Enhancing Code Readability through Automated Consistent Formatting

Thomas Kanoutas, Thomas Karanikiotis * and Andreas L. Symeonidis

Abstract: Code readability is critical to software development and has a significant impact on maintenance and collaboration in evolving technology landscapes. With the increasing complexity of projects and the diversity of developers’ coding styles, the need for automated tools to improve code readability has become more apparent. This paper presents an innovative automated system designed to improve code readability by modeling and enforcing consistent formatting standards. The approach uses techniques such as Long Short-Term Memory (LSTM) networks and N-gram models, allowing the system to adapt to different coding styles and preferences. The system works autonomously by analyzing code styling within a project, identifying deviations from established standards and providing actionable recommendations for consistent styling. To validate our approach, several evaluations were performed on a large dataset of Java files. The results demonstrate the system’s effectiveness in detecting and correcting formatting errors, identifying a formatting error within the first five predictions more than 90% of the time, while providing the correct fix nearly 96% of the time, regardless of formatting convention or programming language. By offering a solution tailored to the specific needs of different teams, our system represents a significant advance in automated code formatting and readability improvement.

Keywords: source code readability; source code formatting; source code styling; LSTM; N-grams

1. Introduction

Quality software development is essential to modern society, affecting many aspects of daily life, from communication and commerce to entertainment and healthcare. High-quality software not only ensures the efficient operation of systems, but also contributes to user satisfaction, reliability, and security [1,2]. However, achieving and maintaining software quality is a multi-dimensional task, covering different dimensions such as functionality, performance, reliability, and maintainability [3].

Among these dimensions, code readability is one of the most critical factors affecting software quality. As well as being visually appealing, readable code has a significant impact on software maintenance, development, and collaboration [4]. Readable code is easier to understand, debug and modify, facilitating smoother development workflows and reducing the likelihood of introducing errors during code changes [5]. Furthermore, in collaborative environments, readable code supports effective communication between team members, promoting knowledge sharing and collective understanding of the code base.

Formatting plays a key role in code readability. Consistent and well-defined formatting conventions improve code clarity and comprehension, allowing developers to quickly identify code structure, logical flow, and relationships between different components [6]. Conversely, inconsistent or poorly formatted code introduces cognitive overhead, limiting developers’ ability to efficiently navigate and understand the code base.

Previous research efforts have explored various approaches to automating code styling and formatting, with the goal of reducing the effort of manual formatting tasks and enforcing consistent coding standards across development teams [7,8]. However, many existing
solutions rely on static sets of formatting rules that may not adequately address the varying preferences and requirements of different projects or development teams. As a result, there is a need for a flexible and adaptive system that can autonomously learn and apply appropriate formatting conventions tailored to specific contexts.

In an effort to tackle this need, we present an automated mechanism for modeling the code formatting preferences of individuals and development teams. Our system works in an unsupervised manner, analyzing the existing code style within a project or repository and identifying deviations from desired formatting standards. Extending our previous work [9], where we proposed an automated mechanism for identifying styling deviations and formatting errors in source code files, our current approach not only provides potential fixes for these formatting errors but also incorporates statistical properties and the concept of “naturalness” in source code, providing actionable recommendations for maintaining consistent and readable code styling, tailored to the specific characteristics of each project.

The main contributions of our approach are summarized in the following points:

- **Automated Mechanism for Code Formatting**: We propose a system that autonomously learns and enforces appropriate formatting conventions within a project, drastically reducing the manual effort required for formatting tasks.

- **Unsupervised Analysis**: Our system operates in an unsupervised manner, analyzing existing code styles and identifying deviations without the need for predefined rules, making it adaptable to various projects and team preferences.

- **Statistical Modeling and Naturalness**: The approach leverages statistical properties and the concept of “naturalness” in code to provide actionable recommendations, enhancing code consistency and readability.

- **Extension of Previous Work**: This research extends our previous work by not only identifying formatting errors but also providing fixes, thus improving the overall efficiency of the code formatting process.

- **Ease of Collaboration**: By promoting consistent coding standards and improving code readability, our system enhances collaboration between developers and contributes to higher software quality.

The primary purpose of our system is to automate the code formatting process by providing developers with a tool that can autonomously learn and enforce appropriate formatting conventions within their projects, according to the formatting preferences of each team. By reducing the manual effort required for formatting tasks and promoting consistent coding standards, our system aims to improve code readability, facilitate collaboration between developers, and ultimately enhance software quality. It is important to clarify that our system does not have an understanding of code in the traditional human sense. Rather than interpreting the semantics or meaning of code entities, our system operates on statistical patterns learned from the order in which tokens appear in code files. Therefore, it is important to note that our system’s evaluation and error detection capabilities are based on statistical modeling and pattern recognition, rather than semantic understanding.

The rest of this paper is organized as follows. Section 2 reviews recent approaches to source code formatting and explains how our work differs from and enhances those approaches. Section 3 depicts our methodology, along with details about the models we have employed, while in Section 4, we evaluate the performance of our approach in the detection and fixing of formatting errors. Lastly, Section 5 delves into an examination of potential threats affecting both our internal and external validity. Following that, in Section 6, we discuss the conclusions drawn from our approach and offer insights for future research directions.

### 2. Related Work

Source code readability is important in efficient software development as it is considered a pillar aspect of maintainability and reusability. Readability, defined as the ease with which code can be understood by humans, plays a vital role in software engineering [10]. It is underlined that readability is essential not only during the development phase, but also
during maintenance, as understanding the code is the first step in ensuring effective maintenance [11]. The readability of a program is closely related to its maintainability, making it a key factor in the overall quality of software [12]. Studies have shown that developers actively work to improve the readability of their code, demonstrating its importance in software development [13].

Improving software readability involves various different approaches, such as improving the readability of code comments [14] and making effective use of method chains and comments [15]. Refactoring, when carried out correctly, can enhance software quality by improving readability, maintainability, and extensibility [16]. In addition, studies suggest that simple blank lines are more critical than comments when assessing software readability [17]. Furthermore, a readable coding strategy is beneficial for future maintenance and facilitates the understanding and transfer of rules in different domains and pipelines [18].

Code formatting, particularly indentation, is a critical factor in improving readability. It helps developers to understand the content and functionality of the source code. Hindle et al. [19] and Persson and Sundkvist [20] have highlighted how indentation shapes correlate with code structure, thereby improving comprehension. In their research, Hindle et al. analyzed over two hundred software projects to explore the relationship between indentation shapes and code block structures. They discovered a remarkable correlation between the patterns created by code indentations and the overall structure of the code. This finding is particularly valuable for developers, as it helps to improve their understanding of the content within a software component. In the same context, Persson and Sundkvist argue that the correct use of code indentation can improve the readability and comprehension of source code.

The development of tools such as Indent [21] and Prettier [7] demonstrates efforts to preserve code formatting and detect deviations from established standards. In addition, a code editor that separates the functional components of source code from its aesthetic and formatting presentation was the aim of the work of Prabhu et al. [22]. The team also included features such as automatic indentation and spacing to provide a formatting solution that reduces the need for manual intervention, while useful, these automated formatting features rely on heuristic algorithms developed by the researchers. They adhere to a standardized styling approach and do not allow for modification.

At the same time, there have been some research efforts to model some globally accepted formatting rules in order to detect deviations from them in new instances [23]. However, these methods rely almost entirely on a set of fixed rules over which developers have limited control. As a result, the ability to customize or adapt them to specific coding needs is limited. This is particularly important for maintaining consistent formatting across projects or files in large teams, which is key to achieving high quality code. Kesler et al. [6] support this point in their research. They conducted experiments to determine how different levels of indentation—none, excessive, or moderate—affect the readability of the source code. Their findings suggest that no single indentation style can satisfy the needs of all teams or individual developers, emphasizing the importance of careful consideration in each case. Miara et al. [24] conducted a study of common indentation styles and concluded that indentation depth is critical to code readability. They found that different programs use a variety of indentation styles, highlighting the need for models that can recognize the specific code formatting used in a project and identify any deviations from it.

Towards this direction, several approaches have been developed to model encoding formats and propose styling corrections. NATURALIZE, by Allamanis et al. [25], was one of the early frameworks in this area, focusing on the specific formatting of a project and proposing fixes for deviations. However, its scope was limited to local contexts, and mainly to indentation and whitespace, rather than more general formatting elements. Another notable approach is Codebuff by Parr et al. [26], which aimed to automatically generate universal code formatters based on the grammar of a given language. Although effective, Codebuff was limited in its generalization capabilities and struggled to handle mixed indentation styles and quote variations. Finally, STYLE-ANALYZER by Markovtsev et al. [27] offered a more complex solution, providing suggestions for correcting formatting
deviations after analyzing a project’s style. Although the approach showed promising results, its application was limited to JavaScript source code and involved a time-consuming process. These approaches, each with their own focus and limitations, have made significant contributions to the automation of code styling and formatting.

Last but not least, in our previous work [9], we presented an efficient approach to code formatting. We introduced an automated mechanism that learns the formatting style of a specific project or file set and identifies deviations from this style. Utilizing a combination of models, the system predicts the likelihood of a token being wrongly positioned and proposes fixes. Expanding on our preceding work [9], our current research introduces the concept of “naturalness” in code formatting, which describes the compliance of a code file with the previously defined desired coding style. We have developed an evaluation system that quantifies this abstract notion through statistical properties and cross-entropy, enabling the assessment of source code formatting and measuring the impact of any formatting changes. The cross-entropy metric captures statistical patterns in code, distinguishing our method from others. We have also improved the performance of the already used LSTM networks [9] in detecting formatting errors by combining them with N-gram language models, resulting in an overall improved performance. We argue that our work significantly contributes to the existing readability research by effectively detecting and correcting multiple formatting errors, through a robust methodology.

Our approach builds on previous work in code readability and formatting, addressing several limitations and introducing novel improvements. Below, we outline the specific differences and unique contributions of our approach compared to existing methods:

- Broader scope of formatting elements: Unlike NATURALIZE and other previous approaches, which focus primarily on local contexts such as whitespace, our approach addresses more general formatting elements, based on the broad tokenization step.
- Adaptability and customization: Our system does not rely on a fixed set of rules like some existing methods, allowing for greater customization to meet the specific needs of different projects and teams.
- Cross-project and cross-language capability: Unlike STYLE-ANALYZER, which is limited to JavaScript, and other similar methods, our approach can be applied to various programming languages, making it more versatile and scalable.
- Comprehensive fixes: Extending beyond identifying formatting errors, our system also proposes fixes, which many previous methods do not.
- Statistical modeling: Our approach incorporates statistical properties and the concept of “naturalness” in code, which is a novel aspect not fully explored in other methods.
- Improved performance: By combining LSTM networks with N-gram language models, our system achieves superior performance in detecting and correcting formatting errors compared to previous methods.

3. Materials and Methods

In this section, the architecture of our system is presented along with the individual subsystems that comprise it, which is illustrated in Figure 1. The subsystems that comprise the system include the data preprocessing, the code formatting evaluation, the formatting error detection, and the formatting error fixing subsystems, which are discussed in the following subsections.

3.1. Data Preprocessing

In the first step of our methodology, the files that will be used to train the system models need to be identified, where the specific formatting standards will be derived from. It should be noted that this is one of the main differentiations of the proposed system with respect to other approaches. Each development team can leverage the formatting system without specific configurations, simply by providing the files that constitute its project (dataset-agnostic). The data used in this work were collected from the database created as part of the research and experiments conducted by Santos et al. [28]. The database contains
a total of 2,322,481 Java files originating from 10,000 open-source repositories on GitHub. Generally, the preprocessing pipeline described in [9] is followed, which will be briefly discussed in the following.

Figure 1. Overview of the system’s architecture.

In the context of the system demonstration, it was considered important to effectively filter the mentioned source code files not only for their syntactical accuracy, but also to guarantee their adherence to widely accepted and well-defined formatting standards. It is worth mentioning that this specific step has been chosen solely for presentation purposes, and in a practical application, any formatting convention may be employed. The formatting rules were formulated using the regular expressions (regex) originally presented in our previous work (https://gist.github.com/karanikiotis/263251dceb86f839a3265cc2306355b2, accessed on 20 April 2024). Table 1 depicts a regular expression of our corpus and a corresponding example of source code, where the formatting error is identified by the expression.

Table 1. A regular expression used to identify a semicolon that appears on the next line.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Source Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\r\n];</td>
<td>System.out.println(&quot;Hello World&quot;)</td>
</tr>
</tbody>
</table>

The rules defined through regex determine the cases where a file contains formatting errors. After discarding any source code files that do not follow the formatting rules defined by the regular expressions, we randomly selected 10,000 files to use in our pipeline, of which 9000 will be used to train our system and the remaining 1000 files will be used to test and tune the models. It should be noted that the final set of 10,000 files used in our approach can be found online (https://zenodo.org/doi/10.5281/zenodo.10978163, accessed on 20 April 2024).

From the final set of files used in our system, each source code file that will be part of the training set must be transformed into an acceptable format to be fed into the system. The process of processing and transforming code files is divided into the tokenization stage and the vectorization stage. The final vocabulary formed during the implementation of the system can also be found in [9].

Special attention is given to characters related to indentation in a code file, such as spaces, parentheses, brackets, line breaks, as they play a significant role in its correct formatting. In order for the initial file to be in a format understandable and processable by the system’s models, each token found in the vocabulary is assigned a unique integer value (vectorization). Table 2 depicts an example of the full process; from the initial code to the vectorization step, as it was initially presented in [9].

Table 2. The tokenization pipeline from the source code to a numeric vector. The vocabulary indexes for this example are “<lit>” = 0, “<space>” = 1, “<word>” = 2, “<equal>” = 3, “<number>” = 4 and “<semicolon>” = 5.

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Tokens</th>
<th>Vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 1;</td>
<td>['&lt;lit&gt;', '&lt;space&gt;', '&lt;word&gt;', '&lt;space&gt;', '&lt;equal&gt;', '&lt;space&gt;', '&lt;number&gt;', '&lt;semicolon&gt;']</td>
<td>[0, 1, 2, 3, 1, 4, 5]</td>
</tr>
</tbody>
</table>
3.2. Formatting Evaluation System

The core idea behind the implementation of the evaluation system was the need for a mechanism that could evaluate the formatting of a source code file, while simultaneously measuring the impact of any formatting changes on the code. The evaluation system was designed based on the existence of statistical properties and patterns in a source code file, the concept of “naturality”, and the ability of language models to capture these properties effectively, as described by Hindle et al. [29]. For statistical language models, N-gram language models were implemented, which calculate the probability of a sequence of tokens based on their frequency in the training set. The evaluation system is designed to prioritize sequences with higher frequencies, thus assigning higher probabilities to sequences of tokens that adhere to formatting standards. Conversely, sequences of tokens representing code files deviating from formatting standards should show lower probabilities.

The proposed evaluation system, given a code file represented by a sequence of tokens $t_1, t_2, \ldots, t_M$ of length $M$ and trained on a training set $C$, can calculate the probability of the occurrence of that specific sequence as follows:

$$ P(t_1, t_2, \ldots, t_M) = \prod_{m=0}^{M} P(t_m|t_{m-1}t_{m-2} \ldots t_{m-n+1}) $$

The calculation of the probability of occurrence through Equation (1) requires a method to compute the conditional probability $P(t_m|t_{m-1}t_{m-2} \ldots t_{m-n+1})$ for different values of $m$. This probability is calculated based on the number of occurrences of the sequence $t_m t_{m-1} t_{m-2} \ldots t_{m-n+1}$ as well as the number of occurrences of the sequence $t_{m-1} t_{m-2} \ldots t_{m-n+1}$ in the training set. One possible method for calculating this probability is as follows:

$$ P(t_m|t_{m-1}t_{m-2} \ldots t_{m-n+1}) = \frac{\text{count}(t_m t_{m-1} t_{m-2} \ldots t_{m-n+1})}{\text{count}(t_{m-1} t_{m-2} \ldots t_{m-n+1})} $$

A significant problem arises in computing the above probability when specific token sequences are absent from the training set $C$, resulting in zero occurrences. To address this issue, the evaluation system was implemented using the Kneser–Ney smoothing technique [30,31], which has been widely employed in the literature [28,29], effectively addressing the problem of zero occurrences. This is particularly useful when the training set comprises source code files.

Since we have the ability to compute the probability, we have used the concept of cross-entropy as a measure of the “surprise” level of the N-gram model [29] when given a sequence of tokens $t_1, t_2, \ldots, t_M$ as input. The cross-entropy metric is expected to increase when calculated for sequences of tokens representing files that deviate from formatting standards (higher “surprise” level) of the training set while decreasing in opposite cases (lower “surprise” level). The cross-entropy of a sequence of tokens $t_1, t_2, \ldots, t_M$ that represent a source code file is calculated using the following formula:

$$ H(t_1 t_2 \ldots t_M) = -\frac{1}{M} \sum_{m=0}^{M} \log P(t_m|t_{m-1}t_{m-2} \ldots t_{m-n+1}) $$

After establishing the logic of the evaluation system, the subsequent step involves defining the primary hyperparameter for the N-gram language model. This critical hyperparameter, denoted as $N$, determines the number of tokens the model considers from recent history to calculate the conditional probability $P(t_m|t_{m-1}t_{m-2} \ldots t_{m-n+1})$. When selecting the order $N$, we partitioned the dataset into a training and a testing set. The files were randomly divided so that 90% comprised the training set, and the remaining 10% constituted the testing set. N-gram models ranging from order 3 to 10 were trained and
evaluated using the testing set. The cross-entropy value was calculated for each file in the set. Figure 2 shows a clear decrease in average cross-entropy as the N-gram model order increases. The reduction rate stabilizes partially after \( N = 8 \). Thus, it can be concluded that an N-gram model of order \( N = 8 \) is better equipped to identify statistical patterns and formatting conventions found in the training set. This is because it assigns higher probabilities (lower cross-entropy) to code files with similar patterns. It is important to note that the search space for the optimal value of parameter \( N \) was not extended beyond \( N = 10 \), because increasing the value of \( N \) exponentially increases the training time of the model, and it was observed that the cross-entropy metric does not significantly decrease beyond \( N = 8 \).

**Figure 2.** Cross-entropy per order N of the N-gram model.

The evaluation system can operate at two different levels, and thus, the cross-entropy is calculated with two different ways.

### 3.2.1. Evaluation of an Entire Source Code File (Source Code Level)

When evaluating an entire source code file, the system follows a straightforward process. Each input code file undergoes preprocessing to represent the file as a sequence of tokens (tokenization). The token sequence is then fed into the 8-gram model, which computes the probability of the specific sequence’s occurrence and, consequently, the cross-entropy as per Equation (3). The cross-entropy score represents the evaluation of the file’s formatting in relation to the training set used for the 8-gram model.

### 3.2.2. Evaluation of Each Token That Constitutes the Source Code File (Token Level)

For evaluating individual tokens composing the code file, the following process is followed:

1. Tokenization of the code file to represent it as a sequence of tokens.
2. Using a rolling window of 20 tokens (as per White et al. [32]), the sequence is divided into individual parts of the total code (code snippet).
3. For each 20-token window, the cross-entropy is computed based on Equation (3).
4. Each token in the sequence is identified within the windows it belongs to. Let the number of windows each token belongs to be denoted as \( M \).
5. The evaluation degree of each token is calculated as following:

   \[
   TokenScore = \sum_{i=1}^{M} \frac{WindowScore_i}{M} \quad (4)
   \]

**Figure 3** illustrates this process. Assuming that the colorized token is the individual token that we would like to evaluate, all the \( M \) 20-token windows that it participates in are taken into account. For each one of them, the cross-entropy score is calculated, and Equation (4) is used to compute the final token score.
3.3. Error Detection System

The main goal of the error detection system is to examine all of the tokens that make up the sequence representing a source code file and detect those that deviate from the acceptable formatting conventions established by the development team or individual programmer. The design of the error detection system was based on modeling a process that approximates the probability of occurrence of a token based on the available information from previous tokens. This process can be represented by the following probability:

\[ P(token | context) \]  \hspace{1cm} (5)

Long Short-Term Memory (LSTM) networks have been used to implement the error detection system. The goal of the LSTM network is to model a process that, given a sequence of tokens (context), predicts the probability of the appearance of the next token. In practice, the LSTM network has been designed to compute the categorical distribution probability of the set of tokens that appear in the vocabulary of the language. Therefore, for each input sequence of tokens provided, the LSTM network outputs a vector representing the probabilities of each token appearing after that specific sequence, as it is depicted in Equation (6).

\[
P(token | context) = \begin{cases} 
P(< \text{word} > | context) \\ 
P(< \text{lit} > | context) \\ 
\ldots \\ 
P(< \text{open curly} > | context) 
\end{cases} \] \hspace{1cm} (6)

The benefit of using the categorical distribution is twofold. On the one hand, we can determine the probabilities of occurrence for each token within the code and thus identify potential formatting errors. On the other hand, it can be used as a means to provide potential corrections for formatting errors. Equation (6) can be modified accordingly to approximate the probability of each token being a formatting error as follows:

\[
P(token_{\text{error}} | context) = 1 - P(token | context) \] \hspace{1cm} (7)

The LSTM network, implemented using Keras, uses the hyperparameters detailed in Table 3. Training involved using a sliding window of 20 tokens, as per White et al. [32], representing past context for predictions. Each source code file was transformed into sequential 20-token windows as input, with the network returning a categorical probability distribution for each window. Based on this distribution and the subsequent token, we derived the error probability using Equation (7). This process yielded the probability of a formatting error for each token in the code file.
Table 3. LSTM hyperparameters.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden Layers</td>
<td>2</td>
</tr>
<tr>
<td>Hidden Units</td>
<td>300</td>
</tr>
<tr>
<td>Activation Function</td>
<td>softmax</td>
</tr>
<tr>
<td>Loss</td>
<td>Categorical Cross Entropy</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Adam</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>0.001</td>
</tr>
<tr>
<td>Batch Size</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 3 illustrates an example, where the LSTM network is applied on the snippet of code “int x = 1;,” and aspires to predict the final semicolon, given the previous tokens as input. The network predicts the tokens `<semicolon>`, `<word>`, `<space>` and `<number>` as possible next tokens, and thus, the error probability of the `<semicolon>` token, which was the actual one in the given code, is $1 - P(<\text{semicolon}> | \text{context})$.

Figure 4 illustrates an example where the LSTM network is applied on the snippet of code “int x = 1;” and aspires to predict the final semicolon, given the previous tokens as input. The network predicts the tokens `<semicolon>`, `<word>`, `<space>` and `<number>` as possible next tokens, and thus, the error probability of the `<semicolon>` token, which was the actual one in the given code, is $1 - P(<\text{semicolon}> | \text{context})$.

In order to improve cases where the LSTM network does not effectively detect potential formatting errors, we use the formatting scores of each token obtained from the formatting score system as follows:

1. Each token is evaluated with an evaluation score as described in Section 3.2.
2. All token evaluation scores are mapped to the interval $[0, 1]$ such that the highest evaluation score corresponds to zero and the lowest to one.
3. We use the token evaluation score to modify the error probability of the LSTM network for each token, using the following equation:

$$\text{modified\_err\_probability = err\_probability} \times \left(1 - \frac{\text{normalized\_score}}{1}ight)$$

The formatting error detection system ranks the error probabilities in descending order and outputs the positions of the tokens corresponding to the top $M$ highest values. The parameter $M$ is variable and can be set by the user, usually with values such as 1, 3, 5, 10. At the same time, the system offers an alternative option to set a specific threshold for the error probabilities, which results in returning only the positions of tokens with an error probability higher than this threshold.

3.4. Error Fixing System

The error fixing system receives two inputs after the detection process: the set of possible positions within the code file that the detection system has identified as formatting errors and the set of tokens that are evaluated as possible corrections based on their probabilities. The detection system computes probability distributions for tokens using a rolling window of 20 tokens. It selects the top $K$ probabilities as potential corrections. With error positions and potential fixes, the fixing system proceeds to correct each error, performing the following correction actions for each potential error position:
1. Deletion of the token corresponding to the error position.
2. Replacement of the token corresponding to the error position with one of the tokens selected as possible fixes.
3. Append a new token belonging to the list of tokens selected as possible fixes, before the token of the potential error position.

In the example depicted in Figure 4, if the error fixing system had been applied, the possible actions that could resolve the “formatting” error of the semicolon are as follows:

- Delete <semicolon>.
- Replace <semicolon> with <word>.
- Replace <semicolon> with <space>.
- Replace <semicolon> with <number>.
- Add <word> before semicolon.
- Add <space> before semicolon.
- Add <number> before semicolon.

Each of the above actions is performed sequentially for each potential error location provided as input to the correction system. After each correction, the system immediately performs a syntactic correctness check on the resulting file to verify that it conforms to the syntax rules of the particular programming language. If the newly generated file passes the syntax check without any errors, it is then evaluated by the formatting evaluation system to assess its formatting quality. Only if the formatted file shows an improvement in score over the original file provided to the system, indicated by a lower cross-entropy value, it is considered as a potential correction suggestion. This criterion requires that the cross-entropy of the corrected file is lower than that of the original file. In addition, the corrected files generated by the fixing system are sorted into ascending order based on their cross-entropy values before being presented to the user.

4. Results

To evaluate the effectiveness of our proposed method in detecting formatting errors and deviations from a desirable coding style, and in providing recommendations for correction, we performed a set of evaluations along several dimensions. Firstly, we tested the error detection capabilities of our system using the CodRep2019 (https://github.com/ASSERT-KTH/codrep-2019, accessed on 20 April 2024) dataset to assess its effectiveness in detecting formatting errors and deviations from common coding styles. Secondly, we evaluated the system’s ability to generate practical and actionable recommendations for fixing the identified errors. Additionally, in an attempt to verify the language-agnostic nature of our approach, we evaluated the performance of our approach in detecting and fixing formatting errors in different programming languages. Finally, we demonstrated the real-world applicability of our approach by deploying it on some real-world scenarios, thereby evaluating its practical utility in everyday development scenarios.

4.1. Error Detection Evaluation

The initial experiment in evaluating the final system focuses on the performance of the error detection system. To evaluate, 1000 files were randomly selected from the CodRep2019 (https://github.com/ASSERT-KTH/codrep-2019, accessed on 20 April 2024) competition dataset. It is important to note that each file contains a unique formatting error, while the position of the error is known in advance. The set of 1000 files was fed to the error detection system to make predictions and detect possible error positions. The value of parameter $M$ for the error detection system was set to $M = 10$, which determines the number of possible error positions that the system will return. The value of $M$ only affects the results provided and represents a typical number of recommendations expected for users.

In Figure 5, the ranking of the actual formatting error position among the top 10 possible positions returned by the error detection system is shown for each of the 1000 files in
the evaluation set. Note that the 0 position denotes files for which the actual formatting error position was not found among the top 10 recommendations of our system. Figure 5 indicates that for approximately 50% of the files in the evaluation set (499 files), the error detection system returns the actual formatting error position ranked first out of the 10 possible positions. Furthermore, the system accurately identifies the error position in the second position for 301 files and in the third position for 94 files. This leads to the conclusion that the error detection system effectively identifies the actual formatting errors within the first three positions of the output for 89.4% of the files in the evaluation set.

![Histogram of Actual Error Positions](image)

**Figure 5.** The histograms of the positions the correct answer was found in.

The precision@K metric was computed for various values of the parameter K, while observing changes in its value with different probability threshold T. The threshold T determines the minimum probability for a proposal to be included in the metric. Figure 6 illustrates the metric for K values of 1, 3, 5, and 10, and T values ranging from 0.75 to 0.95. Across all plots, the metric decreases as T increases, indicating stricter calculation criteria. Conversely, higher K values result in increased metric values. The maximum metric (0.95) occurs at K = 10 and T = 0.75, suggesting the effective identification of formatting errors by the detection system.

Finally, the value of the Mean Reciprocal Rank (MRR), using Equation (9), was calculated and found to be 0.7.

\[
MRR = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{\text{rank}_i}
\]  

(9)

According to this value, the position of the actual formatting error is returned by the system with an average rank equal to:

\[
\text{MeanRank} = \frac{1}{MRR} = 1.43
\]  

(10)

On average, the system locates and returns the actual formatting error between the first and second position. This observation confirms the results shown in the above figures. It is crucial to note the strictness characterizing the metric of MRR. Its computation method results in small values of the metric, even in cases where incorrect predictions are minimal. In the overall evaluation of the error detection system, the MRR metric was utilized, and it should be considered in conjunction with the precision@K metric as well as the histogram of Figure 5.
4.2. Error Fixing Evaluation

In the second experiment, we evaluated the error correction system for formatting errors. We randomly selected 100 files from the CodRep2019 dataset, each containing a formatting error. These files were processed by the system to identify errors and propose fixes. After completion, we manually verified the accuracy of the fixes proposed by our system. It is important to note that the system offers multiple potential corrections per file, assuming the resulting file is syntactically correct and has a better evaluation compared to the original. Figure 7 illustrates the total number of cases where the system successfully provided a fix among its suggestions and where it failed to do so. Results demonstrate that in the vast majority of cases (96%), the system correctly returned the fixed file. However, in a minority of cases (4%), the system failed to provide a correct fix.

Figure 6. Precision@K metric for various values of parameter K and thresholds.

Figure 7. Fixed files.

In the second experiment, we evaluated the performance of the error correction system when handling files with multiple formatting errors. Twenty files were randomly selected from the CodRep2019 competition dataset, each of which intentionally contained an additional formatting error introduced in a variety of locations, including cases where errors occurred alongside the original errors. This diversity in error placement allowed us to investigate whether the correction of one error could affect the correction of the second. The experiment proceeded as follows:
1. Each file in the dataset is provided as input to the system.
2. The system returns as potential fixes only the ones that are syntactically correct and have a difference in the evaluation score greater than 5%.
3. From the list of potential fixed files returned by the system, the file with the highest formatting evaluation score is selected.
4. The file is provided again to the system as input.
5. We examine whether the error positions have been corrected in the file.

The histograms in Figures 8 and 9 present the distribution of the number of fixes for the first and second fixing iterations, respectively. In the first correction iteration, it is particularly interesting to note that whenever the system provided two corrected file suggestions, they both contained the correct fixes for the errors present in each respective file. In the second correction iteration, it was observed that in most cases, the system suggested a single fixed file containing the correct fix. In all 20 cases used for this experiment, the system provided a final output file that incorporated corrected formatting errors after completing two fixing iterations.

![Figure 8](image_url)  
**Figure 8.** The first iteration of the fixing mechanism.

![Figure 9](image_url)  
**Figure 9.** The second iteration of the fixing mechanism.
4.3. Language Agnostic Evaluation

To verify the language-agnostic nature of our formatting detection and fixing mechanism, we conducted evaluations across multiple programming languages, specifically targeting JavaScript (nodejs version 20.13.1) and Python (version 3.12.3). We obtained JavaScript and Python files at random from GitHub (https://github.com, accessed on 20 April 2024) and subjected them to linting using language-specific linters, specifically ESLint (version 9.3.0) (https://eslint.org, accessed on 20 April 2024) for JavaScript and Pylint (version 3.1.0) (https://pylint.org, accessed on 20 April 2024) for Python. These linters were configured to identify only formatting-related rules, such as “trailing-newlines”, which identifies trailing blank lines in a file, and “curly”, which enforces a consistent brace style for all control statements, in Pylint and ESLint, respectively.

The files with no linting errors formed the training set, while those with at least one linting error formed the test set. We collected 500 test files for JavaScript and 500 test files for Python to ensure a diverse representation of formatting errors.

Given the language-agnostic nature of our approach, we only needed to replace the tokenizer in the input phase of our system. For JavaScript files, we used the “js-tokens” (version 9.0.0) (https://github.com/lydell/js-tokens, accessed on 20 April 2024) library for tokenization, while for Python files we developed a custom tokenizer, similar to the one created for Java, tailored to Python syntax. Using these tokenizers, we converted both the training and test files into token sequences that served as input to our system.

Separate models were trained for JavaScript and Python code using the methodology described in the Section 3. The trained models were then applied to the test files in order to detect and correct formatting errors previously identified by the linters.

In Figure 10, we show the positioning of the actual formatting errors among the top 10 recommendations provided by our error detection system for the 1000 files in the evaluation set, in contrast to the evaluation made for Java files in Figure 5. A label of 0 indicates instances where the actual error was not in the top 10 suggestions. Our analysis shows that in the more than half of the files evaluated, the error detection system accurately ranks the actual formatting error as the top recommendation. Furthermore, in many cases, the system correctly identifies the error within the second position. These results indicate that our error detection system effectively identifies the actual formatting errors within the top two positions of the output for the majority of files evaluated.

Table 4 presents the prediction metrics for formatting error detection, focusing specifically on the accuracy of our system in correctly identifying formatting errors, using the
Mean Reciprocal Rank (as it was originally presented in Section 4.1 and Equation (9)). It also highlights the Precision@1 and Precision@3 metrics for a threshold of 0.75, providing insight into the system’s performance at different levels of precision.

Table 4. Language agnostic evaluation metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Reciprocal Rank (MRR)</td>
<td>0.74</td>
</tr>
<tr>
<td>Mean Rank of Error</td>
<td>1.35</td>
</tr>
<tr>
<td>Precision@1</td>
<td>53%</td>
</tr>
<tr>
<td>Precision@3</td>
<td>92%</td>
</tr>
</tbody>
</table>

Figure 11 shows the total number of cases in which the system successfully provided a fix among its suggestions and in which it failed to do so. From the results, it is clear that the system can successfully propose correct fixes for the formatting errors identified and only fails to do so in a minority of cases, around 9%.

From the evaluation presented in this section, we can conclude that the system performs very well across different programming languages. The performance metrics, including mean reciprocal rank and precision@K, indicate that the system’s effectiveness in detecting and correcting formatting errors remains consistent across different languages. The results of the evaluation in JavaScript and Python, which showed similar levels of performance to those initially shown for Java, provide strong support for the language-agnostic nature of our approach.

4.4. Real Case Scenarios of Error Fixing

To assess the practical applicability of our system in providing actionable recommendations to developers and resolving formatting errors, we conducted experiments on specific use cases. We randomly selected small Java files from popular GitHub repositories and applied our methodology. Figure 12 presents an example where the initial source code of the file contains a formatting error involving the removal of a newline character (depicted in the figure). Our methodology prioritizes the formatting error as the primary issue, and the fixing mechanism generates a well-organized list of recommendations. The initial suggestion offered by our approach involves adding the desired newline character.
package com.developmentontheedge.sql.model;

public class AstNullPredicate extends SimpleNode
{
    public AstNullPredicate(int id)
    {
        super(id);
        this.nodeSuffix = isNull;
        Token isToken, notToken, nullToken;
        public SimpleNode getPredicand() {
            return child(0);
        }
    }

    After
    public class AstNullPredicate extends SimpleNode
    {
        public AstNullPredicate(int id) {
            super(id);
            this.nodeSuffix = isNull;
            Token isToken, notToken, nullToken;
            public SimpleNode getPredicand() {
                return child(0);
            }
        }
    }

Figure 12. The initial and final version of the file to be examined. The blue arrow highlights the formatting error in the initial version of the file and, specifically, the removal of a newline character.

Although these examples may seem minor and the corrections insignificant, they can be of significant importance in extensive projects with diverse developers employing various coding styles. Identifying and addressing these formatting errors has the potential to significantly improve the readability and comprehension of the code.

5. Discussion

Our study presents an automated system designed to improve code readability through formatting improvements. The results demonstrate the system’s effectiveness in detecting and correcting formatting errors, as well as its potential practical utility in real-world development scenarios. We discuss the broader implications of these results and highlight the limitations of our work. The effectiveness of the system in detecting and cor-
recting errors is consistent with the importance of code formatting in improving software maintainability and developer productivity. By accurately identifying formatting errors and providing actionable recommendations, our system contributes to ongoing efforts to automate and simplify the code review process, ultimately facilitating code readability and comprehension.

In addition, the ability of our methodology to adapt to different datasets and formatting styles significantly enhances its usefulness in different development environments. Our evaluation underscores the language-agnostic design of our system. This design facilitates the ability to easily adapt for potential use in a range of programming languages, with only minimal customization required.

When considering the performance and computational requirements of our proposed system, it is important to consider these aspects extensively. The computational intensity lies primarily in the training phase of the underlying models, particularly the LSTM and N-gram models. However, once trained, using these models to detect and correct formatting errors becomes a trivial task and does not impose excessive computational load. Consequently, the use of our system, for example, integrating into popular Integrated Development Environments (IDEs) for real-time feedback during code writing, becomes feasible without significant performance overhead. It is worth noting that while the training phase may require computational resources, it is a one-time process and can be performed on standard hardware configurations, making it accessible to most developers.

Despite the promising results, our study is not without limitations. One potential challenge is the scalability of our approach to projects with thousands of files or different formatting styles; while our evaluation demonstrated effectiveness in handling single files and large projects, further research is needed to ensure seamless integration and performance in even larger software development environments. However, it is worth noting that our system’s ability to handle legacy code with established but outdated formatting standards is not a significant concern. Our system is designed to adapt to any formatting standard, even outdated ones, if specified by the user. In addition, the impact of adopting the system in terms of a learning curve and integration into existing development processes is minimal. As our system works in a completely unsupervised manner, it requires minimal intervention from developers, who only need to define the files from which the desired formatting is to be extracted. Therefore, the learning curve for adopting the system is low and integration into existing workflows is straightforward.

In conclusion, our study highlights the significant progress that has been made in using automated formatting systems to improve code readability and maintainability. By demonstrating the effectiveness of our approach and its practical utility in real-world development scenarios, we contribute to the ongoing evolution of automated tools in software development practices. Through our efforts, we are improving the quality and efficiency of software development processes, and enabling improved code comprehension and more efficient code review procedures.

6. Conclusions

In this paper, we extended our previous work [9], presenting an automated system for enhancing code readability through preserving a common formatting style. Utilizing LSTM networks and N-gram models, our approach effectively adapts to various coding styles, demonstrating significant potential for maintaining consistency in code styling. Establishing the concept of “naturalness”, through calculating the cross-entropy of a sequence of tokens, we were able to evaluate the source code formatting and its compliance with the previously defined desired coding style. The experiments conducted using a large dataset of Java files highlighted our system’s ability in detecting and correcting formatting errors, emphasizing its utility in software development for improving code readability and maintainability.

Our research highlights the important role of an automated mechanism in maintaining high standards of code quality, especially in large projects where maintaining formatting
consistency manually is challenging. The capability of our system to adjust to different scenarios with little need for in-depth domain knowledge or extensive manual configuration indicates a significant advancement in automated code formatting.

Future work on our system includes several key areas for exploration and improvement. The integration of our system with popular Integrated Development Environments (IDEs) stands out as a crucial step towards improving usability and accessibility. By seamlessly integrating with developers’ existing workflows, our system could provide real-time formatting suggestions and corrections, facilitating a smoother coding experience and encouraging adherence to desired formatting standards.

In addition, extending the evaluation of our approach to a diverse set of projects, particularly those of significant size and complexity, is essential to gain deeper insights into its performance and adaptability. Specifically, we want to examine projects that have a wide range of formatting styles and involve a large number of developers. By evaluating our system in such environments, we can assess its ability to effectively adapt to different formatting approaches within extensive codebases, thereby enhancing its practical utility and robustness. Furthermore, while the language-agnostic nature of our approach has already been tested and verified, a more extensive evaluation could be conducted to assess the performance of our system in multiple and diverse environments.

Furthermore, the consideration of alternative technologies beyond LSTM and N-gram models represents an interesting direction for future research. Exploring technologies such as transformer-based models, graph neural networks (GNNs), attention mechanisms, and reinforcement learning approaches could provide new insights and potentially improve the performance of our system. These alternative technologies may provide unique capabilities for analyzing and understanding code, complementing the strengths of LSTM and N-gram models.

Finally, conducting a survey of real developers is an important step in assessing the practical applicability and user satisfaction of our mechanism in real development scenarios. By gathering feedback and insights from developers who use our system in their daily work, we can gain valuable perspectives on its effectiveness, usability, and areas for improvement.

Author Contributions: Conceptualization, T.K. (Thomas Kanoutas), T.K. (Thomas Karanikiotis) and A.L.S.; methodology, T.K. (Thomas Kanoutas), T.K. (Thomas Karanikiotis) and A.L.S.; software, T.K. (Thomas Kanoutas) and T.K. (Thomas Karanikiotis); validation, T.K. (Thomas Kanoutas), T.K. (Thomas Karanikiotis) and A.L.S.; formal analysis, T.K. (Thomas Kanoutas), T.K. (Thomas Karanikiotis) and A.L.S.; investigation, T.K. (Thomas Kanoutas) and T.K. (Thomas Karanikiotis); resources, T.K. (Thomas Kanoutas), T.K. (Thomas Karanikiotis) and A.L.S.; data curation, T.K. (Thomas Kanoutas) and T.K. (Thomas Karanikiotis); writing—original draft preparation, T.K. (Thomas Kanoutas) and T.K. (Thomas Karanikiotis); writing—review and editing, A.L.S.; visualization, T.K. (Thomas Kanoutas), T.K. (Thomas Karanikiotis) and A.L.S.; supervision, A.L.S.; project administration, A.L.S.; funding acquisition, A.L.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original data presented in the study and used for the training of our system are openly available in Zenodo at https://zenodo.org/doi/10.5281/zenodo.10978163 (accessed on 20 April 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References


**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.