Differentiating user-written from AI generated codes

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

Ali Osman Topal*
aliosman.topal@uni.lu
University of Luxembourg
Belval, Esch-sur-Alzette, Luxembourg

ABSTRACT

The rapid advancement of *Artificial Intelligence (AI)* and *Large Language Models (LLM)* has made it increasingly difficult to distinguish between AI-generated and human-written content. This can be noticed in many fields and particularly in coding. Nowadays, in the age of rising technologies, it is an increasing interest of technology related companies to have a reliable tool, capable of verifying candidates' codes integrity. Therefore, this *Bachelor Semester Project* was aimed to derive a solution to tackle the problem and this paper describes the progress and the outcomes of it. The proposed solution relies on a novel approach to detect AI-generated code using *Convolutional Neural Network (CNN)* models, by first transforming human-written and 4 *LLMs* generated code into images and then using them to train the models.

1 INTRODUCTION

Since the deployment of the *ChatGPT*[1] by *OpenAI* back in 2021, *Large Language Models'* popularity grew exponentially, as more and more people realized the potential application of this tool. Today, it is becoming a standard for people all around the globe to rely on a model that can find relevant information or guidelines in a matter of seconds, help with the spelling and vocabulary or content generation. Many students started using it for school essays long time ago. However, many other people (especially academic world and teachers) saw this to be concerning, as the verification of work's integrity is difficult for a person. Hence, we saw a rise of platforms that can check for the traces of *AI* in the text (e.g. *GPTZero*[16]). But this problem is not limited to just text: models' abilities grew together with the rise of competition upon the release of other models, such as *Gemini*[3], *DeepSeek*[4], *CoPilot*[10], etc. As a result, soon enough it was not too challenging for a *LLM* to be able to generate images, video content or tackle with programming challenges, all of which complicated the verification of one's creation integrity. This means that people also can ask *AI* models (for example *GitHub CoPilot*[7]) to create the codes instead of writing it all themselves, allowing them, for instance, to create vast amounts of coding projects that could be claimed by an individual to be purely hand-written. This makes it challenging for Computer Science related companies to distinguish people with actual programming knowledge from people that rely purely on these tools. Consequently, the introduction of an effective and universal *AI-traces* detecting tool would be greatly awaited and appreciated.

In the research world, there are some works that describe code detection through *linguistic analysis*, using tools like *GPTZero*[5], *OpenAI Text Classifier*[2] or by token conversion into vectors, much like it is implemented in *GPTSniffer* (introduced by Nguyen and his team in the research paper "*Is this Snippet Written by ChatGPT? An Empirical Study with a CodeBERT-Based Classifier*"[11]). However, in the first case the approach is not successful in classifying codes[11] and in the second case, the classifier only addresses the *ChatGPT* code detection and poses some difficulties, to adapt it for other *LLMs* (due to the usage of vectors and tokens, which are specific to each of the models).

To resolve this issue and derive a universal solution, we propose *Convolutional Neural Network* based approach to distinguish between AI and human-written

*Primary BSP supervisor

codes. The suggested method relies on code transformation into images instead of vectors or tokens, which are specific to each of the models. Hence, the adaptation for multiple *AI LLMs* as well as scaling to many other programming languages is vastly simplified. Image recognition related solution is versatile, with the only difficulty in the approach being the acquisition of valid and good-quality data and transformation of it into corresponding images, which could be interpreted by the Convolutional Neural Network model.

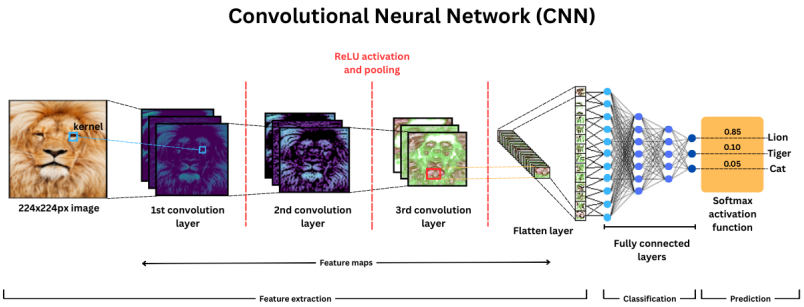


Figure 1: Visualization of a Convolutional Neural Network architecture.

2 CONVOLUTIONAL NEURAL NETWORKS (CNNS)

Before proceeding with the paper, it is worth explaining the main concept of this model that is used in the experiments. I have already mentioned *Convolutional Neural Network*, but what is it actually?

It is a special type of Neural Networks, specifically designed for image recognition, classification problems and computer vision. It relies on image processing using special filters (kernels), which are designed to detect patterns appearing in the image (for example, edges, shapes). This is done in the *convolutional layer*, aimed to extract features from the image. Usually, there are several such layers, allowing to extract higher-level and more complex features from the previous feature maps. Between them, there usually exist special *Pooling* layers, that reduce the number of dimensions, by choosing the highest number for each arbitrary subdivision of the feature map and forming smaller maps. After the extraction, the features are then passed to the *dense* (or *fully-connected*) neural layers, which contain individual nodes, interconnected between with different weights and biases. This is where the actual pattern recognition and classification happens. These parameters are updated after each iteration of the training process, using *backpropagation* algorithms, that allows model to converge towards correct output. Finally, models end with a flattened layer with as many nodes, as there are classes. A special non-linearity activation function is used (such as *Softmax* or *Sigmoid*), resulting in some number (for instance, *Sigmoid* gives a probability $\in [0, 1]$. Then the output is evaluated and the class with biggest probability is chosen and compared with the actual label of the image. If they do not match, the *backpropagation* method is implied and *dropout* is performed, which deactivates some of the nodes for the next learning iteration (this is done to avoid dependence on specific biased features). It is also worth noting, that between the mentioned layers, there usually exist non-linearity activation functions as well (nowadays, preferred *ReLU*), ensuring proper distinction between the classes in the model. More information can be found online[15].

3 METHOD

The method we have chosen is known for its usage and success in malware classification problems. Approach is described in a paper "*Malware Classification Using Convolutional Neural Network*" [5] and in "*Malware Classification Based on Image Segmentation*"[12]. Both papers describe successful results of malwares classification into specific categories.

Following this idea, we adapted the approach for a binary classification problem that we have, where instead of tens, just 2 classes would be present: *AI* and *Human*. We chose to use java programming language for this research and experiments through training, as this language is among the most popular ones and it was feasible to obtain the necessary code samples for it as described in section 4. However, it is worth noting that the choice of this language does not bound the scalability of our proposed method in any way, since the same processes described in the paper can be easily applied for other programming languages as well.

To apply the method correctly, it was necessary to create a unique workflow to ensure the successful CNNs training. It is described in this paper through sections in the following order:

(1) *Data generation and Accumulation*

(2) *Dataset preparation*
(3) *Preprocessing*
(4) *Code to Image transformation*
(5) *Training the CNN models*
(6) *Evaluation*

By relying on the schematic step-by-step procedures, we can ensure the control of the data and the correct measurements to be taken for the training. In short, following the procedures we managed to accumulate 7, 248 java code samples, process and turn them into a dataset of 6, 093 images valid for training and successfully train the CNN models to distinguish between AI and human-written codes at an accuracy of reach an accuracy of 90.5 − 93.5% (more details in *Table 6*).

4 DATA GENERATION AND ACCUMULATION

The primary step before training the models, is obtaining the necessary data and good-quality data, which could be fed into the models.

For this research and training of the CNN models, we used 2 datasets:

- Students generated solutions
- AI generated solutions

Students' dataset consists of purely human-written solutions to 9 distinct exercise sheets, containing 46 questions in total. This dataset comes from the *University of Luxembourg Bachelor in Applied Information Technologies* students from 2021 (noticeably, before the introduction of famous LLM *ChatGPT*) winter semester and were part of the *"Programming 1"* course, written by 58 students and with exercises that vary in length and difficulty.

In the meantime, AI solutions were generated manually using 4 *LLMs*: *ChatGPT 4*, *Deepseek 2.5 and 3.0*, *Gemini Flash 2.0* and *Microsoft Copilot*. Models were given the same exercise sheets for java programming language, as the students and were asked to provide 5 solutions to each of the programming exercises. To achieve this we used 2 prompting patterns: the Introductory and the Continuation prompts.

The Introductory prompt was intended to introduce the *Large Language Model* to the new task and explain that this would be a java programming language exercise and was formulated in the following way:

> *I will give you a Java coding exercise. Solve it:* ***{exercise}***

The Continuation prompt was used to instruct the model to generate additional 4 solutions and is formulated in the following way:

> *Solve it again. [Put the solution in different files.]*

Sometimes *LLMs* tend to put code in different files (reduce complexity), other times everything is put in 1 file instead. For the latter case, the additional prompt in the square brackets("[ ]") part was used. Afterwards, the solution is copied and pasted into a java file, storing it in a predefined dataset hierarchy, described further. Process is iterated for each of the unique exercise and for each of the 4 used *LLMs*.

Talking about the number of files, finally we obtained a total of 7, 248 java files, out of which 4, 303 were human-written and obtained from the University, while the remainder of 2, 945 were AI-generated, which further divide into 695 generated by *ChatGPT*, 729 by *Microsoft CoPilot*, 761 by *DeepSeek* and 760 by *Gemini*.

5 DATASET PREPARATION

As mentioned before, we gathered thousands of java code files. With such amount of data, the organization of the datasets is a very important aspect to ensure good handling of the training samples. The students' dataset had an already pre-defined hierarchy, while the AI dataset hierarchy had to be created manually. Thus, the hierarchy is defined as shown in AI dataset hierarchy (**Figure 2**).

The Root folder contains the *codes* and *images* parts. These folders further devide into *students* and *AI* folders. Accordingly, the folders contain the content as specified in the path. Folder *codes/AI/* contains "labs", ranging from 02 to 10, which are exercise sheets. Inside there are *chatgpt*, *copilot*, *deepseek*, *gemini* folders with 5 solution folders inside denoted *solN*, where N is the natural number ranging in $[1, 5]$. Finally, inside each of the solution folders there are 1-11 java files, depending on the complexity of the exercise.

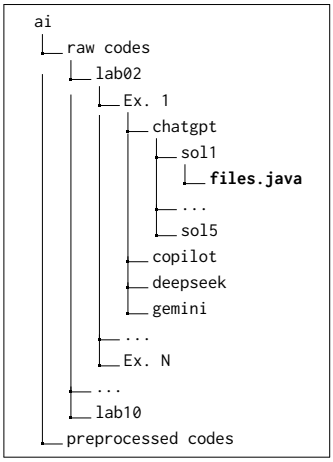


Figure 2: AI dataset hierarchy

6 CODE TO IMAGE TRANSFORMATION

The next important step is the preparation of the dataset, more specifically, the transformation of the java files into PNG images. The simplified workload of this is illustrated by the scheme and explained in steps below:

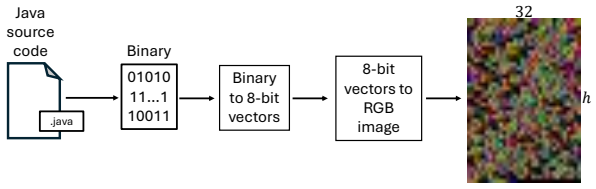


Figure 3: workflow - from code to image

1) We first open each java code file in a *binary mode* and read the *binary representation*:

`public class ...` → 01110000 01110101 ...

2) Then, the *binary code* is divided into *vectors* of a 8-bit length. If the division is uneven and there are remainders, the last vector takes the *remainders* and the rest is filled with *0* values. As a result, each vector's *integer representation* is in range from *0* to *255*:

Vectors with remainders are filled with *0*s:

$$(0\,1) \rightarrow (0\,1\,0\,0\,0\,0\,0\,0)$$

We then get a set of vectors:

$$S = \{(0\,1\,1\,1\,0\,0\,0\,0), (0\,1\,1\,1\,0\,1\,0\,1), \dots\}$$

Decimal representation:

$$\forall\, \underline{\mathbf{v}} \in S,\ \text{int}(\underline{\mathbf{v}}) \in [0,\ 255]$$

3) The vectors are then divided into groups of 3, each vector in the group representing RGB values of a pixel accordingly. If the division has remainders, the $\underline{\mathbf{0}}$ vector fills the gaps:

Each pixel i represented by RGB values:

$$\forall\, i \in \left\{0, 1, \dots, \left\lfloor \frac{|S|-1}{3} \right\rfloor\right\}, \quad \underline{\mathbf{v}}_{3i},\ \underline{\mathbf{v}}_{3i+1},\ \underline{\mathbf{v}}_{3i+2} \in S \cup \{\underline{0}\}$$

$$\implies RGB_i = (\underline{\mathbf{v}}_{3i},\ \underline{\mathbf{v}}_{3i+1},\ \underline{\mathbf{v}}_{3i+2})$$

If there are remainders:

$$(\underline{v}_k, \underline{v}_{k+1}) \rightarrow (\underline{v}_k,\ \underline{v}_{k+1},\ \underline{0}), \quad k \in \mathbb{N}$$

4) We set a fixed length of an image to be $L = 32$ (pixels). Then we get the height h of the image by dividing number of vectors by L and adding 1 if there are remainders. If remainders were found, the rest of the row is filled with padding pixels:

$$h = \left\lceil \frac{|S|}{L} \right\rceil$$

$$|S| \bmod L \neq 0 \implies \text{filling row with pad pixels}$$

Padding pixels: $RGB_p = (\underline{0},\ \underline{0},\ \underline{0})$

This way we obtain h rows of RGB pixels.

5) Then these pixel rows are transformed into a PNG image. The final dimensions of an image are 32 pixels in length and ≥ 32 pixels in height:

- if $h < 32$, then $H = 32$ and the remainder rows are filled with padding (*example in figure 2 below*).
- if $h \geq 32$, then $H = h$ (*example in Figure 4, last picture. There $H = h$*).

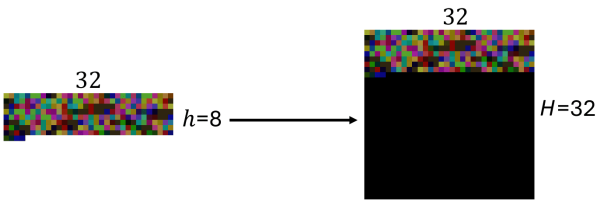


Figure 4: PNG image formatting

6) Finally, the image is saved in PNG format in appropriate branch in the dataset hierarchy. Additionally, a *info.json* file is saved in same directory as *bad, good, over* folders. The json file contains metadata that could be used later for analysis and traceability and contains image name, the relative directory of the java file that is represented by the image and the "quality", denoted *bad* (if $h < 3$), *good* ($3 \leq h \leq 32$) and *over* ($h > 32$). The idea behind this classification is that, in order to have a meaningful data, for the *CNN model* to train on, we must have at least 3 rows of meaningful pixels in the image.

As a result, after filtering out the images that had $h < 3$ and leaving the rest in the dataset, we obtain 3872 images from codes written by humans and 2618 AI generated codes transformed into images. See *Table 1* for more details.

7 PREPROCESSING

Codes written by 2 very different entities naturally contain some evident biases, that can serve as a pattern for CNN to easily distinguish the 2 parties. To ensure proper training, it is very important to diminish "obvious" patterns.

An example of such bias occurrence is shown in the **Table 2**. Students' codes had a specific package and a special type of comment, that could easily give out code's origins, while AI generated code sometimes had a reference to the name of the file containing the code. With the help of Python, regular expressions and automatization, we could clean the codes from evident biases and prepare the dataset for proper training. After preprocessing, the codes are transformed into images in the same way described in *section 6*.

Preprocessed codes and images are saved separately from the raw datasets, allowing us to train on both datasets and compare the results. After this step is completed, the data is now fully prepared for the training. Here is a table, representing the exact amount of data samples in the dataset:

Nature	Codes	Raw Images		Preroc. Images	
		$h < 3$	$h \geq 3$	$h < 3$	$h \geq 3$
Human	4303	431	3872	804	3499
AI	2945	327	2618	351	2594
Total	**7248**	758	**6490**	1155	**6093**

Table 1: Raw and preprocessed images datasets. Only $h \geq 3$ images are considered for the training.

Considering only the already mentioned $h \geq 3$ codes and images, the corresponding file sizes are displayed in **Table 3**. Additionally a more broad view on the average file sizes in the dataset is shown in **Table 4**.

What is interesting to notice in the **Table 3** is that after processing the codes, the average size of the codes that satisfy the condition $h \geq 3$ after transformation decreased for AI generated codes (this could have been expected, as in general the processing erases some parts of the codes), while the Human-written codes average size increased. This can be explained by noticing the differences visualized in **Figure 5**. Majority of students written code sizes are near the border of 3 rows, below which the codes are allocated to *Bad* category (red color in the Figure). After preprocessing, big part of these codes fall within the $h < 3$ category, meaning that they are no longer considered for the training. As a result, the average code size decreases.

8 TRAINING THE CNN MODELS

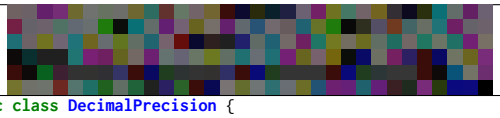
After the data preparation stage, the following step is the most important of all - the *Training*. The broad workflow of training the CNN model and getting the results, output of the model is portrayed in **Figure 6**. The code first gets transformed into the image representation, then before feeding it into CNN model for training, it is reshaped into $a \times b$ dimensions, to fit the expected dimensions depending on the model (in this paper all the used models had an input dimensions of 224x224). Then the image is processed and used for training. Later in the evaluation stage, the model is used to determine its performance, depending on its

<table>
<tr><th>Raw/preprocessed</th><th>Human-generated code</th><th>AI-generated code</th></tr>
<tr><td>Raw codes</td><td>

```java
package lu.uni.programming1.lab2.exercise2;
/**
 *  Class for Exercise 2.3
 */
public class DecimalPrecision {
        public static void main(String[] args) {
                // TOdo: Define two decimal numbers per type
                ↪  float and double
                //       make assign a very small and large value
                ↪  per type
                //       try to add them respectively and print
                ↪  out the value

                float a, b;
                double x,y;

                a = 222222000444f;
                b = -3333330000f;
                x = 444444444;
                y = -66666666;

                System.out.println("Sum of a and b: " + a+b);
                System.out.println("Sum of x and y: " + x+y);
        }
}
```

</td><td>

```java
// PrecisionFloatDouble.jav
public class PrecisionFloatDouble {
    public static void main(String[] args) {
        double small = 1e-15; // very small number
        double large = 1e15;  // very large number

        // Adding small to large
        double result = small + large;

        // Printing the result
        System.out.println("Result of adding a very small number
        ↪  to a very large number: " + result);
    }
}
```

</td></tr>
<tr><td>Raw images</td><td>  </td><td>  </td></tr>
<tr><td>Preproc. codes</td><td>

```java
public class DecimalPrecision {
        public static void main(String[] args) {

                float a, b;
                double x,y;

                a = 222222000444f;
                b = -3333330000f;
                x = 444444444;
                y = -66666666;

                System.out.println("Sum of a and b: " + a+b);
                System.out.println("Sum of x and y: " + x+y);
        }
}
```

</td><td>

```java
public class PrecisionFloatDouble {
    public static void main(String[] args) {
        double small = 1e-15; // very small number
        double large = 1e15;  // very large number

        // Adding small to large
        double result = small + large;

        // Printing the result
        System.out.println("Result of adding a very small number
        ↪  to a very large number: " + result);
    }
}
```

</td></tr>
<tr><td>Preproc. images</td><td>  </td><td>  </td></tr>
</table>

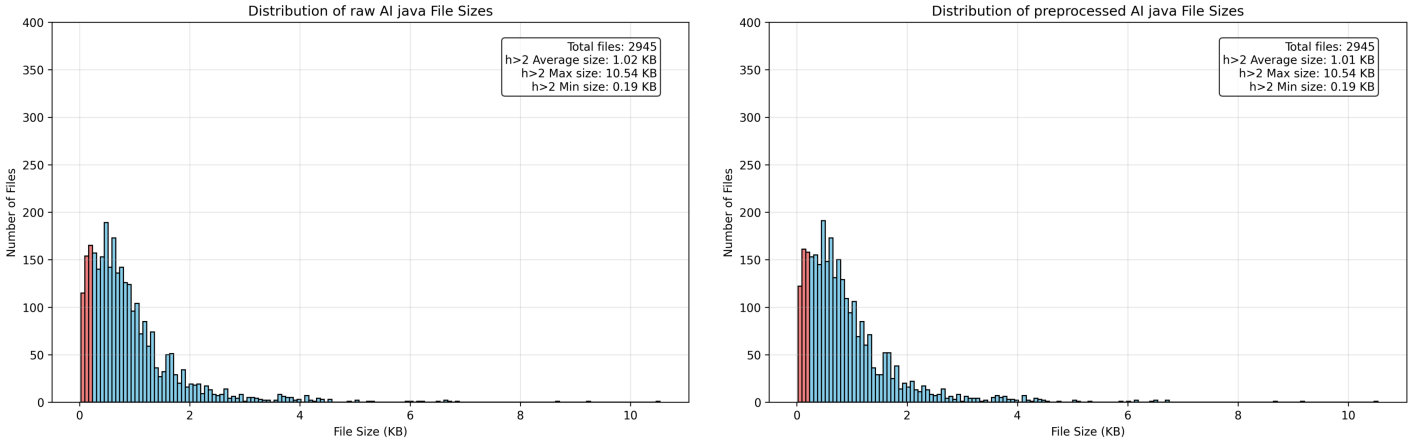
Table 2: Examples of raw and preprocessed codes and images generated from human-written and ChatGPT-4-generated codes, addressing the same question.

Nature		AI		Humans	
Type	Dataset	Range	Average	Range	Average
Codes	Raw	0.19-10.54 KB	1.02 KB	0.19-10.20 KB	1.08 KB
	Preprocessed	0.19-10.54 KB	1.01 KB	0.19-10.14 KB	1.10 KB
Images	Raw	0.25-5.96 KB	0.750 KB	0.25-4.67 KB	0.79 KB
	Preprocessed	0.24-5.96 KB	0.748 KB	0.23-4.73 KB	0.78 KB

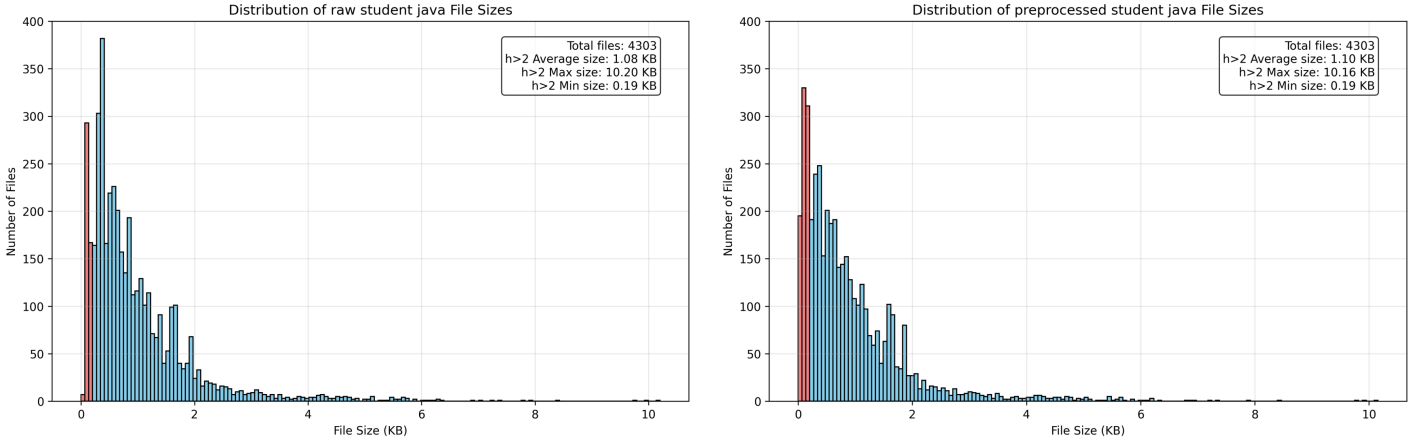
Table 3: Detailed files sizes

Average	Codes	Images
Raw	1.055 KB	0.775 KB
Preprocessed	1.062 KB	0.766 KB

Table 4: Overall file size averages (AI and humans)



(a) AI generated code sizes change visualization



(b) Humans written code sizes change visualization

Figure 5: Comparison of code sizes change: AI vs Humans. Red color indicates $h < 3$.

	Raw Dataset			Preprocessed Dataset		
	DN169	RN101	VGG19	DN169	RN101	VGG19
OTT	1304	1448	1499	1222	1330	1398
Av. TT/epoch	32.6	36.2	37.48	30.6	33.3	35

Table 5: Overall training time for 40 epochs and average training time per epoch (in seconds) of the models on the raw and the preprocessed datasets.

output. The model initially outputs a single number p in range from 0 to 1, which corresponds to probability of the image being AI-generated. To obtain probability of the code being human-written (q), we need to subtract p from 1: $q = 1 - p$. **Figure 6** portrays both of these probabilities.

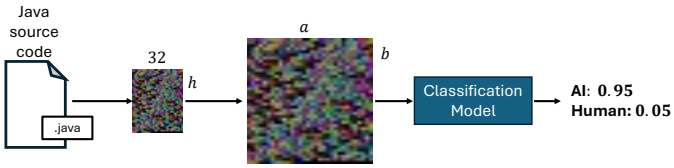


Figure 6: scheme - from code to training and evaluation.

It's worth noting that most computations (training, evaluation) were performed on nodes with NVIDIA Tesla V100 GPUs of the IRIS HPC Cluster [17] at the University of Luxembourg.

For this approach we rely on CNN models pre-trained by *ImageNet*[6]: *VGG-19*[14], *ResNet-101*[8] and *DenseNet-169*[9]. These CNN families are well known for their Image recognition and detection capabilities. Models were adapted with a change of last classification layer to have just 1 output feature (instead of hundreds, as by default) to correlate models specifically with distinction between 2 classes, based on image representation of a code.

The specific steps of the these models training are as follows:

(1) First, 100 images random images are selected from both human-written and AI-generated codes for evaluation.
(2) The remaining data was split into 80% for training and 20
(3) Non-normalized images of size $32 \times H$ (explained in **Figure 3**) are resized through transformations to fit the 224×224 input size of the pre-trained CNNs.
(4) Adam optimizer is used with a learning rate set to 0.0001, batch size is set to 32, and the models are trained for 40 epochs.
(5) At final stage, binary cross-entropy loss is used. The last layer of the CNNs in a binary classification task ends with a *sigmoid activation* function, therefore CNNs output a single value $\in [0, 1]$ to determine the prediction and likeliness of the code being generated by AI, *i.e. if the output is close to 1, it predicts to be AI-generated, while if it it's close to 0, it predicts to be humans-written.*

Training performed over 40 epochs, requires between 21-25 minutes on the raw dataset, and between 20-23 minutes on the preprocessed dataset, depending on the model considered. Consequently, the average time for training per epoch is approximately 32-37 seconds for raw and 30-35 seconds for the preprocessed datasets (for detailed timings see **Table 5**). These times are influenced by the difference in the amount of training data, as preprocessed datasets are smaller (see **Table 1** and **Table 3**).

9 EVALUATION

Models' performance evaluation is conducted separately on *CNNs* models trained with the raw and preprocessed datasets. As mentioned earlier, 200 samples from raw and preprocessed datasets are separated before training to evaluate the performance. Evaluation is performed based on 4 metrics (*note that the boolean part of "**True/False** positive/negative" describes whether the model predicted correctly a positive (AI-generated code) / negative (human-written code) sample*):

1) **Accuracy**. This metric is used widely and describes the performance generally. It describes the proportion of correct predictions in general. It is calculated in the following formula:

$$Accuracy = \frac{True\ positives + True\ negatives}{Total\ predictions}$$

2) **Precision**. This metrics describes the proportion that tells how many of the predicted (AI-generated codes) in total by the model were actually correct and positive. It is defined as follows:

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}$$

3) **Recall**. This metrics describes the percentage of correctly predicted positives in regard to all positives:

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives}$$

4) **F1 score**. This metric provides a balanced overview of *Precision* and *Recall* metrics. It is known to be useful in unbalanced dataset, where 1 class is underevaluated by the CNN model. It is defined as follows:

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Additionally, the training history is saved and plotted to obtain a learning curve and confusion matrix is derived based on the evaluation with the selected samples. All the evaluation graphs and results are provided in **Table 6**, **Figure 7** and **Figure 8**.

10 RESULTS

The results of the DenseNet-169, ResNet-101 and VGG-19 are reflected on **Table 6**. It is worth noting that all of the metrics are ≥ 0.88, which indicates good performance and predictability. In the meantime, accuracy is ranging from 90.5% up to 95%. The highest parameter is precision of Raw images of RN101 and VGG19 models with astonishing 0.99 values. Also worth noting that other competitive detectors: GPTZero and GPT-Sniffer had their best F1 score of 0.82. CNNs surpass it by a matter of ≈ 0.10. This shows that the models are very much capable of identifying and differentiating AI-generated from human-written codes and pose good opportunities in the future to scale these models into reliable and plausible tools for the detection of AI traces, as alternatives to the already existent tools. Especially considering its training time of $20-25$ minutes (or even less with 20 epochs) and evaluation time of a single image taking no more than ≈ 1 second).

Another important factor in training is the learning curve, visualizing the process of training of the models. It can be seen in **Figure 7a** for the Raw dataset training for different models and **Figure 7b** for preprocessed dataset training. Noticeably, the learning curve skyrockets in the first 5 epochs and reaches the best performance result in about 10-15 epochs, after which the benefits of training seem to be declining and the learning curve either more or less stable or slightly increasing. This indicates that the optimal amount of epochs for training is around 20. To conclude the results, a confusion matrix is provided, showing the effectiveness on the evaluation set for the 3 models for raw and preprocessed datasets (**Figure 8**).

11 PROJECT OUTCOME

This paper shows strong performance results of the CNN models in terms of differentiating AI-generated codes from human-written, allowing for an accuracy reaching up to 93.5% accuracy with DenseNet-169 model on preprocessed dataset. Therefore, the project outcome is positive, as the intended goals were reached and the model with the required capabilities was constructed. This successful discovery poses opportunities to be continued in further Bachelor Semester Projects, to scale this research up with larger databases and a variety of other programming languages.

11.1 Contribution

My personal contribution in this project is closely related to the training of the models and the creation of dataset, organization and processing of AI-generated codes, generation of the code and training samples themselves. The hierarchy organization and solving the provided exercise sheets using 4 Large Language Models (as described previously in the paper) was one of the biggest tasks in this project.

More precisely, around ≈ 4.000 different AI java file codes were created, out of which ≈ 1.000 had to be discarded due to the flaws and mistakes naturally introduced in the dataset due to the lack of experience. Consequently, the meaningful data accumulated and used for the training was 2945 samples. These samples were organized in a pre-defined folders hierarchy.

The following step was formatting the data into a trainable format, i.e. transforming java files into PNG format images. This was using *Python* codes and automatization. A workflow was defined and the image transformation and organization criteria were created, separating images depending on the number of rows present into *bad, good* and *over* categories.

Finally, I performed trainings of 12 pretrained models from different families, 3 of which are reflected on this paper. Constructing the codes for training and initiating the jobs on *University's of Luxembourg Hyper Performance Computer (HPC)* allowed me to learn how to use it and gain valuable experience.

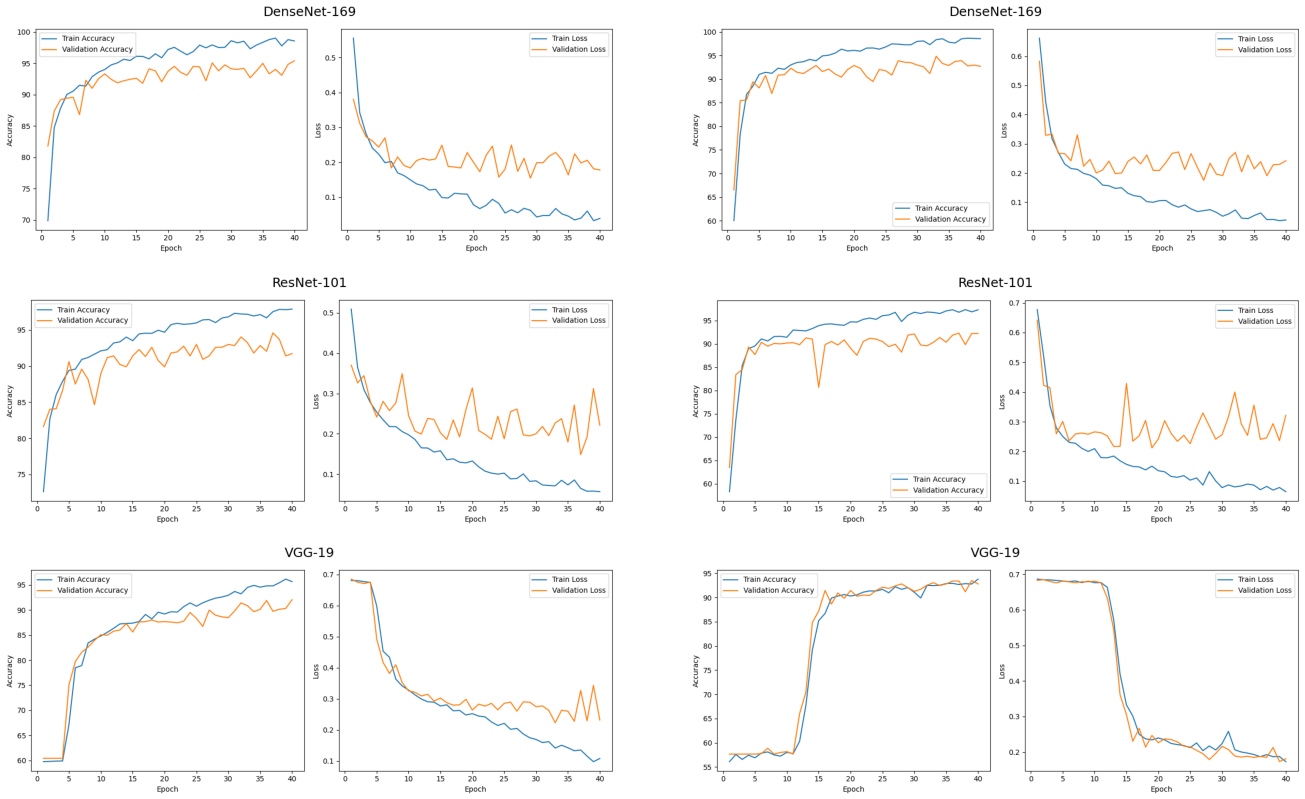
Finally, as a lot of analysis and supplementary data was required for the paper to be sufficient and accurate, it was a good way to discover and learn about how to navigate through datasets and how to obtain relevant information. This was done through *Python* codes and automatization.

11.2 Lessons learned

To begin with this section, it is worth noting that this project was beneficial in terms of getting to know the

	Raw Images			Preprocessed Images		
Metrics	DN169	RN101	VGG19	DN169	RN101	VGG19
F1	0.95	0.94	0.93	0.93	0.91	0.93
Recall	0.92	0.89	0.88	0.92	0.90	0.94
Precision	0.98	0.99	0.99	0.95	0.91	0.91
Accuracy	95%	94%	93.5%	93.5%	90.5%	92.5%

Table 6: Models performance on the raw and preprocessed images (200 images per category).



(a) Raw dataset

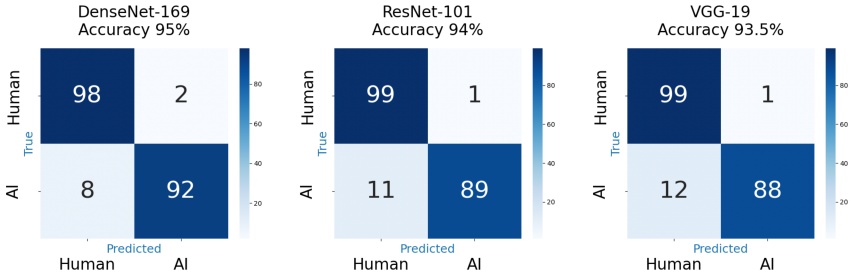
(b) Preprocessed dataset

Figure 7: Learning curves for each model on the raw and preprocessed datasets

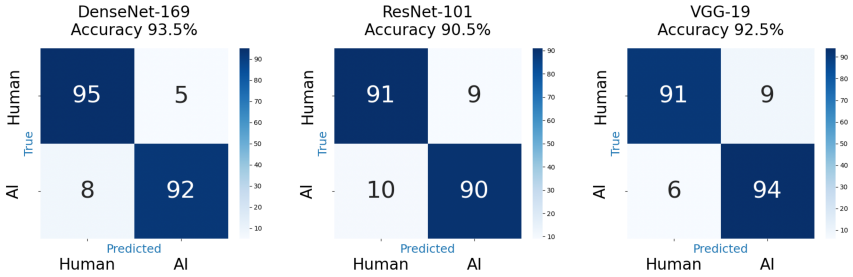
latest technologies related to AI and *CNN* models, learning about them and understanding the reasoning, mechanics and the workflow behind training the Neural Networks: mathematics behind it, how exactly the learning happens and how to build the neural networks, according to the problem we're addressing. Classification and the number of classes matter. Finally, the construction of *CNN model* from scratch gives great insights and experience in this field, allowing me to expand my knowledge in this field even further.

Another important skill and experience I have obtained during the project is the experience of working with *University of Luxembourg HPC*. This Linux based environment, connecting to it and managing it seemed complicated and complex at first. However, we managed to address the issues and connect successfully and train the models accordingly. This experience is important for further scientific researches or projects, requiring something similar.

Moreover, this project allowed me to learn more about how the dataset navigation, analysis and work



(a) Confusion matrices for DN169, RN101, and VGG19 (200 raw images)



(b) Confusion matrices for DN169, RN101, and VGG19 (200 preprocessed images)

Figure 8: Comparison of confusion matrices and accuracies for DN169, RN101, and VGG19 models in evaluation step.

conclusion, report creation works. It allowed me to broaden my knowledge and confidence in *Python* programming language.

During this project, we faced some difficulties, some of which were caused by the natural complexity of the problem itself, while some were caused by our own errors. One of them was the creation of the dataset. An important lesson was to ensure that the data generation and creation and storing it in the hierarchy is consistent and is done with a plan to prevent unnecessary waste of time afterwards, trying to fix, which happened to us. This important lesson is the key component of any work, project or research with large amounts of data. Another important key concept is that quality of data is more important than speed and amount of data, this is to reduce the number of biases in the dataset.

12 CONCLUSION

Overall, Convolution Neural Networks, as reflected by paper's results, have strong potential in terms of being an effective an universal tool in detecting AI-generated code. It is flexible and applicable to multiple Large Language Models and reasonably easily adaptable for other programming languages as well (considering proper preprocessing). All in all, methods provide a valid and strong alternative to the already existing AI detections tools, allowing for higher accuracies than the counterparts. This can be potentially widely applicable in the real world scenarios, especially by the academic world and technology-related businesses, where AI traces detection is critical.

Finally, this research opens door for further experiments with larger databases, such as *CodeNet*[13] with millions of code samples.

REFERENCES

[1] 2022. ChatGPT: Optimizing language models for dialogue. https://openai.com/blog/chatgpt

[2] 2023. OpenAI Text Classifier. https://platform.openai.com/ai-text-classifier

[3] 2024. Google Gemini. https://gemini.google.com/

[4] 2025. Deepseek. https://www.deepseek.com/en

[5] Tekerek A. Ayaz Z. and Yapici M. M. 2021. Malware Classification Using Convolutional Neural Network. *ResearchGate* (2021). https://www.researchgate.net/publication/356587735_Malware_Classification_Using_Convolutional_Neural_Network

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[7] GitHub. 2021. *GitHub Copilot*. https://github.com/features/copilot

[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] Microsoft. 2024. Microsoft Copilot. https://www.microsoft.com/en-us/copilot

[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] 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

[14] 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

[15] Rishabh Singh. 2024. *Convolutional Neural Network (CNN)*. https://medium.com/@RobuRishabh/convolutional-neural-network-cnn-part-1-d1c027913b2b Accessed on 2025-05-26.

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

[17] 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