International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 12 Issue: 04 | Apr 2025 www.irjet.net p-ISSN: 2395-0072

A Review of Real-Time Code Plagiarism Detection Using NLP and Machine Learning for Academic and Industry Applications

Nargis Siddiqui[1], Deepshikha[2]

[1]Master of Technology, Computer Science and Engineering, Lucknow Institute of Technology, Lucknow, India
[2]Assistant Professor, Department of Computer Science and Engineering, Lucknow Institute of Technology, Lucknow, India

---

Abstract - *Code plagiarism is a problem increasingly important in academic and industrial settings both, with respect to intellectual property, academic integrity and software quality. Current plagiarism detection methods (e.g. strings matching, syntax tree comparison) are insufficient in detecting highly sophisticated techniques like logic replication and code obfuscation. In recent years, Natural Language Processing (NLP) and Machine Learning (ML) have made significant advances which have provided new way for real time, accurate and scalable code plagiarism detection. In this review paper, I explore ways to tackle the shortcomings of the traditional way of solving NLP by using NLP and ML techniques together. In this work we analyze the use of NLP for semantic analysis and preprocessing of code, and ML models that address whether code structure and/or logic are similar, using supervised, unsupervised, and deep learning based methods. We further elaborate on the architecture of real time plagiarism detection systems, their usage in academia and industry as well as the scalability, the cross language detection, and the intrinsic ethical problems. Finally, the paper also suggests future research directions about how to leverage more advanced transformer based models, ways to make the solutions more explainable and ways to enable cloud based collaborative solutions. This review synthesizes existing research and identifies gaps, aiming to provide a comprehensive knowledge of current status of the real time code plagiarism detection and inspires further innovation in this critical region.*

Key Words: Code Plagiarism Detection , Natural Language Processing (NLP), Machine Learning (ML), Real-Time Systems, Semantic Analysis, Deep Learning.

1.INTRODUCTION

1.1.Background and Motivation

With so much riding on it, code plagiarism detection has become a very serious concern at the industrial front as well as the academic world. With the growing use of online learning platforms and its adoption as an ingredient in the entirety of academia, including programming assignment's utilization, has contributed to the ability of students to claim copying or reuse other's code without giving all the needed credits. This makes educational system a suspect and crude the process of developing real programming skills in the students. In code too, plagiarism is similar to how it is in research, if you do not cite someone else's work, it amounts to an unethical practice; such as replicating another's algorithm or methodology, which might distort scientific progress. Therefore, code plagiarism is a big menace in the industrial sphere. But one of the common challenges addresses is that proprietary code is seldom reused or stolen, making it financially lossy as well as legal trouble. Additionally, reusing code without proper scrutiny may introduce vulnerabilities, resulting in low quality and security of the software.

Figure-1: Plagiarism detection

Since code can vary in syntax, logic and structure even among functionally similar programs, it's quite difficult to detect plagiarism in code. For example, the two might do the same task but with different variable names, logic or algorithms making it hard for traditional plagiarism detection methods to see something is being duplicated. Moreover, advanced methods such as code obfuscation, transforming the code to change its structure in a way to keep track of it is hard. They underscore the reality that any kind of comparison that is not deeper than the surface will not suffice – we need more advanced methods that will involve going beyond surface comparisons and will analyse the semantic and functional aspects of the code.

1.2.Problem Statement

The expansion of the computational capability of software development and the rise in quantity of code that is

created requires development of a real time, accurate and scalable plagiarism detection system. Often these nuances escape the order and accuracy that the traditional methods — string matching and the syntactic tree comparisons — provide. In cases were code has been paraphrased, restructured or obfuscated, they have a very hard time finding out if it is plagiarism or not. Furthermore, these methods are not meant for real time usage which is needed for eg. automated grading systems in academia or continuous integration pipes lines in industry. They are also not applicable in massive open online courses (MOOCs) and enterprise-level code repositories because of the lack of scalable solutions.

1.3.Objectives

The main goal of this review is to systematically examine the features of existing techniques, tools and frameworks for code plagiarism detection. It consists of an evaluation of existing methods and more recent methods that borrow from Natural Language Processing (NLP) and Machine Learning (ML). The code description is supported by a key focus on exploration of uses of NLP to preprocess and analyze code at a semantic level, in order to detect code similarities beyond just syntax. Furthermore, the review will further address the role of ML models such as supervised, unsupervie and deep learning to improve the accuracy and efficiency of the plagiarism detection systems.

A second aim is to find gaps in the existing research and suggest future ways of innovation. This includes solving cross language plagiarism detection challenge of translating or adapting code to a different programming language, as well as making the outputs of ML models explainable to gain transparency and trust. This review synthesizes existing knowledge, and identifies potential areas of future exploration in order to support the development of more robust and flexible plagiarism detection systems in both academic and industrial terms.

2.OVERVIEW OF CODE PLAGIARISM DETECTION

2.1.Types of Code Plagiarism

Plagiarism of code is the use of some other's code without right attribution or authorization and then presenting the same as one's own. The practice of this kind is a form of unethical practice for which there is no end of complication. One of the simplest types of copying is direct copying when the entire section of code is copied verbatim. This form may be easier to detect, but still has a lot of presence, especially among beginners or under time constrains. The second form is paraphrasing, a method where the original code is slightly altered – such as in changing variable names, changing comments, or reordering statements – to mask its source. While these changes might not seem too drastic they can shield an attacker considerably.

Logic replication is another type of more advanced plagiarism, which involves the code reproduction apart from copying the syntax by directly copying the logic or the algorithm of the original code. This is harder to identify using traditional ways and needs a deeper insight of the code functionality. In the last case, code obfuscation is a very complex approach where the code if intentionally transformed to be hard to understand or to trace. Hide the original source by using techniques like renaming the variables to meaningless strings, inserting redundant piece of code, and using complex control structure. They further warrant robust detection methods that can be applied to a variety of forms of plagiarism.

2.2.Traditional Methods for Plagiarism Detection

The traditional ways of detecting code plagiarism have made used of the structural and syntactic analysis. String matching is one of the earliest and most widely used techniques, and the code is treated as plain text, where the sequences of characters are compared to find out similarities. The advantages of this method are that it is simple and effective at detecting copying, but it doesn't consider even minor changes such as variable name and formatting changes. To overcome this limitation, tokenization was introduced where the code is broken down into tokens (keywords, operators, identifiers) with a finer degree of comparison. The detection accuracies improve using this approach as it only considers the logical aspects of the code as compared to its textual representation.

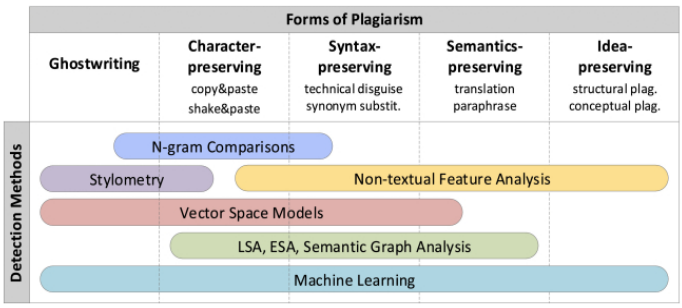

Figure-2: Suitability of detection methods for identifying certain forms of plagiarism.

Another commonly used approach is abstract syntax tree (AST) comparison, which is parsing code into a tree like structure which represents the syntactic hierarchies. This method compares the AST of two code segments and would be much more effective than a string based approach since it compares the structure and organization of code. But AST comparison is powerful, but computationally expensive and it may not be able to compare very much restructured or obfuscated code. Code

plagiarism detection based on these traditional methods has been the foundation for what is nowadays available - they are however very limited when it comes to taking into account more sophisticated forms of plagiarism.

2.3.Limitations of Traditional Approaches

However, traditional methods in code plagiarism detection have limited their contributions. The biggest drawback is that they cannot deal with semantic similarities. These methods are good in word but fell short when they could not identify where the logic or functionality of the code has been replicated but not copied. For example, two programs can have no semantic relationship at all, but implement the same algorithm, they are the same (are) semantically equivalent. But no one has made them work, and traditional approaches are tuned to working with so—called surface—level features, and are ill—equipped to recognize similarities this way.

Another major limitation of them is their vulnerability to code obfuscation. Such techniques as renaming variables, adding redundant code, changing control structures, easily fool conventional detectors, as they are based on the structure of the code which can easily be changed. Moreover, these methods tend to be not scalable or efficient enough for real time applications including automated grading and continuous integration pipelines where the purpose is both speed and correct. However, these shortcomings suggest that more sophisticated techniques, which can deal with code plagiarism's intricacies, are necessary and NLP and ML are expected to play an important role in this field.

3.ROLE OF NLP IN CODE PLAGIARISM DETECTION

3.1.Preprocessing Code for NLP

Preprocessing of raw code is the very first step to apply Natural Language Processing (NLP) to code plagiarism detection. First is the tokenization where the code is broken down into smaller pieces of code such as keywords, operators, identifiers, and literals. It aids in capturing the core constituent of the code so that it is simpler to analyze and compare. Consider, for example a line of code, `int x = 10;` would give us `x = int, x, =, 10, ;`. Tokenization is very helpful for working with variations in formatting or spacings, such as ones that can be found in plagiarized code.

After that, normalization is done to represent code in an standard way. Usually, it is converting tokens to consistent format (e.g. converting all variable names to generic placeholder (like for example VAR1, VAR2) or removing language specific syntax that does not have any impact on the code logic. Normalization reduces the noise and directs the analysis on structural and functional elements of code and does not care about superficial differences. Next, parsing is used to generate a structured representation of the code (an abstract syntax tree (AST), a control flow graph (CFG), etc.). They capture the hierarchical and logical relations among code elements which is then used as a casing for further analysis. However, for NLP techniques to be able to analyze and compare code regardless of whether it has been modified or obfuscated, preprocessing is required.

3.2.Semantic Analysis Using NLP

NLP in code plagiarism detection is one of the most beautiful things and here it performs most of the work and this is the reason for which it can do semantic analysis i.e. it compares not only syntactically but also the logic and functionality. The traditional methods focus on the surface features of the code while NLPs handle the deep thinking of the meaning and intent behind the code. For example, two segments with totally different syntax may perform the same algorithm or solve for the same problem. These semantic similarities can be captured by NLP models, in particular deep learning based models, which can learn such patterns and relationships through large amounts of code in a dataset.

In this regard, techniques, such as word embeddings and transformer-based models (e.g., BERT, CodeBERT) have worked quite well. These allow us to represent code in the form of high dimensional vectors and therefore allow meaningful comparisons of its semantic meaning. For instance, a model provided with a large corpus of code can infer that from a 'for' loop and a 'while' loop that have equivalent logic, their syntax can be said to be functionally equivalent. This capability is very handy for detecting more advanced forms of plagiarism like logic replication or paraphrasing, where in fact the functionality is not changed but the implementation does. NLP takes the semantic analysis approach to overcome the limitations associated with traditional plagiarism detection methodology.

3.3.Challenges in Applying NLP to Code

There are significant challenges to applying NLP to code plagiarism detection. An issue also is how to handle nuances of programming languages. None of the programming languages follow the same syntax, idioms nor the best practices. For example, block structure in Python is radically different from how it is done in C with braces. These language specific features need to be trained on the NLP models and the data can be huge plus it would require lots of computational resources. But continuing to writing for more languages directly translates to multilingual plagiarism detection — where a piece of code needs to be translated (or adapted) to another programming language — which is even more challenging as models are forced to generalize across language and structure differences.

Dealing with code obfuscation (where the code is made to hide its original structure or intent) is another very important issue. NLP models struggle to read and measure code accurately since techniques like variable renaming, inserting redundant code as well as using and complex control structures may make it hard for models to read and understand code. Preprocessing steps such as normalization can reduce some of these issues, but code that can be highly obfuscated may still go undetected. Additionally, there is a lack of standardized datasets and evaluation metrics for code plagiarism detection, which makes the approach of different NLP baselines in this problem space difficult to benchmark and compare. The existence of these challenges emphasizes the importance of further research and invention to make the most of NLP in this area.

4.ROLE OF MACHINE LEARNING IN CODE PLAGIARISM DETECTION

4.1.Supervised Learning Approaches

The use of supervised learning as a tool to detect plagiarized code with the help of already labeled dataset for training classification models has emerged as a powerful tool. Due to this, in this approach, the system is given examples of the plagiarized and non plagiarized code to learn out the pattern and the feature which differentiate between two. The code is fed to the model by extracting features like token sequences and control flow structures, syntactic from the code. For instance, algorithm like decision trees, SVMs and random forests are used for this. For instance, such a model can be trained to look for certain patterns of variable renaming or control flow alteration which are indicative of plagiarism. In particular, supervised learning is effective when large, high quality labeled datasets are available because they allow the model to generalize well to code that is unseen. But one pitfall of using said data is that they rely on the presence of labeled data, and it becomes difficult to formulate expensive datasets.

4.2.Unsupervised Learning Approaches

Supervised methods involve creating labeled data to learn and decide on plagiarism automatically. In contrast with this, these approaches try to understand the patterns and similarity in the code using techniques such as clustering and similarity detection. Structural or semantic similarities of code segments are utilized by clustering algorithms like k-means or hierarchical clustering to group code segments, so that cases of plagiarism can be identified within the clusters. As one specific example, if two code segments have the same control flow graph, such as tokens, then they may be grouped together to point out potential instances of copying or paraphrasing. Another frequently used unsupervised approach is a similarity detection where we use the cosine similarity or Jaccard index to compare the code segments by their features. Since it is particularly useful in cases where there is not enough or even no labeled data available, unsupervised learning is a versatile tool for using in plagiarism detection. However, the effectiveness of this approach is reliant on the quality of feature extraction and the code space is extremely complex, thus more sophisticated methods are needed to learn subtle similarities in the code space.

4.3.Deep Learning Models

By allowing the automatic extraction of complicated useful features and patterns of code, deep learning has revolutionized code plagiarism detection. RNNs and LSTMs are particularly suited to working with tokenized code, and hence process sequential data well. They can also capture long range dependencies and contextual information to find similarities that do not concern immediate surface features. Imagine an example such as an LSTM model that may learn to realize that two code pieces are functionally analogous, despite if they use different variable names or control codes. However, more recent transformer based models, such codes like BERT and CodeBERT have become popular as these models can work with large scale data and also learn semantic relations between code. The models are given the entire context of the code using self-attention mechanisms and are very effective for tasks such as semantic similarity detection. However, deep learning models are capable to learn from complex and obfuscated code but they still need huge computational resources and large quantities of training data to provide acceptable performance.

4.4.Hybrid Models

The hybrid models incorporate the inputs from both NLP and machine learning to boost the accuracy and robustness in detecting code plagiarism. These models can also integrate the semantic understanding yielded through NLP with machine learning's pattern recognition capabilities, but through the integration of both domains. In other words, one of the models could use NLP techniques to preprocess and tokenize the code, and then pass it to a deep learning model to extract high level features and perform similarity analysis. Finally, another method is to combine supervised and unsupervised learning methods, also combining the supervised models to do initial classification and unsupervised techniques for refining the results or identifying edge cases. In practice, hybrid models are an excellent way of overcoming the shortcomings of each approach; for instance, an inherent assumption is that supervised learning requires access to labeled data, while an obliviousness to semantics is a limitation of unsupervised methods. Hybrid models based on the joint use of NLP and machine learning provide a more sophisticated and reliable solution to the identified challenges in detecting code plagiarism both in academic and industrial research.

5.CONCLUSION

We review real time code plagiarism detection using Natural Language Processing (NLP) and Machine Learning (ML) as examples to point out how these technologies can be used to solve an issue that is important to solve, both on the academic and the industrial side. Although basic, traditional methods cannot cope with the intricacies of current code plagiarism like semantic similarity, logic replication, etc., as well as advanced obfuscation. NLP and ML integration provides a paradigm shift, when combined can allow systems to analyze code at higher level, such as seeing not only syntactic patterns but also semantic and functional relationships. Tokenization, semantic analysis, and transformers as deep learning models are some of the techniques that have shown to be very capable in detecting plagiarized code, even in the face of extensive modifications.

NLP and ML hold the possibility of transforming code plagiarism detection in a way code is processed and understood by machines in a manner akin to human reasoning. These technologies leverage large datasets and sophisticated algorithms to pick up on the subtle similarities and patterns that standard approaches frequently miss. Finally, real time systems built using NLP and ML for instance automated grading in academic and continuous code review in industry become probable. In addition to significantly increasing accuracy and efficiency, these systems offer scalable solutions which can accommodate growing volume and complexity of code in the present digital environment.

REFERENCES

1. A. Aiken, "Moss: A system for detecting software plagiarism," Journal of Systems and Software, vol. 83, no. 12, pp. 2504–2515, Dec. 2010, doi: 10.1016/j.jss.2010.07.013.

2. L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," Journal of Universal Computer Science, vol. 8, no. 11, pp. 1016–1038, Nov. 2002, doi: 10.3217/jucs-008-11-1016.

3. S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," Software: Practice and Experience, vol. 37, no. 2, pp. 151–175, Feb. 2007, doi: 10.1002/spe.757.

4. M. S. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," IEEE Transactions on Software Engineering, vol. 43, no. 6, pp. 538–558, Jun. 2017, doi: 10.1109/TSE.2016.2615307.

5. Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," IEEE Transactions on Software Engineering, vol. 32, no. 3, pp. 176–192, Mar. 2006, doi: 10.1109/TSE.2006.28.

6. C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's School of Computing TR, vol. 541, no. 115, pp. 64–68, 2007, doi: 10.15388/ioi.2019.03.

7. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," ACM Computing Surveys, vol. 50, no. 4, pp. 1–36, Aug. 2017, doi: 10.1145/3092566.

8. D. Perez and S. Chiba, "Cross-language clone detection by learning over abstract syntax trees," IEEE Transactions on Software Engineering, vol. 46, no. 12, pp. 1350–1367, Dec. 2020, doi: 10.1109/TSE.2019.2896965.

9. Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 35, no. 8, pp. 1798–1828, Aug. 2013, doi: 10.1109/TPAMI.2013.50.

10. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018, doi: 10.48550/arXiv.1810.04805.

11. Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," arXiv preprint arXiv:2002.08155, 2020, doi: 10.48550/arXiv.2002.08155.

12. A. Vaswani et al., "Attention is all you need," Advances in Neural Information Processing Systems, vol. 30, pp. 5998–6008, 2017, doi: 10.48550/arXiv.1706.03762.

13. S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," ACM SIGPLAN Notices, vol. 49, no. 10, pp. 173–184, Oct. 2014, doi: 10.1145/2714064.2660203.

14. M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," ACM SIGSOFT Software Engineering Notes, vol. 39, no. 1, pp. 1–10, Jan. 2014, doi: 10.1145/2557833.2561957.

15. C. Chen, Z. Liu, and Y. Sun, "A neural framework for retrieval and summarization of source code," IEEE Transactions on Software Engineering, vol. 47, no. 9, pp. 1899–1912, Sep. 2021, doi: 10.1109/TSE.2019.2940987.

16. M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 87–98, 2016, doi: 10.1145/2970276.2970326.

17. H. K. Dam, T. Tran, and J. Grundy, "DeepSoft: A vision for a deep model of software," IEEE Transactions on Software Engineering, vol. 46, no. 12, pp. 1285–1298, Dec. 2020, doi: 10.1109/TSE.2019.2893173.

18. R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning," AAAI Conference on Artificial Intelligence, vol. 31, no. 1, pp. 1345–1351, 2017, doi: 10.1609/aaai.v31i1.10771.

19. U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2Vec: Learning distributed representations of code," Proceedings of the ACM on Programming Languages, vol. 3, no. POPL, pp. 1–29, Jan. 2019, doi: 10.1145/3290353.

20. S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," IEEE Transactions on Software Engineering, vol. 44, no. 3, pp. 209–222, Mar. 2018, doi: 10.1109/TSE.2017.2678538.

21. M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," International Conference on Learning Representations (ICLR), 2018, doi: 10.48550/arXiv.1711.00740.

22. Y. Liu, M. Ott, N. Goyal, and J. Du, "RoBERTa: A robustly optimized BERT pretraining approach," arXiv preprint arXiv:1907.11692, 2019, doi: 10.48550/arXiv.1907.11692.

23. C. Sutton, T. Hoos, and M. Allamanis, "Learning semantic representations for software artifacts," IEEE Transactions on Software Engineering, vol. 45, no. 12, pp. 1206–1223, Dec. 2019, doi: 10.1109/TSE.2018.2867724.

24. P. Yin, G. Neubig, M. Allamanis, and M. Brockschmidt, "Learning to represent edits," International Conference on Learning Representations (ICLR), 2019, doi: 10.48550/arXiv.1810.13328.

25. T. Zhang, G. Upadhyaya, A. Reinhardt, and H. Rajan, "Are code examples on an online Q&A forum reliable? A study of API misuse on Stack Overflow," IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 23–34, 2018, doi: 10.1109/ICSME.2018.00013.