Towards Automatically Generating a Personalized Code Formatting Mechanism

Thomas Karanikiotis[a], Kyriakos C. Chatzidimitriou[b] and Andreas L. Symeonidis[c]
Dept. of Electrical and Computer Eng., Aristotle University of Thessaloniki, Thessaloniki, Greece

Keywords: Source Code Formatting, Code Style, Source Code Readability, LSTM, SVM One-Class.

Abstract: Source code readability and comprehensibility are continuously gaining interest, due to the wide adoption of component-based software development and the (re)use of software residing in code hosting platforms. Consistent code styling and formatting across a project tend to improve readability, while most code formatting approaches rely on a set of rules defined by experts, that aspire to model a commonly accepted formatting. This approach is usually based on the experts' expertise, is time-consuming and does not take into account the way a team develops software. Thus, it becomes too intrusive and, in many cases, is not adopted. In this work we present an automated mechanism, that, given a set of source code files, can be trained to recognize the formatting style used across a project and identify deviations, in a completely unsupervised manner. At first, source code is transformed into small meaningful pieces, called tokens, which are used to train the models of our mechanism, in order to predict the probability of a token being wrongly positioned. Preliminary evaluation on various axes indicates that our approach can effectively detect formatting deviations from the project's code styling and provide actionable recommendations to the developer.

1 INTRODUCTION

Source code readability has recently gained much research interest and is considered of vital importance for developers, especially those working under a component-based software engineering scheme. It is a quite complex concept and includes factors such as understanding of the control flow, the functionality and the purpose of a given software component. At the same time, source code readability is highly related to maintainability and reusability, pillar aspects of software quality.

In this context, the importance of readability is obvious and a number of recent approaches aspire to quantify readability of a source code component (Scalabrino et al., 2018; Scalabrino et al., 2016; Posnett et al., 2011); still, the selection of metrics that properly quantify readability is under heavy debate. What has been proven, though, is that the selection of a suitable code styling and correct formatting approach can improve source code readability and can enhance the capability of the developers to comprehend the content, the functionality and the intention of the source code (Tysell Sundkvist and Persson, 2017); on the contrary, different coding styles affect readability (Lee et al., 2013). At the same time, studies have shown that the indentation applied on source code, which is a part of code formatting, directly affects comprehensibility (Kesler et al., 1984).

There have been several efforts that mainly aspire to identify styling errors and deviations from a priorly accepted set of formatting rules and, possibly, provide styling fixes (Loriot et al., 2019; Markovtsev et al., 2019; Prabhu et al., 2017). The majority of the related approaches make use of a predefined set of "globally" accepted formatting rules and, based on these rules, try to identify pieces of code that diverge from them. These rules can only be turned on/off and the developers are not able to alter any of them, or add their own. In addition, some approaches (Allamanis et al., 2014), aspire to identify alterations on identifier or method names, sequence of function calls or code structure that could make the code more interpretable, focusing on program comprehension, rather than on readability from the code styling perspective.

The majority of the aforementioned approaches aspire to provide a mechanism that models a

[a] https://orcid.org/0000-0001-6117-8222
[b] https://orcid.org/0000-0003-0715-1197
[c] https://orcid.org/0000-0003-0235-6046

Karanikiotis, T., Chatzidimitriou, K. and Symeonidis, A.
Towards Automatically Generating a Personalized Code Formatting Mechanism.
DOI: 10.5220/0010579900900101
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 90-101
ISBN: 978-989-758-523-4

commonly accepted code styling and can help developers apply it in practice. However, as teams vary in skills and have different needs, not all styling guidelines can apply to each one of them. At the same time, the process of properly configuring the set of formatting rules to match the needs of a specific team can be a quite time consuming and complex task, especially when there is a large group of developers that participate in a team. Thus, an unsupervised system that could model the desired formatting style of a team is needed, which, based on an existing project, could identify any styling deviations from it.

In this work, we present an automated mechanism that can learn the code style used in a project or a repository in a completely unsupervised manner. This mechanism can be used by single developers or a team of developers with minimal prior project knowledge, in order to detect and highlight the points within a source code that deviate from the global styling applied in the whole project. Using this tool, a team of developers could keep a common formatting across the whole project, making it easier for them to maintain or reuse certain pieces of code, or cooperate. At the same time, with the appropriate selection of the data that will be used for training, our approach can act like a common formatter, which learns globally accepted styling rules with no supervision and detects deviations from them.

The rest of this paper is organized as follows. Section II reviews the recent approaches on source code formatting and discusses how our work differentiates and further extends them. Our methodology, as well as the models we have developed and the data used to train the models are depicted in Section III, while in Section IV we evaluate the performance of our approach in the detection of formatting errors. Finally, in Section V we analyze potential threats to our internal and external validity, while in Section VI we discuss the conclusions of our approach and provide insights for further research.

2 RELATED WORK

The importance of source code readability and comprehension has gained increasing interest during the recent years, especially in the cases where the software development process involves large teams of developers or the component-based software development paradigm is followed. Such cases require projects to be processed quickly and maintain a standard level of quality and readability is considered a success or fail factor. Additionally, in the context of software quality, readability is one of the software attributes that is closely linked to maintainability, which, as the importance of correct and evolving code is given, has gained the attention of research approaches in the recent years.

While software readability and code comprehension have abstract meanings and their quantification is still to be clarified, the code styling and formatting approach used across the source code can unquestionably ease the process of reading and understanding the functionality, the intentions and the content of a given source code. Code indentation, one of the most important attributes of code formatting, has been proven to change the way a developer perceives the content of the source code, separates the various blocks of code that may appear and apprehends their purpose. Persson and Sundkvist (Tysell Sundkvist and Persson, 2017) argue that the readability and interpretation of source code can be improved by the use of correct indentation, providing faster understanding of the code purpose and functionality as the source code size increases. Hindle et al. (Hindle et al., 2008) conducted a study about the relationship between the shape that is drawn from the code indentation and the code block's syntactic structure. The study concluded that there is a high correlation between the indentation shape and the code structure, which can help the developers better perceive the content of the given code.

Prabhu et al. (Prabhu et al., 2017), on the other hand, focus on building an editor that can separate the content of the source code from the presentation (e.g. styling) and provide features such as auto-format coding, including auto-indentation and auto-spacing. However, the auto-formatting features are strictly based on algorithms developed by the authors, that aspire to follow a global styling pattern and do not allow alterations. Additionally, Wang et al. (Wang et al., 2011) built a heuristic method in order to segment a given Java method into meaningful blocks that implement different functionalities, trying to ease the developer's task to comprehend the code functionality. At the same time, there are a lot of tools, like *Indent* (GNU Project, 2007) and *Prettier* (Prettier, 2017), that, based on a set of expert-defined rules, aspire to detect styling mismatches and pieces of code that diverge from the "ground truth".

While there are a lot of approaches that aspire to detect code formatting errors based on a set of predefined rules and, possibly, provide fixes (Loriot

et al., 2019), developers are not able to edit these rules or add their own, in order to build a custom code styling that best matches their needs. In large teams of developers, maintaining a common formatting, that is also configured according to their requirements, is a crucial factor towards quality code. The work of Kesler et al. (Kesler et al., 1984) supports this argument; they ran an experiment on the effects of no indentation, excessive indentation and a moderate indentation on the comprehensibility of a program. The results of the experiment depicted that, while the moderate indentation seems to yield the best results, one can conclude that there is no perfect indentation style and it should be chosen carefully at each time. Miara et al. (Miara et al., 1983) conducted a study for the most popular and most used indentations, which concluded that multiple and different indentation styles may be found across programs and the level of indentation could be a crucial factor for code comprehensibility. Therefore, there is a need for models that could identify the code formatting that is used across the same project and detect deviations from it.

Towards dynamic adaptation and homogenization of code styling, Allamanis et al. (Allamanis et al., 2014) carried out one of the first approaches towards learning the styling used by one or more developers in a single project and, then, detecting and identifying variations from it. The authors built a framework, called *NATURALIZE*, that could suggest identifier names and styling changes in the given source code, in order to increase styling consistency across the files of a project. While NATURALIZE was proven effective for providing accurate suggestions, it could process only local context and could not incorporate semantically valid suggestions, while it was mainly focused on the use of indentation and whitespaces and not on other aspects of code formatting (e.g. the placement of comments within the code). Furthermore, Parr et al. (Parr and Vinju, 2016) proposed *CODEBUFF*, a code formatter based on machine learning, that could automatically create universal code formatters built from the grammar of any given language. CODEBUFF achieved quite good results, but it was based on a complex model trained by trial and error, with no generalization capabilities. Moreover, it could not handle some ordinary cases, such as mixed indentation with tabs and spaces or mixed quotes with single and double quotes. Towards avoiding these limitations, Markovtsev et al. (Markovtsev et al., 2019) created a tool, called *STYLE-ANALYZER*, which can mine the formatting style of a given Git repository, identify style inconsistencies and propose fixes. Even though STYLE-ANALYZER can achieve pretty good results at modelling the code styling of the respective project, the proposed model is quite complex and time-consuming and is targeted only on javascript source code. Finally, Ogura et al. (Ogura et al., 2018) proposed *STYLECOORDINATOR*, a tool that can help developers maintain a consistent code styling across their project and, also, use their own local formatting style. StyleCoordinator could process the code of a repository and produce a styling configuration stored in a file. This configuration file would then be used to provide consistency in every new or modified file in the user's repository. However, StyleCoordinator is initially based on a common convention configuration, in order to ensure consistency, and is not able to extract the code styling selected by the user from the ground, while its efficiency is yet to be clarified.

In this work, we aspire to confront the main limitations introduced in the aforementioned related approaches. We propose a generalizable model, which can dynamically learn the formatting style applied across a project or set of files and, then, identify and highlight any deviations from it. Using our approach, single developers or teams of developers are able to feed their existing source code files to indicate the desired formatting and then use the generated model to format future code in the same styling. This way they can save time in understanding their source code, while helping their team keep a uniform way of developing software. Our approach requires no specific domain knowledge or even rules customisation, which most of the recent linters and style checkers need.

3 METHODOLOGY

In this section we design our formatting error detection system (shown in Figure 1) based on two approaches that aspire to model the formatting of a given source code from different aspects: the generative model and the outlier detection model.

3.1 User Dataset

In the first step towards creating our system, the set of source code files that will determine the selected formatting style needs to be defined. This is one of the main points where our system differentiates from other similar approaches. Instead of using a set of predefined formatting rules to determine styling deviations, we allow every developer or team of developers to use their own source code files to

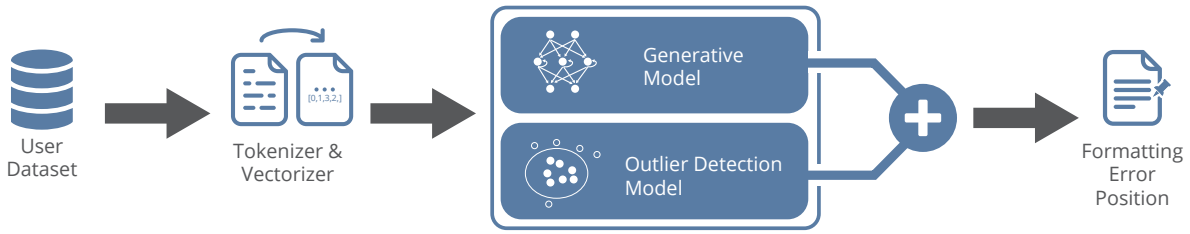

Figure 1: Overview of the Formatting Error Detection Methodology.

define the desired code styling. The rest of our modelling procedure simply adapts to the provided dataset and, thus, our system is dataset agnostic.

In order to showcase the performance of our approach on frequently encountered formatting errors that have been found across projects, we trained our formatting error detection mechanism on a code writing style that is widely used by developers. Specifically, we made use of the dataset used by Santos et al. (Santos et al., 2018). The authors mined the top 10,000 Java repositories and extracted the latest snapshot of the default branch, keeping only the syntactically-valid Java files. In total 2,322,481 Java files were collected.

While these files are syntactically valid and depict the formatting style used by most of the developers, we also want to ensure that they do not diverge from widely known formatting rules, in order to evaluate the ability of our mechanism to identify also commonly found styling deviations. Thus, a set of rules has been defined manually, which describes the occasions when a formatting error occurs. We created a set of 22 regular expressions that are able to identify the corresponding formatting errors that occur in a single Java file. 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. The complete corpus of our 22 regular expressions can be found on our page[1]. It should be noted once again that the selection of widely known formatting errors is used only for showcasing and does not affect the adaptive nature of our approach, while each developer could train the system with his/her own specific code styling guidelines, just by providing a set of source code files.

Using the set of the 22 regular expressions we defined, we were able to identify 10,000 Java files from the dataset used by Santos et al., that did not contain any formatting deviations from the commonly accepted standards and, thus, could be used as the basis for the training of our models.

[1]https://gist.github.com/karanikiotis/263251decb86f839a3265cc2306355b2

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

Regular Expression	”[\r\n];”
Source Code Example	System.out.println(”Hello World”) ;

3.2 Tokenizer & Vectorizer

Before the source code is further processed by our models, it needs to be transformed into a suitable form. This procedure is called *tokenization* and transforms the source code into a set of tokens. Each programming language consists of a list of all possible unique tokens, which is called *vocabulary*, and contains all the possible keywords and operators used by the language. The source code contains also a set of out-of-vocabulary tokens, which are the variable names, string literals and numbers used by the developer. These tokens are usually projected into an abstract form and they are represented by the respective token that indicates the corresponding category the token belongs to (variable, string or number).

In this work, our main target was token differentiation regarding the way they are placed between the rest of the tokens of the source code. The tokenizer identifies and abstracts the set of variables, strings and numbers used by the developer, detects the set of keywords and groups tokens with similar formatting behaviors and returns the set of tokens identified, as well as the number of characters each token occupies in the initial source code. Table 2 depicts some example of keywords identified in the initial source code by the tokenizer and the corresponding token they are transformed into.

Table 2: Examples of keywords and their respective tokens.

Token Name	Token Symbol	Keywords
KEYWORD	<keyword>	*break, for, if, return, ...*
LIT	<lit>	*float, int, void, ...*
LITERAL	<literal>	*true, false, null*
NUMBER	<number>	*123, 5.2, 10, 1, 0, ...*
STRING	<word>	*”a”, ”hello”, ...*

Using the tokenizer, each token from the initial source code is categorized into the appropriate token category. For example, whenever any of the words *true*, *false* or *null* are identified in the source code, they are transformed and treated with the token *LITERAL*. Special attention was required in the correct tokenization of indentation characters (e.g. whitespaces and tabs), brackets and special symbols, such as semicolons, which are vital for an appropriate code styling.

In order to convert source code into a form that is suitable for training our models, a two step procedure is followed. First, the source code is tokenized using the aforementioned tokenizer and, thus, a set of tokens is returned. Subsequently, this set of tokens is processed to extract the total vocabulary of tokens used. Each token is then assigned a positive integer index, that will then be used as the input in the following models. Table 3 depicts an example of a full transformation; the initial source code is transformed to a numeric vector that can be treated by our models.

Table 3: 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*.

Source Code	*int x = 1;*
Tokens	*['<lit>', '<space>', '<word>', '<space>','<equal>', '<space>', '<number>', '<semicolon>']*
Tokens Lengths	*[3, 1, 1, 1, 1, 1, 1, 1]*
Vectorization	*[0, 1, 2, 1, 3, 1, 4, 5]*

3.3 Model Generation

Given the sequence of tokens that appear in the training corpus and that constitute the formatting style that will be followed, we trained two different models that aspire to detect a formatting error. The main goal of our modelling approach is to approximate a function that, given a set of tokens, determines the possibility of a token being wrongly positioned among the others, i.e. a formatting error. In order to do so, each token coming from the source code needs to be assigned a likelihood of being a formatting error, as shown in the following equation:

$$P(formatting\_error\_token|context) \quad (1)$$

Each of the two models used in our approach, aspires to approximate the aforementioned probability from its own perspective for every token that appears in the source code. The final probabilities are then calculated by aggregating the outputs of the respective models.

3.3.1 Generative Model

Given a series of tokens that have already been identified in the source code, the generative model predicts the next token that will be found, assigning a probability at each possible token from the vocabulary. The technology employed to accomplish this goal is a *long short-term memory (LSTM)* (Hochreiter and Schmidhuber, 1997) recurrent neural network. LSTM neural networks are an extension of RNNs (recurrent neural networks), that resolve the vanishing gradient problem and, thus, can memorize past data easier. LSTMs have been proven really effective in processing source code and predicting the next tokens in a sequence of previous ones (Hellendoorn and Devanbu, 2017).

Our modelling goal using the LSTMs is to approximate a function that, given the previous $n-1$ tokens that have been identified in the series of source code tokens, approximates the probability of the next token to be found. LSTMs return an array of probabilities that depict the likelihood of the next token to be the respective one. It can be also considered as a categorical distribution of the probability across the vocabulary of all possible tokens. The following equation depicts the categorical distribution, i.e. the vector of probabilities given by the LSTMs:

$$P(next\_token|context) = \begin{cases} P(<word>|context) \\ P(<space>|context) \\ P(<number>|context) \\ ... \end{cases} \quad (2)$$

Figure 2 illustrates the way the LSTM predicts the token *<semicolon>* given the previous tokens from the source code *"int x = 1;"*.

As already mentioned, the LSTM neural network calculates a vector of probabilities for each token from the vocabulary of the possible ones, which depict the likelihood of each token to be the next one to be identified in the series of tokens. By inverting this probability, we can approximate the probability of identifying any other token, except from the respective one, which can also be considered as the probability of the respective token being wrongly placed in the specific position in the source code. The following equation depicts the probability of the token *<tok>* being wrongly positioned, i.e. the probabilities of every other possible token:

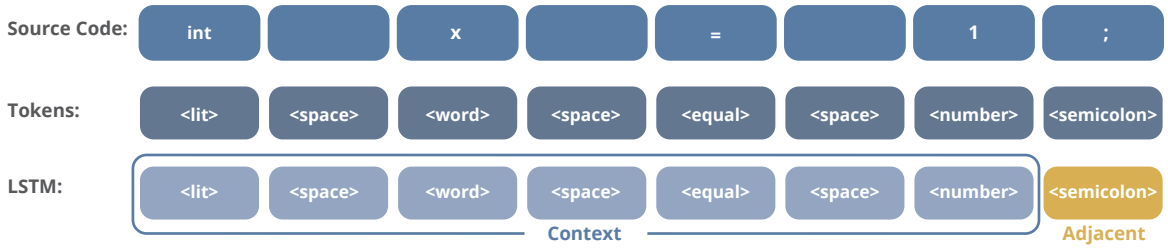

Figure 2: The LSTM prediction of the adjacent token, given the source code *"int x = 1;"*.

$$P(<\text{tok}> \text{ wrong}|context) = 1 - P(next\_tok = < tok > |context) \quad (3)$$

where $<next\_tok>$ is the next token in the sequence of the previously identified *context*.

We feed-forward the LSTM architecture with the tokens coming from the source code that needs to be checked for formatting errors. It should be noted that, in order for the first n tokens of the source code to be checked also by the LSTMs, we manually added a set of n starting tokens, so the first token to be predicted from the architecture is the actual first token of the code. For the creation of the LSTM neural network we made use of the Keras [2] deep neural network framework, with two layers of 400 LSTM nodes each, parameters that were selected upon testing. The sliding window that was applied to the tokens of the source code in order to create the input vector was chosen to have a context length of 20 tokens, as it was proven to be effective on source code (White et al., 2015); A window of 20 tokens is selected at each time-step and is given as input to the LSTM network. The model outputs the probability of each possible token to be the next one in the series. We compare these probabilities with the actual next token and, using equation 3, we transform this probability into an error probability.

3.3.2 Outlier Detection Model

The problem of identifying a piece of source code that diverges from the common formatting style can also be seen from a different perspective, which is the classification approach using n-grams. N-grams are a set of n continuous tokens from the given source code. Figure 3 illustrates the procedure of tokenizing the source code "*int x = 1;*" and splitting the generated tokens into different n-grams, with $n = 1$, $n = 2$ and $n = 3$.

By transforming the source code into a set of n-grams, we can approach the formatting error detection problem from the outlier detection perspective. Using this perspective, we aspire to detect n-grams in the source code that is examined for formatting errors, which have not been previously met in the training corpus, e.g. the set of files that define the developer's coding style. Indeed, if the most of the n-grams a specific token participates in are classified as outliers, then, probably, this particular token has not been previously used in this way and constitutes a formatting error.

The technology we employed for the outlier detection was a *Support Vector Machine - SVM One-Class* algorithm, which has been proven successful in outlier detection problems (Seo, 2007). The SVM One-Class model is trained using only data coming from the original (or "positive") class, this way identifying new data that deviate from the "normal" behavior. In our approach, the model is trained on n-grams coming from the training corpus, which define the developer's preferable code styling, and is then used to detect n-grams that diverge.

As multiple and previously unknown n-grams may be found during the prediction stage of the SVM One-Class, the model has to be flexible and adaptive. However, the tuning of the SVM parameters, which are the *nu* and *gamma* parameters, tends to confine the model. In order to overcome this limitation, we applied the following approach: instead of simply using one single SVM One-Class model, with a certain set of *nu* and *gamma* parameters, we created a set of SVM models with various *nu-gamma* pairs, that aspire to cover a large area of the fine-tuning procedure. Each model is trained separately and, then, all the predictions are aggregated, leading to the final prediction.

As our primary modelling target is the prediction of the probability of a single token being a formatting error, we cannot make use of the classification of an n-gram into the original or the outliers class. Instead, we take into account the prediction probabilities that are produced by each model. The n-gram is feeded into all the different SVM One-Class models, which return their prediction probabilities of the n-gram being an outlier, i.e. the n-gram does not have the same behavior with the ones met in the training

[2] https://github.com/keras-team/keras

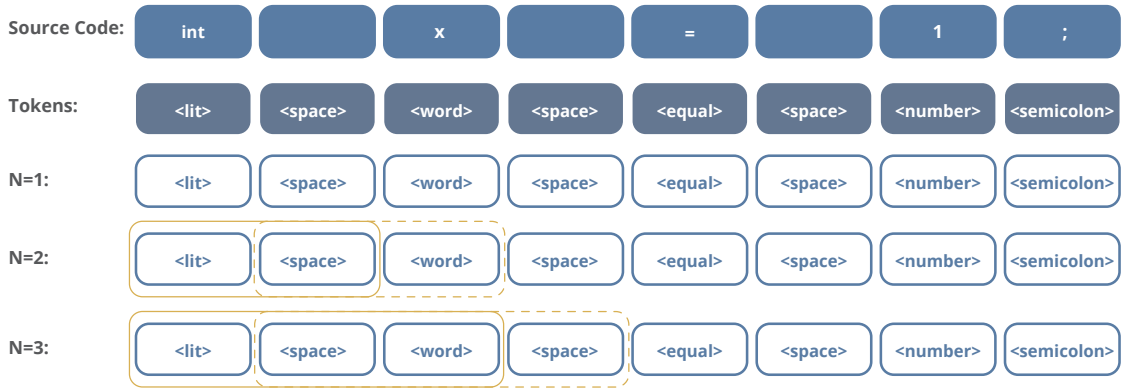

Figure 3: The n-grams that are produced from the source code *"int x = 1;"* for $n = 1$, $n = 2$ and $n = 3$. For $n = 1$ the unigrams are "<lit>", "<space>", etc., for $n = 2$ the bigrams are "<lit> <space>", "<space> <word>", etc. and for $n = 3$ the trigrams are "<lit> <space> <word>", "<space> <word> <space>", etc.

corpus. Subsequently, the predictions of all SVM models are being aggregated and the final prediction for the n-gram is made.

Finally, each token coming from the source code is assigned a probability of being a formatting error as the mean probability of all the n-grams it participates. Equation 4 depicts the calculation of the formatting error probability for the token *<tok>*, where *n-grams* is the set of all n-grams the token participates in, *SVM_models* is the set of all the SVM models we used and *context* is the previously identified context.

$$P(\text{<tok> being wrong}|context) = \sum_{n-grams} \sum_{SVM\_models} P_{model}(\text{n-gram is outlier}) \quad (4)$$

3.3.3 Final Pipeline

The final pipeline of our approach is the merging of the two models, as illustrated in Figure 1. Overall, the source code that needs to be checked for formatting errors is firstly tokenized and then is forwarded into the two selected models, the LSTM and the SVM One-Class. For each token found in the initial source code, the models output their probabilities of the token being wrongful present in the specific position. In the aggregation stage, the two predictions are combined to form the final prediction of the system. By averaging the probabilities calculated from each model, we were able to fix some ambiguous decisions made by a single model, i.e. a token probably misclassified as formatting error but with low prediction probability from the one model, would be correctly classified with a high probability from the second one. Finally, the tokens along with their respective probabilities are being sorted to create a descending order of tokens possibly being formatting errors.

4 EVALUATION

In this section we evaluate our methodology for detecting formatting errors and deviations from a globally used code styling. At first, in an effort to evaluate our system, we apply our methodology on the codrep dataset [3], in order to measure its performance. Additionally, towards the evaluation of the effectiveness of our approach in practice, we apply our system in real-world scenarios, in order to assess its ability of providing actionable recommendations that can be used in practice during development.

4.1 System Evaluation

In the first step towards assessing the effectiveness of our system and identifying code styling inconsistencies and pieces of code that diverge from a common formatting, we tested our system against data coming from the *Codrep* competition (Codrep, 2019). Codrep is a competition for applying machine learning on source code. The main task of the codrep 2019 competition was the identification of the position in a code file in which a formatting error appears. Codrep dataset consists of 8,000 Java files, each one of which contains a single formatting error in a specific character position. An additional file is given, which includes the character position the formatting error appears for each one of the 8,000

[3] https://github.com/KTH/codrep-2019/tree/master/Datasets

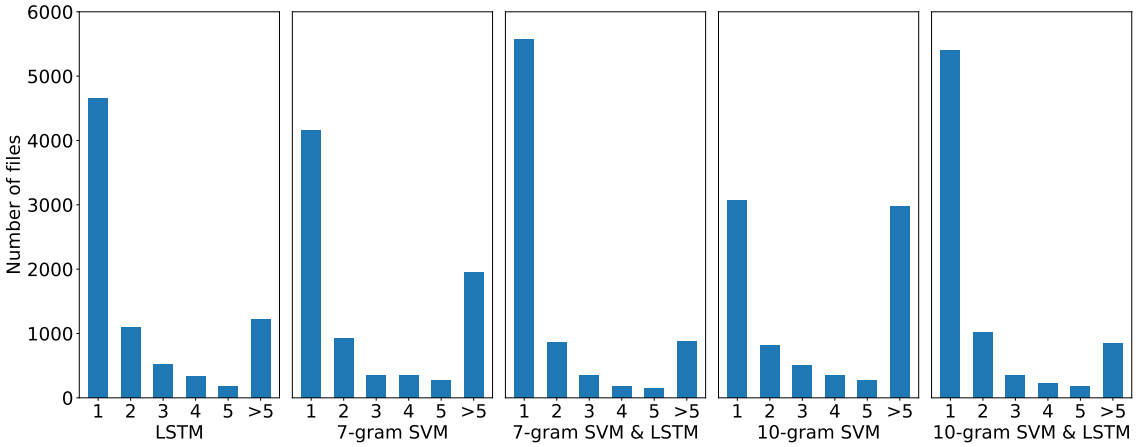

Figure 4: The histograms of the positions the correct answer was found in.

files. Figure 5 depicts an example, in which a formatting error appears. In this case, the formatting error is the unnecessary space that appears in character position 30.

```
1 public class test{
2   int a = 1 ;
3 }
4
```

Figure 5: Example of code file containing a single formatting error in character position 30.

The rules of the codrep competition are quite simple. The program has to take as input a source code file and output a descending ranking of the characters offsets, according to the probabilities that are calculated and estimate their likelihood of containing a formatting error. The final ranking of the characters is compared with the actual character that contains the formatting error and the evaluation metric is calculated.

The evaluation metric used by codrep to evaluate the performance of a system in identifying the formatting error position is the *Mean Reciprocal Rank (MRR)*. The reciprocal rank for one prediction is calculated using the inverse of the rank in an ordered list the correct answer is found for a given file q. The MRR is the average value of the reciprocal ranks for every file q in the set of the total evaluation files Q. The following equation depicts the calculation of the MRR, in a set of files Q to be predicted, where $rank_q$ is the position of the correct answer in the ordered list of predictions and $|Q|$ is the number of files used in the evaluation:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q} \tag{5}$$

The MRR is always a value in the interval [0, 1], where an MRR value of 1 is the best possible score and depicts that the first prediction in the ordered list is the correct position of the formatting error, while an MRR value of 0 means that the correct position was not found. Table 4 depicts the MRR obtained using our approach only with the generative model, only with the outlier detection model and with both the models combined. The n-grams we selected to use were 7-grams and 10-grams.

Table 4: MRRs of LSTM and SVM models.

Model	MRR
LSTM	0.70
7-gram SVM	0.63
7-gram SVM & LSTM	0.78
10-gram SVM	0.57
10-gram SVM & LSTM	**0.85**

According to the MRR values depicted in Table 4, the combined model, consisted of both the LSTM and the 10-gram SVM yields the best results. These values are well above the ones from random guessing the position of the formatting error. Santos et al. (Santos et al., 2018) calculated that, for a file of 100 lines and 10 tokens per line, the random guessing would achieve an MRR of 0.002.

It should be noted at this point that we selected not to compare the MRR values of our approach with the results coming from the actual Codrep competition[4], as the rules of the competition were not quite strict

[4]https://github.com/KTH/codrep-2019

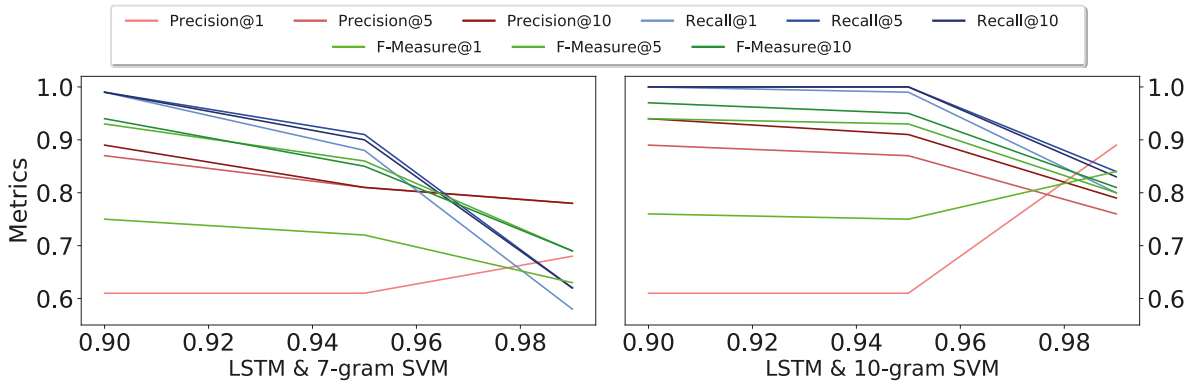

Figure 6: The precision, recall and f-measure metrics for various k and threshold values.

and the participants were allowed to use any possible technique to identify the formatting errors, e.g. using regular expressions, while a training set containing a lot of similar formatting errors was given to them a priori.

However, the MRR is a quite strict and conservative metric and its values can be significantly reduced just by some bad predictions. Indeed, in a case where the correct answer is ranked first 50% of cases and second the other 50%, the MRR value would be just 0.75, despite the fact that this model would probably be considered acceptable. In order to cope with the strictness of the MRR metric, we also calculated the histograms of the position in the ordered list the correct answer was found. Figure 4 illustrates these histograms. The height of each bar displays the number of files for which the correct prediction was found on that position.

The results from Table 4 and Figure 4 show that the correct prediction, i.e. the identification of the formatting error within the source code, is the first one for the most of the times. The combination of the two models, LSTM and SVM, clearly improved the results, while the selection of 10-grams over 7-grams had also a positive impact.

Moreover, in an attempt to further examine the performance of our approach in identifying formatting errors and deviations from the globally used code styling, we evaluated our system in the following scenario. From the sorted list of tokens, along with their probabilities of being a formatting error, only the first k tokens are returned to the user, as long as their probability of being a formatting error is above a predefined *threshold*. For these tokens, we examine whether the actual formatting error is included and, based on that, we calculate the metrics *precision@k*, *recall@k* and *f-measure@k*. Figure 6 illustrates these metrics calculated for the best two of the previous models (the combination of 7-gram SVM and LSTM and the combination of 10-gram SVM and LSTM) and using various thresholds in the range $0.90 - 1.0$ and values $1, 5$ and 10 for k.

From the results illustrated above, we can conclude that the 10-gram SVM along with the LSTM performs better with a precision value of 0.9, a recall value of 1.0 and an f-measure value of 0.95 for $k = 10$ and $threshold = 0.95$. A system with the aforementioned parameters could correctly identify deviations from the globally used formatting and provide useful suggestions to the developer about possible fixes.

4.2 Application of Formatting Error Detection in Practice

In order to further assess the effectiveness of our approach in providing actual and useful recommendations that can be used in practice during the development process, we applied our methodology in certain use-cases, in which we aspire to identify the applicability of the formatting error detection system in practice. Thus, we randomly selected some small Java files from the most popular open-source GitHub repositories, in which our methodology would be applied to.

Figure 7a presents the initial source code of the first file to be evaluated by our system. In this occasion a formatting error is detected in line 8 and concerns the extra use of a new-line character (the character before the red circle). It is obvious that the insertion of just one new character can complicate and possibly reduce the overall code readability and comprehensibility, as there is no correct indentation.

Our system identifies the formatting error position in the first place of the sorted predictions and returns this prediction to the developer, in an attempt to fix this error and improve the readability

```
1 package com.developmentontheedge.sql.model;
2
3 public class SqlQuery
4 {
5     public static AstStart parse(String
          query)
6     {
7         return parse(query,
              DefaultParserContext.getInstance
              ());
8 }
9
10    public static AstStart parse(String
          query, ParserContext context)
11    {
12        SqlParser parser = new SqlParser();
13        parser.setContext(context);
14        parser.parse(query);
15        return parser.getStartNode();
16    }
17 }
```

(a) Initial version of file

```
1 package com.developmentontheedge.sql.model;
2
3 public class SqlQuery
4 {
5     public static AstStart parse(String
          query)
6     {
7         return parse(query,
              DefaultParserContext.getInstance
              ());
8     }
9
10    public static AstStart parse(String
          query, ParserContext context)
11    {
12        SqlParser parser = new SqlParser();
13        parser.setContext(context);
14        parser.parse(query);
15        return parser.getStartNode();
16    }
17 }
```

(b) Final version of file

Figure 7: First use case.

```
1 package com.developmentontheedge.sql.model;
2
3 public class PredefinedFunction
      implements Function
4 {
5     private final int maxNumberOfParams;
6
7     /**
8      * Returns the biggest possible
           number of required parameters,
           or -1 if any number of
9      * parameters is allowed.
10     */
11    @Override
12    ____public int maxNumberOfParams()
13    {
14        return maxNumberOfParams;
15    }
16 }
```

(a) Initial version of file

```
1 package com.developmentontheedge.sql.model;
2
3 public class PredefinedFunction
      implements Function
4 {
5     private final int maxNumberOfParams;
6
7     /**
8      * Returns the biggest possible
           number of required parameters,
           or -1 if any number of
9      * parameters is allowed.
10     */
11    @Override
12    public int maxNumberOfParams()
13    {
14        return maxNumberOfParams;
15    }
16 }
```

(b) Final version of file

Figure 8: Second use case.

of the file. Indeed, the final version of the file, which is displayed in Figure 7b, where the developer has taken into account the system's prediction, is easier for the developers to understand, as the correct indentation can be a valuable guidance towards the code flow comprehension.

Figure 8a illustrates a different example, in which, in the initial source code of the file, the formatting error is detected in line 12 and concerns the extra use of a tab character (depicted in the Figure). Again, the insertion of just one new character alters the way a developer can read the code and understand its content.

Our approach detects the formatting error and ranks it first among the set of all possible positions and returns this suggestion to the developer, in order for the formatting error to be fixed. Once again, the final version of the file, that is displayed in Figure 8b, is easier for the developers to comprehend, as the reading procedure follows a natural flow.

Despite these examples seem small and the fixes seem insignificant, they can be quite important in

large projects, in which different and various developers participate with various coding styles. Detecting and fixing these formatting errors could noticeably improve readability and code comprehensibility.

5 THREATS TO VALIDITY

Our approach towards identifying and detecting formatting errors and formatting deviations from the selected ground truth seems to achieve high internal validity, as it was proved by the evaluation of our system in the previous section.

When it comes to the external validity of our approach, there are some limitations and threats that need to be considered and span along the following axes: 1) the selected use case and 2) the definition of the ground truth along with the selection of the training dataset. Our design choice to apply our methodology on detecting formatting errors based on the codrep competition is just one of the use cases our system can be applied to. One threat to the external validity of our approach lies on the evaluation of our approach on different scenarios, i.e. the generalization of our approach on a set of different code stylings. However, the selected use case is considered as the most common and necessary one, while it does not differ significantly from the other scenarios. Additionally, for the creation of the ground truth, i.e. the set of files that define the most used code styling, we made use of a dataset created by Santos et al. (Santos et al., 2018), mining the top 10,000 Java repositories from Github. While more or different projects could be used for the creation of our ground truth and the training of our system on a selected formatting, our methodology can be applied as-is using a different benchmark or pristine dataset that can cover multiple and different evaluation scenarios.

6 CONCLUSIONS AND FUTURE WORK

In this work, we proposed an automated formatting error detection methodology, which is based on two algorithms, LSTM and SVM, that aspire to model the problem from different perspectives. One of the basic contributions of our approach is that it does not need to be pre-trained based on a dataset or based on a set of predefined rules, that allow only minor modifications, but it can learn the coding style used in a project and detect deviations from it in a completely unsupervised manner, without the need of experts or prior domain knowledge. The evaluation of our approach in two diverse axes indicates that our system can effectively identify formatting deviations from the coding style used as ground truth and provide actionable and useful recommendations to the developers, enhancing the readability degree and ensuring the styling consistency across the project.

While the use of globally adopted code styling in the evaluation stage of our approach indicates that our methodology could also be used as a common formatter, the main contribution of our approach lies on the unsupervised code styling consistency held across a project or set of files. Should a team of developers apply our methodology across a project, every team member will be motivated to follow the common code styling from the ground, improving the maintenance and the evolution of the software.

Future work relies on several axes. Firstly, a fixing mechanism could be built that, based on the formatting error detection approach of this work, would provide actual recommendations to the developers about changes that could fix the error, without changing the functionality of the code. Additionally, a thorough evaluation mechanism could be created that could qualitatively or quantitively assess the performance of the complete system in detecting and fixing formatting errors, as well as the readability improvement achieved. Moreover, we would suggest the creation of a tool or plugin for a set of commonly used IDEs, that would predict the formatting errors, while the developer is typing, highlight these errors and suggest possible fixes. Finally, we could alter the training dataset by using projects with different characteristics and, especially, small projects with developers that use different formatting styles, in order to evaluate the performance of our approach in a small code basis with high formatting fluctuations.

ACKNOWLEDGEMENTS

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code:T1EDK-04673).

REFERENCES

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2014). Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 281–293, New York, NY, USA. Association for Computing Machinery.

Codrep (2019). Codrep 2019. https://github.com/KTH/codrep-2019. Accessed: 2020-09-27.

GNU Project (2007). Indent - gnu project. https://www.gnu.org/software/indent/. Accessed: 2020-09-27.

Hellendoorn, V. J. and Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 763–773, New York, NY, USA. Association for Computing Machinery.

Hindle, A., Godfrey, M. W., and Holt, R. C. (2008). From indentation shapes to code structures. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 111–120.

Hochreiter, S. and Schmidhuber, J. (1997). Lstm can solve hard long time lag problems. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*, pages 473–479. MIT Press.

Kesler, T. E., Uram, R. B., Magareh-Abed, F., Fritzsche, A., Amport, C., and Dunsmore, H. (1984). The effect of indentation on program comprehension. *International Journal of Man-Machine Studies*, 21(5):415 – 428.

Lee, T., Lee, J.-B., and In, H. (2013). A study of different coding styles affecting code readability. *International Journal of Software Engineering and Its Applications*, 7:413–422.

Loriot, B., Madeiral, F., and Monperrus, M. (2019). Styler: Learning formatting conventions to repair checkstyle errors. *CoRR*, abs/1904.01754.

Markovtsev, V., Long, W., Mougard, H., Slavnov, K., and Bulychev, E. (2019). Style-analyzer: Fixing code style inconsistencies with interpretable unsupervised algorithms. volume 2019-May, pages 468–478.

Miara, R. J., Musselman, J. A., Navarro, J. A., and Shneiderman, B. (1983). Program indentation and comprehensibility. *Commun. ACM*, 26(11):861–867.

Ogura, N., Matsumoto, S., Hata, H., and Kusumoto, S. (2018). Bring your own coding style. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 527–531.

Parr, T. and Vinju, J. (2016). Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, page 137–151, New York, NY, USA. Association for Computing Machinery.

Posnett, D., Hindle, A., and Devanbu, P. (2011). A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 73–82, New York, NY, USA. Association for Computing Machinery.

Prabhu, R., Phutane, N., Dhar, S., and Doiphode, S. (2017). Dynamic formatting of source code in editors. In *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, pages 1–6.

Prettier (2017). Prettier. https://prettier.io/. Accessed: 2020-09-27.

Santos, E. A., Campbell, J. C., Patel, D., Hindle, A., and Amaral, J. N. (2018). Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322.

Scalabrino, S., Linares-Vásquez, M., Poshyvanyk, D., and Oliveto, R. (2016). Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10.

Scalabrino, S., Linares-Vásquez, M., Oliveto, R., and Poshyvanyk, D. (2018). A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30.

Seo, K.-K. (2007). An application of one-class support vector machines in content-based image retrieval. *Expert Systems with Applications*, 33(2):491 – 498.

Tysell Sundkvist, L. and Persson, E. (2017). *Code Styling and its Effects on Code Readability and Interpretation*. PhD thesis, KTH Royal Institute of Technology.

Wang, X., Pollock, L., and Vijay-Shanker, K. (2011). Automatic segmentation of method code into meaningful blocks to improve readability. In *2011 18th Working Conference on Reverse Engineering*, pages 35–44.

White, M., Vendome, C., Linares-Vásquez, M., and Poshyvanyk, D. (2015). Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 334–345. IEEE Press.