An Ongoing Design of Dictating Programming That Accepts Noisy Input

Wennong Cai, Shigeru Chiba

We propose a "predict before parsing" architecture to let the dictating programming system be able to handle noisy input. We think the user experience of existing dictating programming designs is restricted by the quality of speech recognition systems, and the carefulness of users speaking. The eventual goal of the proposed architecture is to parse a dictating programming statement into its Abstract Syntax Tree, even if the statement contains noises.

The current progress of building this architecture and several novel points in the implementation are introduced in this paper. We conducted experiments on the "Statement Predictor", which is a component part of the architecture. The results show that an appropriately trained machine learning model can predict the type of noisy statement with high accuracy.

1 Introduction

Programming is essential nowadays, and there are lots of engineers making their products by typing the programming code on the keyboard. However, as electrical devices become more and more portable, the way to do the programming still requires a large keyboard to achieve the best efficiency. Portable devices such as tablets or smartphones are seldom used to type programming, as the tiny-size touch-screen keyboard is hard to type punctuation symbols or indents.

Dictating Programming is an approach to improve feasibility of programming on small devices. As it is common nowadays that a phone is equipped with a speech recognition system, being able to use the voice to support the programming improves efficiency of doing programming on portable devices.

In this paper, we first introduce the background of dictating programming and the issue of noises. Then we present our proposal of a flexible parser with the statement predictor as the solution to the issue. Ongoing experiment results are shown after that. Finally, we also introduce some related works on dictating programming, and compare their works with this paper.

2 Background of Dictating Programming

The idea of dictating programming is to use speaking instead of typing to enter the programming code. A dictating programming system generally has two sub-system parts. The first part is the speech recognition part, which is responsible for converting human speaking voice into text. The second part is an algorithm that translates the recognized spoken text into an existing programming language, or executes it directly. However, one uncertain point in such a system is that the recognized

---

* ノイズ入力を受け入れる口述プログラミングの設計

蔡 文農, 千葉 滋, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

spoken text is hardly as perfect as the algorithm input, because it often contains noises. We consider a dictating programming should allow a certain extent of noises.

Human Speaking and Speech Recognition are two primary noise sources considered in this research. We came to this decision by reading some simple Python code to the existing speech recognition system, in a format of dictating programming grammar we designed. The recognition results are collected and compared with the intended spoken statement, and the differences between them are concluded as the following two categories.

2.1 Noises from Human Speaking

Speaking a sentence strictly following the valid grammar is exhausting, especially for foreign people. People tend to speak a sentence in the most natural way they feel. Therefore, it is reasonable to assume dictating programming users can hardly speak the absolute correct grammar for the most input. Some of the most common examples are adding additional article words such as "the" or "a", using different tenses for verbs, and using singular or plural forms for nouns.

2.2 Noises from Speech Recognition

The existing Speech Recognition systems are not good enough. They have the possibility of making mistakes on recognizing words. Learning from large natural language datasets is also a key to high recognition accuracy for most speech recognition systems. However, such highly accurate recognition does not apply to this research, since there is no large available corpus of dictating programming speaking.

3 Proposal: Flexible Parser with Statement Predictor

To address the issue of parsing noisy dictating programming, we propose an architecture that contains a flexible parser with statement predictor, as shown in Figure 1. Details of overview design and the statement predictor are introduced in this section, while the parser component is still ongoing research.

3.1 "Predict before Parsing" Architecture

We made a hypothesis that for a noisy spoken statement, knowing its intended statement type in advance improves the accuracy of parsing it into its intended version. To examine this idea, the "Predict before Parsing" architecture is proposed.

The "Predict before Parsing" architecture is made of a Statement Predictor and a customized parser. The statement predictor is a trained machine learning model that predicts the statement type given the noisy input statement. The customized parser behaves like a normal parser when parsing a valid statement, but also attempts to parse the noisy statement by taking guess action based on the predicted type, instead of just throwing the exception. The system takes one input statement at a time, and outputs the Abstract Syntax Tree for this statement.

3.2 Input Pre-Processing

We propose input preprocessing steps for our proposed architecture. Proper input processing increases the accuracy of the Statement Predictor, and also improves dictating programming users experience.

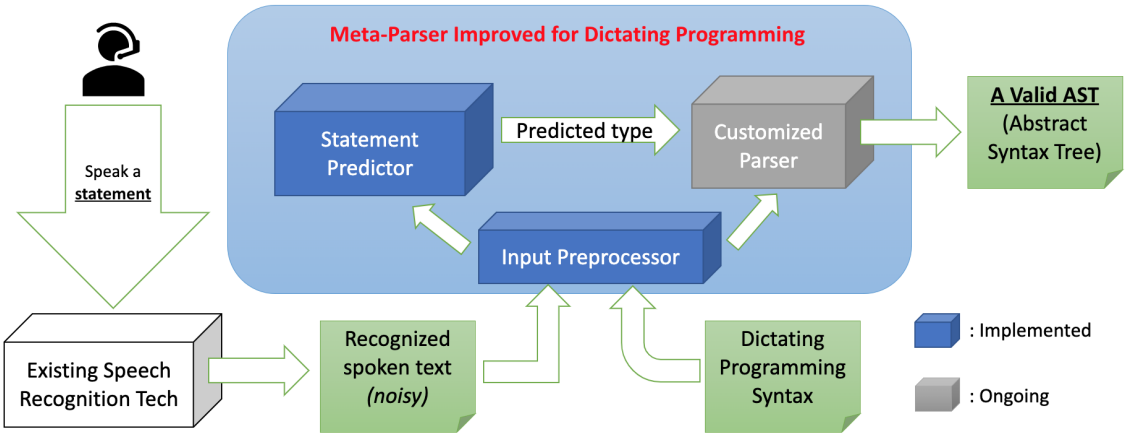

Figure 1 The "Predict before Parsing" Architecture

3.2.1 Programming Language for Dictating

A domain-specific language for dictating programming is designed in this research. Traditional typing programming language is hard to dictate, because there are many punctuation symbols in use. Therefore, we made our own language with grammar that does not use punctuation symbols. As an example, the following Python code:

```
1 if x > y:
2     print(x)
3 else:
4     print(y)
```

can be translated into the following equivalent dictating programming statement:

if x is greater than y then invoke print with x else invoke print with y

3.2.2 Phonetic-based Encoding

A novel encoding approach based on phonetics is proposed in this research. The traditional token-based encoding approach in natural language field is not suitable in the task of dealing noisy sentences, because it treats two similar tokens as completely independent. The normal character-based encoding captures the similarity if a token is mis-spelled, but the most common noises in dictating programming are mis-pronunciation. Therefore, instead of directly using character-based encoding, we turn every token into its phonetic-based format first, then map each phonetic character to a numeric value. The Oxford Dictionary API [1] is used to retrieve the phonetic format automatically.

3.2.3 Flatten Grammar Structure

We decide to divide any statement type containing nested structure into several independent types in order to flatten the nested structure. For instance, a normal FOR-LOOP statement usually contains the keyword "for", the condition expression, the nested body statements, and the optional keyword "else" with its nested body statements. In our dataset, we treat the FOR-LOOP statement as 3 separate parts: The FOR-LOOP header, which contains the keyword "for" as well as condition expression; The body statements, which are no longer part of the FOR-LOOP type but parallel independent statements; The FOR-LOOP ender, which is just a static value "end for".

Mismatching errors between header and ender are then handled during the program execution phase. The flatten strategy makes the parser output not directly contain any nested statement such as FOR-LOOP or IF. As the drawback of this strategy, syntax errors such as providing an ELSE state-

ment without an IF-header statement, cannot be captured during the parsing phase. We consider that this is acceptable since they can still be detected when executing the program.

Flattening nested statements is a key to reducing the difficulty of Statement Predictor training. The nested statement usually has a longer length than non-nested statement, and longer input makes the model harder to find useful information. Also, the existence of inner statements interferes with the prediction on the outer statement.

Rejecting all nested statements as input is also not feasible. Nested statement examples such as the definition of function or class, or the if-else statement, they are widely used in most existing programming languages. Rejecting the usage of those statement types makes a programming language not practical.

4 Experiments

We built a Statement Predictor and conducted experiments by training it in different approaches. The Statement Predictor is evaluated by the accuracy of predicting the statement type of a potentially-noisy input statement.

Also, a large-scale dataset is required to train a Statement Predictor. Due to the lack of existing popularly used dictating programming systems, there is no publicly available dataset of dictating programming corpus. The approach of making a dictating programming dataset based on existing programming corpus is introduced in this section.

4. 1 Model Used

The Long Short-Term Memory Model [7] is used as the Statement Predictor model in this paper. The LSTM model is a practical model to deal with sequential input data, but we found the LSTM in our task tends to focus on the last few tokens more than the beginning tokens. Therefore, we also apply the bidirectional LSTM model [6] to let the model focus on both ends of the sequential input.

4. 2 Dataset Generation

The dataset is generated from existing programming corpus into the format of our specific programming language for dictating as mentioned in section 3. 2. 1. The abstract syntax tree is constructed from the raw python code, then converted into dictating programming statements by traversing from each statement-level node. Each converted statement will be stored along with its statement type.

The python corpus made by the BIFI team is used in this research. BIFI [8] is research on the automatic correction of programming syntax errors. They collect 3 million Python 3 snippets which include valid code and syntax error code. We consider the syntax errors in typing programming, such as missing punctuation symbols, do not represent the situation in dictating programming. Therefore, we only make use of the valid code dataset as the input source of the dataset generation process in our research.

There are 18 labels in the original nested grammar structure, and 24 labels in the flattened grammar structure. To prevent extreme imbalance in the label distribution, each label can have at most 1 million entries.

The nested structure dataset contains 6520568 statements, and contains 19964626 statements after the grammar has been flattened. The reason for the size increment in the flattened dataset is because many statements type such as FOR LOOP statement have been divided into multiple independent statement types.

We have also made a smaller version of the flattened structure dataset by setting a smaller maximum cap on each label, since it contains much more data entries than nested dataset. The light version

Model	Using Noisy Identifiers	Grammar Structure	Dataset size	Accuracy
LSTM	No	Nested	6520568	97.2%
LSTM	Yes	Nested	6520568	26.5%
Bi-LSTM	Yes	Nested	6520568	33.9%
Bi-LSTM	Yes	Flattened	3587531	92.3%
Bi-LSTM	Yes	Flattened	19964626	97.6%

Table 1 Experiment Results.

of the flattened structure dataset contains 3587531 statements.

4. 3 Noise Simulation

Introducing noises to the training dataset is vital to increase the generality of the Statement Predictor. While there is no publicly available dictating programming corpus, noises have to be generated in an imitation approach as well.

For reserved keywords, several pre-defined noisy tokens have been selected that have similar pronunciations. The keywords are replaced by those noisy tokens based on a random probability.

For identifier names, the original identifiers used in Python code are already random noises, but most of them are hard to dictate or impossible to be converted into the phonetic format. We prepare a dataset with no identifier noises by replacing all identifiers with the same token. To simulate noises from identifiers, we use a collection of 10000 random nouns [5], and pick 9012 of them which have phonetic values in Oxford Dictionary API [1], then replaced identifier names with one of them randomly to make another dataset. The "Using Noisy Identifiers" field in Table 1 indicates which dataset is used.

4. 4 Results

All experiments are conducted in 4 epochs. Dataset is shuffled and divided into training, validation, and testing dataset in the portion of 8:1:1.

As Table 1 shows, the accuracy is high when every identifier is treated as the same, with only some keywords been modified to similar pronunciation words. It shows that under a relatively low noises environment, the LSTM can recognize the statement type based on a sequence of phonetic characters input.

However, when introducing the noises on identifiers by using the 9012 nouns mentioned in section 4. 3, the accuracy has a great fall. The model almost cannot recognize any labels but just gives random guesses on the most popular labels. Using a bidirectional LSTM slightly improves accuracy, but still remains at an impractical level.

We think the reason for low accuracy is that identifiers in programming do not make any contribution to determining the statement type. Also, reserved keywords in the nested inner statements do not help to predict the outer statement type as well. Having too much noisy information in the input makes the model hard to learn useful information.

In the flattened dataset, the number of identifiers decreases since there are no more inner statements. All keywords left in the input are useful to predict the statement type as well. Accuracy gets a large increment as the results show.

4. 5 Limitations

We considered 2 possible drawbacks in the current design that can be further improved in future research. The first drawback is the mapping from

phonetic characters to numeric values is intuitively designed in Phonetic-based Encoding. The similarity in numeric values does not accurately represent the similarity in pronunciation. The second drawback is the grammar of dictating programming used in this research is simply designed. The primary goal of designing this grammar is just to remove the necessity of using punctuation symbols, so all tokens used are normal words that do have phonetic form. A more delicate design of the grammar may reduce the difficulty of handling noises, and further improve the feasibility of dictating programming.

5 Related Works

There are existing attempts at designing the dictating programming system. VoiceGrip [3] designs a domain-specific language that is easy-to-dictate, and can be translated into the C language. It also designs an algorithm to recognize consecutive tokens as one single camel case identifier, which enhances the ability to name identifiers in dictating programming. In 2021, our previous dictating programming design, Natural Programming Language [2], introduced the usage of context-dependent pronouns. Using pronouns in dictating programming reduces repetitions of long expressions, which generally appear in typing programming.

However, dictating programming is still far from feasible to be used efficiently. We think one of the difficult points that cause this is the noise in the input. The existing designs of dictating programming are focusing on making the grammar easier to dictate, or implementing simple voice macros to achieve more complicated actions by speaking less words. They assume the input text is the already-recognized perfect text without any error.

Some existing works such as VoiceCode [4] do consider the incorrect recognition by providing the voice macro to retreat the user's last speaking, which indirectly solves the problem by increasing the user's burden. We aim to enable the dictating programming system to accept the spoken text even if it contains some noisy tokens.

6 Conclusion

In this paper, we introduced the issue of noises that keeps the dictating programming not feasible enough. The architecture design of a flexible parser with the statement predictor is presented. As ongoing research, we presented the current progress of some promising experimental results for the Statement Predictor implementation. Limitations of the Statement Predictor, such as phonetic encoding and grammar design, have also been presented as potential future research. For our next step, we will attempt to design the customized parser in our proposed architecture, which parses noisy statements given the predicted statement type.

参 考 文 献

[1] *Oxford Dictionaries API.* Retrieved from: https://developer.oxforddictionaries.com.

[2] Cai, W., Akiyama, S., and Chiba, S.: A Preliminary Design of an Easy-to-Dictate Programming Language with Pronouns, *Proceedings of the 38th JSSST Annual Conference*, 2021.

[3] Desilets, A.: VoiceGrip: a tool for programming-by-voice, *International Journal of Speech Technology*, Vol. 4, No. 2(2001), pp. 103–116.

[4] Desilets, A., Fox, D., and Norton, S.: Voicecode: An innovative speech interface for programming-by-voice, *In CHI'06 Extended Abstracts on Human Factors in Computing Systems*, (2006 Apirl), pp. 239–242.

[5] Eric Price, MIT: *10000 Word List.* Retrieved from: https://www.mit.edu/ ecprice/-wordlist.10000.

[6] Graves, A., Fernández, S., and Schmidhuber, J.: Framewise phoneme classification with bidirectional LSTM and other neural network architectures, *Neural networks*, Vol. 18, No. 5-6(2005), pp. 602–610.

[7] Hochreiter, S. and Schmidhuber, J.: Long Short-Term Memory, *Neural Computation*, Vol. 9, No. 8(1997), pp. 1735–1780.

[8] Yasunaga, M. and Liang, P.: Break-it-fix-it: Unsupervised learning for program repair, *Proceedings of the 38th International Conference on Machine Learning, PMLR*, Vol. 139, 2021 July, pp. 11941–11952.