Source Code Plagiarism Detection with Pre-Trained Model Embeddings and Automated Machine Learning

Fahad Ebrahim
University of Warwick, UK
Fahad.Ebrahim@warwick.ac.uk

Mike Joy
University of Warwick, UK
M.S.Joy@warwick.ac.uk

Abstract

Source code plagiarism is a critical ethical issue in computer science education where students use someone else's work as their own. It can be treated as a binary classification problem where the output can be either 'yes' (plagiarism found) or 'no' (plagiarism not found).

In this research, we have taken the open-source dataset 'SOCO', which contains two programming languages (PLs), namely Java and C/C++ (although our method could be applied to any PL). Source codes should be converted to vector representations that capture both the syntax and semantics of the text, known as contextual embeddings. These embeddings would be generated using source code pre-trained models (CodePTMs). The cosine similarity scores of three different CodePTMs were selected as features. The classifier selection and parameter tuning were conducted with the assistance of Automated Machine Learning (AutoML). The selected classifiers were tested, initially on Java, and the proposed approach produced average to high results compared to other published research, and surpassed the baseline (the JPlag plagiarism detection tool). For C/C++, the approach outperformed other research work and produced the highest ranking score.

1 Introduction

Plagiarism is the ethical and educational issue of taking ideas from other sources and representing them as your own without acknowledgement. Plagiarism can be divided into text and source code. Academic source code plagiarism can be defined as "Source-code plagiarism in programming assignments can occur when a student reuses source code authored by someone else and, intentionally or unintentionally, fails to acknowledge it adequately, thus submitting it as his/her own work. This involves obtaining the source-code, either with or without the permission of the original author and reusing the source code produced as part of another assessment (in which academic credit was gained) without adequate acknowledgement" (Cosma and Joy, 2008). The words reuse, obtain, and acknowledge may also be defined on the basis of the academic requirements.

The detection of plagiarism is a lengthy and demanding process, so new technologies such as Artificial Intelligence (AI) might be used effectively. Plagiarism can be treated as a classification problem as the output can be considered a class of discrete values: 'yes' (plagiarised), 'no' (non-plagiarised), or potentially 'partial'. Source code plagiarism can be also considered to be an application of a source code similarity measurement task (Zakeri-Nasrabadi et al., 2023).

There are several ways to represent source codes (Hrkút et al., 2023) such as graphs, trees and tokens. The source codes must be converted into vectors known as embeddings before being fed into a classifier. Contextualized embeddings not only consider syntax but also the semantics of source codes. These embeddings can be created using pre-trained models. The emergence of pre-trained models has revolutionized the field of Natural Language Processing (NLP) and are being known for their robustness. They can be utilised in various ways such as re-training, fine-tuning and inference. Also, some domain-specific models have been created for certain areas and tasks.

This work inspects the robustness of the embeddings generated by source code pre-trained models (CodePTMs) for the task of source code plagiarism detection. Contextual embeddings are extracted using these CodePTMs and a classifier built on top of the embeddings. For classification, this work utilises the concept of Auto Machine Learning (AutoML) to determine the best classifier given certain training data. The training, testing, and evaluation are based on the SOurce COde reuse dataset

301

(SOCO) (Flores et al., 2014).

The paper is organized as follows: section 2 covers related work, section 3 covers the methodology, section 4 presents the results, and section 5 covers the conclusion and future work.

2   Related work

Several software tools have been used as source code plagiarism detectors. Novak compared several such tools, namely JPlag, MOSS, SIM, Splat, Marble, Plaggie, and Sherlock Warwick in his review paper (Novak, 2016).

Engels et al. introduced the idea of neural networks in source code plagiarism detection and reused the output of MOSS on a dataset containing 20,706 C++ introductory course assignments. The evaluation was performed on the basis of the classification evaluation metrics (precision, recall, and F1 scores) (Engels et al., 2007).

Ljubovic and Pajic tackled the issue of external plagiarism by using a cloud and applying an Artificial Neural Network (ANN) based on the output of the SIM's software and repository monitoring on a dataset containing 3,655 submissions from an introductory C course (Ljubovic and Pajic, 2020). Their setup was compared to JPlag, MOSS and SIM.

Abstract Syntax Trees (ASTs) along with code disassembly have been used by Viuginov et al. who considered the lexical, syntactic, layout, and structural characteristics of the source code on a dataset containing 90,000 C++ solutions (Viuginov et al., 2020). For the evaluation, they calculated the F1-score.

Manahi et al. used a combination of Siamese networks, Bidirectional Long Short-Term Memory (BLSTM), and character embeddings on a dataset including 16,800 introductory course C assignments (Manahi et al., 2022). Siamese networks are multiple similar neural networks with the same configurations and weights. They are mainly used for similarity detection. For their evaluation, they calculated the classification evaluation metrics.

Humayoun et al. used the concepts of tokenization, AST, and upsampling on their public dataset of 60 introductory C++ programming assignments (Humayoun et al., 2022). The features tested were N-gram overlap, Longest Common Substring (LCS), and greedy string tilling. Eight classifiers were implemented using Weka and the authors' model was evaluated based on the classification evaluation metrics.

In the first part of his master's thesis (Heres, 2017), Heres proposed a system using N-grams, term frequency–inverse document frequency (TF-IDF), and cosine similarity. The dataset used was the SOCO Java set and their own private dataset having 16,954 files. The evaluation was based on the average precision score.

Deep learning using char-Recurrent Neural Network (char-RNN) and Long Short-Term Memory (LSTM) was the basis of Katta's approach, which used general deep features that could be applied to any dataset (Katta, 2018). The dataset covered an introductory C course with a total of 4,700 submissions. The model was evaluated using classification evaluation metrics.

2.1   SOCO related works

This work uses the SOCO dataset and the approach followed will be compared to the other approaches based on the same dataset.

Garcia et al. used an approach (UAEM) consisting of four phases (García-Hernández and Lendeneva, 2014). The first phase was related to pre-processing, which was the tokenization of the source code, and the second phase was the similarity measurement, which was based on the longest common substrings. The third phase was related to extracting different parameters such as distance, ranking, gap, and relative difference, and the final phase was decision-making by using the obtained parameters.

Ramırez et al. in their approach (UAM-C) applied three different views to the source code: a lexical view utilising 3-grams, a structural view that utilizes the methods headers, and a stylistic view covering features such as the number of spaces and the number of uppercase or lowercase letters (Ramırez-de-la Cruz et al., 2014).

Ganguly and John developed an Information Retrieval (IR) model (DCU)(Ganguly and Jones, 2014). They built a Language Model (LM) based only on the Java dataset. A Java parser was utilized as a bag of words scheme, and an AST was constructed to capture the structure of the code.

Ganguly et al. proposed an improvement over DCU (Ganguly et al., 2018), where a supervised classifier (random forest) was added to an IR model based only on the Java dataset. The approach had three models: an IR Language Model (LM), LM with AST (LM_AST), and LM_AST with different

index fields (FLM_AST).

Flores et al. used a text comparison approach (Flores et al., 2014). They searched for matching lines between two source codes and calculated the ratio between the number of these lines and the larger number of lines of the two files. The decision was based on a threshold value. Furthermore, Apoorv worked in a similar manner, but the decision was based on the maximum similarity value (ratio of matching lines).

We refer to JPlag (Prechelt et al., 2002) as "Baseline1". JPlag is a well-known tool used for source code plagiarism detection. Source codes are converted into tokens, and the similarity between these tokens is estimated using the 'greedy string tilling' algorithm (Wise, 1993).

The work of Flores et al. is referred to as "Baseline2" (Flores et al., 2011), which divides the activity into three stages: pre-processing, feature extraction, and similarity measurement. Spaces, line breaks, and tabs are eliminated. The features are based on 3-grams and term frequencies, and cosine similarity is used to measure the similarity of two source codes.

The recent work of Setoodeh et al. has developed a four-phased approach (Setoodeh et al., 2021). The first phase is pre-processing, such as removing comments and unnecessary code, and the second phase involves generating a sequence to capture the structure of the source codes. The third phase is similarity measurement by applying multiple methods such as comparing the sequence strings, trees, and edges. The final phase is related to the evaluation including the calculation of the precision, recall, and F1 score, and a comparison with other SOCO-related works.

The approach taken in this paper differs from existing approaches in three respects. Firstly, the dataset used is open source, accessible, and adequate in size containing two PLs (Java and C/C++). Secondly, the approach is language-independent and does not depend on tokenizers or specific language syntax, so could be applied to any PL. Thirdly, the approach utilises the state-of-art CodePTM contextualized embeddings.

3 Methodology

This work follows the typical process of applying supervised ML to a classification problem, as mentioned in (Schlegel and Sattler, 2023). The cycle starts with data collection and feature engineering, followed by model selection. The model needed to be trained and tested. Enhancements with parameter tuning could be applied, prior to the model being evaluated.

3.1 Dataset

There is a lack of a proper dataset related to source code plagiarism due to potential legal or social issues, and it was therefore difficult to have an open-source academic dataset of students' data which could be used for this research. Possible solutions included using a privately created dataset (as suggested by several related works) or applying a code reuse dataset such as SOCO.

The SOCO dataset contains training and testing data written in C++ and Java. The training set in C++ included 79 files with 26 reuse cases, while there were 259 files with 84 reuse cases written in Java. For the testing set, in C++, there were 19,895 files with 322 reuse cases, while in Java, there were 12,080 files with 222 reuse cases. There were six different scenarios per language, labelled A1, A2, B1, B2, C1, and C2.

There were a few assumptions in this dataset. First, the reuse occurred within the same programming language, therefore multi-programming reuse was not covered. Second, reuse occurred in the same scenario without overlapping the testing set. One challenge in the SOCO dataset was that the training set was smaller than the testing set. Another challenge was that the testing data were severely imbalanced, while the training set was less imbalanced.

3.2 Data Pre-processing and Encoding

The source code was written in different files and had to be arranged into a suitable data structure. Then, the code needed to be converted into a clean format that could be fed into a classifier that accepts only numbers. Embeddings are vector representations of source code that can be created with pre-trained models. The embeddings are created using Sentence Transformers (Reimers and Gurevych, 2019) with mean pooling.

3.2.1 Selection of Source Code Pre-Trained Models

Multiple surveys have been written for CodePTMs as in (Niu et al., 2022, 2023; Zeng et al., 2022; Xu and Zhu, 2022). CodePTMs are models trained on a large corpus of code.

The suitable models in this work would be selected based on the following criteria.

- The model should be accessible and could be found in Huggingface[1] for ease of use. The models available in Huggingface are Code-Bert (Feng et al., 2020), GraphCodeBert (Guo et al., 2020), UnixCoder (Guo et al., 2022), CodeT5 (Wang et al., 2021), CodeGPT (Lu et al., 2021), PLBART (Ahmad et al., 2021), and CodeBERTa (Wolf et al., 2019).

- The model should pass a simple test. It will be given a pair of totally different source codes. For instance, the first Java program prints 'Hello World' and the other contains a function that calculates the average of two numbers. Then, the similarity of the generated embeddings will be calculated. If the score is above 0.8, it would not be used. Otherwise, the model passes this test. As these two programs are totally different, the similarity score should be low. If it is high, then the source code is not represented adequately. For example, the model GraphCodeBert generates a similarity score of 0.94 if given this pair of code fragments. Further experiments on this test results can be seen in Table 1.

Pre-Trained Model	Similarity Score
PLBART	0.0257
Unixcoder	0.2988
CodeBERTa	0.7868
CodeGPT	0.8899
GraphCodeBert	0.9442
CodeBert	0.9918

Table 1: Cosine Similarity Scores

The three pre-trained models that yielded acceptable similarity scores were UnixCoder, PLBART and CodeBERTa. Each of these captures different aspects of source codes, as UniXcoder considers AST and code comments in addition to the source code, while PLBART captures the style and data flow. The similarity being referred to is the cosine similarity which will be explained in the following subsection.

3.3 Similarity Measure and Feature Selection

Cosine similarity is a common measurement of similarity used in NLP. It represents the angle between two vectors, and the angle (θ) is equal to the dot product of the two vectors (A and B) over the product of their norms, as shown in Equation 1. The higher the similarity score, the more similar the vectors are.

$$Cosine\ Similarity = cos(\theta) = \frac{A.B}{||A||.||B||} \quad (1)$$

The three features of the model would be the cosine similarity scores between the three generated embeddings per source code. Using each model embeddings separately achieved acceptable results. However, the combination of the three of them would add more data to train at the expense of complexity. The classifier would figure out which combination of these three features would be better in terms of evaluation.

3.4 Classification Model Selection

The concept of AutoML utilises several algorithms and selects the best-performing models for certain training data automatically. The main reason behind using AutoML in this work is to reduce time consumption. The testing dataset is large and testing different classifiers and comparing them would take a substantial amount of time. Thus, AutoML would search for the most appropriate classifier in less time. The library selected for this work is AutoSklearn (Feurer et al., 2015, 2022) which chooses the leading algorithm given specific training data and certain time intervals. AutoSkLearn handles both the model selection and parameter tuning. There are other AutoML libraries surveyed by Elshawi et al. (Elshawi et al., 2019). AutoSklearn is selected for its familiar syntax to Sklearn[2] and simplicity.

The inputs of the classifiers were the three cosine similarity values between three different CodePTM embeddings. The parameters configured for AutoSklearn were the duration of 30 minutes with 10-fold cross-validation. The best classification model selected for Java with the configured parameters and Java training set was extra trees, while for C/C++, the best classification model selected was gradient boosting. These two classifiers are relevant in this task as both of them are ensemble

techniques based on decision trees. They perform well in case of imbalanced data as in the SOCO dataset. Also, both methods are known for their high performance in Kaggle[3] competitions. Ensemble methods are known for potential lower loss and less over-fitting.

For testing, the similarity scores are fed to the selected algorithms to create prediction probabilities that are compared to a dynamic threshold determining whether the files are plagiarised or not. Once pairs of plagiarism files are available, the models are evaluated per the next subsection.

3.5 Evaluation

Classification could be evaluated with several metrics (Joshi, 2020). The fundamental metrics were true positive (TP), false positive (FP), true negative (TN), and false negative (FN). TP is to predict the positive value correctly, and FP is to mispredict a positive value. FN is to mispredict a negative value, while TN correctly predicts a negative value. Applied to plagiarism, positive could indicate that plagiarism was found, and negative could indicate that plagiarism was not found. Some other metrics that use TP, FP, and FN in their calculation are as follows: precision, recall, and F1 score, as presented in equations 2, 3, 3, respectively (Joshi, 2020).

$$precision = \frac{TP}{TP + FP} \qquad (2)$$

$$recall = \frac{TP}{TP + FN} \qquad (3)$$

$$f1score = 2 * \frac{precision * recall}{precision + recall} \qquad (4)$$

Furthermore, for the SOCO dataset evaluation (Flores et al., 2014), these three metrics, namely recall, precision, and F1 score, were utilized as a part of the model's standard evaluation, while for the ranking of the model, only the F1 score was used.

The detailed technical methodology illustrated in this section is represented in Figure 1. Following are the simplified summarized steps.

1. Extracting contextualised embeddings using three different CodePTMs using sentence transformers with mean pooling.

2. Calculating the cosine similarity scores between pair of embeddings of the source codes. These scores form the input to the automated machine-learning process.

3. Selecting the leading classifier using AutoML during the training phase.

4. Generating the prediction probabilities with the selected classifier.

5. Decision-making based on whether the probabilities are larger than a dynamic threshold.

6. Evaluating the model and calculating the classification accuracy metrics including the F1 score.

4 Results

4.1 Results and Discussion

The results of the Java scenarios are presented in Table 2. For the scenario of C2 of identical plagiarism files, the metrics values were 1. Scenarios B1 and B2 produced high metric values, but for scenarios A1 and A2, the scores are lower as the file sizes and number of files are high.

Parameter	F1	Precision	Recall
C2	1	1	1
B1	0.724	0.977	0.957
B2	0.772	0.957	0.647
A1	0.643	0.9	0.5
A2	0.623	0.8	0.511

Table 2: Java metrics

The results of the C/C++ scenarios are presented in Table 3. For C1, the scores were high. While for other scenarios, the metrics were similar, falling in a similar range.

Parameter	F1	Precision	Recall
C1	0.8	0.857	0.75
B1	0.458	0.4	0.535
B2	0.473	0.44	0.512
A1	0.521	0.491	0.556
A2	0.47	0.389	0.593

Table 3: C/C++ evaluation metrics

The overall results of the Java files are represented in Table 4. For the proposed work, the F1 score was 0.69, the precision was 0.908, and the
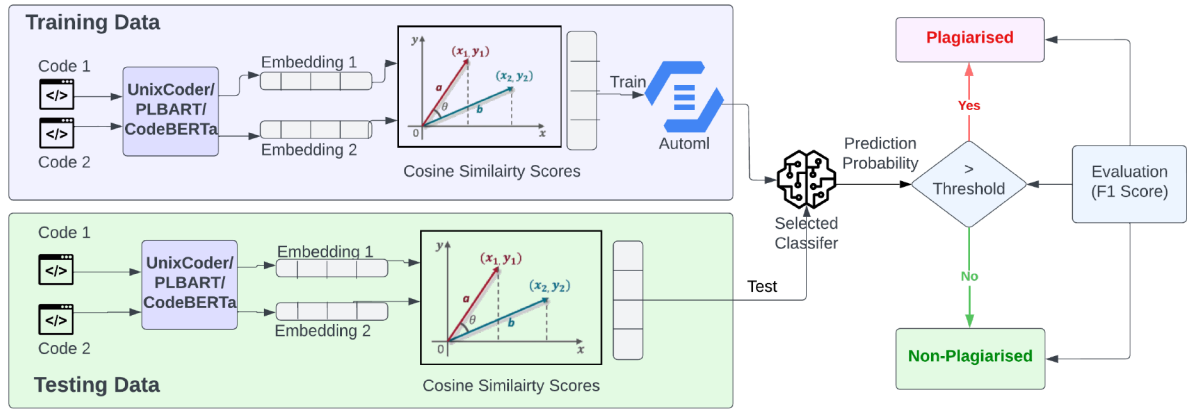
---

[3]https://www.kaggle.com/

Figure 1: Detailed Methodology

recall was 0.559. For the F1 score, the minimum score was 0.031, the average was 0.54, and the maximum was 0.855. For the precision score, the minimum score was 0.016, the average score was 0.46, and the maximum score was 0.951. For the recall, the minimum score was 0.293, the average was 0.882, and the maximum score was 1. In our approach, the precision and F1 scores were between the average and the maximum values. The recall value was lower than the average value. The approach exceeded both baselines and ranked approximately third after Shiraz and UAM-C alongside DCU, LM_AST and FLM_AST in terms of F1 score and ranked second after Shiraz in terms of precision. The high value of precision indicates having fewer false positives, which means non-plagiarized cases are not detected as plagiarized. As the task of plagiarism is sensitive, then higher precision is more suitable. The lower value of recall means that some actual plagiarism cases were not detected. The main reason is due to having severely imbalanced data which can be fixed in future work. Therefore, the results related to the Java dataset were average to high.

The overall results of the C/C++ files can be seen in Table 5. The F1 score of our work was around 0.493, the precision was 0.443, and the recall was 0.561. For the F1 score, the minimum score was 0.01, the average was 0.2, and the maximum was 0.38. For the precision score, the minimum score was 0.005, the average score was 0.192, and the maximum score was 0.35. For the recall, the minimum score was 0.13, the average was 0.59, and the maximum score was 1. The approach taken in this work yielded the highest F1 and precision scores and outperformed both baselines. Recall was around the average values. Therefore, the re-

	Run	F1	P	R
Our work	1	0.69	0.908	0.559
Shiraz	1	0.751	**0.951**	0.621
	2	**0.855**	0.884	0.828
	3	0.836	0.831	0.842
UAEM	1	0.556	0.385	**1**
	2	0.273	0.158	1
	3	0.273	0.158	1
UAM-C	1	0.517	0.349	1
	2	0.037	0.019	0.928
	3	0.807	0.691	0.968
DCU	1	0.602	0.432	0.995
	2	0.692	0.53	0.995
	3	0.68	0.515	1
Baseline 1	1	0.38	0.542	0.293
Baseline 2	1	0.556	0.457	0.712
APoorv	1	0.031	0.016	0.855
LM	1	0.602	0.432	0.995
LM_AST	1	0.692	0.53	0.995
FLM_AST	1	0.68	0.515	1
Rajat	1	0.447	0.32	0.732

Table 4: Comparison with SOCO Java related works

sults on the C/C++ dataset were competitive.

The F1 score in C/C++ is lower than in Java (0.493 compared to 0.69) but compared to other works it is high. This is due to Java having more training data and a higher κ value than C/C++ which implies that the Java training set is more representative (Flores et al., 2014). The main limitation of this approach is the maximum input length to the pre-trained models, which is 512. If the input is larger than 512, it would be truncated. So, for larger files, the end of the files may not be captured. Therefore, if plagiarism occurs at the end of the

	Run	F1	P	R
Our work	1	**0.493**	**0.443**	0.561
Shiraz	1	0.332	0.33	0.335
	2	0.278	0.251	0.313
	3	0.332	0.344	0.322
UAEM	1	0.38	0.306	0.5
	2	0.38	0.306	0.5
	3	0.342	0.26	0.5
UAM-C	1	0.013	0.006	**1**
	2	0.01	0.005	0.95
	3	0.013	0.006	0.977
Baseline 1	1	0.19	0.35	0.13
Baseline 2	1	0.295	0.258	0.345
Apoorv	1	0.014	0.007	0.903
Rajat	1	0.126	0.068	0.927

Table 5: Comparison with SOCO C/C++ related works

code files, it would not be captured.

The usage of contextual embeddings generated by CodePTMs is efficient in the task of source code plagiarism detection producing highly competitive results in the SOCO dataset.

5 Conclusion

Plagiarism in programming assignments is a critical issue in the field of computer science education. It can be treated as a machine learning binary classification problem. So, this research introduced a simple yet effective approach to the task of source code plagiarism detection. It started by selecting the open-source SOCO dataset with two PLs (Java and C/C++). Source code files were converted to embeddings to be part of any machine learning classifier. Three different CodePTMs (PLBART, UnixCoder, and CodeBERTa) were used to generate their own embeddings. Cosine similarity scores between these three models were calculated and considered to be the selected features. The classification models were selected using the concept of AutoML and the library AutoSklearn. The initial testing was conducted on Java, and the proposed model produced high metrics as compared to other approaches and exceeded both baselines. For C/C++, the model produced the highest F1 and precision scores as compared to other approaches and outperformed both baselines.

Exploring other CodePTMs that are not available on HuggingFace for source code plagiarism detection is an idea for future work, along with increasing the training time for AutoSklearn. The dataset is severely imbalanced, hence, different techniques could be used to tackle such issues. Also, chunking can be used to overcome the limited input size of the pre-trained models.

Acknowledgments

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Georgina Cosma and Mike Joy. 2008. Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, 51(2):195–200.

Aarón Ramírez-de-la Cruz, Gabriela Ramırez-de-la Rosa, Christian Sánchez-Sánchez, Wulfrano Arturo Luna-Ramırez, Héctor Jiménez-Salazar, and Carlos Rodrıguez-Lucatero. 2014. UAM@ SOCO 2014: Detection of source code reuse by means of combining different types of representations. *FIRE [4]*.

Radwa Elshawi, Mohamed Maher, and Sherif Sakr. 2019. Automated machine learning: State-of-the-art and open challenges. *arXiv preprint arXiv:1906.02287*.

Steve Engels, Vivek Lakshmanan, and Michelle Craig. 2007. Plagiarism detection using feature-based neural networks. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 34–38.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2022. Auto-sklearn 2.0: Hands-free automl via meta-learning. *The Journal of Machine Learning Research*, 23(1):11936–11996.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28.

Enrique Flores, Alberto Barrón-Cedeno, Paolo Rosso, and Lidia Moreno. 2011. Towards the detection of

cross-language source code reuse. In *International Conference on Application of Natural Language to Information Systems*, pages 250–253. Springer.

Enrique Flores, Paolo Rosso, Lidia Moreno, and Esaú Villatoro-Tello. 2014. On the detection of source code re-use. In *Proceedings of the Forum for Information Retrieval Evaluation*, pages 21–30.

Debasis Ganguly and Gareth JF Jones. 2014. DCU@FIRE-2014: An information retrieval approach for source code plagiarism detection. In *Proceedings of the Forum for Information Retrieval Evaluation*, pages 39–42.

Debasis Ganguly, Gareth JF Jones, Aarón Ramírez-De-La-Cruz, Gabriela Ramírez-De-La-Rosa, and Esaú Villatoro-Tello. 2018. Retrieving and classifying instances of source code plagiarism. *Information Retrieval Journal*, 21(1):1–23.

René Garcıa-Hernández and Yulia Lendeneva. 2014. Identification of similar source codes based on longest common substrings. *FIRE [4]*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcode-bert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Daniël Heres. 2017. Source code plagiarism detection using machine learning. Master's thesis, Utrecht University.

Patrik Hrkút, Michal Ďuračík, Štefan Toth, and Matej Meško. 2023. Current trends in the search for similarities in source codes with an application in the field of plagiarism and clone detection. In *2023 33rd Conference of Open Innovations Association (FRUCT)*, pages 77–84. IEEE.

Muhammad Humayoun, Muhammad Adnan Hashmi, and Ali Hanzala Khan. 2022. Measuring plagiarism in introductory programming course assignments. In *2022 8th International Conference on Information Technology Trends (ITT)*, pages 80–87. IEEE.

Ameet V Joshi. 2020. *Machine learning and artificial intelligence*. Springer.

Jitendra Yasaswi Bharadwaj Katta. 2018. *Machine learning for source-code plagiarism detection*. Ph.D. thesis, International Institute of Information Technology Hyderabad, University of Science and Technology.

Vedran Ljubovic and Enil Pajic. 2020. Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories. *IEEE Access*, 8:96505–96514.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Mohammed Manahi, Suriani Sulaiman, and Normi Sham Awang Abu Bakar. 2022. Source code plagiarism detection using Siamese BLSTM network and embedding models. In *Proceedings of the 8th International Conference on Computational Science and Technology*, pages 397–409. Springer.

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: A survey on pre-trained models of source code. *arXiv preprint arXiv:2205.11739*.

Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. *arXiv preprint arXiv:2302.04026*.

Matija Novak. 2016. Review of source-code plagiarism detection in academia. In *2016 39th International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 796–801. IEEE.

Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. 2002. Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.*, 8(11):1016.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Marius Schlegel and Kai-Uwe Sattler. 2023. Management of machine learning lifecycle artifacts: A survey. *ACM SIGMOD Record*, 51(4):18–35.

Zahra Setoodeh, Mohammad Reza Moosavi, Mostafa Fakhrahmad, and Mohammad Bidoki. 2021. A proposed model for source code reuse detection in computer programs. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, 45(3):1001–1014.

Nickolay Viuginov, Petr Grachev, and Andrey Filchenkov. 2020. A machine learning based plagiarism detection in source code. In *2020 3rd International Conference on Algorithms, Computing and Artificial Intelligence*, pages 1–6.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Michael J Wise. 1993. String similarity via greedy string tiling and running Karp-Rabin matching. *Online Preprint, Dec*, 119(1):1–17.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Yichen Xu and Yanqiao Zhu. 2022. A survey on pre-trained language models for neural code intelligence. *arXiv preprint arXiv:2212.10079*.

Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software*, page 111796.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 39–51.