Analyze Programming Languages

Yiqiao Yin
W.Y.N. Associates, LLC

Abstract

This assignment reports the research findings for analysis of programming languages. The goal is to understand basic programming languages such as C++. There are two parts for this assignment report. First, the report focuses on understanding the basics of programming languages. Second, the report summarizes some current research built using C++ where the C++ forms the basis of the research (all related papers are two years recent and from peer-reviewed sources).

1 Understanding C++

1.1 Introduction

The modern day advancement in fields such as healthcare, technology, finance, and even agriculture all require the usage of many computers at different perspectives of the business parts. It is extremely challenging to operate or manage business without the participation of computers. Hence, the development of problem-solving skills with coding is extremely important.

Fundamentally, the task of coding by itself is a form of problem-solving. In practice, it is often time the case to be able to write many different approaches of solutions tackling one problem. Hence, a flexible coding style is also very important. The understanding of the programming languages can largely increase a person's logical thinking as well as execution in a neat and elegant manner.

With such motivation discussed as the above, it is important to, first of all, address two basic types of languages: assembly language and machine language. Assembly language or ASM is a low-level language where the relationship between the language instruction and the skeleton of code presentation is strong Juneja (2009); Sinkov (1963). We use ASM as a replacement for assembly language for the rest of the paper. The style ASM is short and elegant. It usually consists of a simple statement or execution of code. The coding style is also short and tight with on an one-to-one basis. For the execution of ASM, the code is converted into a small program that is called assembler, which matches the concept where the name ASM suggests. The ASM uses the term assembler to refer to this unity program that is created by this executable machine. A fun fact is that the work Wilkes et al. (1951) introduces the term "assembler" in the first time in history. This concept also lays the ground of today's terminology "assemble machine". It is a necessary building block to bridge the gap between software programs and the hardware platforms, because ASM translates high-level language into machine language. The most

basic example in a computer for ASM is the binary code consists of zeros and ones. It is rare for an organization to put humans directly in charge of writing code such as ASM. This is where machine language shows its strength. Machine language is a low-level programming language that consists of language instructions Juneja (2009); Sinkov (1963). It is used to control the central processing unit or CPU. The instructions to control CPU often carries a very particular task. Ideally, the code is written to execute the jobs or tasks sent to the CPU as efficient as possible which also requires the knowledge of data structures. As opposed to the ASM, machine language can be done by human but the management style requires detailed investigation. Machine language allows us to build more complicated computer program to send command to CPU inside a computer. A comparison is presented between the ASM and machine language in Table 1.

Assembly Language	Machine Language
Between high- and low-level	Low-level language
Uses English alphabet	Uses zeros and ones
Understood by human	Understood by CPU
Memory exists for instructions	No memory

Table 1: **Comparison between ASM and Machine Language**.

1.2 Famous Applications

A famous coding language is C++. In real world application, there are many areas where C++ can provide crucial developmental role in the process.

The first example is the gaming industry. The design of a video tracks down to the core of the hardware, which is something C++ is really good at. The programming language is designed to easily operate. The procedure of game programming over CPU consists of intensive coding and fast executed coding structure. It is sometimes organized over multiple network and can involve 3D design with multi-player setup. Famous games such as Witcher 3, Counter-Strike, Warcraft, and Doom are great application products using C++.

The next field of application is Graphical User Interface or GUI, which is a type of user interface that bridge the gaps between user and graphical icons instead of code. It is a system of interactive visual components for computer software of which C++ shows significant strength at. Famous applications in GUI are Adobe and Windows Media Player.

For data scientists and senior data scientists, the workload cannot leave the realm of data structures. This is an area where database management can be extremely important for the role of a data scientist. The C++, in this case, serves as a great tool to develop writing of the database management software. Famous examples are MySQL and Postgre.

A whole list can be found in Section 1.5 of the work by Juneja (2009). We present a few selected applications here.

1. Computer operational systems such as Windows XP and OS.
2. Real-time control applications.
3. 2D and 3D animations and gaming design.
4. Speech recognition and dictation systems.

1.3 Execution Diagram

The purpose of a compiler is serves as a processor in the creation of the binary code. The process the procedure of developing a program using a language such as C++ is summarized below in Figure 1.

To understand the executing order of the diagram from the work by Juneja (2009). The procedure starts with a text editor. This is the place where the code or the English syntax is typed into a script. This syntax is tailored to each computer language. It is recommended to have a round of bug check. This is indicated using a diamond shape box in the diagram. If there is no error, the procedure continues. If there is an error, it is recommended to go the beginning which is the text editor to rewrite the code which is the English syntax. Next, a preprocessor is used to go through the source code for the compiler. Then a compiler takes over to create a binary file for source code. This source code needs to be linked, which gains its name "linker". The linker rends the code and reports bugs when the code has errors present. If that is the case, it is recommended to go back to the syntax part which is the first step to start over. However, if there is no bugs reported, the program successfully renders and compiles the code. Then the program executes the binary file and it is ready to run. The user can then execute the program and runs the program. There is another round of bug report. If there is any bug present when running the program, it is recommended to go back to the syntax or text editor for a third time. If the program is free of bugs, the program runs, executes, and the results are generated.

2 C++ in Research

2.1 Background

This portion of the assignment investigates and reviews the work by Abuhamad et al. (2021). The field of authorship identification is fundamentally a natural language processing task. Many former scholar has conducted thorough research in this field Juola (2008); Kešelj et al. (2003); Koppel et al. (2009); Stamatatos (2009). However, there is little work done in authorship identification in source code and proposed computer programs Burrows et al. (2014). Based on the unique stylometric features of an author or a programmer, source code can be structurally written in a fundamentally different approach. It is this distinct nature that allows each programmer to tailor their experience according to the pipeline of the particular coding project. From linguistic perspective, the notion of stylometric serves a key purpose in this context, because stylometric is the analysis of features of the writing style that can be statistically and quantitatively measured which can be a range of features such as sentence length, vocabulary diversity, frequency, order of phrases, and so on.

Their work investigated different stylometry of the writings. This means that experiments are conducted to compare for the intrinsic writing style in machine learning algorithms. In this case, the concept of stylometry of stylometric feature is referring to the writing style of machine learning algorithms or code in general that can be quantified statistically. These features can refer to sentence length, vocabulary diversity, and frequencies (of words, word forms, and so on).

The motivation for the work by Abuhamad et al. (2021) is to develop a system for a numerous of applications that can take advance or be built upon the understanding of authorship syntax and writing style. Software identification is a major component of software forensics and security analysis. A major benefit is to establish potential

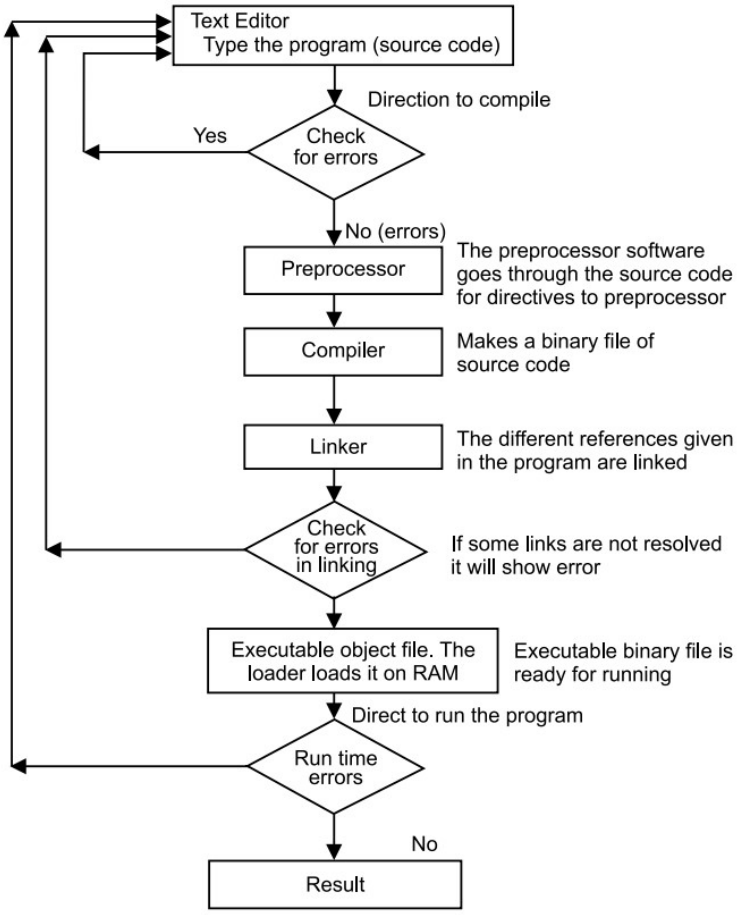

Fig. 1.4: Sequence of processes in development of a program in C/C++ language. (Also See Appendix E for more details on starting programs in C++)

Figure 1: **Diagram of the Developmental Procedure**.

malicious code or even malware that intend to deliberately deliver malicious activity. Another application can be the adaption and potential development of sophisticated plagiarism detection Burrows et al. (2007), authorship disputes Wilcox (1998), copyright infringement, and software integrity investigations Malin et al. (2008).

2.2 Review of Literature and Similar Work

To tackle the above problems, recent work in software authorship identification has proposed several interesting techniques Koppel et al. (2009); Spafford and Weeber (1993); Burrows et al. (2007); Caliskan-Islam et al. (2015). However, their work cannot be generalized to other field and it is challenging to apply the same feature extraction technique to other programming languages such as Java or Python. In other words, the model built for identifying authorship in C++ should only stay in documents and source code that is C++ based. A former study proposed writeprints which is a technique that captures the author's stylometry in diverse corpus Abbasi and Chen (2008). The work by Uzuner and Katz (2005) conducted comparative study to investigate different stylometric information in the author's writing to identify the author's attribution and identification. The work is taken further into the possible attack of cybercriminals by using textual entries Afroz et al. (2014).

2.3 Role of C++ in This Research

The research uses Term Frequency-Inverse Document Frequency or TF-IDF method to extract features from the code syntax and input the extracted features into a Recurrent Neural Network (RNN). The authors showed state-of-the-art prediction performance on a number of experiments. Amongst all 5 listed contributions in the paper, 4 of the data sets consists of C++ and the language can also be compared with C, Java, and Python.

2.4 Research Question

Though the authorship style is unique, the source code is practiced in the following way. In order to process the source code, a program is compiled to the original source code and this is in executable binary bits. This is then decompiled to generate pseudo-code. The syntax at a low-level construction is largely dependent of the language itself and can impose an inherent inflexibility in the architecture of the code. This nature poses great challenge for machine learning scientists to learn about the stylometric of the text using source code. In plain English, there are only so many ways to write a chunk of code and at a basic level of the code it is common that most people write the code in a similar way in terms of the style features used in the stylometric analytical system.

Several questions raised by the author Abuhamad et al. (2021) that deserve attention are listed in the following. Is it possible to use deep learning technology to help investigate or potentially classify the software authors? If yes, how can this experimental procedure be done? In addition, how can we generalize deep learning based approach to wider syntax? If deep learning can be used in one type of syntax, can this methodology be used in other syntax without the prior knowledge? In the case of using this technology in other languages, can the result still perform robustly or will the performance be damaged due to new language or syntax environment?

2.5 Research Findings

The authors developed a sequential model with Recurrent Neural Network or RNN. The model aims to extract the features from the syntax database and use these extracted features to construct the machine learner. For each instance, the authors fed in either complete or incomplete source code or syntax to create explanatory features. These features are generally believed to have high quality and contains distinctive authorship stylometric. Unlike former research that relies on prior knowledge for data transformation, the proposed work extract features automatically using RNN-related deep learning methods. The author proposes to use Term Frequency-Inverse Document Frequency (TF-IDF) Burrows and Tahaghoghi (2007); Frantzeskou et al. (2006); Kešelj et al. (2003). Since TF-IDF is non-bayesian, it does not require any prior knowledge of any particular programming language and nor does it need any high-level translation. Hence, the author believed that it is a more resilient approach to language specifics. It is also beneficial because it can be generalized language-wise when the source code is accessible at hand and the program can be processed into binary format. The important features that help explain the predictions, after the model is trained and fitted on the training dataset, are famous keywords of the used programming language. This result poses an interesting complexity, because this indicates that the authorship cannot be easily identified by variable names but by their dramatically different programming style. The authors are able to achieve the state-of-the-art accuracy using the extracted features.

The authors provided a sequential model that is RNN based and delivered the state-of-the-art prediction performance using features extracted from TF-IDF. Over large dataset of 150 C++ programmers, the prediction performance is near perfect and at a 99% accuracy. An even larger dataset with a huge number of programmers (8,903 programmers) is tested using the same proposed approach and the accuracy is 92.3%.

In addition, to ensure the features extracted are meaningful, the authors also used a random Forest Classifier or RFC as a classifier to learn from the TF-IDF-based features. The RFC reads in the features and then build a classifier by splitting different trees. The number of trees is a tuning parameter. Each tree behaves like a decision tree and grows many branches. Each branch is a split from comparing the feature at that branch with a particular threshold. An advantage of the proposed model is that it requires only seven files for each programmer in a dataset that has 5.5 times the number of programmers than that of the prior work and that it is able to create similar performance.

The third contribution from Juneja (2009) is that analysis is tested on different programming languages such as C++, C, Java, and Python. Analysis is also done using combination of different programming languages (C++/C, C++/Java, and C++/Python). The purpose is to test out whether the proposed model can be generalized enough for multiple different computer languages.

2.6 Limitations and Future Research

There are some limitations to the author's paper and these limitations lead to future research opportunities. This report summarizes below.

First, the experiments are conducted under the assumption that each source code sample is finished with a single programmer. In other words, in a tabular data form, each row can be considered as a single observation or an instance where this observation happens to be only a single programmer. There assumption dictates that there is no situation where more than one programmers collaborate and finish the source code sample together. This assumption rarely holds true in practice, because the common practice is a team

of programmers collaborate and check each others' code. In that case, this assumption does not hold true and we need to release the assumption by accepting the situation that collaboration can happen in the industry.

Second of all, the experiments assumes that the authorship and stylometric of every programmer is unique and poses great differences amongst each other. In addition, the experiment assumes that the programming styles do not vary sample by sample on a single programmer. These assumptions rarely hold true. The reality is that there is honor and respect amongst programmers. The manager could influence the style of junior programmers. The reason is because it is unlikely for junior programmers to start writing an entire code repository for their first industry project. In practice, the senior engineers usually lead the junior employees into the doors and get them started by some shadowing. This formality ensure a safe transition to deliver sophisticated code production but also is able to train new employees for higher-level job tasks. However, this nature implies that it is likely for the junior employees to take on some of the flavor or style from senior engineers. Since this is the first real project, junior employees may take it to heart and set it as a "benchmark" for all future project. This means that the coding styles can be passed from a senior engineer to a junior employee, which means the coding style is not unique to each individual.

A third assumption is the size of the code in production follows that of the dataset. This assumption cannot be held true in practice. In a clean, processed, and manipulated data set, each instance could have similar length. In practice, it is unlikely that each code project has the same length. In similar job tasks, this can sometimes be true. However, it is a very difficult assumption to defend. In reality, the length of the code tends to be dependent on the nature of the job tasks. Machine learning code can be short when the model is called from a previously trained project. The same code can also be long because sometimes it may be beneficial to design a customized model that can be tailored to the particular dataset in hand which can produce extremely lengthy code.

References

Abbasi, A. and Chen, H. (2008). Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Transactions on Information Systems (TOIS)*, 26(2):1–29.

Abuhamad, M., Abuhmed, T., Mohaisen, D., and Nyang, D. (2021). Large-scale and robust code authorship identification with deep feature learning. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–35.

Afroz, S., Islam, A. C., Stolerman, A., Greenstadt, R., and McCoy, D. (2014). Doppelgänger finder: Taking stylometry to the underground. In *2014 IEEE Symposium on Security and Privacy*, pages 212–226. IEEE.

Burrows, S. and Tahaghoghi, S. M. (2007). Source code authorship attribution using n-grams. In *Proceedings of the twelth Australasian document computing symposium, Melbourne, Australia, RMIT University*, pages 32–39. Citeseer.

Burrows, S., Tahaghoghi, S. M., and Zobel, J. (2007). Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, 37(2):151–175.

Burrows, S., Uitdenbogerd, A. L., and Turpin, A. (2014). Comparing techniques for authorship attribution of source code. *Software: Practice and Experience*, 44(1):1–32.

Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., and Greenstadt, R. (2015). De-anonymizing programmers via code stylometry. In *24th USENIX security symposium (USENIX Security 15)*, pages 255–270.

Frantzeskou, G., Stamatatos, E., Gritzalis, S., and Katsikas, S. (2006). Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*, pages 893–896.

Juneja, B. (2009). *Programming with C++*. New Age International.

Juola, P. (2008). *Authorship attribution*, volume 3. Now Publishers Inc.

Kešelj, V., Peng, F., Cercone, N., and Thomas, C. (2003). N-gram-based author profiles for authorship attribution. In *Proceedings of the conference pacific association for computational linguistics, PACLING*, volume 3, pages 255–264.

Koppel, M., Schler, J., and Argamon, S. (2009). Computational methods in authorship attribution. *Journal of the American Society for information Science and Technology*, 60(1):9–26.

Malin, C. H., Casey, E., and Aquilina, J. M. (2008). *Malware forensics: investigating and analyzing malicious code*. Syngress.

Sinkov, A. (1963). Programming the ibm 1401: A self-instructional programmed manual.

Spafford, E. H. and Weeber, S. A. (1993). Software forensics: Can we track code to its authors? *Computers & Security*, 12(6):585–595.

Stamatatos, E. (2009). A survey of modern authorship attribution methods. *Journal of the American Society for information Science and Technology*, 60(3):538–556.

Uzuner, Ö. and Katz, B. (2005). A comparative study of language models for book and author recognition. In *International conference on natural language processing*, pages 969–980. Springer.

Wilcox, L. J. (1998). Authorship: the coin of the realm, the source of complaints. *Jama*, 280(3):216–217.

Wilkes, M. V., Wheeler, D. J., and Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer: With special reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley Press.