From Syntax Trees to Embeddings: A Comparative Study of AI-Generated Code Detection

Marino Jakimoski*, Martina Toshevska*, Sonja Gievska*
* Faculty of Computer Science and Engineering, Skopje, North Macedonia
marino.jakimoski@students.finki.ukim.mk

***Abstract*—Large language models (LLMs) are increasingly used to generate program code, which complicates code authorship attribution in settings where originality is required (e.g., academic assessment). This paper studies binary classification of source code as either *AI*-generated or *human*-written using the CoDeT-M4 dataset (500,552 samples across Java, Python, and C++). We compare three families of approaches: (i) traditional machine learning on engineered features extracted from concrete syntax trees, (ii) embedding-based deep models built on pretrained CodeBERT representations, and (iii) graph-based models operating on concrete syntax tree-derived graphs. Our results show that embedding-based architectures achieve the highest performance, while graph-based methods remain competitive and highlight the value of structurally grounded representations, including simple comment-indicator nodes, for authorship detection.**

***Keywords*—authorship detection, AI-generated code, CodeBERT, graph neural networks, syntax trees, embeddings**

I. Introduction

With the advent of LLMs (large language models), writing tasks across domains have become much easier and more accessible. This accessibility has also found its way into tasks related to program code [1]. Coding-related tasks are divided into many categories, most notably: modification, error correction, code completion tasks, and direct generation of code from natural-language prompts [1]. The robustness of large language models makes programming more accessible, much easier, and faster, even for people who have little or no experience in the field [2].

The capacity of LLMs to generate code from natural-language prompts that range from short sentences, to lists of functional requirements is of interest to individuals from many different backgrounds: seasoned software engineers, scientists looking to visualize their data, and even students who have just started their academic or professional journeys in the fields of Computer Science.

This creates a problem in spaces where code must be either fully written by the human, or where plagiarism is strictly forbidden. Naturally, one such example of a space like that is the academic setting, in which students must be evaluated on their skills. The ease of accessibility of LLMs makes this a pressing issue. Students can easily copy and paste the description of their exercises or tasks as a prompt into these LLMs and get a working solution in one, or only a few iterations with the LLM, thereby avoiding the solitary solving of the task [2]. In doing so, students undermine their academic integrity, and create an "inflation" of the final grades they receive, thereby creating a reduction of the value of having an academic degree [3].

In this work, we perform a systematic comparison of code authorship detection methods on CoDeT-M4, targeting the binary labels *human* and *AI*. We evaluate traditional machine learning baselines (Random Forest, Logistic Regression, Naive Bayes, and CatBoost), embedding-based deep models operating on pretrained CodeBERT representations, and graph neural networks built from concrete syntax trees. Beyond reporting performance, we analyze how architectural and representation choices (e.g., pooling strategies, comment-indicator nodes, and positional encodings) affect results, with the broader goal of clarifying the trade-off between peak predictive performance and structurally grounded modeling.

II. Methodology

This section presents the compared methods and the dataset used for training and evaluation.

A. Experimental configuration

To ensure fair comparison and reproducibility, all experiments use a fixed random seed of **872002** and the same train/validation/test split from Section II-B. Implementation code and experiment scripts are available at https://github.com/marin-o/recognizing-ai-generated-code.

B. Data and data setup

We use CoDeT-M4 [4], which contains 500,552 samples: 246,221 *human* and 254,331 *AI*. AI samples are tagged by source model (5 models) and cover Java, Python, and C++. The data is split into *train, validation, and test*. Figure 1 shows split ratios, Figure 2 language distribution per split, and Figure 3 code-length distribution.

The dataset also includes per-sample features (e.g., *avgFunctionLength, avgIdentifierLength, avgLineLength*) that were not used in this work.

During data exploration, we identified minor leakage: 1234 samples appeared in more than one subset. Before experiments, we removed overlaps by giving priority to

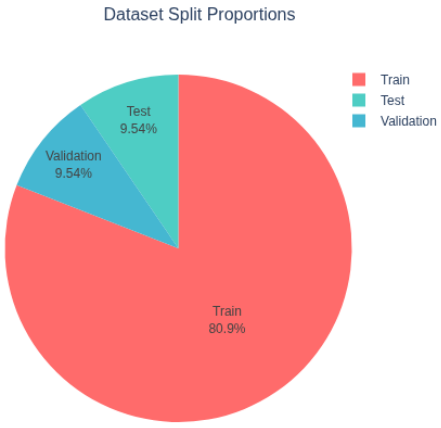

Fig. 1: Size of train, validation and test subsets in CoDeT-M4.

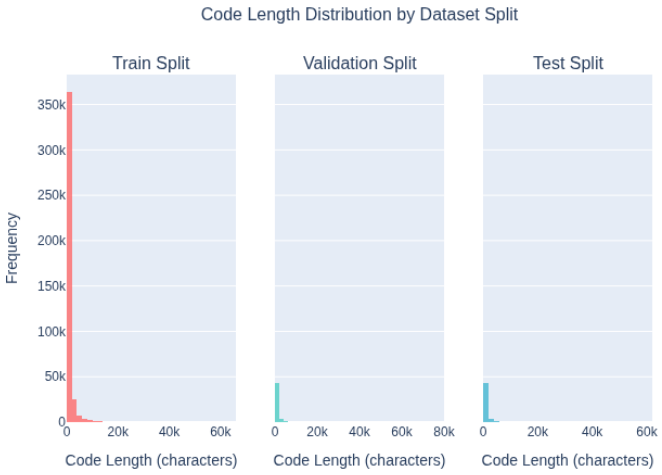

Fig. 3: Code length (by number of characters) in each subset.

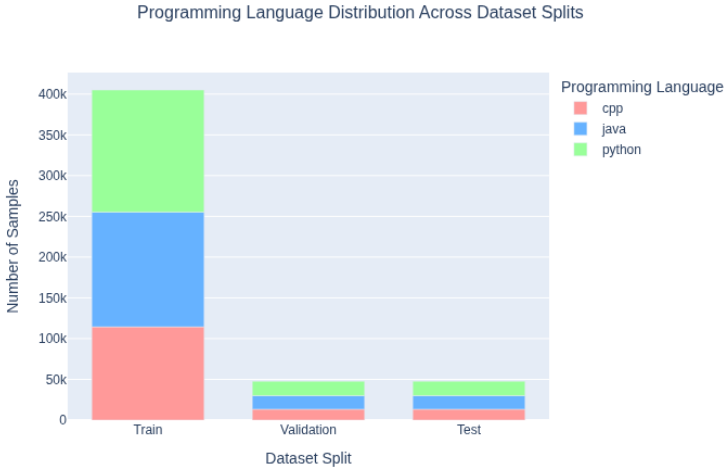

Fig. 2: Distribution of programming languages in each subset of CoDeT-M4.

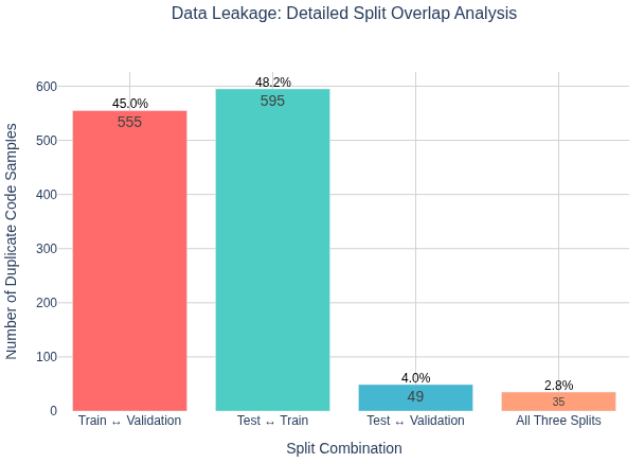

Fig. 4: Number of samples that occur in the corresponding combination of subsets. The percentage above each graph indicates the proportion of the total number of duplicate instances that appear in the respective combination of subsets.

train, then *test*: samples shared with *train* were removed from the other subset, and samples shared between *validation* and *test* were removed from *validation*. Figure 4 summarizes the affected subset combinations.

C. Feature engineering

This section describes preprocessing and feature construction used by the methods in Section II-D.

1) Creating syntax trees: We build *concrete* syntax trees (CSTs, i.e., parse trees) with *Tree-Sitter* [5] by loading the grammar of each target language and querying the resulting trees for features or full-structure processing.

We use CSTs instead of ASTs because this task targets authorship cues, where low-level syntactic decisions (including formatting- and delimiter-related structure and comment placement) may carry signal that AST abstractions often discard. To reduce noise, our graph construction does not encode token text and relies on node types and parent-child structure; comment information is included only as indicator nodes in the corresponding dataset variants.

2) Features for traditional ML methods: The traditional ML methods in Section II-D1 use feature vectors extracted from CSTs (Section II-C1) according to Table I. The resulting features are normalized with *StandardScaler* from *Scikit-Learn* [6].

3) Creating Graphs: After creating a CST (Section II-C1), we convert it to a PyTorch-Geometric (PyG) graph [7], [8] for the models in Section II-D2. Each

Feature	Description
Functions	Number of function or method declarations
Classes	Number of class and interface declarations throughout the code
If Statements	Number of `if` statements
Loops	Number of loop constructs (`while`, `for`, `do-while`)
Imports	Number of import declarations
Comments	Number of comments (single-line or multi-line)
Binary Operations	Number of binary expressions (arithmetic, logical, or comparison operators)
Errors	Indicates whether a syntax error was detected during tree parsing

TABLE I: Features extracted from each syntax tree and their descriptions.

tree is traversed to extract node types, which are mapped to integer IDs used as node attributes; edges carry no attributes and encode directed parent-child structure.

Types of graph datasets. We created multiple graph-based CoDeT-M4 variants, summarized in Table II.

Dataset Type	Description
Base	Base graph dataset as in Section II-C3, no preprocessing.
Comments	Comment nodes retained; textual content excluded.
Cleaned	Duplicates removed from validation and test subsets.
Comments + Cleaned	Comment nodes retained and duplicates removed.
Cleaned + Comments + Depth & Positional Emb.	Comment nodes retained, duplicates removed, nodes augmented with tree depth and child index.

TABLE II: Graph-based dataset variants used in this study.

D. Methods

1) Traditional ML: This subsection presents the traditional ML methods evaluated in this work, implemented with Scikit-Learn and trained on features from Section II-C2. Table III contains the models, and only those hyperparameters that we manually set differently from the default values.

2) Deep Learning: This subsection presents embedding-based and CST-graph-based deep learning methods. All models are built with PyTorch [9] and PyTorch-Geometric [7], [8], and trained with Adam [10].

a) Token-embedding based methods: These methods learn and classify token embeddings of each code sample.

CodeBERT Embeddings Classification This method directly classifies embeddings from a pretrained, frozen CodeBERT [11] backbone. CodeBERT outputs 768-dimensional vectors, followed by a classifier head with one or two fully connected layers.

Model	Hyperparameters	Selected
Random Forest	`n_estimators`: [10, 20, 30, 50]	30
	`max_depth`: [10, 20, 30, 40]	20
	`min_samples_split`: [2, 3, 5]	5
Logistic Regression	`C`: [0.01, 0.1, 1, 10]	0.01
	`penalty`: [l1, l2]	l2
	`solver`: [liblinear, saga]	liblinear
Naive Bayes (Gaussian)	`var_smoothing`: [$1e^{-9}$, $1e^{-8}$, $1e^{-7}$, $1e^{-6}$]	$1e^{-9}$
CatBoost	`iterations`: [100, 200, 300]	200
	`learning_rate`: [0.03, 0.1, 0.2]	0.1
	`depth`: [4, 6, 8]	4
	`l2_leaf_reg`: [1, 3, 5]	3

TABLE III: Traditional ML models and their hyperparameters.

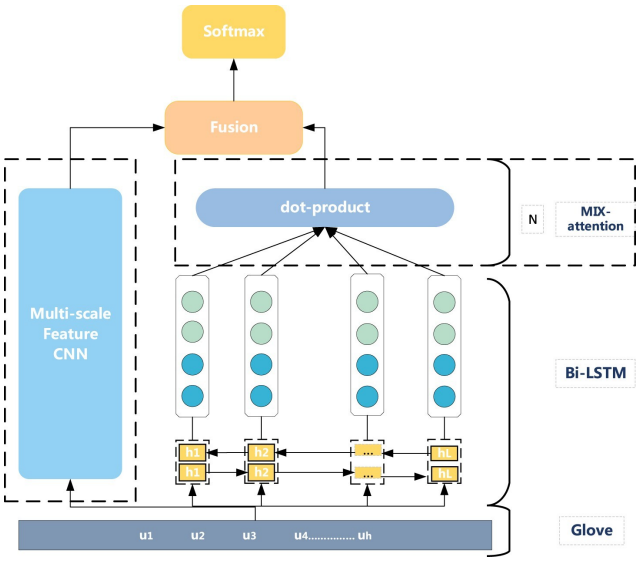

Fig. 5: Multi-Scale CNN+Bi-LSTM network. *Source: [12].*

Multi-Scale CNN + Bi-LSTM The model proposed by [12], combines a convolutional neural network with multiple scalings (MCNN), and a Bi-LSTM layer, for the purposes of text classification using *GloVe* [13] textual embeddings. The MCNN layers serve to capture local or short-range dependencies or characteristics in the textual embeddings, while the Bi-LSTM captures long-range characteristics. The MCNN module contains kernels with different sizes in order to extract differently-sized features from the embeddings. Figure 5 shows the visualization of the MCNN+BiLSTM model.

In this work, we adapt the network to code by replacing *GloVe* with embeddings from pretrained CodeBERT, which are more representative of programming languages. The

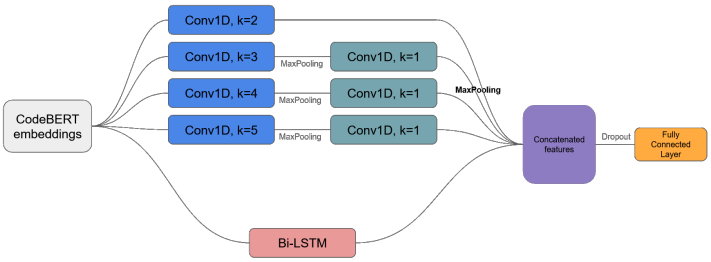

Fig. 6: Multi-Scale CNN + Bi-LSTM network adapted for this work.

adapted MCNN uses kernel sizes 2–5 to capture local n-gram-like features, while the Bi-LSTM captures longer-range dependencies. The modified architecture is shown in Figure 6.

For ablation, we also evaluate classification with only the MCNN module to measure the Bi-LSTM contribution.

b) Graph-based Methods: We evaluate several graph neural network architectures on the tree-structured representations described in Section II-C1.

We consider a graph-convolutional baseline in which the convolution operator is tuned between GCN [14] and GraphSAGE [15], as well as Graph Attention Networks (GAT) [16] and GraphTransformer layers [17]. The selected graph-convolutional setting uses GraphSAGE. Each model employs a learnable embedding layer initialized according to the number of node types. After the final graph layer, node representations are aggregated using global mean pooling and passed to a classifier head. The overall architecture is illustrated in Figure 7.

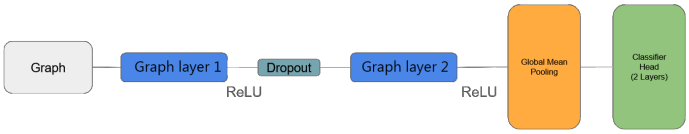

Fig. 7: Graph neural networks architecture.

Hyperparameters, including hidden dimensions, embedding size, learning rate, and convolution type (GCN vs. GraphSAGE), were optimized; the final configuration is reported in Table IV.

In addition to global mean pooling, we also evaluate attention pooling [18] and global max pooling.

For the Graph Transformer, we use 8 attention heads per layer. Due to the absence of edge attributes in our graphs, edge features are disabled. We evaluate two variants: (i) without positional embeddings, and (ii) with structural positional embeddings encoding node depth and child order.

III. Results and Discussion

This section reports quantitative results across three methodological categories: traditional, embedding-based, and graph-based approaches. All experiments were conducted with the fixed random seed for reproducibility.

Hyperparam.	**Search Range**	**Selected**	**Description**
Hidden Dim 1	[128, 512]	512	First graph conv. layer dimension
Hidden Dim 2	[128, 512]	400	Second graph conv. layer dimension
Embedding Dim	[64, 512]	176	Node embedding vector dimension
Use SAGE	True, False	True	GCN [14] vs. GraphSAGE [15] layers
Learning Rate	$[10^{-4}, 10^{-2}]$	1.2697×10^{-3}	Adam optimizer learning rate
Classifier Hidden Dim 1	N/A	200	First classifier hidden layer dim
Classifier Hidden Dim 2	N/A	100	Second classifier hidden layer dim

TABLE IV: Optimized hyperparameters for the graph convolutional model.

Model	**Acc.**	**Rec.**	**F1**	**Prec.**	**Spec.**	**AUC**
RF	**0.7374**	**0.6930**	**0.7210**	0.7513	**0.7800**	**0.8242**
LR	0.6329	0.6422	0.6314	0.6209	0.6240	0.6819
NB	0.6083	0.5151	0.5628	0.6203	0.6976	0.6561
CB	0.7363	0.6597	0.7101	**0.7688**	0.8097	0.8223

TABLE V: Performance of traditional ML models on the *test* split of CoDeT-M4. RF: Random Forest, LR: Logistic Regression, NB: Naive Bayes, CB: CatBoost. **Bold** = best, underlined = second-best.

We report test-set accuracy, precision, recall, specificity, F1-score, and AUROC. AI-generated code is treated as the positive class, making specificity particularly important because mislabeling human-written code can have serious consequences in academic and related settings.

A. Traditional ML

As shown in Table V, Random Forest and CatBoost achieve the strongest results, with nearly identical accuracy and AUROC, likely due to better modeling of non-linear feature interactions. Logistic Regression and Naive Bayes lag behind, suggesting that linear boundaries and conditional-independence assumptions are insufficient for this task.

Compared with the CoDeT-M4 paper's in-domain binary results [4], our best traditional setup (Random Forest: F1 0.7210, Acc 0.7374) is close to their SVM baseline (F1 0.7219, Acc 0.7219), but remains clearly below their reported CatBoost (F1 0.8878, Acc 0.8879).

B. Embedding-based methods

Table VI shows that embedding-based models substantially outperform traditional approaches. Performance improves with model complexity, from a linear head to a

Model	Acc.	Rec.	F1	Prec.	Spec.	AUC
Base (Linear)	0.8452	0.8393	0.8404	0.8414	0.8507	0.9193
Base (2-layer)	0.9436	0.9289	0.9416	0.9547	0.9577	0.9847
M-CNN	0.9734	0.9735	0.9734	0.9733	0.9735	0.9735
M-CNN+ Bi-LSTM	**0.9836**	**0.9837**	**0.9836**	**0.9835**	**0.9837**	**0.9837**

TABLE VI: Performance of embedding-based methods on the *test* split. Base models use CodeBERT embeddings. M-CNN: Multi-Scale CNN
Bold = best, underlined = second-best.

Model	Data / Config	Acc.	Rec.	F1	Prec.	Spec.	AUC
GraphSAGE	Clean. / Mean	0.8841	0.8620	0.8801	0.8971	0.9052	0.9509
GraphSAGE	C+Cmt / Mean	**0.9453**	**0.9280**	**0.9432**	**0.9589**	**0.9619**	**0.9849**
GraphSAGE	C+Cmt / Max	0.9300	0.9096	0.9272	0.9454	0.9497	0.9797
GraphSAGE	C+Cmt / Attn	0.9377	0.9138	0.9349	0.9571	0.9607	0.9821
GAT	C+Cmt / Mean	0.8794	0.8379	0.8718	0.9086	0.9192	0.9501
GT	Clean. / No PE	0.9201	0.9034	0.9165	0.9301	0.9359	0.9744
GT	C+Cmt / No PE	0.9348	0.9122	0.9319	0.9525	0.9564	0.9807
GT	C+Cmt + PE	0.9351	0.9156	0.9325	0.9499	0.9537	0.9812

TABLE VII: Performance of graph models on the *test* split. GT = Graph Transformer, C+Cmt = Cleaned+Comments, Mean/Max/Attn = Pooling type, PE = Positional Embeddings
Bold = best, underlined = second-best.

two-layer classifier, then to Multi-Scale CNN, and finally Multi-Scale CNN + Bi-LSTM, which reaches ≈0.98 across metrics. This trend indicates that CodeBERT embeddings benefit from non-linear transformations, informative local convolutional patterns (token n-grams), and a smaller additional gain from Bi-LSTM sequence modeling.

Relative to the CoDeT-M4 paper [4], our best embedding model (F1 0.9836, Acc 0.9836) is slightly below their UniXcoder (F1 0.9865, Acc 0.9865), and nearly matches their CodeT5 (F1/Acc 0.9835), while clearly exceeding their CodeBERT result (F1 0.9573, Acc 0.9577).

C. Graph-Based Methods

As shown in Table VII, graph-based models substantially outperform traditional ML but do not surpass the strongest embedding-based architectures. The best configuration is the graph-convolutional baseline with the GraphSAGE setting, comment indicator nodes, and mean pooling (accuracy 0.9453, F1 0.9432), showing that comment presence and placement, even without text, carries stylistic signal relevant to authorship. Among pooling strategies, mean pooling performs best, followed by attention pooling, while max pooling is weakest.

Against CoDeT-M4 paper binary benchmarks [4], this best graph model remains below their transformer-based top results (CodeBERT F1 0.9573; UniXcoder F1 0.9865), but it narrows much of the gap while using structurally grounded syntax representations.

GAT does not improve over the graph-convolutional variants, and the Graph Transformer gains only marginally from positional embeddings. These findings suggest that graph topology already encodes sufficient structural information, with diminishing returns from more complex attention or positional mechanisms.

D. Qualitative error analysis of false negatives

To complement aggregate metrics, we inspected false negatives (AI code predicted as human) for the strongest embedding-based run (*cbm_codet_full*) on the held-out test split. Out of 47,046 samples, 270 were false negatives (0.57%).

Most errors are concentrated in low-information snippets: 122/270 (45.19%) have at most 10 lines and 67/270 (24.81%) at most 5 lines, often short utility wrappers or one-expression routines where AI and human style overlap. A Java-specific boilerplate effect appears as well: 13/69 (18.84%) Java false negatives are setter/getter/delegation-style templates (8 of these are also $\leq$10 lines). Still, 38/270 (14.07%) false negatives are $\geq$50 lines, indicating residual confusion on longer but highly conventional implementations. These patterns suggest robustness should focus on terse and templated code.

E. Overall Assessment

Embedding-based methods achieve the best overall performance, with Multi-Scale CNN + Bi-LSTM near 0.98 across metrics, indicating that pretrained contextual embeddings capture strong authorship cues.

Graph-based methods remain competitive (best graph-convolutional baseline accuracy 0.9453) and show that syntax-structure representations are highly informative; adding comment indicator nodes consistently helps, suggesting comment placement contributes stylistic signal even without comment text. Traditional ML lags both deep-learning families, highlighting limits of handcrafted features.

Compared with CoDeT-M4 [4], our best in-domain binary score is close to strong reported baselines, but we did not reproduce their broader protocols (authorship attribution, OOD splits, ternary human/LLM/hybrid), so comparisons should be interpreted only for shared binary in-domain evaluation. While embeddings maximize accuracy, CST-derived graph models remain structurally grounded and more directly auditable at the syntax level.

F. Threats to Validity

Threats to validity arise mainly from data representativeness and evaluation scope. CoDeT-M4 is sourced from GitHub and competitive-programming contexts, where human code is often more polished than typical student submissions. Because our target use case is academic assessment, performance may differ on classroom assignments and mixed-skill student populations.

A second limitation is robustness coverage. We evaluate cleaned CoDeT-M4 under a standard supervised train/validation/test protocol, but we do not include a dedicated benchmark of semantics-preserving obfuscations or post-edit transformations. Therefore, our conclusions support in-domain binary detection on this dataset, not worst-case robustness.

IV. Conclusion

The increasing use of large language models for code generation has created demand for reliable methods to distinguish AI-generated from human-written source code. In this work, we compared traditional machine learning, embedding-based models, and graph neural networks operating on concrete syntax tree–derived graphs for code authorship detection.

Embedding-based approaches achieved the strongest overall performance, showing that contextual token representations capture highly discriminative signals for this task. Graph-based methods achieved competitive results, demonstrating that structural representations derived from syntax trees provide substantial discriminative power, while traditional feature-based models were considerably weaker. Future work should expand robustness benchmarking across multiple models, datasets, and semantics-preserving edit strategies.

References

[1] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 2, Jan. 2026. [Online]. Available: https://doi.org/10.1145/3747588

[2] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming is hard - or at least it used to be: Educational opportunities and challenges of ai code generation," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, Mar. 2023, p. 500–506. [Online]. Available: https://dl.acm.org/doi/10.1145/3545945.3569759

[3] N. V. Hanh and N. T. Duyen, "Ai-assisted academic cheating: a conceptual model based on postgraduate student voices," *Frontiers in Computer Science*, vol. Volume 7 - 2025, 2025. [Online]. Available: https://www.frontiersin.org/journals/computer-science/articles/10.3389/fcomp.2025.1682190

[4] D. Orel, D. Azizov, and P. Nakov, "CoDet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings," in *Findings of the Association for Computational Linguistics: ACL 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 10 570–10 593. [Online]. Available: https://aclanthology.org/2025.findings-acl.550/

[5] M. Brunsfeld, A. Qureshi, A. Hlynskyi, ObserverOfTime, W. Lillis, J. Vera, dundargoc, P. Turnbull, T. Clem, D. Creager, A. Helwer, R. Rix, D. Kavolis, H. van Antwerpen, C. Clason, M. Davis, R. Bruins, A. Delpeuch, Ika, A. Ya, T.-A. Nguyễn, bfredl, M. Massicotte, S. Brunk, N. Hasabnis, J. McCoy, M. Dong, S. Moelius, and S. Kalt, "tree-sitter/tree-sitter: v0.25.9," Sep. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17069743

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[7] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[8] M. Fey, J. Sunil, A. Nitta, R. Puri, M. Shah, B. Stojanovič, R. Bendias, A. Barghi, V. Kocijan, Z. Zhang, X. He, J. E. Lenssen, and J. Leskovec, "PyG 2.0: Scalable learning on real world graphs," in *Temporal Graph Learning Workshop @ KDD*, 2025.

[9] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. [Online]. Available: https://docs.pytorch.org/assets/pytorch2-2.pdf

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," no. arXiv:1412.6980, jan 2017, arXiv:1412.6980 [cs]. [Online]. Available: http://arxiv.org/abs/1412.6980

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/

[12] H. Huan, Z. Guo, T. Cai, and Z. He, "A text classification method based on a convolutional and bidirectional long short-term memory model," *Connection Science*, vol. 34, no. 1, p. 2108–2124, Dec. 2022.

[13] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162/

[14] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[15] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[16] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[17] Y. Shi, Z. Huang, S. Feng, H. Zhong, W. Wang, and Y. Sun, "Masked label prediction: Unified message passing model for semi-supervised classification," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2021, pp. 1548–1554, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2021/214

[18] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 3835–3845. [Online]. Available: https://proceedings.mlr.press/v97/li19d.html