RSI) and related disabilities may find typing on a keyboard difficult or impossible [4]. Instead, handwriting with a stylus may be a preferred input mechanism for some of these users [27]. In addition, some physical configurations (e.g. seated on a plane) may simply be more suited to the writing posture than a typing posture for many users.

Handwriting has also been shown to have potential cognitive benefits [3]. In particular, Mueller and Oppenheimer found that students who took longhand notes performed better on conceptual questions than those that typed notes on a laptop [30]. Given these findings, and that fact that many programmers write pseudocode by hand before typing, it is reasonable to consider that handwriting may provide cognitive benefits for programming, especially on mobile devices. Furthermore, recent advances in pen-based input and handwriting recognition technology are quickly making handwriting a viable alternative to typing.

In this paper, we explore the use of handwriting as a means for source code text input. There are two ways to approach this problem. One alternative is to develop or modify a handwriting recognition engine to take source code directly into account. Given that source code often includes English language words, another alternative is to leverage the capabilities of an existing English language handwriting recognition engine. We explore this latter option. First, we collect and present a publicly available dataset of handwritten *Python* source code for use in handwriting recognition research. Second, we explore the use of the state-of-the-art recognition system, MyScript [31] for recognizing *Python* source code. We characterize the errors made by the MyScript engine and present a method for post-processing the engine's results to improve recognition performance on handwritten *Python* source code.

After presenting related work and necessary background information, we describe our data collection process and the resulting publicly available dataset. We then describe the performance of MyScript on recognizing the handwritten source code and present our algorithm for leveraging the MyScript engine to produce improved results. We discuss those results in Section 7 and conclude with avenues of future work.

2 BACKGROUND AND RELATED WORK

Many alternatives to typed source code have been considered, typically in the context of making programming more accessible. Most of those appoaches fall in the realm of speech-based programming [7, 15], however speech is not always an acceptable solution, especially in quiet environments or with applications that require privacy. Given the recent advances in pen-based input, handwriting is a potentially viable alternative. The pendragon supports people who are unable to use a keyboard and seeks to find new interaction techniques for input which may improve communication speed [26]. Mankoff et al.also suggest that word prediction, sentence completion and the syntax of programming languages could be used for handwriting source code [27]. Most closely related to our work is a programming IDE integrated with a handwriting area in which the handwritten code is recognized by an enhanced handwriting recognition system [14]. This work, however, does not present an evaluation of the recognition engine or guidance for how to improve general handwriting recognition engines for application to source code recognition. In this paper, we focus on the recognition step of handwritten source code as the most important step for developing

*e-mail: qzhi@nd.edu

†e-mail:rmetoyer@nd.edu

16-19 May, Edmonton, Alberta, Canada

an effective handwriting interface for source code input and editing.

Research in handwriting recognition has a long history dating back to the 1960s [40]. Hidden Markov Model (HMM) based handwriting recognition [19, 23, 32] is one of the most widely used approaches while neural networks are gaining in popularity [20]. Some approaches also leverage additional constraints for recognizing handwriting in specific domains such as postal addresses [38, 39] and banking checks [1, 16]. These handwriting recognition systems are developed to take advantage of the English language [44], which is intrinsically different from source code. For instance, variable names are often created from concatenated words (e.g. camelCase or underscore naming), which poses a problem for the traditional handwriting recognition system as it expects spaces to appear between words contained within its dictionary. We do not aim to contribute to the extensive literature in handwriting recognition, but rather, we intend to examine how we can leverage this existing work for application to handwritten source code recognition interfaces.

3 Data Collection

Our first contribution in this paper is a data collection study designed to generate a sample set of handwritten source code for research purposes. The first question to consider is what programming language to study. We decided to collect handwritten *Python* source code because of the current popularity of *Python*[1] and its projected growth rate [36]. We chose a "copying task", where three code samples are provided for every participant to copy on the tablet using the stylus. While we understand that a "copying task" may be cognitively quite different from other writing tasks that require synthesis, we sought to eliminate sources of cognitive load that could impact timing as well as writing quality for the purposes of this data collection task. The three shared code samples allow for comparison across participants. To broaden our dataset of unique handwritten source code samples, we also randomly selected a fourth source code sample function (per participant) that was unique to that participant. In this section, we describe our data collection process and the resulting database of handwritten *Python* samples.

3.1 Participants

We recruited 15 participants (9 females) from the University of Notre Dame for our study. Thirteen of the participants were computer science majors and all participants had at least two semesters of programming experience. Their ages ranged from 19 to 29 (mean = 22.3). Two participants were left-handed. Eight participants had used a pen/stylus for handwriting on a touchscreen device and only one participant had used a tablet for inputting source code (via the virtual keyboard). All participants were compensated $5 for the study which took approximately 30 minutes each.

3.2 Apparatus and Software

We used a 12.9 inch iPad Pro with a 2732-by-2048 screen resolution at 264 pixels per inch (PPI) and fingerprint-resistant oleophobic coating. Participants used the Apple Pencil as the stylus device. We implemented a web application with a writing area to record user input. This application was responsible for converting touch points of the stylus into handwriting strokes and saving strokes to a JSON file. Each stroke consists of the coordinates of the sampled points and time-stamp information for each coordinate. The writing area in this application measured 795 * 805 pixels with subtle lines on the background to provide guides for the participants (See Fig. 1). We also implemented functions to undo or redo the previous stroke as well as clear the writing area of all strokes.

[1] http://www.tiobe.com/tiobe-index/

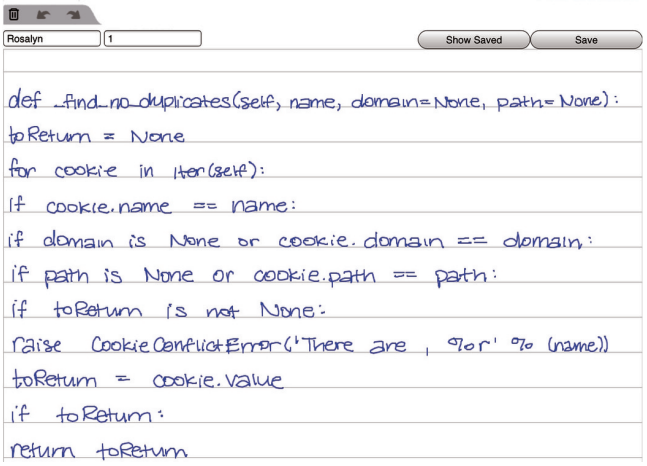

Figure 1: Screen shot of our data collection web application. Participants entered their name and code sample number into the boxes in the upper left. They then entered the code sample using the Apple Pencil and selected 'Save' when finished.

3.3 Representative Source Code Material

Our goal was to create a database of representative samples of handwritten *Python* source code for use in evaluating the performance of a handwriting recognition system. Because different *Python* samples contain different language elements, there is no single representative corpus [2]. Ideally, representative code samples should contain a variety of language constructs and not be restricted to a single project.

Our process for choosing source code samples is based on that used by McMillan et al. [29, 37]. First, we selected six popular *Python* projects on Github. Table 1 summarizes the details of these projects. We then extracted all functions from the project source code and eliminated comments in order to focus solely on the source code of the samples. Next, to obtain functions that were sufficiently long to collect a substantial amount of handwriting, but not so long as to require multiple pages of handwriting, we filtered the functions to those with between 9 and 18 lines of source code and those with no lines greater than 60 characters (to eliminate long, wrapping lines). We also manually filtered out highly repetitive functions, such as a function that includes only assignment statements for variables. The result was 1324 eligible functions. We randomly selected the three shared test code samples from this set for use by all participants and one additional unique code sample to be entered by each participant in the study. Although *Python* syntax considers whitespace, we decided to ignore indentation for the purposes of focusing purely on handwriting recognition.

3.4 Procedure

For every participant, we began our data collection with an informed consent process. Each participant then filled out a pre-study questionnaire about demographics and experience using touchscreen devices and a stylus. Participants were given a practice task to familiarize them with the process. For each of the four tasks, participants were given a sheet of paper with the sample typed *Python* source code. Participants entered their name and code sample number into the web application and then entered the code sample using the Apple Pencil and selected 'Save' when finished. After completing all four input tasks, participants were compensated and the session ended.

3.5 Data Collection Results

The final dataset includes stroke data for four code samples for each of 15 participants resulting in a total of 60 handwritten source code

Project	Lines	Fuctions	Eligible Functions
AlphaGo	1,963	151	1
Bittorrent	7,164	570	39
Blender	265,684	12,774	1,126
Instagram	1,265	145	8
Requests	14,009	862	84
Webpy	10,199	1,029	66

Table 1: *Python* projects used for selecting code samples

samples. So, for each of 3 given source code input samples, we have 15 copies of handwritten source code (for a total of 45 handwritten source code samples). The remaining 15 are handwritten samples of unique input source code examples from each participant. The handwritten source code data can be downloaded at `http://www.purl.org/recognizinghandwrittencode/data`.

4 Source Code Recognition Errors

Current commercial handwriting recognition systems are built to take advantage of the rules of the English language as opposed to that of a programming language, therefore it is not surprising that these systems might perform poorly on source code recognition [14]. There is, however, no previous research that evaluates how well existing state-of-the-art handwriting recognition systems perform on handwritten source code. Here we describe the state-of-the-art handwriting recognition system we employed and characterize the errors based on the dataset we collected.

4.1 State-of-the-art: Myscript

Automatic recognition of handwriting is now a mature discipline that has found many commercial uses [33]. MyScript [31] is an online handwriting recognition engine that supports more than 80 languages and achieved the best recognition rate in the International Conference on Document Analysis and Recognition competition [11]. Here we use the MyScript engine as our baseline for comparison and study the typical recognition errors produced when applied to handwritten source code to better understand the complexities introduced by *Python* source code and source code in general.

4.2 Data Pre-Processing

In order to use MyScript efficiently and to make a fair comparison between its performance and our algorithm, we apply two simple pre-processing steps to the data. First, we provide MyScript with *Python* specific context through the Subset Knowledge (SK) facility and a custom lexicon. SK is a MyScript feature for telling the recognizer that we only want it to enable recognition of certain characters. For example, for a phone number field, we may want only digits to be recognized. We created an SK resource in MyScript to allow only characters that can legally appear in *Python* source code. We also provide the legal *Python* keywords through a user-defined lexicon.

The MyScript Cloud Development Kit (CDK) is an HTTP-based set of services that take handwritten strokes as input and produce potential recognition results as output. To use the CDK in experiments, we must send strokes to the recognizer at some level of granularity (e.g. single character, whole word, whole line, etc). We chose to break the stroke data into lines assuming developers might write a statement at a time on a single line. To do so, we analyze stroke coordinates and create a new line each time the user moves to a new vertical position. Each line is then sent one at a time to the MyScript CDK. This simulates a developer writing one programming statement (one line) at a time, pausing at the end of each line. Alternative pre-processing is possible given the raw stroke data and timestamp information (e.g. sending incomplete lines when a participant pauses).

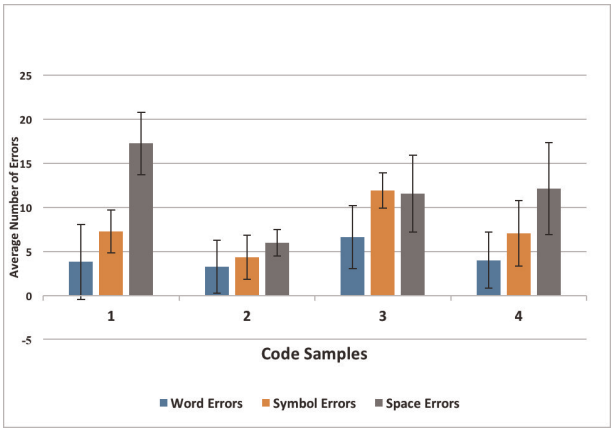

Figure 2: Average error numbers of all participants for each code sample from MyScript general handwriting recognition engine

4.3 Characterizing errors

We processed all of the handwritten data as described above to collect baseline recognition results for all handwritten samples in our dataset. We then set out to understand the types of recognition errors that were present in the final recognized text. We identified three major types of recognition errors: word errors, symbol errors, and space errors.

Word errors occur when MyScript simply incorrectly recognizes a written word. This is typically due to poor writing and can occur for keywords as well as non-keywords. For example, when the handwritten word 'self' is recognized as 'silt', we characterize this as a word error. Symbol errors represent incorrect recognition of symbols or non alpha-numeric characters. For example, an '_' (underscore) is often recognized as a '-' (dash). Finally, a space error results when the system inserts an unexpected space. For example, when 'ConflictError' is recognized as 'Conflict Error', we characterize it as a space error.

Most of the word errors and symbol errors can be attributed to poor writing or cursive writing (characters are written joined together in a flowing manner) which is inherently more difficult for MyScript to recognize than block writing (characters are written separately). Space errors, on the other hand, appear to depend on the language model of the recognizer, which most likely does not include training on CamelCase[2] or proper English words separated by dot notation (e.g. student.name). The result is that MyScript inserts space at these word and dot notation separators.

In summary, from the statistical results for each type of error presented in Fig. 2, space errors, mainly caused by the internal mechanism of English handwriting recognition system, represent the most prevalent recognition error. In addition, poor writing and the tendency to return an English word for a non-English word in the source code lead to word errors, which also represents a significant portion of all errors. Symbol errors are also a prevalent error type. This makes sense given that MyScript is designed to recognize general words, however, symbols, dot notation, and combinations of symbols and words are typically not present in general text, especially in the way that they are used in source code. For example, the most problematic symbols includes underscore '_', parentheses '( )' and equal '='.

[2]https://en.wikipedia.org/wiki/Camel_case

Class	Frequency	Class	Frequency
def	3	except	1
if	7	while	1
for	3	try	1
raise	2	break	1
return	2	else	1
yield	2	assignment	13

Table 2: Frequency for each statement class in three test code samples

5 Handwritten Source Code Recognition Pipeline

A programming language is governed by grammar rules, which stipulate the positions of keywords and symbols. For example, in *Python*, a *def* sentence must end with a ':'. However, handwritten symbols are often problematic. For example, colons ':' are sometimes recognized as semicolons ';'. In addition to grammar rules, programming languages are highly repetitive with predictable properties [18]. Function names and variable names are the most common repetitive words in a single source code project. If a function name appears more than once in the same handwritten code sample, however, it is impossible for users to hand write the *exact* same strokes for this function name, which makes different recognition results of the same handwritten function name a possibility that we must account for.

In this section, we present an approach to improve the recognition rate for handwritten source code by addressing these issues as well as those common errors characterized in Section 4.3. We leverage what we know about the predictability and structure of source code to improve recognition results beyond that of the state-of-the-art recognizer.

The general premise of our approach is that state-of-the-art engines can produce excellent results given good writing and the absence of symbols and programming practices like camelCase. Our framework, illustrated in Fig. 3, is therefore aimed at analyzing and post-processing the recognition results produced from MyScript to utilize its recognition capabilities but correct for those common errors. This framework can be divided into four parts: statement classification, statement parsing, token processing, and statement concatenation. The source code for this post-processing algorithm can be found at `http://www.purl.org/recognizinghandwrittencode/code`.

5.1 Statement Classification

As we mentioned before, we process the handwritten source code data considering each statement as a unit. According to the *Python* grammar specification, we can restrict *Python* source code statements into a limited number of classes, each of which has specified structure rules[3]. Here we use the first token in the statement as the symbol for classification. For example, a 'def' statement starts with 'def' and its structure is defined as 'def' + 'function name' + '(parameters0, parameters1 ...):'. We define 14 classes for *Python* code statements, including an 'assignment' statement, which means the first word in this statement is not a keyword but rather a variable name. In Fig. 3, the recognition result is classified as an 'if' statement. Table 2 presents statistics for the various statement classes in the three code samples.

5.2 Statement Parsing

After classifying the statement, we need to break it down into independent parts according to the grammar rules. Similar to a recursive-descent parser [43], our system consists of a series of functions, each

[3]https://docs.python.org/2/reference/grammar.html

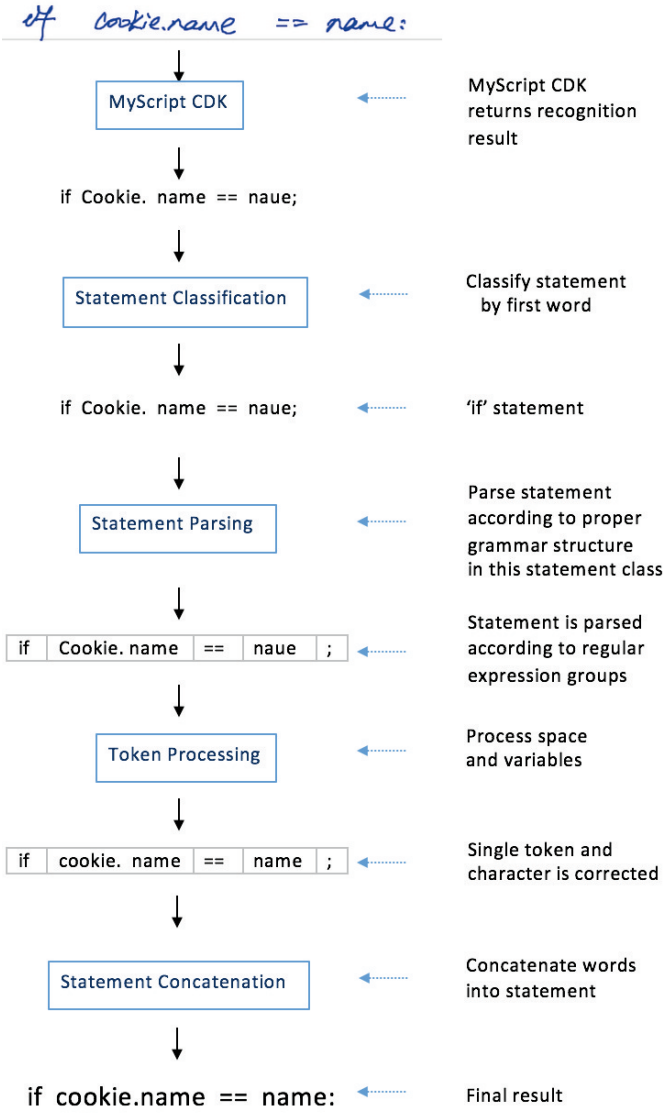

Figure 3: Framework for augmenting MyScript to correct for common recognition errors in handwritten source code.

of which is responsible for one class of statement. Each function includes a set of mutually recursive procedures where each such procedure implements one of the productions of the grammar as a regular expression. We implement a top-down LL parser to parse the input from left to right and perform a leftmost derivation [12] of the statement. As a result, a statement is parsed into a list of single tokens and/or characters. For example, the statement in Fig. 3 is parsed into five individual tokens. Specifically, 'if' is a keyword token; 'Cookie. name' is a variable token; '==' is a symbol token; 'naue' is a variable token; ';' is the last symbol token.

5.3 Token Processing

The previous stage results in a list of single tokens and/or characters that make up the statement. We assume all non-keywords are properly recognized and add them to the lexicon assuming they are *variable* names. Then for all non-keywords in each statement that follows, we first compare the token to all the words in the non-keyword lexicon. If a 'similar' token already exists in the lexicon, we replace it with the 'similar' token in the lexicon. For example,

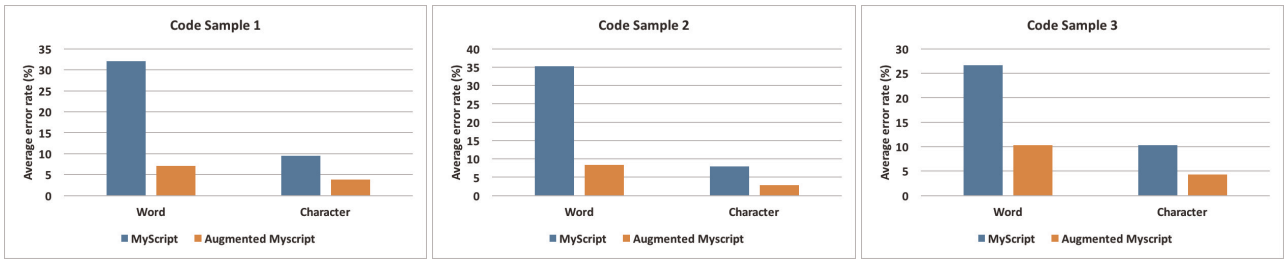

Figure 4: Average recognition error rate of MyScript and our augmented MyScript system for three test code samples

	t-test score (t_{14})	P-value
WER on sample 1	-9.02	$P < 0.00001$
WER on sample 2	-8.29	$P < 0.00001$
WER on sample 3	-6.57	$P < 0.00001$
CER on sample 1	-3.88	$P < 0.001$
CER on sample 2	-5.45	$P < 0.0001$
CER on sample 3	-6.13	$P < 0.0001$

Table 3: Statistical evidence (T-test and P-value) for WER and CER on three code samples

System	WER(%)	CER(%)
Augmented MyScript	8.6	3.6
Kozielski et al. [9]	9.5	2.7
Keysers et al. [21]	10.4	4.3
Zamora et al. [45]	16.1	7.6
Poznanski et al. [34]	6.45	3.44

Table 4: Performance of our system compared to handwritten English recognition systems on the IAM dataset

in Fig. 3, 'naue' is very similar to 'name', which is already in the lexicon, so we just replace the token 'naue' with 'name'. If there is no 'similar' token in the lexicon, we accept this token as it is and add it to the lexicon. We calculate similarity using the Levenshtein distance [25] with a threshold of 0.7, determined empirically.

5.4 Statement concatenation

After processing all tokens, we remove all extra spaces in any single token, then concatenate each token with a single space between them to reconstruct the final statement. Additionally, we ensure that the last recognized character of a statement is a ':'. For example, in Fig. 3, we first remove the space in 'cookie. name' and then replace the last character ';' with ':'.

6 Evaluation

To assess the performance of our system, we measure the Character Error Rate (CER) and Word Error Rate (WER). WER and CER are percentages obtained from the Levenshtein distance between the recognized sequence and the corresponding ground truth. They are calculated as

$$\frac{D+I+S}{L} \times 100\%$$

where D is the number of deleted units, I is the number of inserted units, S is the number of substituted units, and L is the total number of units in the ground truth transcriptions. A unit is a word for WER or a character for CER.

We evaluate our recognition approach by applying our framework to the 45 code samples in our database. In the following section, we compare the results of our enhanced recognizer to the results of using MyScript alone.

7 Results

As shown in Fig. 4, our augmented recognition approach results in an 8.6% word error rate and 3.6% character error rate, on average, over the three code samples, which outperforms the original MyScript recognizer with 31.31% and 9.24% in word and character error rate respectively. We also find statistical evidence for an effect of our augmented recognition approach on both WER and CER (See Table 3).

Since there is no existing handwriting source code recognizer for comparison, we compare the recognition rate of our our augmented MyScript recognition system (on source code) to that of four state-of-the-art general handwritten English recognition systems (on general text). The IAM handwriting database [28] consists of 9,285 lines of general handwritten text written by approximately 400 writers with no restrictions on style or writing tool. This database has been widely used to evaluate English handwriting recognition systems. The four systems in Table 4 were tested based on this IAM handwriting database. Table 3 shows that the WER and CER of our augmented source code recognition system are comparable with other state-of-the-art handwritten English recognition systems on general handwritten text.

8 Discussion

Our approach achieved an 8.6% word error rate and a 3.6% character error rate on the collected dataset by taking the language grammar rules into account. Overall, improvement of our recognition pipeline over the baseline MyScript recognition engine can be attributed to addressing the three main error types identified in Section 4.3. After statement concatenation, all unnecessary space errors in a single token are removed. Ensuring the last character of a statement eliminates 32% of the symbol errors. Token processing fixes around 78% of the word errors.

Recognition results, however, are still not 100% accurate. Initial inspection indicates that this is mainly due to the illegible or cursive handwriting of the participants and the incorrect recognition of symbols. Also, since one of our lexicons is dependent on the non-keywords already recognized in the code, incorrectly recognized words will also be added to the lexicon, thereby corrupting the lexicon and preventing it from enhancing the recognition of the following words. Additionally, it is difficult to identify incorrectly recognized symbols; for example, if '(' appearing in the middle of the text is recognized as 'l', it becomes impossible to rectify it using our approach. Errors like unmatched '(' and ')' in a statement can be detected, but not reliably corrected. For example, '(name' can be recognized as 'cname', but we have no evidence to correct 'cname' to '(name'. Two methods can be employed to resolve remaining errors such as this. The first is to develop a widget in the handwriting interface to highlight all errors that are identified but can't be corrected and let users correct them manually. Another option is to

train a language model to identify words that do not exist [45].

Because typing on a virtual keyboard is the standard input method on touchscreen devices, it is useful to examine how virtual keyboard typing error rates compare to those of handwritten source code recognition. Almusaly et al. report a 7.81% total error rate (TER) for typing *Java* programs on a standard virtual keyboard as measured from 32 participants [2]. TER, similar to CER, is a measure of the total number of errors (i.e., omissions, substitutions, and insertions) and corrections that are made in the resulting typed text. Our handwriting results are comparable.

This approach can also be generalized to other programming languages with strict grammar rules. For instance, one can define statement classes for *Java* according to the first word in the statement and then replace the regular expressions with productions of *Java* grammar rules. Algorithms for searching and replacing similar words can be kept unchanged. Other heuristic steps like concatenating tokens are also trivial to implement for new languages.

9 Conclusion and Future Work

The keyboard is not an ideal input mechanism for every person and situation. Alternatives to typing, such as speech, have been considered in the past [7, 15]. However, speech is not always an appropriate option given social conventions and privacy issues. Given advances in pen-based technology that provides an opportunity for users to engage with devices in a potentially more 'natural' way than that supported by a virtual keyboard, handwriting input is a viable alternative to virtual keyboard input. In this paper, we have explored handwriting recognition specifically for source code with the ultimate goal of supporting handwriting as a means for programming. We collect and present a small database of publicly available handwritten source code samples and we propose an approach to recognize handwritten source code by leveraging a commercial handwriting recognition system. Experiments on the data collected from 15 participants show our framework has an average 8.6% word error rate and 3.6% character error rate which outperforms the baseline recognition system and produces rates comparable to the recognition of general handwritten English text. We are encouraged by these initial results but believe there are several avenues of future work.

From the view-point of human-computer interaction, usability and user satisfaction is critical. For handwriting text input, users expect recognition technology with a low error rate and responsive recognition speed. LaLomia et al. [24] reported that users are willing to accept a recognition error rate of only 3% (a 97% recognition rate), although Frankish et al. [13] concluded that users will accept higher error rates depending on the text-editing task. It would not be surprising, therefore, if higher error rates were acceptable for source code entry and editing which is inherently difficult due primarily to the use of symbols. Input speed is another concern with respect to handwriting. Modest touch typing speeds on a virtual keyboard in the range of 20 to 40 words per minute (wpm) are achievable. Handwriting speeds are commonly in the 15 to 25 wpm range [6, 8, 10]. We suspect that this decrease in speed, however, will be acceptable to the particular groups for whom handwriting is the most viable input option. Additionally, in professional programming, most of the code that developers write involves reuse of existing example code and libraries [5]. This 'reuse' typically amounts to editing existing code to suit a new context or problem and generally provides benefits to developers in terms of time and error reduction [22]. For these reasons, we envision our system as being particularly useful in the code editing domain as opposed to writing extensive source code from scratch. Studying how the algorithms perform in editing tasks is left as future work.

While databases exist for research in general handwritten text recognition [17, 28], there is no such dataset for handwritten source code. This paper represents the first such contribution of a handwritten source code dataset consisting of 555 lines of *Python* code written by 15 participants. While we recognize that using the same three code samples for all users and employing a "copy task" may lessen the generality of the dataset, we sought to eliminate all effects of cognitive complexity (e.g. actually solving programming problems) to focus solely on the handwritten source code quality. Collecting data for other programming languages and for actual programming tasks is left as future work.

The next most obvious area of future work is to develop a handwritten source code recognition system from scratch instead of augmenting the results produced by an existing system. We suspect this approach would lead to comparable and most likely improved recognition rates. Building a universal handwritten source code reading system could employ deep learning techniques such as Concurrent Neural Networks [34] or neural network language models [45] trained purely on the source code.

Additionally, there are several opportunities to explore the integration of handwriting recognition into source code IDEs [14]. For example, how do we now integrate source code completion into a handwriting-based interaction? Can we integrate elements such as syntax insertion and highlighting? Exploring the affordances of handwriting in the context of an IDE is an exciting area of future work that is enabled by these initial findings.

Multimodal methods present another area of future work. Perhaps the combination of handwriting and speech input or handwriting and occasional keyboard input [30] begin to produce interaction experiences that rival those of typed source code input.

Finally, we will never reach a perfect recognition rate for handwritten text (general or source code). How do we effectively support efficient editing of the recognized text so that users can quickly correct mistakes? Natural and effective text entry and editing is an interesting topic for future studies.

Acknowledgments

The authors wish to thank all the study participants as well as Poorna Talkad Sukumar, Jason Liu, and Suwen Lin for their valuable discussions and input.

References

[1] A. Agarwal, A. Gupta, K. Hussein, and P. Wang. Bank check analysis and recognition by computers. *Handbook of Character Recognition and Document Image Analysis*, pp. 623–651, 1997.

[2] I. Almusaly and R. Metoyer. A syntax-directed keyboard extension for writing source code on touchscreen devices. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, pp. 195–202. IEEE, 2015.

[3] M. A. P. Alonso. Metacognition and sensorimotor components underlying the process of handwriting and keyboarding and their impact on learning. an analysis from the perspective of embodied psychology. *Procedia-Social and Behavioral Sciences*, 176:263–269, 2015.

[4] A. Begel. Programming by voice: A domain-specific application of speech recognition.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9), 2007.

[6] S. K. Card, A. Newell, and T. P. Moran. The psychology of human-computer interaction. 1983.

[7] A. Désilets, D. C. Fox, and S. Norton. Voicecode: an innovative speech interface for programming-by-voice. In *CHI'06 Extended Abstracts on Human Factors in Computing Systems*, pp. 239–242. ACM, 2006.

[8] D. B. Devoe. Alternatives to handprinting in the manual entry of data. *IEEE Transactions on Human Factors in Electronics*, (1):21–32, 1967.

[9] P. Doetsch, H. Ney, et al. Improvements in rwth's system for off-line handwriting recognition. In *2013 12th International Conference on Document Analysis and Recognition*, pp. 935–939. IEEE, 2013.

[10] M. D. Dunlop and M. M. Masters. Pickup usability dominates: a brief history of mobile text entry research and adoption. *International Journal of Mobile Human Computer Interaction (IJMHCI)*, 1(1):42–59, 2009.

[11] H. El Abed, M. Kherallah, V. Märgner, and A. M. Alimi. On-line arabic handwriting recognition competition. *International Journal on Document Analysis and Recognition (IJDAR)*, 14(1):15–23, 2011.

[12] H. Fernau. Regulated grammars with leftmost derivation. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pp. 322–331. Springer, 1998.

[13] C. Frankish, R. Hull, and P. Morgan. Recognition accuracy and user acceptance of pen interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 503–510. ACM Press/Addison-Wesley Publishing Co., 1995.

[14] J. Frye and B. Franke. Pdp: pen driven programming. In *Proceedings of the 22nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction-Volume 2*, pp. 127–130. British Computer Society, 2008.

[15] B. M. Gordon. Improving spoken programming through language design and the incorporation of dynamic context. 2013.

[16] N. Gorski, V. Anisimov, E. Augustin, O. Baret, D. Price, and J.-C. Simon. A2ia check reader: A family of bank check recognition systems. In *Document Analysis and Recognition, 1999. ICDAR'99. Proceedings of the Fifth International Conference on*, pp. 523–526. IEEE, 1999.

[17] E. Grosicki[1], M. Carré, J.-M. Brodin, and E. Geoffrois[1]. Rimes evaluation campaign for handwritten mail processing.

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847. IEEE, 2012.

[19] J. Hu, M. K. Brown, and W. Turin. Hmm based online handwriting recognition. *IEEE Transactions on pattern analysis and machine intelligence*, 18(10):1039–1045, 1996.

[20] S. Jaeger, S. Manke, J. Reichert, and A. Waibel. Online handwriting recognition: the npen++ recognizer. *International Journal on Document Analysis and Recognition*, 3(3):169–180, 2001.

[21] D. Keysers, T. Deselaers, H. A. Rowley, L.-L. Wang, and V. Carbune. Multi-language online handwriting recognition. 2016.

[22] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):21, 2011.

[23] A. Kundu and P. Bahl. Recognition of handwritten script: a hidden markov model based approach. In *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, pp. 928–931. IEEE, 1988.

[24] M. LaLomia. User acceptance of handwritten recognition accuracy. In *Conference companion on Human factors in computing systems*, pp. 107–108. ACM, 1994.

[25] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, vol. 10, p. 707, 1966.

[26] J. Mankoff. Pendragon, 1999. `http://www.cc.gatech.edu/fce/pendragon/`.

[27] J. Mankoff and G. D. Abowd. Cirrin: a word-level unistroke keyboard for pen input. In *Proceedings of the 11th annual ACM symposium on User interface software and technology, http://www.cc.gatech.edu/fce/pendragon/*, pp. 213–214. ACM, 1998.

[28] U.-V. Marti and H. Bunke. The iam-database: an english sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition*, 5(1):39–46, 2002.

[29] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[30] P. A. Mueller and D. M. Oppenheimer. The pen is mightier than the keyboard advantages of longhand over laptop note taking. *Psychological science*, p. 0956797614524581, 2014.

[31] MyScript. Myscript website, 2016. `http://myscript.com/`.

[32] R. Nag, K. Wong, and F. Fallside. Script recognition using hidden markov models. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'86.*, vol. 11, pp. 2071–2074. IEEE, 1986.

[33] R. Plamondon and S. N. Srihari. Online and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on pattern analysis and machine intelligence*, 22(1):63–84, 2000.

[34] A. Poznanski and L. Wolf. Cnn-n-gram for handwriting word recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2305–2314, 2016.

[35] F. Raab, C. Wolff, and F. Echtler. Refactorpad: editing source code on touchscreens. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, pp. 223–228. ACM, 2013.

[36] A. Radenski. Python first: A lab-based digital introduction to computer science. *ACM SIGCSE Bulletin*, 38(3):197–201, 2006.

[37] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*, pp. 390–401. ACM, 2014.

[38] S. N. Srihari. Recognition of handwritten and machine-printed text for postal address interpretation. *Pattern recognition letters*, 14(4):291–302, 1993.

[39] S. N. Srihari, V. Govindaraju, and A. Shekhawat. Interpretation of handwritten addresses in us mailstream. In *Document Analysis and Recognition, 1993., Proceedings of the Second International Conference on*, pp. 291–294. IEEE, 1993.

[40] C. C. Tappert, C. Y. Suen, and T. Wakahara. The state of the art in online handwriting recognition. *IEEE Transactions on pattern analysis and machine intelligence*, 12(8):787–808, 1990.

[41] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 49–60. ACM, 2011.

[42] N. Tillmann, M. Moskal, J. De Halleux, M. Fahndrich, J. Bishop, A. Samuel, and T. Xie. The future of teaching programming is on mobile devices. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pp. 156–161. ACM, 2012.

[43] J. L. van de Snepscheut. Recursive descent parsing. In *What Computing Is All About*, pp. 101–120. Springer, 1993.

[44] J. Van West. Using tablet pc: Handwriting recognition 101. *Windows XP Home Web Site, http://www. microsoft. com/windowsxp/using/tabletpc/getstarted/vanwest03may2*, 2003.

[45] F. Zamora-Martínez, V. Frinken, S. España-Boquera, M. J. Castro-Bleda, A. Fischer, and H. Bunke. Neural network language models for off-line handwriting recognition. *Pattern Recognition*, 47(4):1642–1652, 2014.