Differentiating user-written from AI generated codes

Summer Bachelor Semester Project S2 (Academic Year 2024/25)
University of Luxembourg

Erikas Kadiša
erikas.kadisa.001@student.uni.lu
University of Luxembourg
Belval, Esch-sur-Alzette, Luxembourg

prof. Franck Leprevost*
franck.leprevost@uni.lu
University of Luxembourg
Belval, Esch-sur-Alzette, Luxembourg

ABSTRACT

This paper reports on the progress and outcomes of a Summer Bachelor Semester Project (BSP) for the academic year 1, which is a scale-up and continuation of the previous BSP aimed at developing a Convolutional Neural Network (CNN) model capable of classifying source code as either human- or AI-generated based on its origin. We describe the methodology used to construct a dataset of 67,760 Java files, combining human-written code from external datasets and AI-generated code obtained via API, with half of the total files being from each category. All files were transformed into image representations and used to train CNN models for classification. The proposed approach achieved very high accuracy ($96.0 - 97.7\%$) and F1 scores ($0.96 - 0.98$), suggesting that this method offers a competitive alternative to existing approaches.

1 INTRODUCTION

Large Language Models (LLMs) and other AI technologies have made code generation easier and more accessible than ever. However, this advancement comes at a cost: it has become increasingly difficult to accurately assess programmer's true skills, especially in academic and recruitment settings. Reliable verification tools, capable of confirming that a given program was authored with the help of AI, remain scarce. This gap motivated the Bachelor Semester Project (BSP) in Semester 2, which aimed to develop an effective solution for assessing the integrity of the source code.

The aforementioned previous BSP focused on building a Convolutional Neural Network (CNN)-based model to distinguish between human-written and AI-generated code. That work achieved a promising classification accuracy of $90.5\% - 95\%$, demonstrating competitiveness with established detectors such as GPTZero [19] and GPTSniffer [11]. However, its dataset was limited to 7,248 Java files (with 6,093 used for training), primarily sourced from academic exercises designed to teach core Java principles. This narrow scope limited the model's applicability to real-world programming scenarios.

To overcome these limitations, the current project expanded the dataset in both size and diversity. We used the CodeNet database [15] to collect human-written Java code from various real-world contexts and employed multiple LLM APIs to generate corresponding AI-written solutions for the same problems. For the experiments described in the current paper, we used GPT-4.1 [13], Claude Sonnet 4 [2], DeepSeek V3 0324 [5], and Gemini Flash 2.5 [4] as code generation sources. In the end, this process yielded 33,880 files for each of the two categories, resulting in a balanced dataset of 67,760 Java files. Section 4 describes the data collection process and dataset preparation techniques in detail.

We evaluated three CNN architectures: DenseNet-121 [9], ResNet-50 [8], and VGG-16 [18], given their established performance in image-based classification tasks and previous BSP. Models were trained for 40 epochs (averaging $\simeq$ 3h 20min), with EarlyStopping [16] implemented to prevent overfitting, allowing us to obtain optimal, high-accuracy models within $\simeq$ 1h 18min. The used hyperparameter settings, together with detailed explanations, are provided in Section 4.1.

Model performance was assessed using standard metrics (accuracy, precision, recall, F1), classification statistics, and inference speed. Model achieved accuracies of $96 - 97.7\%$ and a F1 score of $0.96 - 0.98$, delivering predictions in just under 0.5–3 seconds, depending on

*BSP supervisor

architecture, for a single code classification and in under $0.17 - 0.4$ seconds if the model is loaded beforehand. Full results are presented in Section 6.

Most training and computations were performed on the IRIS HPC Cluster [20] at the University of Luxembourg, using nodes equipped with NVIDIA Tesla V100 GPUs.

2 METHODOLOGY

The approach used in the experiments remains the same as in the previous BSP and is based on [12]. To describe it simply, code files are transformed into image representations for classification. Below is the rationale for this approach.

At the most fundamental level, any file can be represented as a sequence of binary values — the machine-level encoding of its content. By reading a source code file as raw bytes, we can obtain a consistent binary representation composed of 0s and 1s, regardless of the original programming language.

Differences in the code's style and structure manifest directly in these binary encodings. AI-generated code often exhibits distinct characteristics compared to human-written code, such as:

- Clean and uniform formatting with perfect indentation.
- More sophisticated or optimized problem-solving approaches.
- Detailed and frequent inline comments.

In contrast, human-written code may contain inconsistencies in formatting, simpler or less optimized logic, and fewer comments. These stylistic and semantic differences — ranging from variable naming conventions to whitespace usage — result in unique binary patterns, which can be visually represented as images. CNN models are able to capture and learn these subtle yet consistent features.

While adversarial attempts could be made to disguise AI-generated code by artificially introducing imperfections (e.g., extra whitespace or unnecessary comments), appropriate preprocessing techniques can mitigate it. This ensures that classification relies on deeper structural and semantic patterns rather than superficial formatting cues.

3 CODE TO IMAGE TRANSFORMATION

Code to Image transformation logic remains the same as well: source code files are read as binary, grouped to 24-bit vectors with padding applied if there are missing bits to fill, to obtain RGB values for a pixel. Pixel by pixel we construct the image.

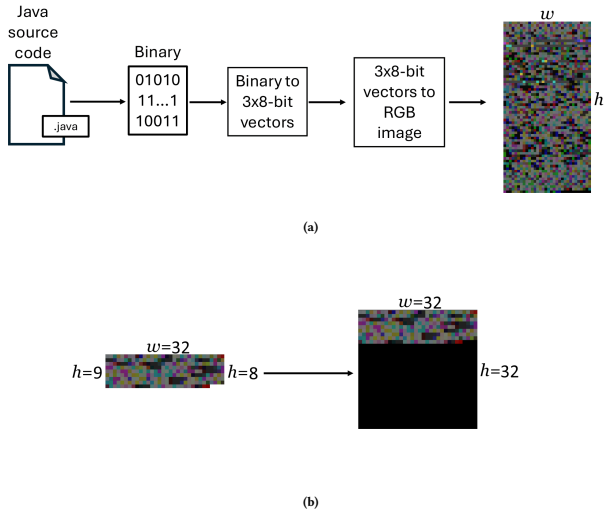

Figure 1: (a) Conversion of source code into an RGB image representation, and (b) Black pixel padding applied to ensure a minimum height of $h \geq w$.

Following the transformation process, we obtain an image with dimensions $w \times h$ (width $\times$ height), where w is determined by the original size of the source code file, as shown in Table 1 (adapted from Table 1 in [10]). If the final row of an image is shorter than w, black pixels (RGB values of 0) are appended to complete it. Likewise, if the resulting height h is less than w, additional rows of black pixels are added to create a square image of size $w \times w$ (see Figure 1b). This ensures a square aspect ratio, which helps reduce distortion when resizing to the CNN input dimensions. However, this step can be skipped if h is greater than or equal to the CNN's required input height, thereby avoiding unnecessary information loss during rescaling.

4 DATASET

We derived a dataset composed of 67,760 source codes, written in Java. Half of the codes originate from humans,

and half from AI. The codes addressed 1,694 specific programming exercises from CodeNet. The raw and preprocessed codes are sorted according to their size ranges, as indicated in Table 1. Before the training phase, the dataset is split into 80% for the training set, 10% for validation, and 10% for testing. Training and validation sets were actively used during training, while the test set is intended to evaluate the model performance after training.

	Image Dimensions		Raw Codes		Preprocessed Codes	
Source code file size range (in kB)	**Width w**	**Height Range**	**Human**	**AI**	**Human**	**AI**
]0, 0.192]	32	]0, 2]	0	0	6	0
]0.192, 10[	32	]2, 105]	32, 430	33, 546	32, 553	33, 872
[10, 30[	64	[53, 157]	1, 179	307	1, 075	8
[30, 60[	128	[79, 157]	153	27	135	0
[60, 100[	256	[79, 131]	68	0	94	0
[100, 200[	384	[87, 174]	48	0	17	0
[200, 500[	512	[131, 326]	2	0	0	0
[500, 1000[	768	[218, 435]	0	0	0	0
[1000, +∞[	1024	[326, +∞[	0	0	0	0
Total considered for training			33, 880	33, 880	33, 774	33, 880

Table 1: Distribution of code types by file size and image dimensions and Human-written and AI-generated Java codes answering 1,694 exercises. The raw and preprocessed codes are sorted according to their file size range. Note that the first row codes result in images with $h < 3$.

4.1 Sources

The human-written dataset was sourced from the CodeNet database on HuggingFace [1], released in 2021 — prior to the deployment of LLMs. The CodeNet dataset contains millions of code submissions in various programming languages, spanning 4,053 distinct programming exercises. For our study, we focused exclusively on the Java subset, which comprises 696,249 files. We selected only those exercises with a sufficient number of correct solutions of meaningful size — specifically, at least 20 submissions with an "Accepted" status and a file size greater than 192 bytes. This filtering yielded 1,694 coding exercises with 348,362 valid submissions. From each exercise, we randomly sampled 20 solutions, resulting in a final set of 33,880 human-written Java files.

AI-generated code was obtained from GPT-4.1, Claude Sonnet 4, Gemini 2.5 Flash, and DeepSeek V3 0324 through API. We gave our four major LLMs, each of which is widely known for their popularity and strength in logical exercises, the task of creating Java code addressing the same 1,694 exercises as those selected for the human dataset. Models generated precisely 8,470 solutions each (5 different solutions per exercise), leading to 33,880 AI-generated codes.

To obtain these AI-generated codes, we defined the values of the hyperparameters and a prompt structure for the requests.

The two main hyperparameters considered are the temperature and the maximum number of tokens per response. Temperature controls the randomness and creativity of the model's output: a value of 0 yields deterministic, most probable responses, whereas values above 1 typically produce more diverse but potentially less accurate outputs [17]. Since most LLM providers — including the four used in this study — do not disclose the exact temperature settings of their web-based interfaces, we adopted a value of 0.7 to balance determinism and diversity. This choice was guided by general recommendations [7], prior studies [3, 6], and the guidelines in [14, Page 24]. The maximum number of tokens constrains the size of the generated outputs. To ensure sufficient capacity for complete and detailed solutions, we set this limit to 8,000 tokens.

The prompt template used for each model request consists of two components. The system-role section contains high-level instructions, specifying the desired behavior and response style of the model. The user-role section provides the concrete task, embedding the extracted exercise from the dataset as `HTML` content. Each exercise is submitted to the LLMs five times, with a slightly modified prompt during each iteration. The variation is introduced by prepending a randomly selected phrase from a predefined set, simulating the variability of human requests. The overall prompt structure is illustrated in Figure 2. After receiving the model's response, the Java code is extracted and saved as a file, resulting in a dataset of 33,880 AI-generated files.

4.2 AI code generation system

One of the biggest challenges during this BSP was automating the AI-code generation. Since our approach was unique and we needed to generate AI code from the `html` exercise sheets, we needed to create a system to send dynamic requests to the chosen LLM models,

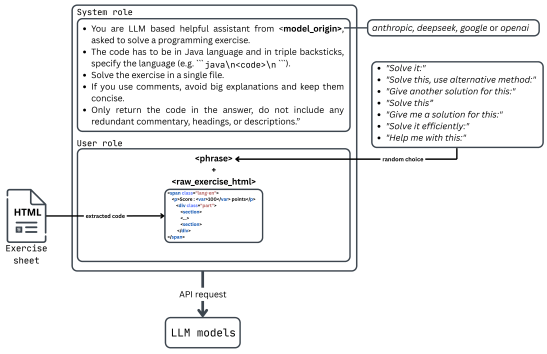

Figure 2: Prompt engineering scheme used to send requests to LLMs through API.

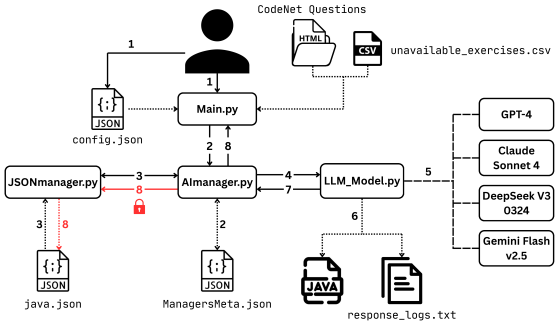

Figure 3: AI code generation using API system scheme.

obtain the response, process it to determine if the response contains the required code, and save it as a file, logging the process.

Figure 3 shows our created python-based system. It contains several modules to reduce the overall program complexity. First of all, it has a module called `Main.py` that acts as a simplified interface for users to generate the solutions. This module takes `config.json` as a configuration file, which can be edited by the user to upload their API key. Then, the code also expects to find a directory containing the exercise sheets (`html` files) and a `csv` file containing a column “Exercises" with the problem ids (filenames without extension) considered unavailable for generation. The `AImanager.py` module is intended to act as the coordinator or the main “brain” of the whole process; it ensures synchronization and logging of the solutions or failures depending on the outcome of generation. `JSONmanager.py` is controlling the metadata of each exercise sheet and model — tracking how many solutions are generated per sheet and model. It also contains information on whether the exercise sheet is available or not. Finally, there is the `LLM_Model.py` module, inherited by subclasses specific to each LLM and saved inside `AImanager.py`. This module takes care of API calls to LLMs, retrieving the responses and saving the files after code extraction.

Let us now define the procedures for using the system and how the system works with the help of arrows.

(1) The user can select between Linear and Parallel generation and choose a range of ids in `Main.py` to obtain the solutions for partitioning the exercises. Additionally, programming language is also selected for the generation.
(2) The code either creates a new instance of AImanager.py or loads it from the `ManagersMeta.json` file if such a file exists in the directory. After the creation of a new instance, such a file is created by default to track the metadata information about AImanager. If the parallel generation function was called, `Main.py` calls the `AImanager.py` generation function multiple times — once per CPU worker.
(3) Manager checks `JSONmanager.py`, specific to the selected language, and makes sure that the problem id is available and still needs solutions (looked up in `<program_lang>.json` file, containing metadata specific to exercises).
(4) A call is sent to `LLM_Model.py` with one of the exercise sheets raw `html` content as an argument.
(5) Request is sent using API to one of the LLMs, preselected by `AImanager.py`.
(6) If LLM responds with correct and expected java code, it is saved in the output directory as a file. Otherwise, the failure is registered in the `response_logs.txt` file.
(7) The caller module is informed about the outcome and either the API call is repeated, going back to the arrow 4 or `JSONmanager.py` is called to register the outcome (notably, after some retries, the failed generation is also registered).
(8) Before calling the `JSONmanager.py`, this function call is locked by a global lock to prevent

other parallel workers from editing the `json` file at the same time. Then either failure is logged or a successful generation is registered. Finally, `AImanager.py` checks if there are still any missing solutions (i.e. if user requested more) and either repeats from arrow 3 the process again, or returns indicating the outcome of generation to the `Main.py` module.

Parallelization is a very important concept to implement. We needed to generate 33, 880 Java files, and since one API call could take several to tens of seconds, linear generation would have taken around 78h 10min to obtain all solutions ($\simeq$ 2h 22min for 1, 000 codes). With parallelization, this time was reduced by $\simeq$ 2.8 times, leading to $\simeq$ 28h 48min ($\simeq$ 51min per 1, 000 codes), which is still a significant amount of time; however, but the system could complete the job over a weekend.

4.3 Preprocessing

As mentioned before, both datasets naturally contained artifacts that could reveal their origin. Notably, in the CodeNet dataset, nearly all Java files included a trailing blank line at the end of the code, a feature rarely present in AI-generated code. Furthermore, human-written code often contained additional whitespace for visual separation and readability, and avoided comments. In contrast, LLM-generated code tended to be more compact and contained detailed comments. Moreover, it was observed that confused LLMs tended to embed their reasoning within comments, despite explicit instructions to keep the comments concise.

To mitigate these distinguishing features, we applied preprocessing techniques to the source files from both categories by removing all comments and normalizing whitespace in the code. The latter technique removed trailing whitespace from each line, trimmed leading and trailing whitespace, and collapsed multiple consecutive blank lines into a single one. Consequently, some of the resulting code files produced images with height $h < 3$, as indicated by the first row of Table 1, and they were excluded from the training. This ultimately led to preprocessed AI and human datasets, comprising 33, 880 and 33, 874 files respectively (see Table 1).

Most codes and the corresponding images are moderate in size. However, there are some files with large file sizes, resulting in a high amplitude. Size comparison between AI and Human datasets is visualized in Figure 4.

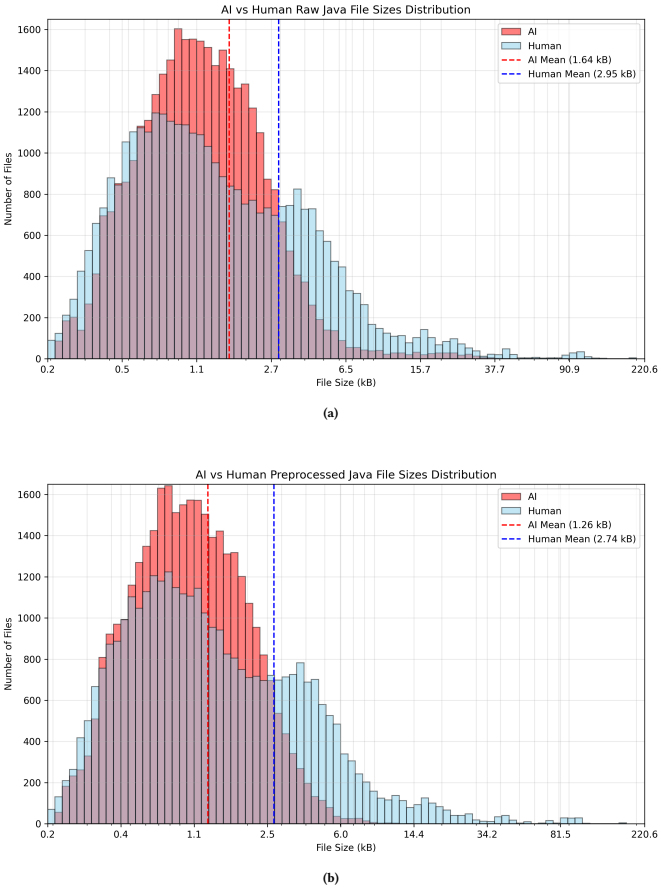

Figure 4: File size comparison histogram with overlapping AI and human classes. Raw (a) and Preprocessed (b) datasets. Note that X axis is not linear.

The size of the code files (raw or preprocessed, AI or human-written) is $\in [0.12, 200.58]$ kB, and the size of the corresponding RGB images is $\in [0.26, 59.92]$ kB. Most of the enormously large files (particularly from the Human dataset) contained custom libraries that help professional coders in competitive programming tasks. The average size of raw human-written code (2.95 kB) is about 1.8 times larger than the average size of AI-generated code (1.64 kB). It indicates that AI tends to solve the same coding exercises in fewer lines of code. In addition, we can notice that after preprocessing, the average human-written code size is approximately 2.2 times larger than that of AI. It can be explained by the fact that AI codes, on average, contained more

comments than human codes, resulting in a higher file size reduction after the removal.

5 TRAINING THE CNN MODELS

The broad workflow of the CNN model training process is visualized in **Figure 5**. The code first gets transformed into the image representation (arrow 1), and before feeding it into a CNN model for training, it is reshaped into $a \times b$ dimensions (arrow 2) to fit the dimensions expected by the model (in this paper, all the used models had input dimensions of 224×224). Then, the image is given for the model to train on (arrow 3). Finally, the model outputs a prediction (arrow 4), and depending on the phase of training (training or evaluation), backpropagation is applied with the calculated loss. The model initially outputs a single number in the range from 0 to 1, which corresponds to the probability of the image being AI-generated.

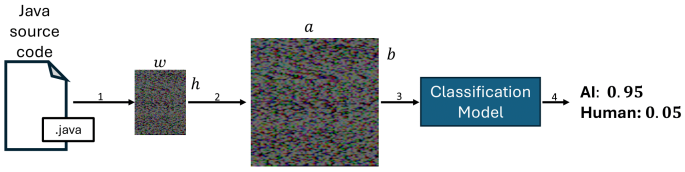

Figure 5: scheme - from code to training and evaluation.

For this experiment, we considered the same CNN models as during the last BSP: VGG-16, ResNet-101, and DenseNet-169. These CNN families are well known for their image recognition and detection capabilities. Models were adapted by changing the last classification layer to have just 1 output feature (instead of hundreds, as is the default) to correlate models specifically with the distinction between 2 classes, based on the image representation of a code.

We fine-tuned our models with the following metrics:

(1) The Adam optimizer is used with a learning rate set to 0.0001, the batch size is set to 32, and the models are trained for 40 epochs.
(2) At the final stage, the binary cross-entropy loss is used. The last layer of the CNNs in a binary classification task ends with a sigmoid activation function; therefore, the CNN outputs a single value $\in [0, 1]$ to determine the prediction and likelihood of the code being generated by AI.
(3) Additionally, EarlyStopping techniques are implemented to prevent overfitting, with patience set to 5 epochs and delta set to 0.0001.

Training performed over 40 epochs required between 2h 57min and 3h 48min, depending on the model and dataset considered. Consequently, the average time for training per epoch is between 4min 25s and 5min 41s. However, in practice, one would only need to train until the optimal epoch. The training time to reach the optimal epoch ranges between 50min and 1h 59min, significantly cutting the initial training margin.

Figure 6 illustrates the overall training progress on the preprocessed dataset for the three evaluated models. The raw dataset exhibits similar trends.

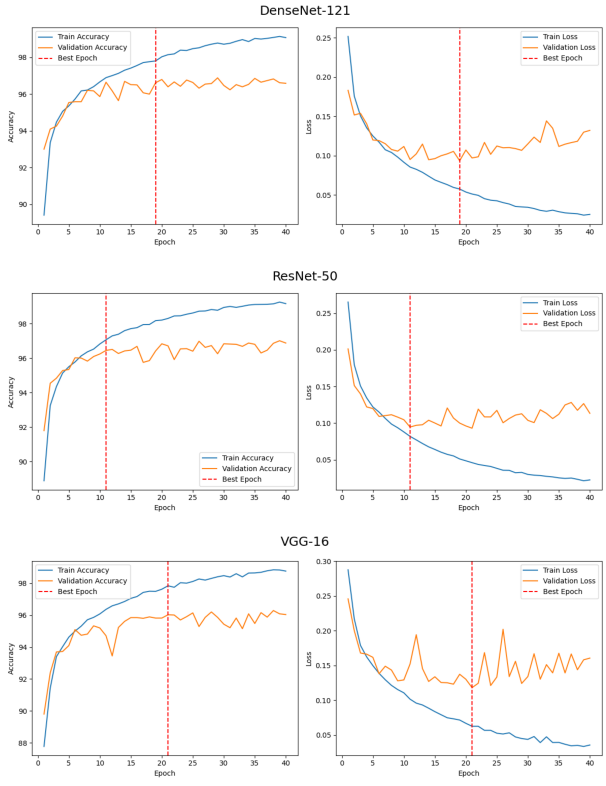

Figure 6: Training and validation curves for VGG16, ResNet50, and DenseNet121 showing generalization performance on the preprocessed dataset.

Training accuracy starts above 85% from the very first epoch, indicating rapid and effective learning. As training progresses, the accuracy curves plateau between epochs 10 and 20. The optimal epoch — marked by a red dotted line representing the checkpoint where the model is saved — falls within this range for all models except VGG-16, where it occurs at epoch 21 (though not far from the boundary). Beyond the optimal point, the training and validation loss curves begin to diverge, suggesting the onset of overfitting. Consequently, training past the optimal epoch is both inefficient and detrimental to generalization.

6 RESULTS

Models' performance evaluation is conducted separately on CNNs models trained with the raw and preprocessed datasets. As mentioned earlier, 6,776 samples from raw and preprocessed datasets are separated before training to evaluate performance. Evaluation is performed based on accuracy, precision, recall, and F1 score - popular metrics for model performance evaluation. The results of the evaluations of DenseNet-169, ResNet-101, and VGG-19 are reflected on Table 2. All metrics show great results for all three models, reaching an accuracy of 97.2 – 97.7% on the raw dataset and 96 – 96.8% on the preprocessed dataset (corresponding F1 score ranges are 0.97 – 0.98 and 0.96 – 0.97). The classification speed for a single code file sample took 0.51 – 2.81 seconds on average, depending on the model. However, the bottleneck in this inference workflow is the loading of the model (taking 0.35 – 2.42 on average, depending on the model). Hence, if the model is loaded beforehand and multiple code files are considered for classification, the time per sample shrinks to just 0.17 – 0.38 seconds per code sample, making this approach time-efficient.

	Raw Images			Preprocessed Images		
Metrics	**DN121**	**RN50**	**VGG16**	**DN121**	**RN50**	**VGG16**
F1	0.98	0.98	0.97	0.97	0.97	0.96
Recall	0.98	0.98	0.99	0.98	0.97	0.98
Precision	0.97	0.97	0.96	0.96	0.96	0.95
Accuracy	97.7%	97.6%	97.2%	96.8%	96.5%	96%

Table 2: Models performance on the 6,776 **raw and** 6,776 **preprocessed images.**

Figure 7 presents the confusion matrices for the preprocessed dataset, showing high performance (96.0% – 96.8% accuracy) with a low false-positive rate (2% – 3%) and a false-negative rate of approximately 1%. Despite these strong results, a small risk of misclassifications remains. Therefore, ethical considerations are essential: model predictions should not be taken as definitive proof that code is AI-generated but rather as an indicator warranting further verification through additional tests.

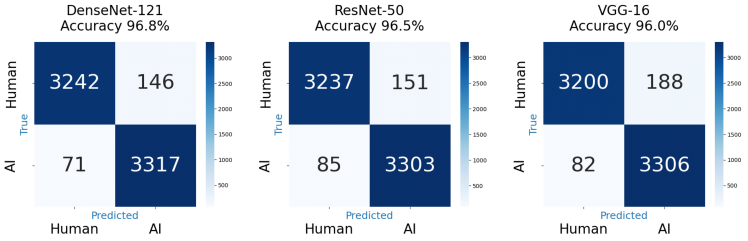

Figure 7: Confusion matrices for DN121, RN50, and VGG16 (6,776 **preprocessed images)**

7 CONCLUSION

The results demonstrate that the employed CNN architectures are highly effective for this classification task, surpassing the performance of the previous BSP. The proposed approach is both effective and scalable, accurately detecting image-based patterns to distinguish AI-generated from human-written code with an accuracy of up to 96.8%. This capability holds practical value in educational and technological settings where the integrity of human-authored code must be verified.

Beyond its immediate applications, this work opens avenues for further exploration. Future research could extend the methodology to other programming languages or develop multiclass CNN models capable of identifying the specific LLM responsible for generating a given code sample. Such advancements would not only broaden the scope of the method but also deepen our understanding of AI code generation patterns.

REFERENCES

[1] [n. d.]. CodeNet Database CodeNet Database on HugginFace. https://huggingface.co/datasets/iNeil77/CodeNet.

[2] Anthropic. 2024. Claude 4 Sonnet: High-performance Language Model. https://www.anthropic.com/claude/sonnet. Accessed: 2025-08-07.

[3] Chetan Arora, Ahnaf Ibn Sayeed, Sherlock Licorish, Fanyu Wang, and Christoph Treude. 2024. Optimizing Large Language Model Hyperparameters for Code Generation. arXiv:2408.10577 [cs.SE] https://arxiv.org/abs/2408.10577

[4] Google DeepMind. 2024. Gemini Flash 2.5: Efficient Multimodal Language Model. https://deepmind.google/technologies/gemini/. Accessed: 2025-08-07.

[5] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[6] Weihua Du, Yiming Yang, and Sean Welleck. 2025. Optimizing Temperature for Language Models with Multi-Sample Inference. arXiv:2502.05234 [cs.LG] https://arxiv.org/abs/2502.05234

[7] Prompt Engineering. 2025. Prompt Engineering with Temperature and Top-p. https://promptengineering.org/prompt-engineering-with-temperature-and-top-p/. The article focuses on describing the optimal parameters to use with LLMs and propose to use temperature with a value of 0.7 as the starting point for general tasks..

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778. https://arxiv.org/abs/1512.03385

[9] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708. https://arxiv.org/abs/1608.06993

[10] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath. 2011. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*. 1–7. https://doi.org/10.1145/2016904.2016908

[11] Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2023. Is this Snippet Written by ChatGPT? An Empirical Study with a CodeBERT-Based Classifier. arXiv:2307.09381 [cs.SE] https://arxiv.org/abs/2307.09381

[12] Wanhu Nie. 2024. Malware Classification Based on Image Segmentation. arXiv:2406.03831 [cs.CR] https://arxiv.org/abs/2406.03831

[13] OpenAI. 2024. GPT-4.1 Model. https://openai.com/index/gpt-4-1/.

[14] OpenAI, Achiam J, Adler S, et al. 2024. *GPT-4 Technical Report*. Technical Report. OpenAI. https://doi.org/10.48550/arXiv.2303.08774.

[15] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. arXiv:2105.12655 [cs.SE] https://arxiv.org/abs/2105.12655

[16] PyTorch Team. 2024. EarlyStopping — PyTorch-Ignite v0.4.12 documentation. https://docs.pytorch.org/ignite/generated/ignite.handlers.early_stopping.EarlyStopping.html

[17] Matthew Renze. 2024. The Effect of Sampling Temperature on Problem Solving in Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 7346–7356. https://doi.org/10.18653/v1/2024.findings-emnlp.432

[18] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV] https://arxiv.org/abs/1409.1556

[19] Edward Tian and Alexander Cui. 2023. GPTZero: Towards detection of AI-generated text using zero-shot and supervised methods". https://gptzero.me

[20] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. 2014. Management of an Academic HPC Cluster: The UL Experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. IEEE, Bologna, Italy, 959–967. https://doi.org/10.1109/HPCSim.2014.6903792