Automatic Programming Code Explanation Generation with Structured Translation Models

by

Yihan Lu

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved February 2020 by the
Graduate Supervisory Committee:

I-Han Hsiao, Chair
Kurt VanLehn
Hanghang Tong
Yezhou Yang
Thomas Price

ARIZONA STATE UNIVERSITY

May 2020

ABSTRACT

Learning programming involves a variety of complex cognitive activities, from abstract knowledge construction to structural operations, which include program design, modifying, debugging, and documenting tasks. In this work, the objective was to explore and investigate the barriers and obstacles that programming novice learners encountered and how the learners overcome them. Several lab and classroom studies were designed and conducted, the results showed that novice students had different behavior patterns compared to experienced learners, which indicates obstacles encountered. The studies also proved that proper assistance could help novices find helpful materials to read. However, novices still suffered from the lack of background knowledge and the limited cognitive load while learning, which resulted in challenges in understanding programming related materials, especially code examples. Therefore, I further proposed to use the natural language generator (NLG) to generate code explanations for educational purposes. The natural language generator is designed based on Long Short-Term Memory (LSTM), a deep-learning translation model. To establish the model, a data set was collected from Amazon Mechanical Turks (AMT) recording explanations from human experts for programming code lines.

To evaluate the model, a pilot study was conducted and proved that the readability of the machine generated (MG) explanation was compatible with human explanations, while its accuracy is still not ideal, especially for complicated code lines. Furthermore, a code-example based learning platform was developed to utilize the explanation generating model in programming teaching. To examine the effect of code example explanations on different learners, two lab-class experiments were conducted separately

in a programming novices' class and an advanced students' class. The experiment result indicated that when learning programming concepts, the MG code explanations significantly improved the learning Predictability for novices compared to control group, and the explanations also extended the novices' learning time by generating more material to read, which potentially lead to a better learning gain. Besides, a completed correlation model was constructed according to the experiment result to illustrate the connections between different factors and the learning effect.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Language is the foundation of human communications, the medium of knowledge passing, and the precondition of civilization continuation. Although there are many languages in the world, automatic translators with computer science techniques are helping eliminate the barrier caused by different languages in the model society. With the help of computer science, humans are getting easier to communicate with each other around the world. But how about the communication between humans and computers? In this thesis, I explored the gaps to learn communicating with computers, or programming. I also built multiple assistants to help the learners.

Motivation

The computers are deeply involved in our modern life, they are “taught” to do almost anything done by humans in the past. They learn quickly, accurately, and never make mistakes, but the way they learn is not through human language. They learn through machine language, or “programming”. To this degree, the translators between humans and computers are called “programmers”.

There is no doubt that the explosion of information technology (IT) and its corresponding industry has deeply changed the world and our life, and there is no clue shows it will stop. Computers are learning to finish more jobs, so more teachers for computers, or programmers, are required. There have been 18.2 million programmers contributed to this industry, and the number is still growing. Even though the group of programmers is huge, the urge of well-educated programmers is still extreme. A fact is that programmer is believed to be one of the highest paid jobs in the market, while the

supply of high-quality programmers' human resource is still not fully satisfying IT companies.

Potentially, one of the reasons for the shortage in programmer human resources is the difficulties in programming education. Programmers must learn the language to communicate with a computer, and this learning process takes a long period, and costs a lot. Programming education is identical to traditional subjects in many degrees: the education of programming is relatively late for most students compared to other basic subjects (math, physics, etc.); it takes a large amount of cognitive load, and the effect of learning deeply depends on computational thinking (Lahtinen, E., et al. 2005). These factors lead to a fact that programming education is not friendly for novices.

Since computers are designed to automate jobs and improve efficiency, it is natural for educators to utilize computers to automate the process of programming education. Researchers have begun tackling the challenge in several ways, such as intelligent tutor to basic programming problem solving(Reiser, B. et al. 1985), providing recommendations of materials or examples to learn (Vesin, B. et al. 2012; De Oliveira, M. G. et al. 2013), or drawing connections for collaboration among learners (Serrano-Cámara, L. M. et al 2014). For most of the studies, their common premise is that humans are the best teachers in programming, so when humans try to involve machines to promote education, machines should learn from human teachers and try to teach as human as close as possible. In this way, programming education systems are able to serve a larger volume of learners, while the cost is reduced.

Nonetheless, programming education involves a large volume of learning materials, which are generated manually. It becomes a bottleneck that current supportive mechanisms still rely on manually created materials, whose volume is limited for extensive automatic education systems, and it is not guaranteed to solve all problems. Potentially, if the computers can be trained to generate materials for learners based on specific requirements, the benefits will be large.

To address this gap, this thesis presents a research for MG programming code explanation. This approach utilizes deep learning models in natural language translation, which has been improved for decades and relatively mature. The purpose of this research is to broaden the range of learning material for novices and make it easier to learn from code examples by generating explanations for any code example.

To evaluate the value of this model, a learning system was developed, and two lab-class studies were conducted. The experiment results revealed the effect of example explanations on programming novices compared to experienced students. Furthermore, a model of factors affecting the learning gain from code examples was built.

Research Questions

This research attempts to research:

- **What do novices need in programming learning?**
- **How to assist novices by explaining codes with machine generated (MG) language?**

- **How do programming learners benefit from machine generated (MG) annotated examples in declarative and procedural knowledge learning?**

To address this question, a set of questions are further raised.

What are the obstacles for novices in programming learning? This research question was partitioned into four in my research:

- *How do programming learners browse and learn from online discussion forums?*
- *How do programming learners explore and search on online discussion forums?*
- *How do novices seek information on search engines?*
- *How to better explain materials to learners?*

To tackle the obstacles for novices in programming learning, I set a series of studies to analyze the learning behavior of novices. The learner behavior analysis included the browsing behavior on discussion forum and the searching behavior on search engine. The searching behavior was further decomposed into query forming behavior and result browsing behavior to deepen the understanding of their obstacles. These studies identified a set of obstacles and requirements for programming novices and proposed corresponding potential assistance for future studies. Among the obstacles identified, the difficulty in understanding example code was one of the biggest according to the analysis result and student feedback. To solve this problem, natural language explanation of codes was proposed.

How to generate natural language explanations for programming code with deep learning methods? There have been a series of studies (Brusilovsky & Weber, 1996;

Brusilovsky, 1992; Burow & Weber, 1996; Faries & Reiser, 1988; Guzdial, 1995; Hohmann, Guzdial & Soloway, 1992; Linn, 1992a; Linn, 1992b; Redmiles, 1993) proved the value of examples in learning. However, studies (Lu, Y., Hsiao, I-H.& Li, Q. 2016; Lu, Y., Hsiao, I-H. 2017.) have shown that novices still have problems in understanding code examples, which indicates the requirement of code explanation. To research this problem, generating explanations with machine learning models is one of the ways to free the limitation of content volume. In this line of research, multiple models were built and compared to determine the most practical method of code explanation generation. Then the model will be evaluated in explanation readability and accuracy and compared with human explanation in experiments. The expected result of the experiments is that human explanation outperforms the MG content, but the MG content still helps novices in some level. I hypothesize that the deep learning models could generate code explanations as translations, and the quality of generated explanation is compatible with human explanation in a degree.

How do learners benefit from machine generated (MG) code explanations in declarative and procedural knowledge learning? To evaluate the potential learning effect of code explanations in education, the cognitive process should be investigated when a learner benefits from code explanations. Since the effect of code examples and explanations has been proved and evaluated in previous studies (Malan, K., & Halland, K. 2004, October; Burow, R., & Weber, G. 1996), in this research I will examine the effect of MG code explanations in learning. Besides learning performance, I am also curious about the detailed impact of MG content, such as its readability, explanation accuracy, and attractiveness compared to human explanation. Experiments will be

conducted to evaluate the MG contents entail to examine the impact compared to human content. I hypothesize that the learners could benefit from MG code examples by deepening their understanding of the code examples while drawing the connection between explanations and codes, and with a reliable quality, MG explanations can impact learners in the same way as human explanations.

Contribution

By investigating these research problems and its associated challenges, this research made the following main contributions:

- **Identify the obstacles for novice learners in programming.**

  Programming education is believed to be one of the fields unfriendly for beginners. As a result, studies have been established to identify the obstacles for novices, and methods were proposed to assist them. In my research, I conducted a series of studies focusing on the information seeking behavior of novices, and proposed systems to assist novice in self-learning with search engines. Furthermore, although the value of example has been proved in studies (Atkinson, R. K., et al. 2000; Brusilovsky, P. 2001; Chi, M. T. et al. 1989), we still identified the obstacle of learners in understanding code examples. Deep-learning-based code explainer was proposed correspondingly.

- **Utilized a deep-learning-based model to generate natural language explanations for code lines for the purpose of education.** Education based code explanation is currently a blank field, while researches have highlighted that the explanation of code is as important as the code itself in learning (Nasehi, S. M. et

al. 2012). A well selected and explained code example will stimulate learners to seek further information and think more. Although the model proposed in this dissertation still has space to improve, it will fill in the blank in programming education and become a potential tool for other programming learning platforms. This model will also play the role of a baseline for future research to reference. Besides the model, the data collected from crowd source is also valuable for future studies for model training and validating.

- **Evaluate the effect of code explanation in learning from code examples over a long period of time in real communities**. After building up the model, a series of experiments were established to evaluate the performance in real learning scenarios. In this dissertation, experiments were designed and established focusing on both novices and experienced learners to monitor their learning effect and real task performance. Potential long-term mechanisms can be conducted in the future to monitor their learning achievements and the change of learning customs.

Overall, the goal of these contributions is to be the first step in the creation of code learner assistants. Such systems aim to help learners search, learn, and practice programming. Deep learning models in the natural language have a potential to play the critical role in content recommending, information retrieving, content explaining and reasoning.

In the remainder of this thesis, I discuss related work in the domains of programming learning and code analysis, concluding with a description of the existing work at their intersection. I follow that with a brief discussion of my previous work in the

area. Furthermore, I discuss my research approach: system development, microtask workflows, content modelling, experiment designing, and user modeling. I conclude with a description of my plans for evaluating the effectiveness of this system. This thesis then concludes with a description of the expected timeline, risks, and an overall conclusion.

CHAPTER 2

RELATED WORK

Programming learning

Code analysis

Cheang, B., et al (2003) proposed to evaluate programming assignment from three perspectives: correctness, efficiency, and maintainability. According to these criteria, the research team built up a system to evaluate programming code automatically by examining executing results and measuring time and memory cost. The researchers also analyzed the similarity of submissions in lexical level to detect cheater, which is another degree of code analysis. This system is proved to be useful in real course situations, and even helped detect plagiarism. However, it could not evaluate programming code in the degree of maintainability, and it does not provide further hints to learners.

Helmick, M. T. (2007) proposed an auto grader based on the interface and reflection concept in Java. He configured a JUnit test class to use multiple implementations that can be enabled for classes containing all tests. By defining a set of tests, this grader can examine the results of code in standard format. The system also utilized Programming Mistake Detector (PMD) to analyze the coding style and quality in detail level, which is an improvement. The code style analyzer automatically detects unused variables or functions, redundant implementation, and empty blocks, which reflects the quality of code besides the execution result. However, this study did not investigate the code logic, and programming related feedback or hint cannot be provided.

Besides grading, an alternative in code analysis is to provide hints to help students move forward. Zimmerman and Rupakheti (2015) used a pq-Gram tree edit distance algorithm to match a student's program to its closest part in a set of solutions. By identifying the set of insertions, deletions and changes from given code to solution, this method provides hints for novices to correct their code. Rivers and Koedinger (2015) used tree edit distances to compute similarities between syntax trees of Python programs to identify adjacent states. Gross and colleagues (2015) similarly applied edit distances on syntax trees to infer clusters of computer programs and select the most similar sample solution for feedback.

To provide more for open-ended programming assignments, Price, T. W., et al (2016) provided next-step hint to students by analyzing the Abstract Syntax Tree (AST) of student's code with Contextual Tree Decomposition (CTD) algorithm. In this system each student's state is represented with AST, and contextual interaction networks (CINs) are built to "learn" from previous student submissions and generate hints for new students based on their learning. According to CTD algorithm, the current student's code will be compiled into AST, and matched to its closest previous success submission according to its CIN. Then, the route path from the current code to the successful submission will be calculated to generate the next step hint. According to the experiment, this work is proved to be significant in improving the quality of final solutions and reducing the likelihood of undoing assignment objectives.

Intelligent tutor is another promising direction in providing educational assistance in programming, especially with the help of code analysis. Paaßen, B., et al (2016)

utilized code analysis in syntax level, memory level and execution trace level to classify coding strategies, detect errors, locate errors, and finally provide hints in intelligent tutoring system. This work is unique since it analyzed programming code in different levels, which enabled them to provide accurate help in code correction.

Piech and colleagues (2015) propose a neural network-based approach to infer a vectorial representation of programs instead of AST, such that standard machine learning methods can be applied in the resulting Euclidean space. Like Paaßen, B.'s approach, Piech and colleagues intend to represent a program function with a direct mapping between input and output of program segments.

Recommendation in education

The educational recommender systems have many similarities compared to traditional recommenders. Both systems use user/learner history as the indicator of recommendation; both systems mainly use collaborative recommender and content-based recommender as fundamental methods. However, education recommender systems have unique challenges different from traditional recommenders.

There are two main unique challenges in educational recommender systems:

In the educational recommendation system, the learners are expected to take part in a continuous process of learning, usually long (days, months, even years). During this learning process, intelligent tutoring (Butz, C. J., Hua, S., & Maguire, R. B. 2006) can be applied, proper tasks/materials can be recommended, and even personalized learning route can be arranged (Labutov, I., & Studer, C. 2016). As a result, the user stickiness is critical to help learners finish the whole process of learn.On the other hand, recommender

systems of Amazon, movies, or books are designed for one-time purchase, their purpose is to maximize the probability for the current visitor to give an order, which is a single, quick decision (Herlocker, J. L., Konstan, J. A., Terveen, L. G., & Riedl, J. T. 2004). Although there are some recommendation system features involving the visitor's purchase history, its weight is still not as large as in the educational recommender system.

The other challenge for the education recommender system is the requirement of learning effect. The educational recommendation system does not only attract learners to choose information and material to learn, but also need to improve the learning effect. Compared to Amazon, movies, and books recommendation, the effect of learning is more important than the learner's interest. As a result, the cognitive load (Sweller, J. 1988), emotional status (Pekrun, R., 2011), and knowledge transfer mode are considered important after a learner receives recommendations.

Learning from examples

It has often been claimed that humans use solutions to previous problems to solve new problems or planning tasks. As a result, the power of examples in learning has been tackled and proved in many studies, especially for programming (Bartha, S., & Cheney, J. 2019; Brusilovsky & Weber, 1996; Brusilovsky, 1992; Burow & Weber, 1996; Chozas, A. C., Memeti, S., & Pllana, S. 2017; Faries & Reiser, 1988; Guzdial, 1995; Hohmann, Guzdial & Soloway, 1992; Linn, 1992a; Linn, 1992b; Redmiles, 1993). In most of the works, the only function of example-based programming systems was to help the student find a relevant example by having the student pick a static program example from a large list or search for an example using keywords.

In a system designed by Redmiles (1993) explanations for an example were "hardwired" into the system's code by the author himself. This method focused on the knowledge-based interface, and proved that the subjects using “EXPLAINER” performed the programming task more directly, and they performed it with less inter-subject variability, while subjects using the online documentation tool proceeded in a trial-and-error fashion and exhibited great inter-subject variability. This result indicated that the explainer tool is becoming increasingly important in the software field.

ELMPE (Burow & Weber, 1996) applied an Artificial Intelligence approach to provide learning assistance according to the learner’s behavior history. This work proved that debugging and helping facilities specifically designed for beginners are integrated into the programming environment.

Brusilovsky, P. proposed an online programming learning platform named WebEx (2001), which enabled teachers to use example-based programming approach with heterogeneous classes. The idea of WebEx was to provide self-explaining examples instead of bare code. According to Brusilovsky, P., the explanations served at least two different purposes: First, it explains the student the meaning of each program line and its role in the overall solution of a programming problem; Second, it comments on a particular way of using language constructs in every line of code thus bridging the gap between student general knowledge about programming language constructs and practical skills of their use for solving programming problems.

My work is unique in several fields compared to these related works. First, I designed a completed model generating explanations of code, which broadened the

application of explained example in programming learning; Second, I developed a learning system with the help of code examples to teach novices programming concepts; Finally, I conducted an experiment to fully reveal the cognitive process of learners when learning from examples.

Programming novice education

Novices and experts have differences in learning patterns from the perspective of both behavior and cognitive (Wiedenbeck, S. 1985; Weiser, M., & Shertz, J. 1983). In other words, the learning patterns shift when a novice transfers to an export. In education, it is a common method to improve the learning effect by providing hints to lead novices "learn as an export".

In these related works (Denny, P. et al. 2014; Pettit, R. S. et al. 2017; Becker, Brett A. 2016), the researchers tried to compare the number of "failed submissions" between experiment and control group, in which the expectation is that the error explanations could reduce the number of failures. However, the experiment results did not meet the expectation well.

Among the related works, Denny, P. et al. (2014) conducted enhanced feedback to help students solve syntax errors in their code. As a result, they claimed that the enhanced feedback did not reduce the number of non-compiling submission. Pettit, R. S. et al. (2017) generated C++ enhanced error messages by analyzing the submissions, then generated enhanced interpretation of errors and hints for current students according to code similarity. In their result report, there was no significant difference in submissions failed to compile after enhanced messages were introduced. Similar results were also

reported by Becker, Brett A. (2016), who generated enhanced error message of Java for novice students. In my previous work, I provided searching hints to novices to reduce their "try and failure" iterations in search. Like code compiling studies, it is also a conclusion of our previous studies (Lu, Y. & Hsiao, I-H. 2017) that novices could receive little benefit in reducing the number of iterations of their "trial and failure" in searching.

This result does not mean that explaining errors has no meaning. On the contrary, it could help a lot, but not reflected in the number of failed submissions. Pea, R. D. pointed out that debugging errors is a "constructive and plannable activity" that applicable to any problem-solving, learners are not supposed to be error-free, but learn from errors instead (Pea, R. D., & Kurland, D. M. 1984). The lesson I learned from these studies is that I should indeed enhance the activity of "learning from errors" and conduct environments that can help learners focus on errors, understand more about errors, and be more motivated to solve the errors.

In our study, one step forward could be explaining not only correct code, but also code with bugs or errors. Learners can use our tool to explain their own code, identify what is not expected, and try to fix it by themselves.

Natural language processing (NLP)

Natural language processing (NLP) is a subfield of computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human (natural) languages, especially how to program computers to process and analyze large amounts of natural language data.

Challenges in natural language processing frequently involve speech recognition, natural language understanding, and natural language generation.

NL Understanding

As a tool connecting computers and humans, NLP techniques must understand human language first. In traditional NL systems, the understanding of NL is refined into multiple levels including syntax, semantics, context, and reasoning (Bates, M. 1995).

The syntax analysis is relatively mature in NLP. Besides checking the input format, syntactic analysis has two uses: one is to simplify the process of subsequent components as they try to extract meaning from the input; the second use of syntactic analysis is to help detect new or unusual meanings (Bates, M. 1995). By applying NL templates with tokens and formulas, terms in NL are replaced by class expression so that the semantics can be inferred in different logic representations (McAllester, D. A., & Givan, R. 1992).

The semantics component in NL understanding aims to infer the "meaning" of the language input. Three representation logics of meaning are widely used in semantic analysis: propositional logic (most frame-based semantic representations are equivalent to this, since they do not allow quantification); First- Order Predicate Logic (FOPL, which does allow quantifiers); and various representations that can handle expressions not representable in FOPL (McAllester, D. A., & Givan, R. 1992). However, the semantic meaning is still greatly influenced by the context in which the words are used and by the purpose the words are intended to achieve context, so context analysis is necessary to capture the logic in language.

Language context analysis is the least well understood and most difficult aspects of NLP (McAllester, D. A., & Givan, R. 1992). Unlike context in speech, which is quite localized in time, NL context is all pervasive and extremely powerful; it can reach back (or forward) hundreds of words; it considers wide range of contents around the target text to understand elliptical sentence fragments, dropped articles, false starts, misspellings, and other forms of nonstandard language. In practice, there is usually a parameter limiting the range of context considered.

Reasoning is the final purpose of NL understanding. It involves deep logic of knowledge collecting, inferring, and querying from representations of previous knowledge collections. The deducting of semantics is investigated with negative polarity, concord marking (Dowty, D. 1994, November), and concept net (Liu, H., & Singh, P. 2004, September) for decades. In this study, the reasoning level is not reached, instead a semi-translation model will be conducted without semantic reasoning.

NL generation

The goal of the NL system is to interact with humans in NL, so there must be an interface to interact with humans with natural language. The generation of natural language is as critical as understanding natural language to this degree. However, in some cases the expected output is not easy for machines to generate. For example, taking the input “How to calculate Fibonacci sequence with recursion”, should that be interpreted as a request to list the codes solving the problem, or should the effect merely to change the discourse state so that subsequent queries will take into account the intended itinerary ("Why the recursion function parameter is coded in this way?"), or should it cause the

system to plan and produce a response to clarify the user's goals ("Do you want to see all the codes?")?

Is the decision to treat the input as a command, or merely as information to the system, really part of the "NL understanding" process, or part of the backend process that takes place after understanding, and is independent of it? The same questions can be asked about the response planner and response generator components in Figure. 1. Is it a part of the NL processing (a part that just is not used when the input comes from text instead of an interactive user) or is it part of the post-NLP system? Computational linguists do not agree on where to draw these boundaries or on how to represent the information that passes between them (Bates, M. 1995).

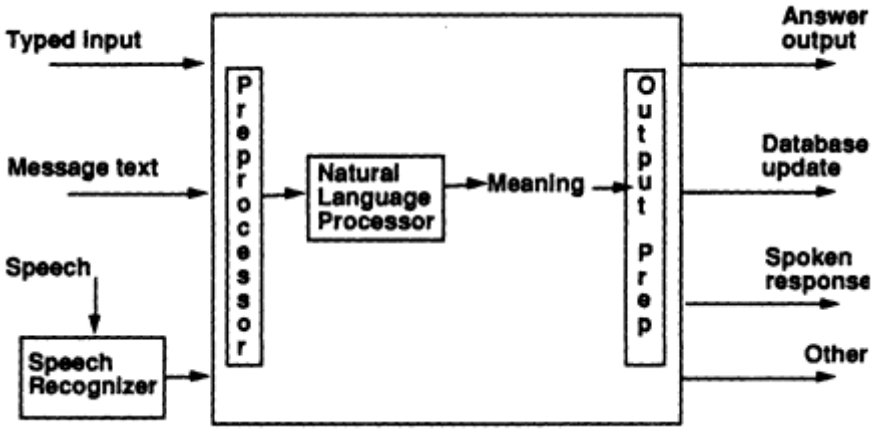


Figure 1. A Generic NL System. (Bates, M. 1995)

Reiter E. (1994) surveyed a set of NL generation systems to investigate psycho-linguistically plausible and consensus NL generation architecture. He analyzed and compared NL systems in architecture, content determination, sentence planning, surface generation, and morphology and formatting. Then he reached a controversial conclusion

that although it is believed that a certain linguistic phenomena is best handled by a certain architecture, the next evolutionary process in NL is not about “proper” handling of specific cases, but more related to non-deterministic human-like language generator.

Code explanation with NLP

In programming, NLP technology has also been considered as a tool to explain machine language.

Wong, E. et al. (2015) proposed “CloCom” to generate code comments by detecting existing code with comments that are similar to the current Java code. They obtain the Abstract Syntax Tree (AST) of the Java source code with Eclipse AST-Parser, then calculate the similarity between pairs of codes. By extracting and selecting comments from similar codes and ranking them by similarity and comment quality, CloCom generates at most two comments for each code input. This study involved term similarity and filtering, which are basic NLP techniques, so improvement space is created by utilizing more intelligent algorithms.

Hu, X. et al (2018) chose a different approach to generate code comments. They utilized AST and structure-based traversal (SBT) to transform programming code into sequential structured language and build up deep learning models with RNN and LSTM to process the programming and translate it into natural language. By evaluating the result with BLEU-4 (Papineni, K. et al. 2002, July), the research team claims to have a better performance compared to their baseline. In this work, the model is designed to generate comments for completed methods for industrial purposes, so they used existing

code comments as training data. On the other hand, our study focused on education purpose, so industrial comments are not proper in general cases, and explanation for single lines is more agile and helpful in our scenario. Also, Hu, X. and his team utilize LSTM, which is a sequence-based model, by transforming the tree-structured code into a sequence, while I try to build the model with tree-structure, which is a novel and valuable attempt.

Deep learning in NLP

Deep learning, or deep neural network, has been used in NLP for decades in different tasks including part-of-speech (POS) tagging, chunking, named entity recognition, semantic role labeling, language models, and semantically related words, etc. (Collobert, R., & Weston, J. 2008, July). In this work, I focused on the existing translation models, and utilized them for new purposes. Among all translation deep learning models, long short-term memory is widely used and performs well.

Long short-term memory (LSTM)

First proposed in 1997, LSTM is a model based on RNN, an extension of conventional feedforward neural network. Specifically, LSTM cells are capable of modeling long-range dependencies, which other traditional RNNs fail to do given the vanishing gradient issue (Hochreiter and Schmidhuber, 1997). The LSTM architecture addresses this problem of learning long-term dependencies by introducing a memory cell that can preserve state over long periods of time. Each LSTM cell consists of an input gate *i*, an output gate *o*, and a forget gate *f*, to control the flow of information.

Studies have been established for different purposes in NLP utilizing LSTM. Zhou et al. (2016) indicates the benefit of using such networks to incorporate contextual information in the classification process. Tai et al. (2015) further implemented a tree-structured LSTM to improve its reasoning ability, which lead to better sentiment classification and semantic relatedness

Meanwhile, sequence-to-sequence (seq2seq) models (Sutskever et al., 2014, Cho et al., 2014) have enjoyed great success in a variety of tasks such as machine translation, speech recognition, and text summarization. Neural Machine Translation (NMT) was the very first testbed for seq2seq models with wild success utilizing LSTM. This model is based on RNN Encoder–Decoder, which takes input into encoder, captures the deep logic as a "thought" vector, then exports into another language with decoder.

In this thesis, LSTM is involved to build up a translation model, which takes programming language as input, and output human language, English, in our case.

Besides the sequence base LSTM, the tree-structured LSTM was also proposed by Tai, K. S. (2015). The difference between sequence-based model and tree structure-based model is illustrated in Figure 2.

To conquer the limitation of the LSTM architecture that they only allow for strictly sequential information propagation, the N-ary Tree-LSTM is chosen to connect the tree structures. The variants allow for richer network topologies where each LSTM unit is able to incorporate information from multiple child units. As in standard LSTM units, each Tree-LSTM unit (indexed by j) contains input and output gates i_j and o_j, a memory cell c_j and hidden state h_j. The difference between the standard LSTM unit and

Tree-LSTM units is that gating vectors and memory cell updates are dependent on the states of possibly many child units. Additionally, instead of a single forget gate, the Tree-LSTM unit contains one forget gate f_{jk} for each child k. This allows the Tree-LSTM unit to selectively incorporate information from each child. For example, a Tree-LSTM model can learn to emphasize semantic heads in a semantic relatedness task, or it can learn to preserve the representation of sentiment-rich children for sentiment classification. As with the standard LSTM, each Tree-LSTM unit takes an input vector xj . In our applications, each x_j is a vector representation of a word in a sentence. The input word at each node depends on the tree structure used for the network. For instance, in a Tree-LSTM over a dependency tree, each node in the tree takes the vector corresponding to the head word as input, whereas in a Tree-LSTM over a constituency tree, the leaf nodes take the corresponding word vectors as input (Tai, K. S. et al. 2015).

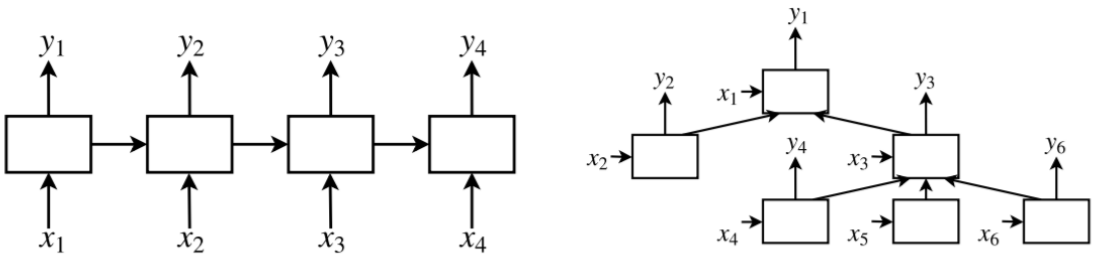


Figure 2. Left: A Chain-structured LSTM Network. Right: A Tree-structured LSTM Network with Arbitrary Branching Factor (Tai, K. S. et al. 2015).

Furthermore, Shido, Y. et al. (2019) proposed Multi-way Tree-structured LSTM, which is an extension of tree-structured LSTM to handle a tree containing a node having an arbitrary number of ordered children to interpret programming code AST. The method

was to adopt bidirectional LSTMs on each gate to encode the information on forward children to backward children and vice versa (Figure. 3). Shido, Y, et al also conducted an experiment to compare the Multi-way tree-structured LSTM with sequence LSTM and other tree-structured models with a data set of 243183 samples for training, 29155 for validation, and 33010 for testing. In the experiment, comments were utilized as explanation of codes and matched with corresponding code. The result shows the Multi-way tree-structured LSTM outperforms other models in BLEU-1 to BLEU-4 and other metrics.

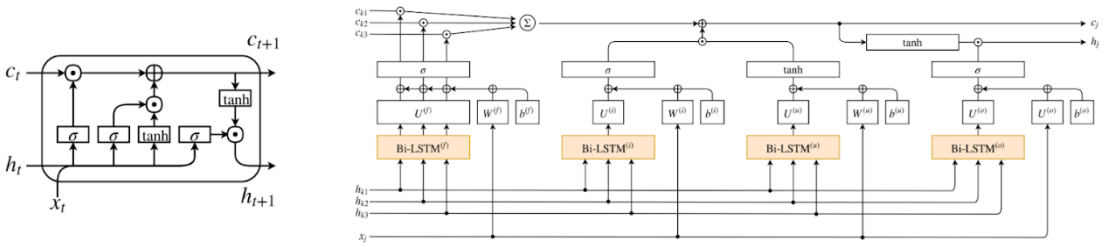


Figure 3. Node Structure of LSTM (Left) and an Example Node Structure with Three Children in Multi-way LSTM (Right) (Shido, Y. et al. 2019)

In this thesis, I refer to the Multi-way tree-structured LSTM and train it with human explanations intended to teach novices about programming.

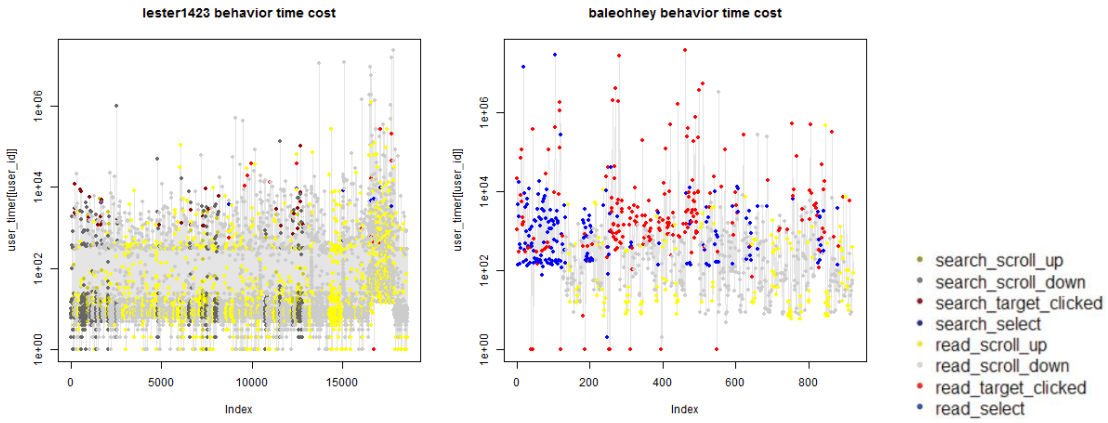
CHAPTER 3

METHODOLOGY IN OBSTACLE IDENTIFICATION

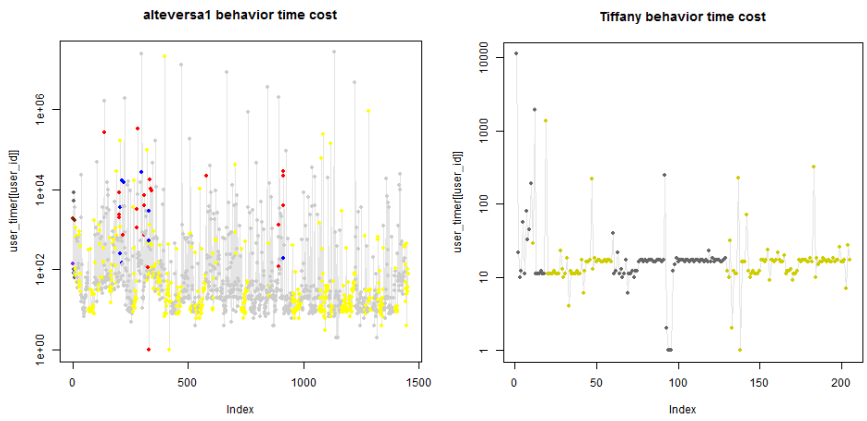
To identify the obstacles for novices in learning and solve them, this research followed a thread of problem identifying and solving.

Discussion forum learning behavior analysis

The beginning of this programming education research was the discussion forum learning analysis (Lu, Y., Hsiao, I-H.& Li, Q. 2016). I design engines to capture programming learners' activities on StackOverflow site, such as problem verbalization in queries, query revision and other information seeking processes. Then I collect a semester long of informal programming learning activities from a programming discussion forum. After serial data analysis and K-mean clustering, the patterns in students' behavior were revealed that the students are clustered into four groups (Figure 4). In the figures, the x axis is the index of each operation, and the y axis is the time cost on each operation.



(a) Hyper-user (b) Selecter



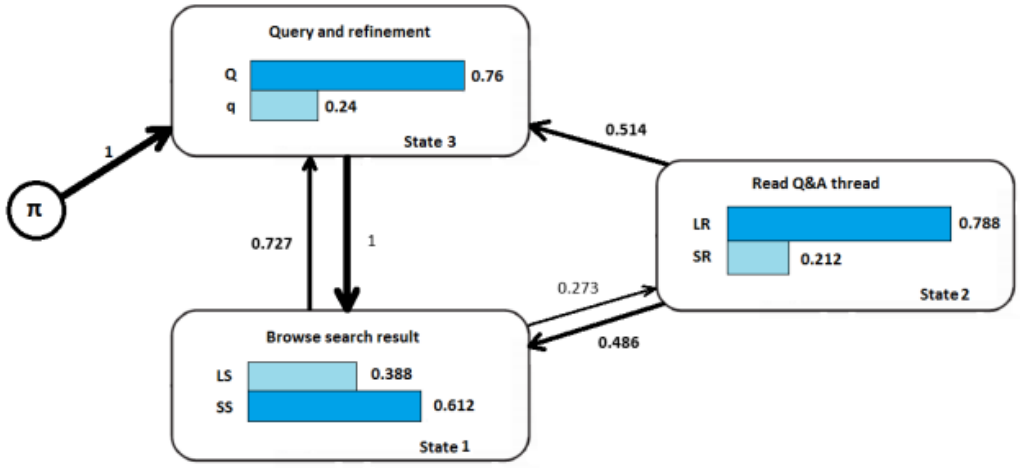
(c) Impatient-reader (d) Passive-user

Figure 4. Typical User Timelines in Each Behavior Cluster

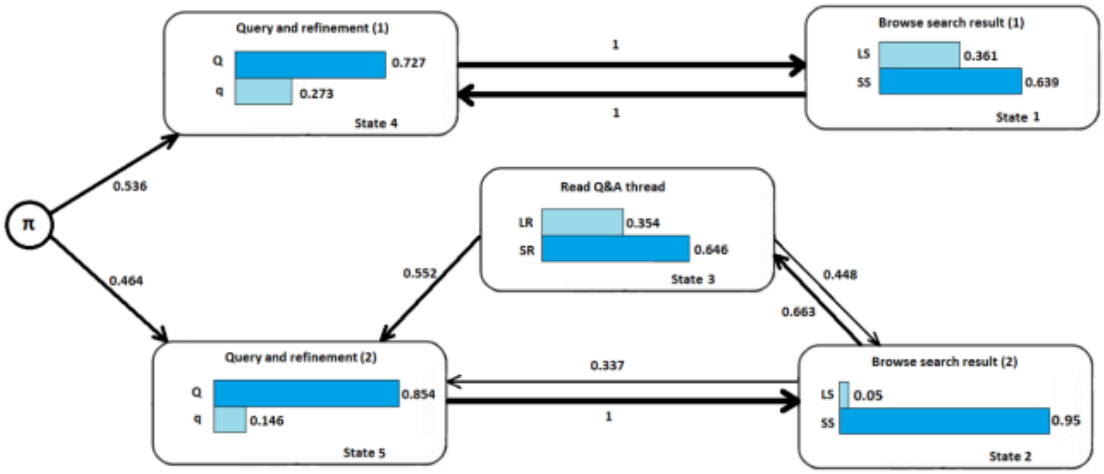
In this research, the result showed that many programming novices spend a lot of time browsing search results and reading, and there are usually multiple iterations of "try and failure" in query searching. All the study results shed light on programming learners seeking for learning resources from extensive online discussion forums. I anticipate this work serves as guidelines for educational technologists to design better effective tools to facilitate learning via programming information seeking process.

Information seeking behavior analysis

Following this clue, I continue to investigate the information seeking behavior of programming learners (Lu, Y. & Hsiao, I-H. 2017). In the next study, I studied how programming novices explore and search on online discussion forums. The method was to collect novices' intentions and search logs. I model their information seeking activities by using Hidden Markov Model and mining the post of their readings (Figure 5).



a. HMM for Advanced Student Browsing Behavior



b. HMM for Novice Student Browsing Behavior

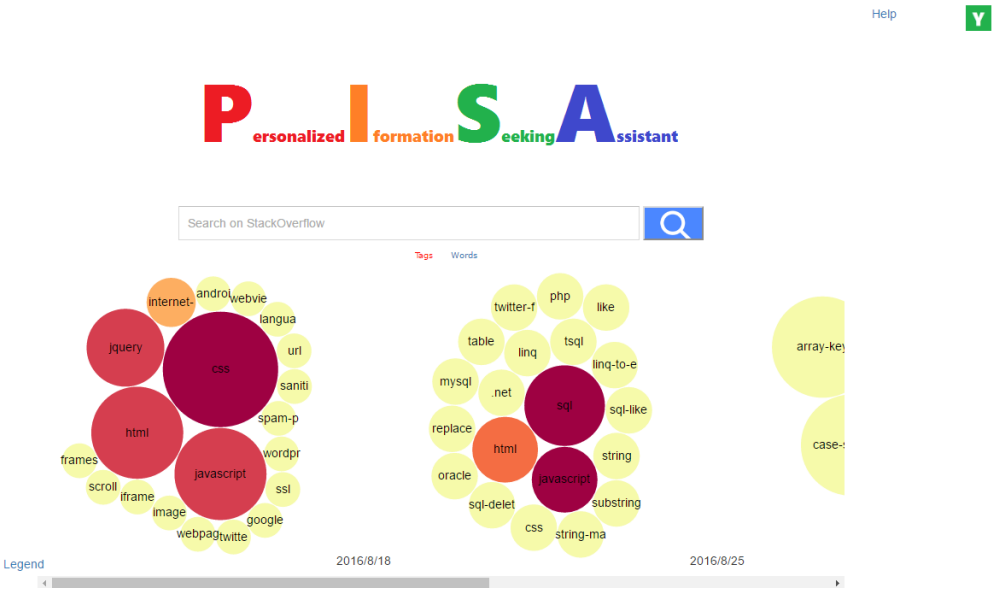
Figure 5. Information Seeking Behavior Modeling for Novice and Advanced Students

The results indicated that novices had significant different behavior patterns compared to experienced learners. Different from searching, novices do more skimming, while experienced learners read contents more carefully. Analysis also reflected that if

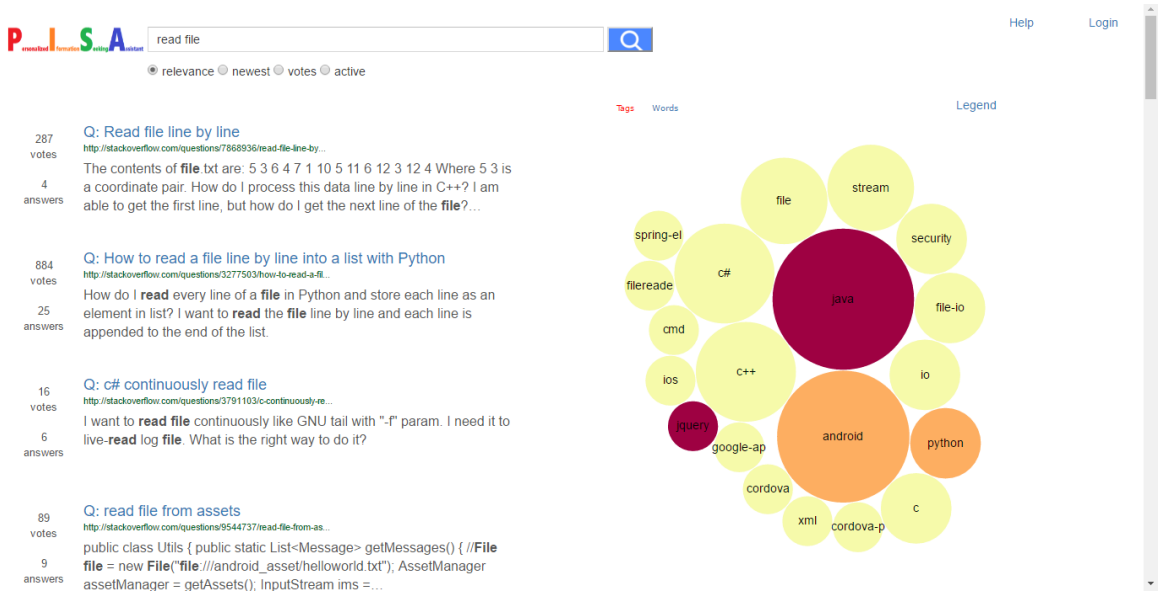
novices can read as well as advanced students, they can learn as much as advanced students according to the results of learning evaluation. This fact suggested that novices indeed require specific assistance in generating search queries and filtering helpful information among the search results.

Information seeking system design

In the next research, I design PiSA (Personalized Information Seeking Assistant) (Lu, Y. & Hsiao, I-H. 2017), a programming related search platform, to facilitate learners look for programming related information (Figure 6).



a. PiSA Search Page with Query Term Recommendation Considering User History and Social Features



b. PiSA Browse Page with Searching Results Summary

Figure 6. PiSA System Interface Example

This platform is designed visually to combine social features, user history, and to personalize query recommendation and conclude frequent words in search results to help learners filter results and refine their queries. However, the limitation of this work is obvious that in most cases, learners prefer to use general search engine rather than specific tool designed for learning programming due to their inertia in custom and the design detail issues in learning tools according to their feedback.

As a result, to provide further assistance to learners, I move on to a public search engine, Google, to analyze the novices' searching behavior and identify their common obstacles. In this study, I conducted a lab study to investigate students' programming information seeking behavior via Google search engine, in which students were given a programming task with limited time, and they were also required to report their online

search process including search query and web pages browsed. I analyze the web pages they browsed, model student's behavior, and cluster them into groups with different search tactics. The results show that the web pages they browsed during the task consisted of either conceptual knowledge or coding technical content. The students who performed better would browse more about conceptual knowledge. Students who set more and smaller units of sub-goals outperformed the students with fewer and larger sub-goals. In this study, I also observed that learners have obstacles after retrieving valuable material, because of the lack of prior knowledge to understand the material.

Human Language-Programming Code Connection

To address the new obstacle identified, I turned to material interpretation analysis (Lu, Y., & Hsiao, I. H. 2018, July). To explain materials for learners, I investigated semantics in programming. In the first study in this field, I utilized deep learning models to build up connections between descriptive language and programming concepts. By analyzing existing codes and descriptive comments, I was able to predict concepts involved given descriptive language with a neural network model.

To further assist novice learners, I finally reached the current study. I build up another model to interpret not only human language materials, but also programming codes. The whole process includes data collection from crowd source, build up model, pilot study evaluation, and long-term lab study evaluation.

CHAPTER 4

METHODOLOGY IN PROGRAMMING CODE EXPLANATION

Crowdsourcing

Amazon Mechanical Turks (AMT) is a crowdsourcing website for businesses (known as Requesters) to hire remotely located "crowd workers" to perform discrete on-demand tasks that computers are currently unable to do. It is operated under Amazon Web Services and is owned by Amazon. This platform is easy to use, script driven, and safe for both requester and workers. In this work, I utilize AMT to collect human code explanations. In AMT, the data collection has two phases: qualification phase and HIT phase.

Qualification

In this work, crowd source is utilized to collect true human explanations for code line, which will be used as ground truth in neural network training. To this degree, the quality of the explanation is critical, code experts are expected as the data source. For this purpose, a qualification is conducted for the Turks on the platform.

There are two purposes to establish the qualification. First, the Turks with little programming knowledge must be excluded, since the content collected are assumed to be experts explaining code for novices. Second, the programming knowledge level is also critical in model training. There is a potential in the future that the model could positively learn from dynamic content input with a weight representing the quality of the content, and the knowledge level of the provider is a good representation.

The qualification is designed as a survey with 20 questions, in which each question is multiple choice. The topic of questions covers programming related knowledge from basic to advance to differentiate the Turks in different levels. Turks must achieve at least 60% of the questions to pass the qualification and participate in the data collection phase.

Data Collection Design

In data collection, Turks are expected to read a complete java code, and explain the code line by line. To organize the tasks, AMT partitions the tasks in HIT level and assignment level, Figure 7 is an example. In this example, each java code is the original data awaiting human reply. The first step is to process the original data into a HIT, which is formatted with basic descriptions, highlights of what to answer, how to answer, and even good examples and bad examples. After formatting a HIT, it is published to the qualified Turks with a limited number of assignments, 3 in the example. After 3 Turks take all the assignments, this HIT is closed.

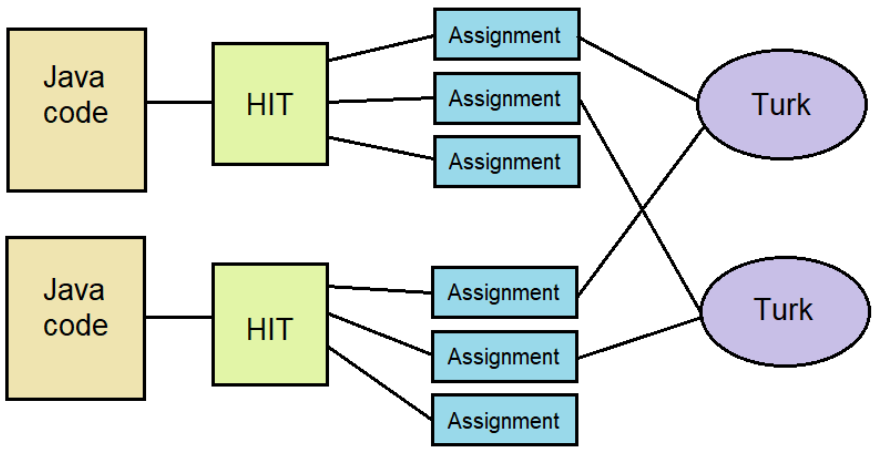


Figure 7. Amazon Mechanical Turks - Task Structure

In the code explanation collection, the original code data set is a collection including code example in programming textbooks and practical open Java projects on GitHub. Before posting, the codes are filtered by their degree of completion and length. Only the codes with complete logic (different from codes like “hello world”) and a medium length (10 to 20 lines) are left for AMT. An example of HIT to explain the first two lines in a code is shown in Figure 8. In real HITs, Turks will explain every line in a code.

Figure 8. A HIT Example for the First Two Lines in a Code

After filtering, each code is processed by a modified compiler to remove comments and standardized format. Then a script is executed to format each code into a HIT, where each code line is required to be explained as a short-answer question. The

HITs are designed to have a one-week lifespan, which means after a week, even though the three assignments are not all finished, the HIT will still be closed.

After 6 weeks of data collection, more than 450 Turks participated in the data collection, over 800 assignments are finished on ATM platform, and 8569 lines of code are explained.

Pilot study

The main contribution of this work is based on the code explanation deep learning models. After data collection, the translation model will be trained and evaluated in pilot study.

Model Training

In this work, a Neural Machine Translation (NMT) model is trained in iterations. In each iteration it first took training data to form the model, then validated it for the next iteration. The Figure 9 is an example of NMT translation on GitHub. To train the NMT model, both the code data and the explanation data collected are pre-processed to extract their vocabulary. Then the whole data is split into training, validating, and testing data set.

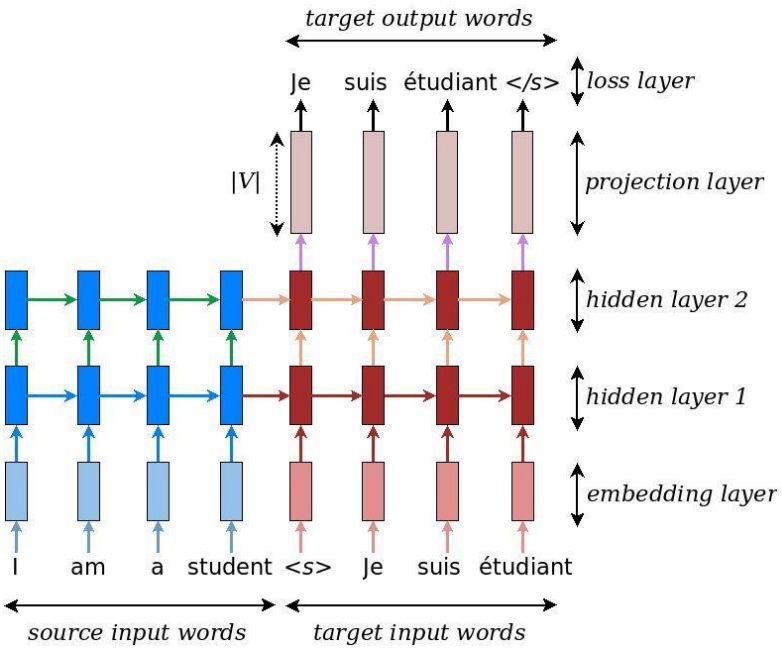


Figure 9. Neural Machine Translation - Example of a Deep Recurrent Architecture

In this work, one more step is required since the variable names are more variance in programming language, which should not be considered as part of vocabulary. To correctly tackle this set of terms, a tokenizing process is applied before the training. This tokenizing process is close to compiler tokenizing, in which an internal vocabulary is maintained to filter "unknown" term. These unknown terms are masked with specific token, and matched to the explanation terms, or translation.

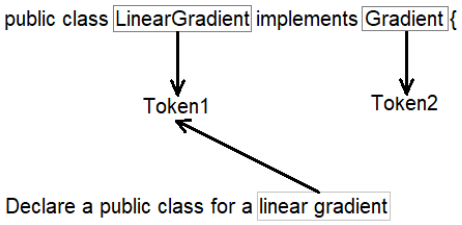


Figure 10. Example of Token Masking Match

Furthermore, considering the structure of human language sentence is in a tree structure, which is agreed by programming code line, a translator would perform better if it can translate a tree structure into another. With this assumption, the tree-structured LSTM is employed as an advanced model (Tai, K. S. et al. 2015).

Since tree-structured LSTM has been compared with sequence LSTM in many previous studies (Shido, Y. et al. 2019; Tai, K. S. et al. 2015), we will only focus on the impact of tree-structured LSTM in example learning.

Study Design

After model training, the code lines in test data set is taken as the input to the model, and MG explanations are collected to compare with human explanations.

In the experiment, 6 complete codes with 103 lines in total were randomly sampled in the test data set, and 4 programming experts participated in the study. Among the 103 sample lines, 50 of them were randomly selected and explained by humans, and the other 53 were explained by tree-structured LSTM. The pilots did not know the source of each explanation. The pilots were required to evaluate each explanation in two perspectives: whether it is readable (Y=readable; N=unreadable) and the level of accuracy (1 to 5, 1=totally inaccurate; 5=totally accurate).

Study results

In Figure 11, the results of pilot study are illustrated.

In the left figure, the readability was a binary evaluation for the pilots as “Y” or “N” to judge whether they can understand an explanation. After taking “Y” as score 1

and “N” as score 0, the average of all pilots is taken as the evaluation of different explanation sources. The result shows that most of the explanations are readable, in other words, they can understand the explanations. Meanwhile, the tree-structured model outperforms sequential model, and both outperforms the human baseline. The average Cohen’s kappa among each pair of the 4 pilots is 0.37. This result shows that MG explanations can compete with humans in readability to some degree.

In the right figure, the comment accuracy is calculated similarly as readability. The average Cohen’s kappa among all pairs of the 4 pilots is 0.27, which is not strong considering there are 4 pilots selecting from 5 options for each code line. On the other hand, the distribution of results shows that human explanations as a relatively better performance since the accuracy are skewed to the right. Compared to humans, machine explanations have a distribution of two peaks, which indicates there is a certain part of code lines not well explained by model.

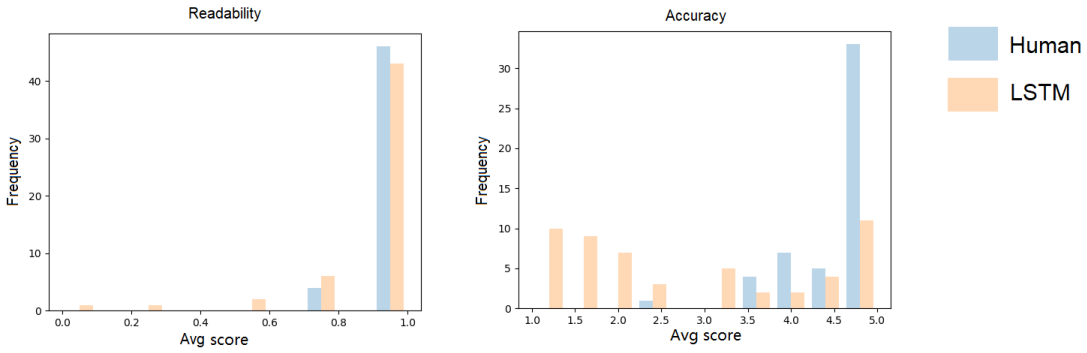


Figure 11. Pilot Study Result of Readability and Accuracy

Table 1

Comparison of Human and LSTM Model in Readability and Accuracy.

	Readability (0 to 1)	Accuracy (1 to 5)
LSTM	0.92 +- 0.20	2.82 +- 1.48
Human	0.98 +- 0.07	4.62 +- 0.57

The examples of different explanations given by human and machine are as follows:

- Example of both human and machine success to explain

```
Code: public class Stopwatch{
```

Human explanation: ***Create a class with class name as Stopwatch***

Model explanation: ***Declare a public class***

- Example of both human and machine fail to explain

```
Code:  for (int i = numbers.size()-1; i  > = 0; i--)
result.println(numbers.get(i));
```

Human explanation: ***Usies the PrintWriter to print out each number from last to first to the result.dat file.***

Reason of failure: too much information, not necessary for novices, lack of further explanation

Model explanation: ***add the listener to the question***

Reason for failure: irrelevant terms involved.

- Example of machine outperforms human

```
Code: while (in.hasNext()) {
```

Human explanation: ***while statement executes the code in the while block unless it returns a boolean value true to the expression in.hasNext()***

Reason for failure: too complicated for novices, logic of boolean value returning is wrong.

Model explanation: ***while loop for all in the input***

- Example of human outperforms machine

```
Code: Rectangle box = new Rectangle ( 5 , 10 , 20 ,
30 ) ;
```

Human explanation: ***Constructs a rectangle and saves it in a variable***

Model explanation: ***Constructs a NoSuchMethodException and saves it in a variable***

Reason for failure: irrelevant term "***NoSuchMethodException*** " involved.

In these examples, Human explanations are labeled inaccurate mostly because they are not clear or too complicated illustrating complex concepts, while model explanations fail because they involved irrelevant terms.

CHAPTER 5

LAB EXPERIMENT DESIGN

The result of pilot study indicates that the MG code explanations are compatible in quality, which has answered the first part in the research questions. However, it is still important to measure the effect of MG explanations in learning and compare with human explanation in impact to answer the second research question. I planned to investigate the effect by conducting a lab study, which measures the value of MG content in educating novices in real classroom for a long-time period.

Implementation

At the beginning of the whole lab study, there was a background survey together with a pre-test given to examine the background knowledge of the learners, which helped to measure the improvement of students during the study. In the pre-test, programming related questions involving all concepts they would learn in the following two months was tested. For each concept, learners were given specific multiple-choice questions to test the level of their knowledge and followed by another question asking their confidence level from 1 (lowest confidence) to 5 (highest confidence).

In the lab study, there were a set of sessions for the students:

In the first session, the students were given a general introduction about the lab, including the purpose of the lab, the sessions they would experience, and the time setup for each session. The introduction also interpreted the meaning of the time bar at the top of the page to help them manage their time.

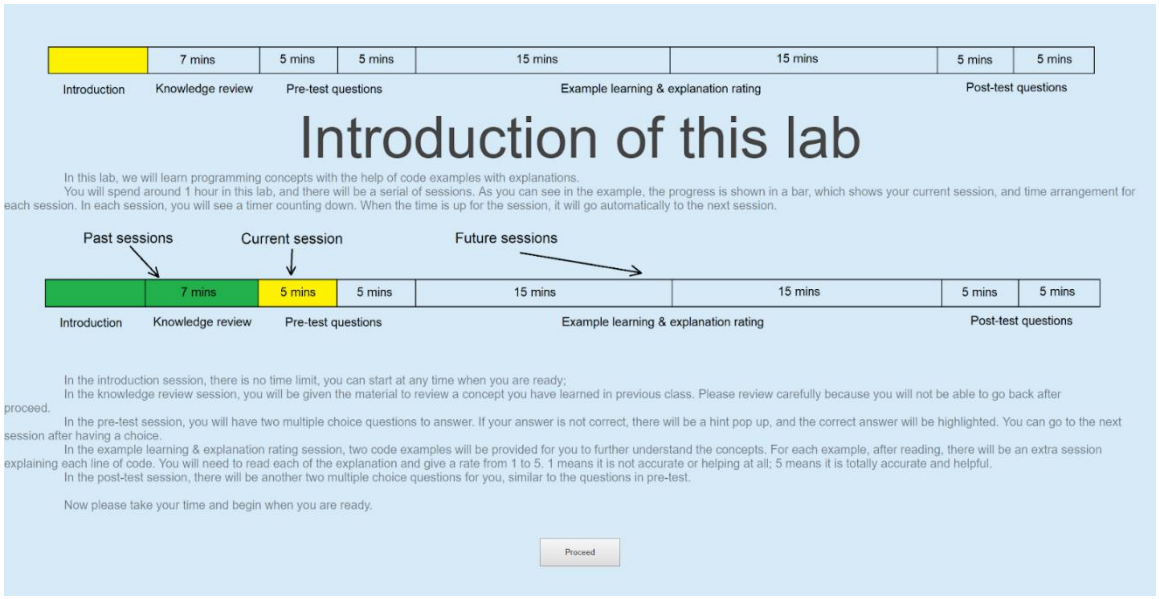
In the next session the students reviewed the programming related concepts they have learned in class, in which users can browse the concept definitions, usages, and precautions. The purpose of this view was to have students remind the knowledge they have just learned in class and unify the knowledge baseline before the students start the learning process. In the middle of the description, there can be images or charts embedded to better introduce the concept.

After reviewing, the students were given four coding related multiple-choice questions in the third session to examine their level of understanding. This session was treated as the pre-activity test since it is before the example learning. If they did not choose the correct answer, hints would be provided, and the correct answer would be highlighted. The purpose of this session was to examine the level of students' knowledge about the concept before reading all example codes with explanations.

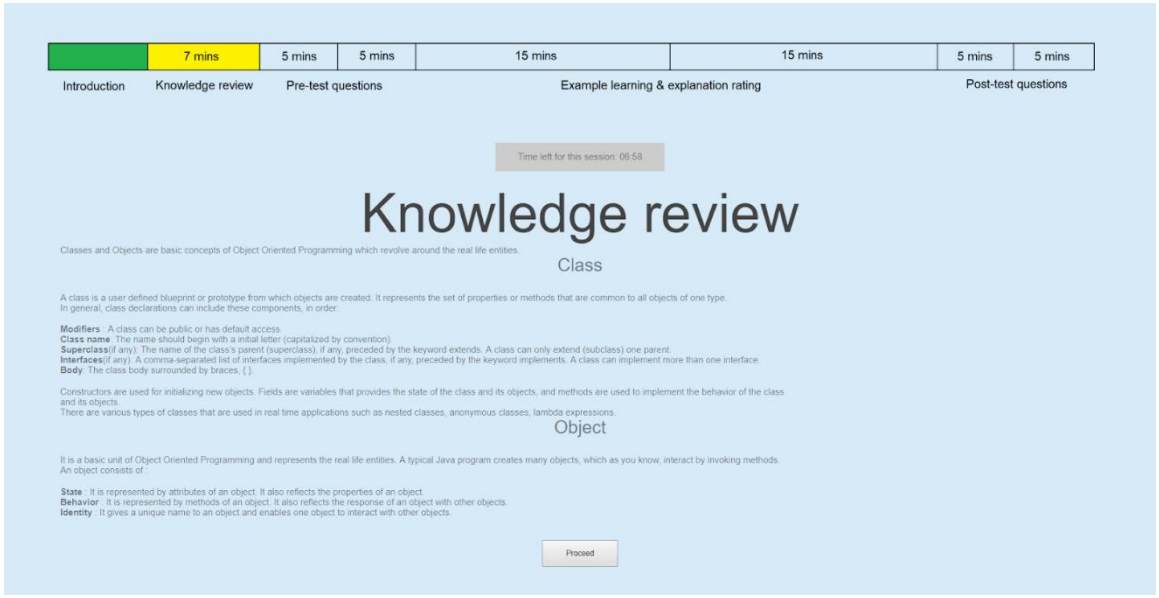
In the fourth session, five different code examples were provided to the students to learn. For each line of the code examples, the students in experiment group were required to provide an explanation rate from 1 to 5 to evaluate its helpfulness. In order to evaluate the effect of the proposed tree-structured LSTM model, there was no explanation for the students in control group.

The interface design for the task system is shown in Figure 12.

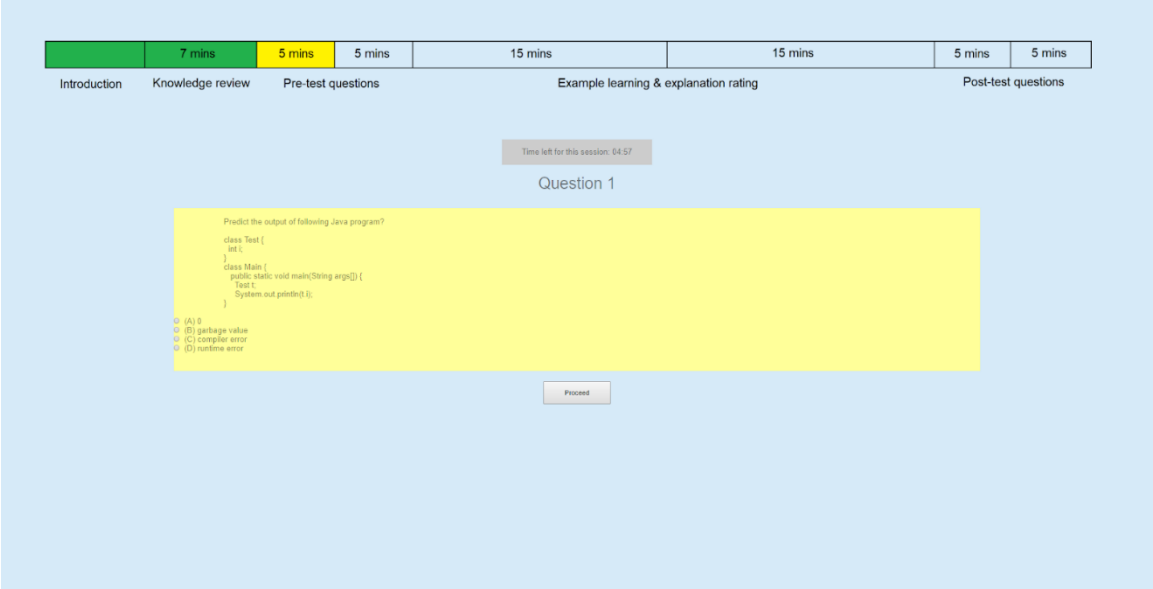
After example learning, the students will have another session of coding related multiple-choice question as the post-test.



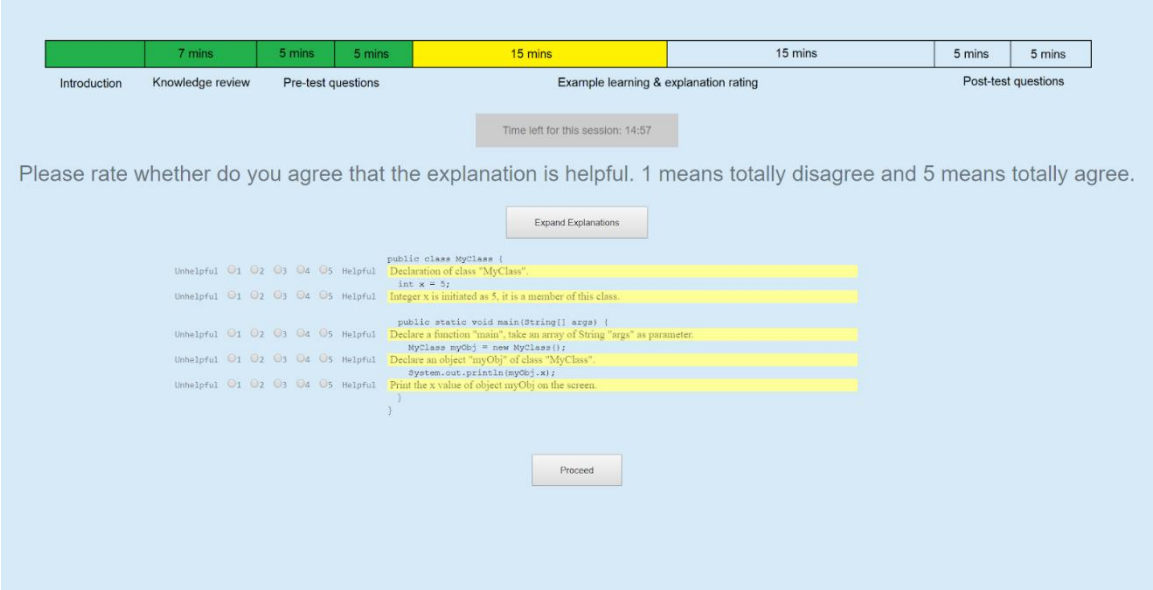
(a) The Lab Introduction Session Page.



(b) The Concept Review Session Page.



(c) The Question Session Page for Pre-test and Post-test.



(d) The Example Learning & Explanation Rating Session Page.

Figure 12. Interface of Lab Study Task System.

Through the whole process on this website, the students' mouse clicks, scroll, and hover will be all logged together with the timestamp and user identity for further analysis.

To prove the usability of the system and interface, there have been three ASU101 classes that took part in the system evaluation. Over 50 students have taken the pre-survey, knowledge background pre-test, and experienced the lab study system to learn "for loop" in Java. The pre-survey result shows that the students' background is uniformly distributed from no background to more than 1-year experience. Additionally, the pre-test with confidence supported this conclusion. During the system evaluation, all students went through all the sessions of the system without problem.

There will be a formal lab class conducted in the course CPI101, which has 90 students registered in total. By tracking the performance of students and the ratings of code explanations, there will be a detailed, multidimensional understanding of the effect of MG content in programming education.

Workflow

The completed process of the lab study has two main phases: pre-test phase and task phase. Each phase has a detailed workflow.

In the pre-test phase, there are three main stages. In the first stage, the whole lab study will be introduced to the students. The introduction will include the purpose, process, what they are expected to do, and how they will benefit from the study.

After confirming the introduction, the students will come to the second stage, in which they will answer a set of questions to self-evaluate their programming background,

including their programming year-age, most experienced language, and level of understanding in some specific programming concepts.

Then, the students will start the last stage to test their mastery of knowledge in detail. In this stage, concepts that the students will learn during the semester will be involved in multiple choice questions to examine the students' level of understanding. Each concept will have two multiple choice questions, which are differentiated by difficulty to examine the students' knowledge in detail. After each multiple-choice question, there will be an additional question asking about their confidence rating from 1 to 5. This design helps to remove the bias caused by the students who have not learned the concepts and randomly answer the questions.

The whole pre-test will take an hour in total, in which the introduction stage will take 5 minutes, the background self-evaluation will take 10 minutes, and the knowledge test will take the rest 45 minutes.

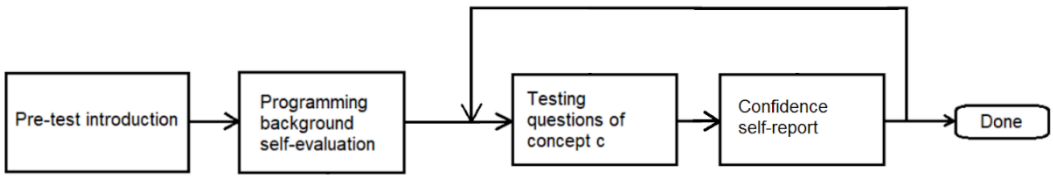


Figure 13. Workflow of Pre-test

The regular lab study will be held in two single classes. During a lab study, each student will be randomly assigned a source of code explanation selected from human content, sequential model generated content, tree-structured model generated content, or

no explanation. The workflow of each regular lab study will have another three main stages.

In the first stage, the students will go through a brief review of the concepts they have just learned in the last class. The review will not cover too much detail, but just illustrate definitions and suitable scenarios for the concept.

Then, a set of task questions will be given to the students to examine their concept knowledge at this point. The questions are multiple choices to fill code line into a complete code block. This test result will be considered as the baseline before learning from code examples and explanations.

In the next stage, the student will be required to carefully read and learn from three code examples closely related to the concept reviewed. Concurrently, they will read the code explanations for each line, try to draw the connection between codes and explanations, then evaluate the explanation by giving a rate from 1 to 5. After reading and evaluating all contents in the second stage, the students are expected to have a better understanding about the concept.

In the final stage, the students will be tested with programming related tasks. In this experiment, the tasks are two multiple choice questions about filtering blanks in programming codes, but different from the questions tested before example learning. If the students successfully answer the two questions, the study will end. Otherwise, if they make another wrong choice, there will be hints given to them and have them try again until they reach the correct choice. The task stage is designed in this way since it is

critical to measure the mastery level of knowledge for students to prove the lift in their learning, while the number of attempts they consume is a reasonable metric.

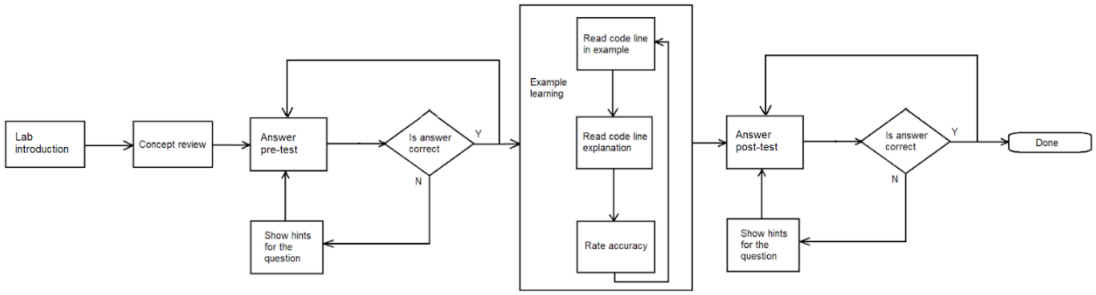


Figure 14. Workflow of Regular Lab Study

At the end of the long-term lab study, the students will be surveyed about their experience in the lab study. They will give feedback regarding the interface design, content arrangement, and learning experience. The students will also be asked about their experience about the code explanation, and further suggestions for learning assistance.

Evaluation

To evaluate the whole model and prove the value of MG code explanation, the learning effect from code examples with explanations were analyzed from two perspectives. The first analysis was focus on the task performance comparison to examine the final learning effect. The second analysis was turn to the ratings given to the explanations to measure the effects of three content sources on students with different knowledge backgrounds.

Task Performance Analysis

In the first analysis, the learning effect of the students was evaluated by comparing their task performance before the example learning and their first attempt after the example learning. In this way, with a relatively large sample pool, the statistical difference between these two performances were reflect the learning effect difference among human code explanation, sequential model, and tree-structured model.

In this analysis, three questions were investigated:

- How does the explanation content affect the task performance?
- How does the knowledge background affect the task performance?
- How does the background knowledge impact the students' learning effect in class (before example learning)?

To address the first question, students were grouped into Four according to the explanation content and the test questions they encounter. The explanation content groups are named T (Tree-structured LSTM model generated content), and N (no explanation). Besides, there were two sets of questions for students to take before and after the example learning. The students were take the question sets in random order to remove the bias caused by the question difficulty, which further split the groups into four (Table. 2).

Table 2

Student Grouping with Different Example Explanation Content and Test Question Sets.

	Tree-structured model explanation	No explanation
Question set 1 as pre-test, 2 as post-test	T1	N1
Question set 2 as pre-test, 1 as post-test	T2	N2

By comparing the task performance enhancement among the four groups, the effect difference can be revealed. Assuming the number of correct first attempts of a student is C, the number before example learning is C_{Before}, the number after example learning is C_{After}, and the difference between the two numbers is $D = C_{After} - C_{Before}$. The comparison were performed by an one-way ANOVA test with hypothesis as follows:

Hypothesis 1:

Hypothesis A: average D_T > average D_N; Hypothesis null : average $D_T \leqq$ average D_N

In this hypothesis, the groups with model explanation are estimated to perform better than non-explanation group, which was verify the conclusion of our previous work that the novices have a requirement of example explanation (Lu, Y. & Hsiao, I-H. 2017), and the tree-structured LSTM model can assist novices in a level.

The next two questions are exploratory questions, which could impact the strategies we may apply on students with different knowledge backgrounds. First, the students were grouped into H (higher experienced students) and L (lower experienced students) according to their background survey and pretest. The analysis method in these two questions are also ANOVA tests with the following hypothesis:

Hypothesis 2:

Hypothesis A: average D_H > average D_L; Hypothesis null : average $D_H \leqq$ average D_L

Hypothesis 3:

Hypothesis A: pre-test C_H > pre-test C_L; Hypothesis null : pre-test $C_H \leqq$ pre-test C_L

Explanation Rating Analysis

The collection of ratings of explanations is also a critical information reflected from the students. In the rating analysis, this data was matched with the learning effect to calculate the correlation and validate that the quality of explanations is relevant to the learning effect. Also, the ratings were matched with the students' pre-test result and task performance before example learning to investigate the potential bias of the rating caused by the knowledge level of students.

In other words, the questions to investigate in this analysis are:

- What is the connection between the students' rating and their task performance improvement after example learning?

- Whether a well knowledgeable student understand explanations better and give higher ratings than a student with less background knowledge?

In this analysis, the second question is essentially aiming to remove a potential bias in the first question, so the first question is the core of the analysis. To reveal the connection between explanation ratings and source of content, a Pearson's r analysis was be conducted to capture the significance of the correlation between students' ratings and their submission correctness. The expected result is to confirm the significant rating difference aligned to the performance difference in the last analysis. Let the average ratings of a student i is R_i, and the task performance difference after example learning is D_i. The hypothesis is as follows:

Hypothesis 4:

Hypothesis A: the correlation between normalized set R and D is higher than 0.2; Hypothesis null: the correlation is 0.

Similar method was be used on the second question to examine the bias of ratings brought by the students' background knowledge. The expected result is to see no significant rating difference between students with different background knowledge. If the code examples are properly chosen, which means they are neither too hard nor too simple, the result of this analysis had a higher chance to be as expected. Let the pre-test performance of a student i is P_i, the hypothesis is as follows:

Hypothesis 5:

Hypothesis A: the correlation between normalized set *P* and *D* is higher than 0.2;

Hypothesis null: the correlation is 0.

CHAPTER 6

EXPERIMENT RESULT

Research Questions

To answer research question #1 “How to explain code for novices with MG language?”, I designed a deep learning-based code explanation generator and developed a system to evaluate the generator by helping learners review programming concepts and deepen their understanding with code examples. Furthermore, to evaluate the impact of code explanation, the system serves pre and post learning tests to the users to measure the improvement during example learning.

The research question #2 “How do learners benefit from MG code explanations when learning concepts and algorithms?” was answered by establishing experiments on two classes of students: one formed by novices and the other had more experience. By comparing the impact of MG code explanation on the two classes, the mechanism of how novices’ benefit is revealed. Moreover, the impact was analyzed from different perspectives including the test performance improvement after learning from example, the efficiency of time spent in example learning, and the relation between students’ rating of explanations and their performance.

Analysis Methods & Results Summary

In the experiment, the performance score of students was evaluated by the correctness of their test answers, each correct answer earned 1 point. Since there were 4 questions in the pre-activity and 4 in the post-activity test, the total score of a student ranged from 0 to 8, and the improvement could range from -4 to 4.

To analyze the experiment result, a set of features and learning analytics were extracted from the experiment log including the students' total performance score (Perf), their normalized performance improvement after example learning as learning gain (LG), their performance in the background knowledge pre-test (Pre), the time they spent on example learning (Time), whether they were provided the code explanations (Explain), and their total ratings on the explanations (Rate). The following table summarizes the experiment log analytics and definitions.

Additionally, the learning gain (LG) was ranged from -4 to 4, since in the worst case, a student could get full score 4 in the pre-activity test, and get 0 in the post-activity test, then the LG would be -4; while in the best case, a student could get 0 score before learning, and 4 after, then the LG would be 4.

Table 3

Data Feature Definitions

Feature	**Definition**
Perf	The total score of a student in the lab class (0 to 8).
LG	The learning gain is the normalized improvement of a student's score after example learn (-4 to 4)
Pre	The pre-knowledge score of a student (0 to 150)
Time	The time spend of a student in the example learning phase

Explain	Whether a student was provided code explanation in examples
Rate	The total rating given to the explanations from a student

Table 4

Subject Group Definition

Group	**Definition**
Experiment group (E)	Code examples with MG explanations.
Control group (C)	Code examples without explanations.

According to the result of analysis, the value of MG code explanation is reflected by improving the Predictability of learning for novices. The Predictability here means that with more time spent in learning from example by a student, how much better he/she can expect in test performance. The analysis result indicates that with the help of code explanations, students can indeed achieve a more predictable and positive improvement by putting more time in learning, while this fact is not shown when explanations are not provided.

However, results also reveal that the value of MG explanations is not reflected directly by improving the students' test performances. Moreover, students' subjective ratings on the explanations do not indicate their achievement. These facts suggest that educators should be cautious about how to use the MG annotations, and the subjective ratings from students are not fully reliable compared to their performance in tests.

Additionally, the MG annotations can play the role attracting students to spend more time in learning while maintaining their attention.

Learning Effects on Novices

Student Background

To evaluate the impact of MG code explanations in programming examples, an experiment was conducted. The experiment was established in an introductory web-programming class. There were 53 students who took part in the experiment.

At the beginning of the semester, the students were asked to complete a pre-survey and a pretest about their knowledge background. According to the survey, students' backgrounds were evenly distributed into three major groups: 1) students with zero or less than a month programming experiences; 2) students with less than a year programming experiences; and 3) students with more than a year experience. According to the background distribution, group 3 (students with more than a year experiences) may seem to potentially result in a ceiling effect. Note that such effects should be minimized by controlling the concepts involved in the experiment were fundamental and essential knowledge to the programming language instead of language specifics.

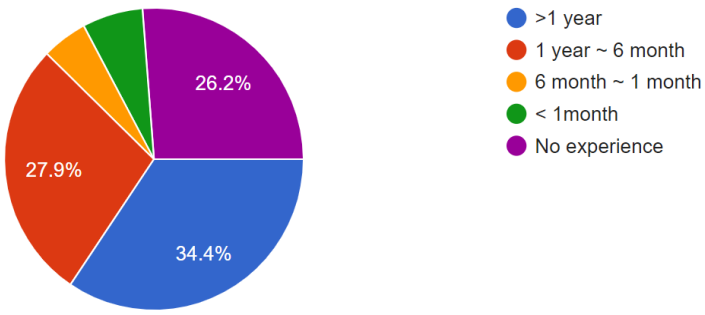


Figure 15. Novice Class Programming Background Distribution

Besides general programming experience, we also asked students rate their web development experience specifically, their HTML and JavaScript coding experience. The results showed that students reported relatively low web programming experience. Most of them rate themselves between 1 and 3 out of 5. It indicated their self-reported novice skills in JavaScript.

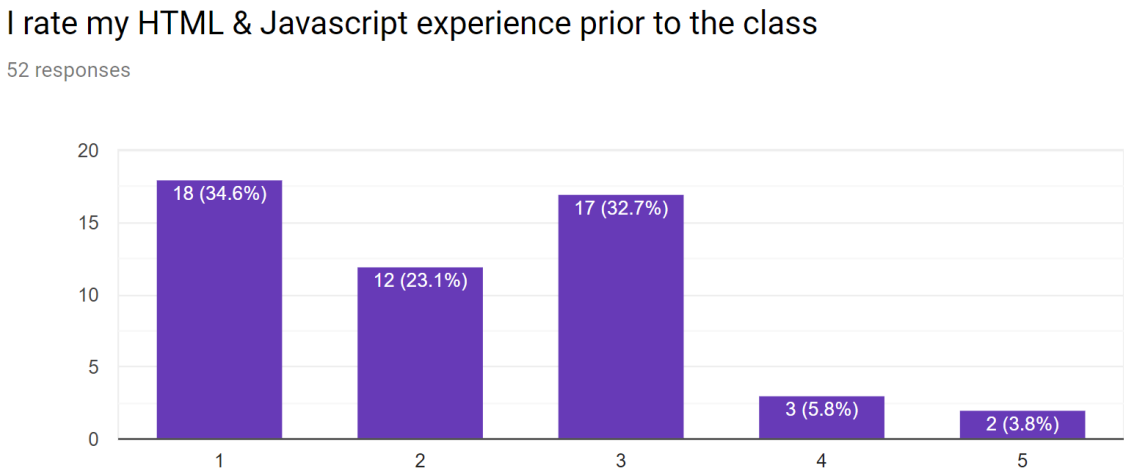


Figure 16. Novice Class Self-evaluation in JavaScript

Besides the subjective self-evaluation survey, a pre-test was administered to investigate students' pre-knowledge of the domain. To avoid the impact when a student does not understand a concept and randomly guess as the answer, the students are required to provide their confidence besides the answer for each question.

The distribution of the average confidence for each student is supported by the survey result, in which most of the students had an average confidence of 3 or less.

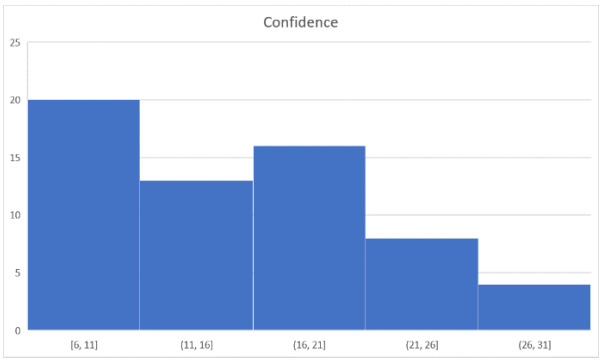


Figure 17. Novice Class Distribution of Total Confidence in Pre-test

To involve both the students' answer quality and their confidence, the score for each question is calculated as the product of the correctness of an answer and the student's confidence level. Both factors are normalized between 0 and 5. In the final total score distribution, most of the students achieved less than 20% of the total score (30 out of 150), which supports that this class is feasible to conduct experiments for novices.

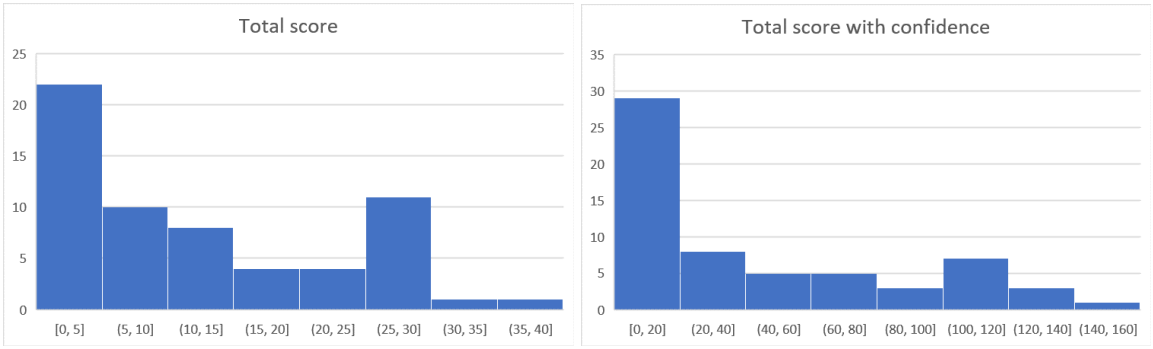


Figure 18. Novice Class Total Score Distribution

Classroom Experiment Setup

The experiment was established as a lab class, which is designed to help students learn JavaScript from examples. The concept "for loop" was chosen as the topic in the lab class, since it was introduced a week before the lab class.

At the beginning of the lab class, students were required to land on the experiment and enter their identity. Before the formal experiment, they were given a brief introduction about the purpose of the experiment and what they can expect from it. In phase 1 of the experiment, the students went through a material helping review the concept, which set them to the same baseline. Then in phase 2, the students took a pre-activity test before example learning by answering 4 multiple choice questions. For each question, after they made a choice and submit, there would be a hint pop out and the correct answer would be highlighted. Then in phase 3, the students were given 5 code examples to learn. In this phase, the students were randomly split into control group (group C) and experiment group (group E). In group C, the students were only provided the example code and the execution result; in group E, besides the example code, the students were also provided code explanations generated by machine. The group E students are required to read and rate the helpfulness for each explanation from 1 to 5. After the example learning, in phase 4 the students took a post-activity test by answering another 4 multiple choice questions, which has the same setup as pre-activity test.

After the experiment process, a post-survey was given to the students to collect their feedback, and suggestions for the system and example learning experience.

Result analysis

Explanation helps students improve learning predictability.

When investigating the difference between group E and group C, one of the major findings is the impact of code explanation on learning predictability. Predictability is defined as the correlation between learning gain represented by the test performance

improvement after example learning (pre & post test score difference) and time spent in example learning (Time).

Note that the learning predictability was adopted as the learning metric instead of using the raw value of time spent in example learning, the reasons are illustrated as followed: Since the students in group E had an additional task that they need to rate the quality of explanations, they spend more time and there is no value to compare the time spent on example learning between group E and C. However, the correlation between learning time (Time) and learning gain (LG), namely predictability, can reflect the value of explanations and remove the bias caused by time spent difference. Thus, a higher correlation between LG and Time indicates that a student's learning gain is more predictable to have a positive correlation with given his/her learning time. This predictability is important since it indicates a more stable and positive effect in learning.

In group E (N=38), the correlation (Learning Predictability) between Time (mean=408.4, sd=150.9) and LG (mean=0.29, sd=1.56) is 0.17. Meanwhile, in group C (N=37) the correlation between Time (mean=271.2, sd=118.0) and LG (mean=10.1%0.03, sd=19.1%1.22) is only -0.07. The positive correlation between Time and LG indicates the more time a student spent on example learning and resulted in higher learning gain was achieved.

Table 5

Novice Class Example Learning Time Spend, Learning Gain, and Predictability

	Time spent(s)	**Learning Gain Performance improvement**	**Correlation (Predictability)**
Experiment Group (N=38)	408.4 ± 150.9	0.29 ± 1.56	0.46 (p = 0.01)
Control Group (N=37)	271.2 ± 118.0	0.03 ± 1.19	0.02 (p = 0.89)

According to this result, in group E the time spent on example learning has a positive effect on task performance improvement, while such effect was not observed in the control group. In other words, with code explanations, students are more likely to achieve better performance after spending more time in learning from examples.

Potential bias may still be involved in this conclusion since the students in group E are designed to spend more time on example learning. The performance improvement may not be caused only by the explanation content, but also possibly brought by the extra time spent in code reading. However, this possibility does not deny the value of the code explanation because one of the benefits of code explanation is leading students to spend more time and stimulate them to have a deeper understanding in programming. As a result, the value of MG code explanation can be reflected by the predictability on novices.

Impact of explanation on task performance.

It is intuitive to examine the value of code explanation in learning by comparing the learning gain (LG) between the experimental group (E) and control group (C). To capture the LG difference, a two-way ANOVA test is conducted to measure the effect of code explanation and learning time on learning gain, and the result is as follows.

Table 6

Two-way ANOVA Test Result in Novice Class Experiment

	Source	**SS**	**DF**	**MS**	**F**	**p-unc**	**np2**
0	group	1.171	1	1.171	2.342	0.265573	0. 539
1	time	2.503	48.0	0.052	0.104	0.999590	0.715
2	group * time	102.969	48.0	2.145	4.290	0.207018	0.990
3	Residual	1.000	2.0	0.500	NaN	NaN	NaN

Although the mean LG of group E (mean = 0.29) is higher than group C (mean = 0.03), the p-value between the two groups is 0.266, which indicates that the difference is not significant. With an effect size 0.5 and a sample size larger than 42 (N=53), this result means a student is not guaranteed to perform significantly better when given explanations when learning from code example. In conclusion, the explanation does not lead to better learning effect in code examples without considering other factors such as knowledge background, time spent, and confidence in learning.

Background knowledge matters a lot in learning.

In education, there is a common sense that the background knowledge is an important factor in learning. Students with better experience can usually learn faster and better. This fact is also reflected in this experiment.

To tackle the connection between pre-knowledge (Pre) and task performance, I involved both total performance (Perf) and learning gain in performance improvement after example learning (LG). The correlation between Perf and Pre is 0.46, which is a strong indicator that better pre-knowledge leads to better total performance. On the other hand, the correlation between Pre and LG is 0.1. The correlation is positive but close to 0, which is reasonable because students who already performed well in pre-activity tasks would have little space to improve in the post-activity tasks. However, the correlation is still positive, so strong background knowledge indeed impacts learning impact in a degree.

Another potential impact of pre-knowledge is on the predictability. To investigate the impact, all students are evenly split into two groups according to their pre-knowledge score. The correlation between time spent in example learning (Time) and task performance improvement after example learning (Imp) in higher-knowledge group is 0.22, which the correlation in lower-knowledge group is 0.01. This result indicates that students who have both better pre-knowledge and more time spent in learning could achieve considerable improvement, while spending more time could not help a lot for students without necessary pre-knowledge.

Explanation ratings are not reflecting the effect.

Besides the task performance, another valuable information collected from the students is their rate on the explanations in group E. The average rate for each code line explanation fits in a skewed normal distribution (mean = 4.34, sd = 0.27) as shown in Figure. The rating is much higher compared to the results in pilot study. This result is not expected, it indicates further problems.

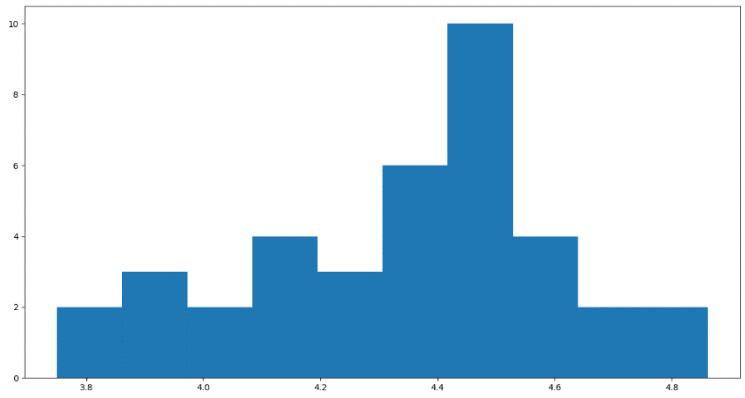


Figure 19. Explanation Average Rating Distribution

The reason for the higher rates is revealed when the average rate for each student is analyzed. The distribution of student rating means is not normal, instead there is a certain part of students rate all explanations as 5. This part of students made a huge impact on the average ratings.

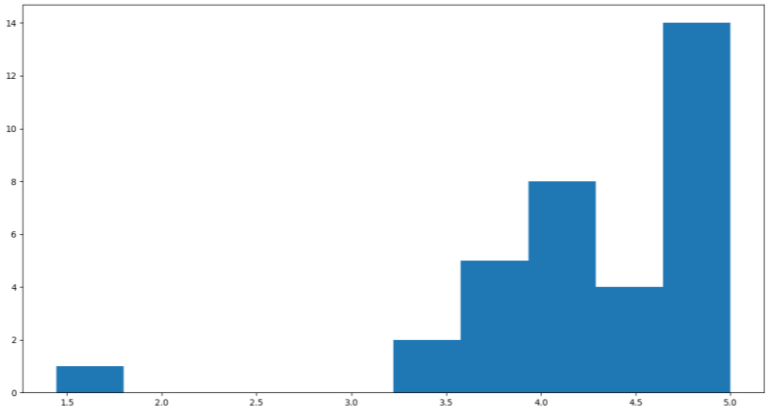


Figure 20. Student Average Rating Distribution

To further understand the behavior of these students, I analyzed the connection between their Imp, knowledge background, and rate. The correlation between knowledge background and ratings is 0.06, which suggests non-correlation, while the correlation between LG and rate is -0.17. This negative correlation indicates that when students do not have a good understanding of the code example, they would rate higher. This finding is unexpected but reasonable since the readability of MG explanations are as good as human explanations according to pilot study. This finding also suggests to us that in novice educating, the subjective ratings from novices are not fully reliable.

Students feedback

Although the LG difference between group E and C is not significant, the students expressed their interest in learning from examples and explanations in their post-survey and feedback. In general, the feedback is positive in the post survey. Students reported

positive ratings toward the interface usability (mean), concept understanding after learning, and the help from code examples.

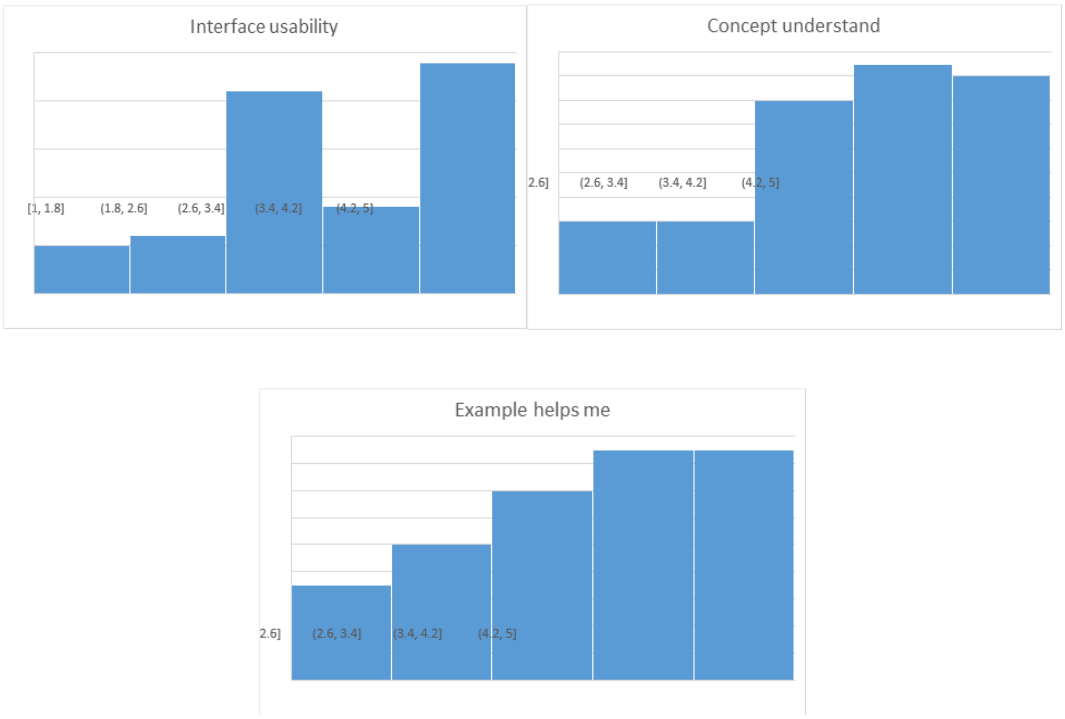


Figure 21. Novice Post-survey Result Distribution

In the detailed feedback, students also expressed their positive feelings on the code examples, explanations, and even time limitations in tasks. On the other hand, some students provided valuable suggestions. Some specific valuable feedback is list below:

Positive feedbacks.

"*I feel like the examples were really helpful.*"

"*This instructive program has given me more in-depth explanations of certain keywords and functions than in class.*"

"*I really liked this. Because I come from a no-coding background, the explanations were super helpful! I would love to see this again.*"

In general, the feedback from students is positive. Many students claimed that this learning system with code examples and explanations helpful. The positive effect was also reflected in their post-survey rating questions that these students gave relatively higher ratings to the interface, concept learning, and effect of examples.

Better explanation is expected.

"*Need to put more detail into explanations, still a little vague*",

"*The examples need to have more information on how they specifically work with quotes or words because we do not understand what they mean.*"

On the other hand, some students expected to have explanations with higher quality and more details. In the LSTM model training, there is a trade-off between the richness of information and the accuracy.

When training the translation model, the frequency of each word in the vocabulary of training dataset is important. The low frequency words in the vocabulary could bring accuracy problems since there are not enough samples for the model to learn the correct usage of these words. As a result, to avoid involving wrong words in explanation generation impacting the accuracy, it is necessary to filter low frequency words under a threshold.

However, if the threshold is too high, and too many words are filtered, the explanation generated will only include high frequency words, which are usually vague

and could not provide much specific information. As a result, there is a trade-off between the richness of explanation and the accuracy with the model.

Considering the scenario of programming novice learning, erronic words in explanation could potentially lead to misunderstanding and further increase the cognitive load in understanding, which is more harmful compared to the lack of deeper information. So, in this experiment, more words are filtered in purpose.

Compilable platform expected.

"*I would like it if there was a 'try your own' code section where there would be pre-written code that you could modify and "play" with to get a better understanding of how each concept works*"

Some students even suggested providing them an embedded compiler to try out the code examples, and even test their own code modification on the example. This is a valuable suggestion, which gives the explanation generator an even wider dimension to impact the novices in learning. If there can be an online code example learning platform with a compiler and an automatic explanation generator, students could understand the knowledge better by modifying the examples in different parts and remove any confusion by reading the explanations to their own code generated dynamically.

Learning model construction

The connections among factors in the experiment are addressed in Figure 22. As shown, the two dependent variables are task total score performance (Perf) and the performance improvement after example learning (Imp), and the four independent

variables are pre-knowledge (Pre), whether provided example explanation (Exp), time spent in example learning (Time), and their ratings given to the explanations (Rate).

These variables are not completely independent. The time spent is impacted by whether explanations are provided, since the students in the experiment group are supposed to spend more time on explanation reading and rating. Also, the level of pre-knowledge has the potential to impact the time spent and ratings in example learning. So, the correlations between pairs of independent variables are also calculated.

As shown in the figure, the total score is highly correlated to the pre-knowledge level of students, but the pre-knowledge has limited impact on the improvement of performance. On the other hand, pre-knowledge has little connection with time spent and ratings in example learning.

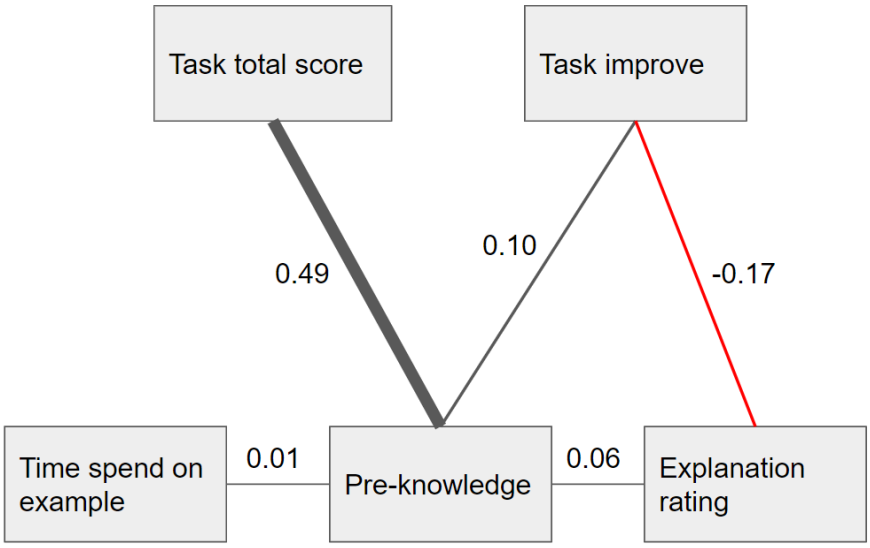


Figure 22. Feature Connection Modeling for Novice Class.

To tackle the impact of code explanation, the students in group E and C were separated when calculating the correlation between Imp, pre-knowledge and time spent. The result showed that when explanations are provided, both pre-knowledge and time spent has a positive effect on Imp, while in the control group the connections are weak.

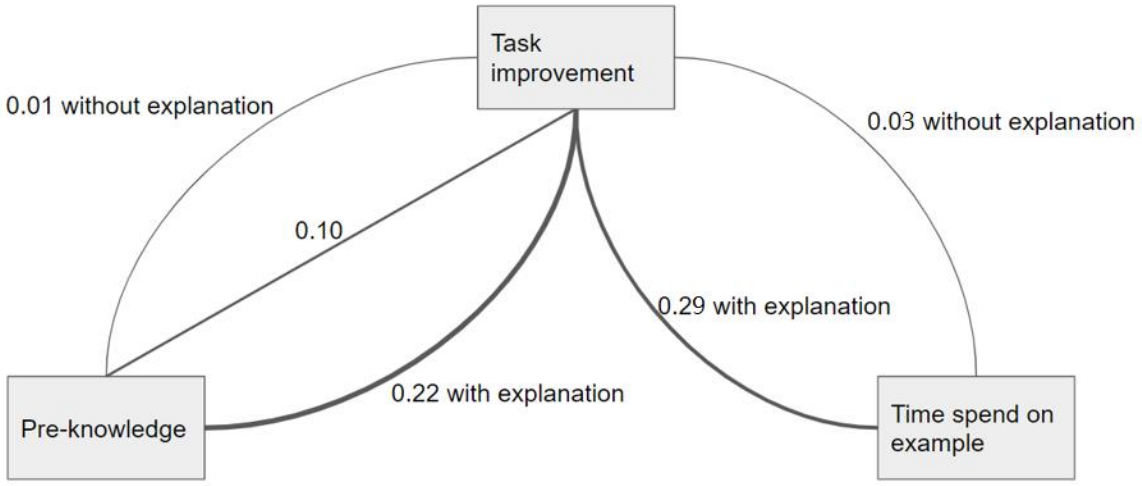


Figure 23. Feature Connection Modeling with and without Explanation Feature.

Learning Effect on Experienced Students

To answer the research question "How do programming learners benefit from MG annotated examples in declarative and procedural knowledge learning?", in this section the impact of MG code on advanced students is investigated and analyzed. To tackle the impact, another experiment was conducted in a class teaching advanced knowledge in Java programming, in which students had at least one semester experience of programming learning.

Student Background

The general knowledge background of the students is shown in Figure 24. 46.3% of the students had experience for more than a year; 28.3% students had experience more than a semester and less than a year; another 18.5% students had experience more than a month.

This result reflected the fact that the prerequisite of the course only accepted students who had passed the first course in programming, and this class was indeed experienced.

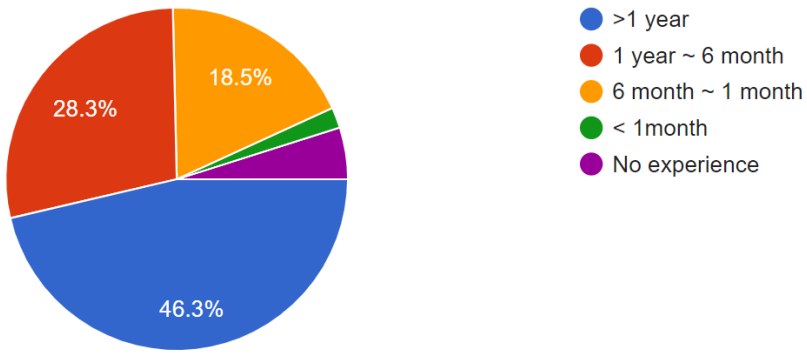


Figure 24. Advanced Class Programming Background Distribution

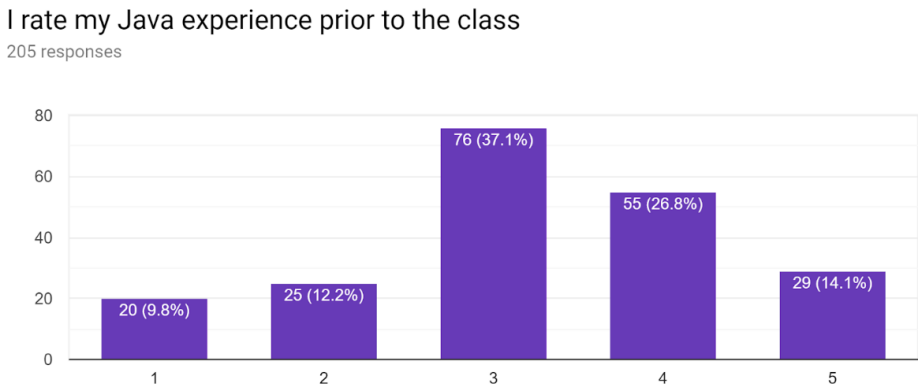


Figure 25. Advanced Class Self-evaluation in Java

The experience advantage was also supported by the student's self-rate on their background in Java language. In this class, the majority (78.0%) of students rate their experience as medium or higher.

Figure 26. Advanced Class Pre-test Total Confidence and Score Distribution

The knowledge background advantage was fully validated in the pre-test result, where the student's total score fits in skewed normal distribution, and the mean value of total score is higher than 50% of full score.

Result analysis

In this experiment, the students were provided knowledge review of the concept "Big O notation", and code examples in Java were provided to further deepen the understanding of the concept. An analysis is applied on the result of this experiment close to the novice class to tackle the impacts on advanced students.

The two-way ANOVA analysis on the LG in the advanced class experiment shows that the LG gained from explanation is not significant ($p = 0.518$).

Table 7

Two-way ANOVA Test Result in Advanced Class Experiment

	Source	SS	DF	MS	F	p-unc	np2
0	group	6.038	5	1.208	0.604	0.518446	0.602
1	time	5.123	73	0.070	0.035	0.999992	0.562
2	group * time	1307.532	365	3.582	1.791	0.425580	0.997
3	Residual	4.000	2.0	2.000	NaN	NaN	NaN

Table 8

Advanced Class Example Learning Time Spend, Learning Gain, and Predictability

	Time spent	Learning Gain	Predictability
Experiment Group (N=46)	362.4 ± 167.2	-0.17 ± 1.80	-0.10 (p = 0.50)
Control Group (N=57)	210.5 ± 128.3	0.13 ± 2.00	-0.01 (p = 0.96)

Learning time is not correlated to learning gain.

When examining the correlation between task performance improvement (Imp) and the time spent on examples in the experiment group (E) and control group (C), both correlations were negative and close to zero (E_cor = -0.10, C_cor = -0.01). This result indicated that for both groups, learning time is not correlated to learning gain. For advanced students, providing code explanations did not lead to a higher correlation between learning time spent and learning gain, or the learning gain is not reflected in reading.

This result supported that the predictability impact for novices was not relevant to the increased time spent.

Lower ratings for explanations.

Since the learning time spent is not reflected in the learning gain, the rating of explanations in the advanced class was analyzed to tackle their reading. Figure 27 shows the distribution of average ratings for each student. Besides the peak close to 4.0, there is

another peak at 2.5. This result suggested that there is a set of students who feel not satisfied with the explanations in general.

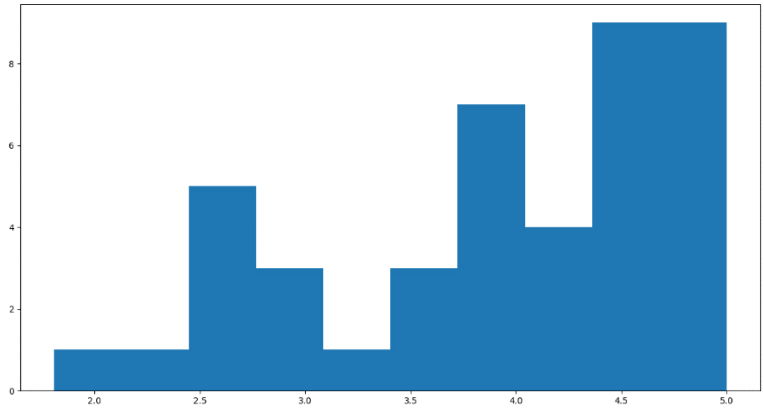


Figure 27. Student Average Rating Distribution in Advanced Class

Considering the topic of the lab class “big O notation” and the background knowledge of students, the peak of lower rate is reasonable due to the importance of deeper logic in the examples, which is not fully reflected in explanations. Although machines cannot explain logical concepts directly, it still has potential in removing obstacles in fundamental syntax, so learners can focus on the logics in codes.

Feedbacks are more positive than novices.

Compared to the novice class, the feedback ratings are even higher. This results indicated that the advanced students had less problem in understanding the interface (mean 4.29, STD 1.06), and had better background knowledge to achieve help from examples (mean 3.76, STD 1.16), and understand the concept better (mean 3.71, STD 1.23).

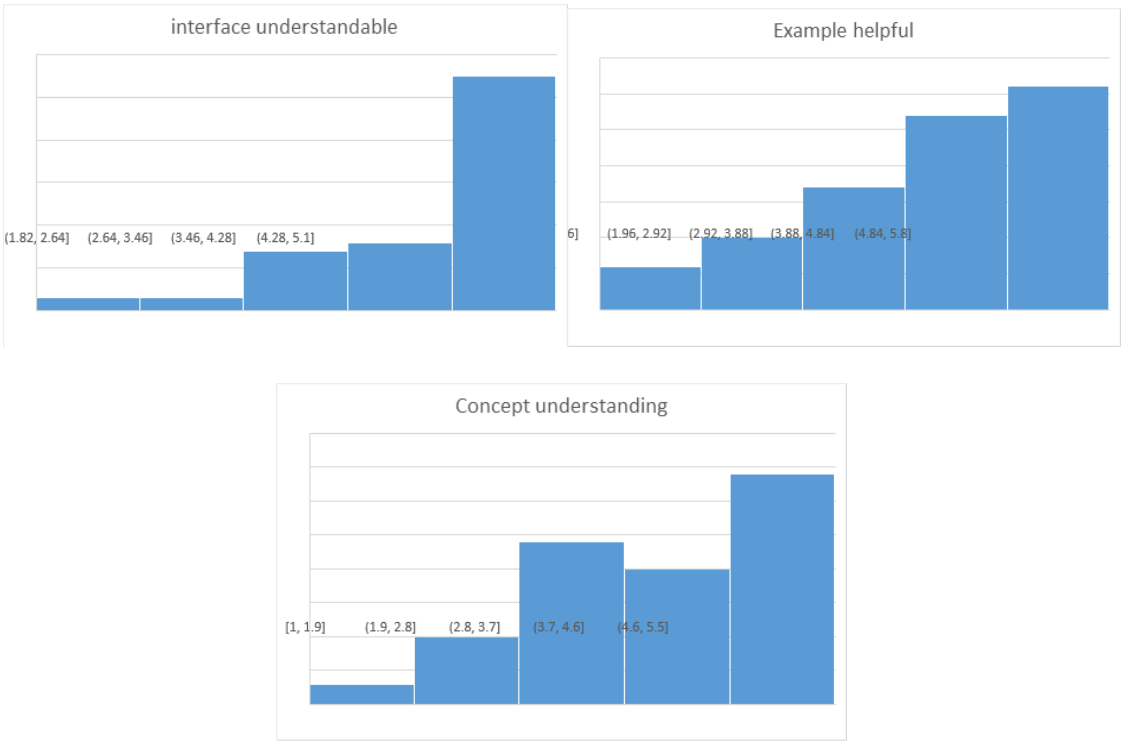


Figure 28. Advanced Class Post-Survey Result Distribution

The reason for this positive feedback is possibly because the advanced students had better experience in understanding coding related websites, some even had better experience in learning from resources online.

Moreover, advanced students also provided fluent suggestions.

Feedback and suggestions

Positive feedbacks

"It was a good system"

"it was intuitive"

"Great idea! It feels simple and easy to learn."

"Really loved the in-class activity because it engaged me and forced me to learn the concept. I hope the professor does more in-class activities that we work on our laptop!"

"I think it's a good tool to use for learning how to read code and learn from the examples."

The majority feedback is positive in ratings, and this fact is also reflected in textual feedback. Samples are shown above. The students expressed their interest in this innovative system and expected more learning experience in this system. Besides the help of the system, some students also highlighted the fact that having them spend time in class to focus on learning concepts was helpful.

More explanations are expected to clarify logics in code

Besides positive feedback, suggestions were also collected:

"You could use the specific parts of the code where n is used in the explanation because I still am slightly unsure where to look for each n/m. Also I didn't understand how O(log(n)) works or how to know why it is that as well as n^2 vs 2^n. More examples of those complex ones would help"

In this feedback, the student expected to have more mathematical explanations besides the syntax, and more complex examples are also expected to extend their understanding in big O notation. This suggestion is reasonable since the concept big O notation is partially a mathematical concept instead of syntax concept. Also, there are many examples of problems and algorithms involving larger time complexity, which are worthy to show as advanced materials.

"Make more of the explanations have to do with the concept at hand, not just helping with understanding code."

This feedback complained that the explanations only focused on syntax instead of the concept logics. However, due to the limitation of the current model, the explanations generated could not accurately capture logic connections among a whole block of code. As a result, the explanation generating model works well on syntaxial concept explaining, but does not work ideally on logic related concepts such as big O notation.

"I wish the written explanations given with each example would be more comprehensive."

In these suggestions, the students expected to have explanations with more details related to the "big O notation" concept. This feedback was exactly expected since the MG explanations are at syntax level, while the topic "big O notation" is a logic level concept. This fact indicated that one of the limitations of my model is it only focused generating syntax level explanation, so it does not well meet the requirement of the advanced students who are learning concepts involving complicated logics.

Learning model construction

Similar to the first study, there is a learning model constructed based on the data analysis on features including the two dependent variables: task total score performance (Perf) and the learning gain after example learning (LG); and the four independent variables: pre-knowledge (Pre), whether provided example explanation (Exp), time spent in example learning (Time), and their ratings given to the explanations (Rate). The connection analysis is shown in Figure 29.

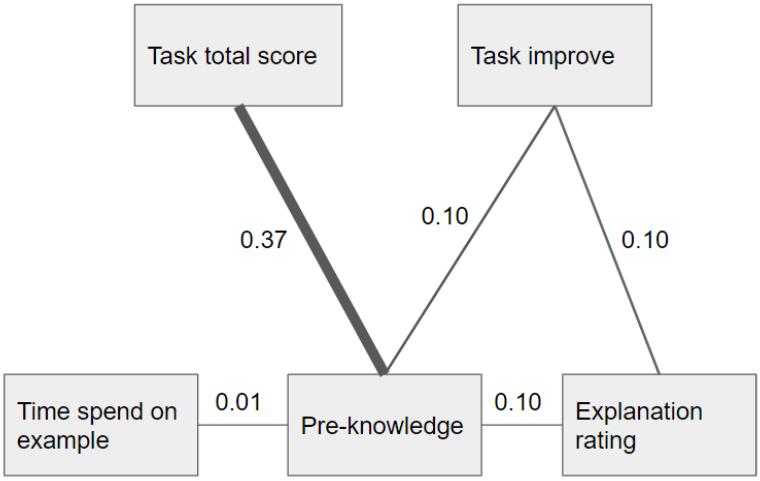


Figure 29. Feature Connection Modeling for Advanced Class

Moreover, the effect of MG explanation has little impact on LG compared to the novice class.

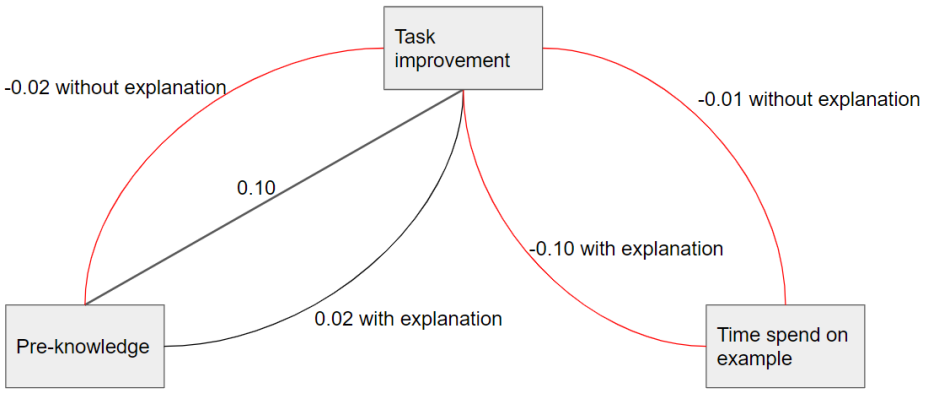


Figure 30. Feature Connection Modeling with and without Explanation Feature.

Hypotheses Results

The experiment clearly revealed the result of hypotheses.

The hypothesis 1 compares the learning gain between the experimental group and control group. As revealed in the experiment result, for both novice class (p-value = 0.31)

and advanced class (p-value = 0.76), the hypothesis A are rejected. In other words, the improvement of learning effect brought by code explanation in a single lab class is not significant.

The hypothesis 2 and 3 compares the pretest performance and learning gain between novices and advanced students. The experiment result shows that the pretest performance ($p < 0.001$) difference between novices and advanced students is significant, while since these two classes took different materials to learn different concepts, their learning gain cannot be compared directly.

The hypothesis 4 & 5 are related to factors potentially affecting learning gain other than the code explanations. In the experiment result, the correlations between ratings and learning gain are not significant for both novices ($cor = -0.16$, $p = 0.41$) and advanced students ($cor = 0.03$, $p = 0.84$), and the connections between pretest performance and learning gain are also not significant for novices ($cor = 0.07$, $p = 0.61$) and advanced students ($cor = 0.09$, $p = 0.43$). In conclusion, both hypotheses are rejected.

Impact Comparison

In this section, I will discuss the difference between the experiment results in the two classes. The experiment on novice class is named Study I, and the advanced class experiment is Study II. The comparison will be conducted from perspective including the students' pre-knowledge, the learning gain effect, the predictability in learning, the ratings of explanations, and their subjective feedback in post-survey.

The two studies were conducted in one two classes with pre-knowledge differences, in which the Study I was conducted on the class formed by novices; the

Study II was conducted on advanced students. There was a pre-knowledge survey and test established in both classes at the beginning of the semester to verify their background knowledge level difference.

To control the variables, the studies for these two classes are conducted with the same system and procedure. However, considering the different contents taught in both classes, the topics involved in the two studies are different: in Study I, the novices learned “for loop” from code examples; while in Study II, the advanced students learned “big O notation”.

The procedure of both studies started from an introduction phase, in which they are informed about the purpose of the study, the process of the study, and what they can expect from the experiment. Then in the second phase, the topic concept of the experiment was introduced in text for students. The purpose of this phase was to have students review it and reach the same knowledge baseline. In the third phase, the students were given a pre-activity test to record their level of understanding of the topic concept according to their performance on a set of coding related multiple-choice questions. Then in the fourth phase, the students were required to further learn the topic concept referring to a set of code examples and execute results. In this phase, the students were randomly split into the experiment group (group E) and the control group (group C). In the experiment group, the students were provided not only code examples, but also line-wise explanations. Moreover, they were required to read the explanations carefully and give rate for each explanation. Meanwhile, for the students in group C, the code examples

were provided without any explanations. In the last phase, the students were given a post-activity test to record their understanding of the concept after learning from examples.

After the experiment process, both classes were given a post-survey to collect their feedback and comments about the experiment system and content provided.

Pre-knowledge Difference

The pre-knowledge difference in programming between the two classes in the two studies is critical since they should be able to represent different groups of learners.

According to the pre-survey and pre-knowledge test, in Study I class, 30% of students had zero experience in programming; 67.3% of students achieved less than 20% points in pre-test; in Study II, 74.6% of students had more than a semester; 56.9% students achieved more than 50% points in pre-test.

This result validated that most students in Study I are novices, while students in Study II are more advanced.

Learning Gain Comparison

In both studies, the learning gain was compared between the control group (group C) and experimental group (group E). In study I, the difference between group C (10.1% +- 19.1%) and group E (20.4% +- 25.2%) is larger than the study II (group C 17.2% +- 27.0% and group E 22.3% +- 29.5%). This result indicates that the MG syntaxial explanations are more suitable for novices when learning syntax concepts, while provides less help for advanced students learning concepts more related to complicated logics.

Predictability Comparison

The predictability of a student is defined as the correlation between time spent in learning and learning gain after learning. With this definition, a student with good learning predictability will achieve better learning gain if he/she spent more time.

In study I, the overall predictability for group E is 0.17, while group E is -0.07. This difference indicates that students benefited from the MG explanations by achieving a better predictability, and passively spend more time in reading the explanations.

However, in study II, the difference between group E (-0.01) and group C (-0.16) is less obvious, and both are negative. This result suggests that for advanced students, the MG explanations do not affect in the same way as novices. A different model should be built to assist this community.

Explanation Ratings

In experimental groups, the students were not only provided code explanations, but also required to rate each of them. By calculating the average rating for each code line explanation, the subjective evaluation of the contents generated can be achieved. Meanwhile, by calculating the average rating given by each student, the connection between student's rating and his/her learning gain is also revealed.

According to the ratings collected, both studies received unexpectedly high ratings compared to the accuracy ratings in pilot study, and the study I is higher than study II. This result suggests that there is a trend that the learners would over evaluate materials compared to experts, especially novices.

This conclusion is supported when analyzing the average rating for each student. The correlation between ratings and learning gain is negative for study I -0.16. Furthermore, by filtering the students rating all explanations 5, the correlation between ratings and learning gain became higher for both studies.

Subjective Feedback

In both studies, students were required to finish a post-survey after experiencing the learning system. In the post-survey, students were asked to rate their experience (1 to 5) about the system interface design, concept understanding, and example helpfulness, and further suggestions about the system. Both studies received positive ratings. In study II, more complaints were received that the explanations were not reflecting the deeper logic related to the concept "big O notation", which supported the findings about the limitation of MG explanations for advanced students.

In the result analysis, the result comparison is shown in Table 9.

Table 9
Comparison Summary Between Novice Class and Advanced Class

	Study I (Novice Programming Learners)	**Study II (Advanced Programming Learners)**	**Comparison discussion**
Pre-knowledge	30% of students had zero experience in programming; 67.3% of students achieved	74.6% of students had more than a semester; 56.9% students achieved	The majority of students in Study I are novices, while

	less than 20% points in pre-test.	more than 50% points in pre-test.	students in Study II are more advanced.
Learning gain	After example learning, students with explanations obtained (20.4% +- 25.2%) more questions on average correct; Without explanations, students gained 10.1% +- 19.1%. No significant difference was found.	Average improvement for students with explanation was 22.3% +- 29.5%; The students without explanations achieved 17.2% +- 27.0%. No significance was found	In both studies, the groups provided explanations obtained more improvement after learning. However, this difference is not significant.
Predictability	Correlation between time spent in learning and performance improvement after learning is 0.17 for students with explanations; The	Both groups had a negative correlation. Students with explanations are -0.01 and the others got -0.16.	For novices, with the help of code explanation, they can learn better by spending more time in example learning, while for novices

	other students got -0.07.		without explanations or advanced
Explanation ratings	The average rating for explanations in the study was 4.34 +- 0.27, while the accuracy rating in pilot study was 2.82 +- 1.48.	The average rating in this study (3.93 +- 0.25) was higher than pilot study. Besides unexpected high ratings, a set of students rated low.	In novices' class, students rate every explanation high performed worse than average. While in advanced class, some students are not satisfied.
Subjective Feedback	The feedback collected is positive in general. Students requested for better explanations and higher-level functions in the system.	The feedback is even more positive. Students claimed that high level logic is not captured.	The feedback in both classes is generally positive. However, the novice class expected to further learn from examples by running them, while advanced students expected the explanations to be

logic related, instead of syntax related.

The impact of MG code on advanced students was not observable in learning gain and learning predictability. A possible reason was that for advanced students, the “Big O notation” was a logical concept instead of syntax concept, so the logic meaning of the code examples was more important than their syntax meaning. To this degree, the effect of code explanation was limited even though the students in group E still need to spend more time in explanation rating.

CHAPTER 7

DISCUSSION & CONCLUSIONS

Summary Discussion

Obstacles Identified

The series of studies of mine (Lu, Y., Hsiao, I-H. 2015; Lu, Y., Hsiao, I-H.& Li, Q. 2016; Lu, Y., Hsiao, I-H. 2017) have revealed that the programming novices have obstacles in learning when they are learning with materials retrieved by themselves.

To assist novices, I have established studies to record and analyze the behavior of learners when they are learning from materials online and implemented learning platforms to help students form search queries, filter search results, and understand coding requirements. However, one of the biggest obstacles remains that they have problems in understanding online materials, especially code examples. To overcome this obstacle, I utilized deep-learning models for the machine generated (MG) code example explanations.

MG Explanations are Readable

With the help of Amazon Mechanical Turks (AMT), I collected human explanations for almost 10K lines of codes. With the data collected as training data, a deep-learning translation model was trained to generate syntax level code explanations. In the pilot study validating the quality of MG explanations, the quality of MG code explanations was proved to be compatible with humans in readability, while the accuracy of explanations was not ideal, especially for concepts involving further contexts and deeper logics. This result indicated that the MG explanations are feasible to help novices

with syntax problems, but its effect on advanced students may be limited due to the inaccuracy.

MG Explanations Helps Novices

According to the two lab-class experiments, the novices received help from MG code explanation by improving their predictability, which means by spending more time, the novices provided code explanations will achieve more stable learning gain and it is positive. The improvement is potentially brought in multiple potential ways including content reminding, focus keeping, and more time spending. However, the novice control group and all advanced students did not have significant coefficients between learning gain and time spend. This result illustrated the function boundary of the current explaining model and highlighted the unique requirements of novices.

New Model Required for Advanced Students

The advanced students have expectations on code explanations to illustrate deeper logics in code, which is difficult for the current machine model to fulfill. This fact indicates future works to develop more applicable models that have higher accuracy of MG explanations and involve more code contexts to cover more complicated concepts and deeper logics.

Contributions

In this study, the following contributions are made:

- Identify the obstacles for novice learners in programming.
- Utilized deep-learning-based model to generate natural language explanations for code lines for the purpose of education.

- The effect of MG code explanation in learning was evaluated in classes of students with different knowledge backgrounds.

After years of study, I have identified multiple requirements and obstacles for novices in learning programming. To fulfill their requirements and help the novices learn programming, I utilized the example learning theory in programming, which has been studied for decades (Brusilovsky & Weber, 1996; Brusilovsky, 1992; Burow & Weber, 1996; Faries & Reiser, 1988; Guzdial, 1995; Hohmann, Guzdial & Soloway, 1992; Linn, 1992a; Linn, 1992b; Redmiles, 1993), extended it with machine generated (MG) code explanation to further help novices understand examples. The model performance could be taken as a baseline for future studies. In this work, I also investigated the potential and effects of MG code explanations in programming education. Based on this background, my research questions for this study is raised as:

RQ1: **What do novices need in programming learning?**

RQ2: **How to explain code for novices with machine generated (MG) language?**

RQ3: **How do programming learners benefit from machine generated (MG) annotated examples in declarative and procedural knowledge learning?**

To answer these research questions, I designed a system helping novices learning programming syntax concepts by providing explained code examples. To achieve better feasibility, the system is designed to automatically generate code explanations utilizing tree-structured LSTM translation model to translate programming code into descriptive English.

To start the model training, labeled data is required as the training data. In this research, a set of Java code was collected from textbooks and GitHub projects as the code examples, and the codes were posted on the Amazon Mechanical Turks (AMT) platform to collect human explanations. In order to guarantee the quality of explanations collected, a set of programming questions were given to turkers as qualifiers on AMT to qualify code experts. During the explanation collection, each Java code was assigned to three turkers to maintain the diversity of human explanations.

To evaluate this translation model, a pilot study was conducted. In the experiment, 6 complete codes with 103 lines in total were randomly sampled in the test data set, and 4 programming experts participated in the study. Among the 103 sample lines, 50 of them were randomly selected and explained by humans, and the other 53 were explained by tree-structured LSTM. The pilots did not know the source of each explanation. The pilots were required to evaluate each explanation in two perspectives: whether it is readable (Y=readable; N=unreadable) and the level of accuracy (1 to 5, 1=totally inaccurate; 5=totally accurate). In the evaluation result, MG explanations were compatible with human explanations in readability, while the accuracy is not as good as human's, especially for codes with complicated logics. However, the model worked well generating simple syntaxial explanations. This result indicates that programming novices can potentially benefit from the system by achieving syntaxial explanations of codes.

To further prove the effect of the system in learning, and clarify its boundary or limitations, two studies were established in real programming classes. The study I was established in a programming novice class teaching JavaScript, and the study II was

conducted in a class of relatively advanced students learning Java. Both experiments were conducted with the same online learning system, in which the students experienced 5 phases: In phase 1 of the experiment, the students went through a material helping review the concept, which set them to the same baseline. Then in phase 2, the students took a pre-activity test before example learning by answering 4 multiple choice questions. For each question, after they made a choice and submit, there would be a hint pop out and the correct answer will be highlighted. Then in phase 3, the students were given 5 code examples to learn. In this phase, the students were randomly split into control group (group C) and experiment group (group E). In group C, the students were only provided the example code and the execution result; in group E, besides the example code, the students were also provided code explanations generated by machine. The group E students are required to read and rate the helpfulness for each explanation from 1 to 5. After the example learning, in phase 4 the students took a post-activity test by answering another 4 multiple choice questions, which has exactly the same setup as pre-activity test. Considering the learning progress of both classes in the semester, the concept introduced in the novice class was “for loop”, and the concept topic for advanced students was “big O notation”.

In the analysis result, for novices the learning gain difference between group E and group C were not significant. However, the predictability difference between the group E and group C for both studies are large. This result indicates that with code explanations, students are more likely to achieve better performance after spending more time in learning from examples. Besides the value of the model and system, the limitation of MG code explanation is also investigated in the experiments. In study II, the effect of

MG explanation on predictability has disappeared on the advanced students, which indicates that the syntaxial explanations are not suitable for advanced students who are learning concepts with complicated logics.

Limitations

The experiment results revealed a set of limitations.

The accuracy of the model is not ideal enough for complicated code explanation generation. Since the explaining model is based on LSTM translation models, the training dataset has a huge impact on the translation quality. In the model training process, the "understanding" of a code term is strongly impacted by the frequency of the term. However, the frequency of terms is not uniform. There is always a long tail of low frequency terms. When training the model, there is a lack of information input related to these low frequency terms, which lead to misunderstandings and inaccuracy.

Besides the long-tail problem, the explaining model also has problems in capturing the larger scale context and deeper logic. Different from natural language, programming language is structured more densely, which means context from further is more involved. This difference brings a difficulty to the model training, because the human language translation model does not consider long-distance context, which is another limitation of my work.

Besides the model developing, the experiment also has limitations. The essential value of the code explanations is not fully revealed in the experiment. The first reason is that the lab-class is not a long-term experiment, which could not fully capture the

learning effects brought by the explanations, and the result is heavily noised by factors including students' pre-knowledge and class-learning effect.; The second reason is that the experiment was designed to evaluate the learning gain brought by example code explanation, while the model does not only explain examples, but can also explain the student's own code and illustrate the unexpected parts, which is another potential of the model.

Future Work

Due to the limitations discussed above, there are a lot of promising future works in this study. In the translation model design, more context can be involved to provide explanations for codes with complicated logics; In the model training process, more data is required since the performance of the model still has large space to improve; In the experiments, long term studies could evaluate the model and system in a more comprehensive perspective, and capture more clues proving the learning effects caused by MG code explanations. Moreover, long term studies also help collect fluence details of the learning process of different students, which further help to build up a more complete model describing how novice learn programming from code and its explanations.

Besides the system of example learning with code explanation, another promising work is the self-explaining system, which could illustrate the bugs or unexpected parts in the students' own code. This system requires a larger data set for training, which includes codes with bugs or errors, and explained correspondingly.

REFERENCES

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. Review of educational research, 70(2), 181-214.

Bartha, S., & Cheney, J. (2019). Towards meta-interpretive learning of programming language semantics. arXiv preprint arXiv:1907.08834.

Bates, M. (1995). Models of natural language understanding. Proceedings of the National Academy of Sciences, 92(22), 9977-9982.

Becker, Brett A. "An effective approach to enhancing compiler error messages." Proceedings of the 47th ACM Technical Symposium on Computing Science Education. ACM, 2016.

Brusilovsky, P. (2001, October). WebEx: Learning from Examples in a Programming Course. In WebNet (Vol. 1, pp. 124-129).

Brusilovsky, P. and Weber, G. (1996) Collaborative example selection in an intelligent example-based programming environment. In: D. C. Edelson and E. A. Domeshek (eds.) Proceedings of International Conference on Learning Sciences, ICLS'96, Evanston, IL, USA, AACE, pp. 357-362, http://www.contrib.andrew.cmu.edu/~plb/papers/icls96.html.

Brusilovsky, P. L. (1992) Intelligent Tutor, Environment and Manual for Introductory Programming. Educational and Training Technology International 29 (1), 26-34.

Butler, M., & Morgan, M. (2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. In *Proceedings ascilite Singapore* (No. 99–107).

Burow, R., & Weber, G. (1996, June). Example explanation in learning environments. In *International Conference on Intelligent Tutoring Systems* (pp. 457-465). Springer, Berlin, Heidelberg.

Chang, K. E., Chiao, B. C., Chen, S. W., & Hsiao, R. S. (2000). A programming learning system for beginners-a completion strategy approach. IEEE Transactions on Education, 43(2), 211-220.

Chi, M. T., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. Cognitive science, 13(2), 145-182.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

Chozas, A. C., Memeti, S., & Pllana, S. (2017). Using cognitive computing for learning parallel programming: An IBM watson solution. arXiv preprint arXiv:1704.01513.

C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In Proceedings of the 32nd International Conference on Machine Learning, International Conference on Machine Learning, pages 1093–1102, 2015

Collobert, R., & Weston, J. (2008, July). A unified architecture for natural language processing: Deep neural networks with multitask learning. In Proceedings of the 25th international conference on Machine learning (pp. 160-167). ACM.

De Oliveira, M. G., Ciarelli, P. M., & Oliveira, E. (2013). Recommendation of programming activities by multi-label classification for a formative assessment of students. Expert Systems with Applications, 40(16), 6641-6651.

Denny, P., Luxton-Reilly, A., & Carpenter, D. (2014). Enhancing Syntax Error Messages Appears Ineffectual. In Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (pp. 273–278). http://doi.org/10.1145/2591708.2591748

Dowty, D. (1994, November). The role of negative polarity and concord marking in natural language reasoning. In Semantics and Linguistic Theory (Vol. 4, pp. 114-144).

Faries, J. M. and Reiser, B. J. (1988) Access and use of previous solutions in a problem solving situation. In: Proceedings of Tenth Annual Conference of the Cognitive Science Society, Montreal, 1988, Lawrence Erlbaum Associates, pp. 433-439.

Guzdial, M. (1995) Software-realized scaffolding to facilitate programming for science learning. Interactive Learning Environments 4 (1), 1-44.

Hohmann, L., Guzdial, M., and Soloway, E. (1992) SODA: a computer-aided design environment for the doing and learning of software design. In: I. Tomek (ed.) Proceedings of 4th International Conference, ICCAL'92, Berlin, Wolfville, Canada, June 17-20, 1992, Springer-Verlag, pp. 307-318.

Linn, M. C. (1992a) Can experts' explanations help students develop program design skills. Int. J. Man-Machine Studies, International Journal on the Man-Machine Studies 36, 511-551.

Linn, M. C. (1992b) How can hypermedia tools help teach programming. Learn. Instr., Learning and Instruction 2, 119-139.

Redmiles, D. F. (1993) Reducing the variability of programmers' performance through explained examples. In: Proceedings of INTERCHI'93, New York, Amsterdam, 24-29 April 1993, ACM, pp. 67-73.

Herlocker, J. L., Konstan, J. A., Terveen, L. G., & Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. ACM Transactions on Information Systems (TOIS), 22(1), 5-53.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

Hu, X., Li, G., Xia, X., Lo, D., & Jin, Z. (2018, May). Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension (pp. 200-210). ACM.

Hulme, C., Maughan, S., & Brown, G. D. (1991). Memory for familiar and unfamiliar words: Evidence for a long-term memory contribution to short-term memory span. Journal of memory and language, 30(6), 685-701.

John R Anderson, C Franklin Boyle, Albert T Corbett, and Matthew W Lewis. 1990. Cognitive modeling and intelligent tutoring. Artificial intelligence 42, 1 (1990), 7–49.

Labutov, I., & Studer, C. (2016). Calibrated Self-Assessment. International Educational Data Mining Society.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. Acm Sigcse Bulletin, 37(3), 14-18.

Li, Y. (2017). Deep reinforcement learning: An overview. arXiv preprint arXiv:1701.07274.

Lin, Y. C., Hong, Z. W., Liao, Y. H., Shih, M. L., Liu, M. Y., & Sun, M. (2017). Tactics of adversarial attack on deep reinforcement learning agents. arXiv preprint arXiv:1703.06748.

Liu, H., & Singh, P. (2004, September). Commonsense reasoning in and over natural language. In International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (pp. 293-306). Springer, Berlin, Heidelberg.

Lu, Y. & Hsiao, I-H. (2016) Seeking Programming-related Information from Large Scaled Discussion Forums, Help or Harm? The 9th International Conference on Educational Data Mining, Raleigh, NC, USA

Lu, Y. & Hsiao, I-H. (2017) Personalized Information Seeking Assistant (PiSA): From Programming Information Seeking to Learning, Information Retrieval Journal

Lu, Y. & Hsiao, I-H. (2017) Toward understanding novices' search process in programming problem solving, Frontiers in Education Conference 2017

Lu, Y., & Hsiao, I. H. (2018, July). Modeling Semantics between Programming Codes and Annotations. In Proceedings of the 29th on Hypertext and Social Media (pp. 101-105). ACM.

Lu, Y., Hsiao, I-H.& Li, Q. (2016) Exploring Online Programming-related Information Seeking Behaviors via Discussion Forums, 16th IEEE International Conference on Advanced Learning Technologies, July 25-28, 2016, Austin, Texas, USA

Malan, K., & Halland, K. (2004, October). Examples that can do harm in learning programming. In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (pp. 83-87). ACM.

McAllester, D. A., & Givan, R. (1992). Natural language syntax and first-order inference. Artificial Intelligence, 56(1), 1-20.

Montague, R. (1970) Synthese 22, 68-94.

Nasehi, S. M., Sillito, J., Maurer, F., & Burns, C. (2012, September). What makes a good code example?: A study of programming Q&A in StackOverflow. In 2012 28th IEEE International Conference on Software Maintenance (ICSM)(pp. 25-34). IEEE.

Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002, July). BLEU: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics (pp. 311-318). Association for Computational Linguistics.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. New ideas in psychology, 2(2), 137-168.

Pekrun, R., Goetz, T., Frenzel, A. C., Barchfeld, P., & Perry, R. P. (2011). Measuring emotions in students' learning and performance: The Achievement Emotions Questionnaire (AEQ). Contemporary educational psychology, 36(1), 36-48.

Pettit, R. S., Homer, J., & Gee, R. (2017). Do Enhanced Compiler Error Messages Help Students?, 465–470. http://doi.org/10.1145/3017680.3017768

Price, T. W., Dong, Y., & Barnes, T. (2016). Generating Data-Driven Hints for Open-Ended Programming. *International Educational Data Mining Society*.

Reiser, B. J., Anderson, J. R., & Farrell, R. G. (1985, August). Dynamic Student Modelling in an Intelligent Tutor for LISP Programming. In IJCAI (Vol. 85, pp. 8-14).

Reiter, E. (1994, June). Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In Proceedings of the Seventh International Workshop on Natural Language Generation (pp. 163-170). Association for Computational Linguistics.

Serrano-Cámara, L. M., Paredes-Velasco, M., Alcover, C. M., & Velazquez-Iturbide, J. Á. (2014). An evaluation of students' motivation in computer-supported collaborative learning of programming concepts. Computers in Human Behavior, 31, 499-508.

Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., & Matsumura, T. (2019). Automatic Source Code Summarization with Extended Tree-LSTM. *arXiv preprint arXiv:1906.08094*.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in neural information processing systems (pp. 3104-3112).

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. Cognitive science, 12(2), 257-285.

Tai, K. S., Socher, R., & Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint arXiv:1503.00075.

Vesin, B., Ivanović, M., KlašNja-MilićEvić, A., & Budimac, Z. (2012). Protus 2.0: Ontology-based semantic recommendation in programming tutoring system. Expert Systems with Applications, 39(15), 12229-12246.

Weiser, M., & Shertz, J. (1983). Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies*, *19*(4), 391-398.

Wen, T. H., Gasic, M., Mrksic, N., Su, P. H., Vandyke, D., & Young, S. (2015). Semantically conditioned lstm-based natural language generation for spoken dialogue systems. arXiv preprint arXiv:1508.01745.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, *23*(4), 383-390.

Wong, E., Liu, T., & Tan, L. (2015, March). Clocom: Mining existing source code for automatic comment generation. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER) (pp. 380-389). IEEE.

Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent trends in deep learning based natural language processing. ieee Computational intelligenCe magazine, 13(3), 55-75.