Exploring the Impact of Code Style in Identifying Good Programmers

Rafed Muhammad Yasir[1], Dr. Ahmedul Kabir[1]

[1]Institute of Information Technology (IIT), University of Dhaka, Dhaka, Bangladesh

Abstract

Code style is an aesthetic choice exhibited in source code that reflects programmers' individual coding habits. This study is the first to investigate whether code style can be used as an indicator to identify good programmers. Data from Google Code Jam was chosen for conducting the study. A cluster analysis was performed to find whether a particular coding style could be associated with good programmers. Furthermore, supervised machine learning models were trained using stylistic features and evaluated using recall, macro-F1, AUC-ROC and balanced accuracy to predict good programmers. The results demonstrate that good programmers may be identified using supervised machine learning models, despite that no particular style groups could be attributed as a good style.

Keywords

code style, identify good programmer, stylistic features

1. Introduction

Code style represents the physical layout of code (e.g., indentation, bracket placement), which reflects an individual's personal programming habits that do not affect its functionality [1]. Figure 1 shows two code snippets that are functionally similar but written in two different styles. Code style has an impact on various aspects of software engineering, including software maintenance [2] and speed of software development [3]. However, no prior studies have been conducted to see whether good programmers can be detected by looking at their coding style. This paper investigates the potential for using code style to identify good programmers.

```
1 int sum(int a, int b){
2     return a+b;
3 }
```

(a) Style 1

```
1 int sum(int a, int b)
2 {
3   return a+b;
4 }
```

(b) Style 2

Figure 1: Two functionally same code snippets written in different styles

Establishing a link between code style and good programmers can have several implications. Many software repositories contain style guidelines that are used to enforce a specific code style in order to maintain software quality [4]. However, these style guides are often opinionated and arbitrary [5, 6]. If a specific code style exhibited by programmers can be identified as a good style, it can be used to create non-arbitrary style guidelines for better software maintenance.

In the software industry, the developers hired by a company directly affect the quality of the codebase that they maintain. During recruitment, the candidates who apply for jobs often have to solve a set of programming problems. However, existing hiring practices do not account for the possibility that a skilled programmer could have a bad day and fail to answer a question correctly. Thus, in some circumstances a judgment may be unfair. If positive stylistic features can be identified in a programmer's code, they can be used as an additional criterion to enhance recruitment processes. This study is an initial attempt to determine whether such relationships between competent programmers and their code style can be established.

To conduct the study, the solutions collected from Google Code Jam (GCJ) [7] were used as the dataset. 30 stylistic metrics were extracted from the codes and used as features for analysis. Two methods of analysis were used. At first, clustering algorithms were applied to the data to discover style groups and check whether good programmers belonged to a particular style group. Secondly, supervised machine learning models were trained using stylistic features to predict good programmers. The models were evaluated using recall, macro-F1, area under curve of ROC (AUC-ROC), and balanced accuracy.

Results show that, although style groupings were found, there were no specific groups with which good programmers could be associated. However, supervised machine learning models showed that good programmers can be predicted to some extent. Based on the evaluated metrics, a Balanced Random Forest achieved the best re-

QuASoQ'22: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual

bsse0733@iit.du.ac.bd (R. M. Yasir); kabir@iit.du.ac.bd (Dr. A. Kabir)

https://rafed.github.io/ (R. M. Yasir)

sult with an average of 0.65 recall, 0.51 macro-F1, and 0.69 AUC-ROC.

2. Related Work

To the best of our knowledge, this is the first time code style has been used to identify good programmers. Early research conducted by Oman and Cook proposed a taxonomy for code styles to help people grasp a coherent view on the basis and application of code styles [8]. The four major categories of their taxonomy are general practices, typographic style, control structure style and information structure style. They also concluded in further research that code style is more than cosmetic and that it can affect areas such as code comprehension [9].

Caliskan et al. proposed a Code Stylometry Feature Set (CSFS) with which they performed source code authorship attribution [10]. Their feature set is language agnostic and can be used for other programming languages as well. With their method, they achieved 94% accuracy in classifying 1600 authors and 98% accuracy in classifying 250 authors. They concluded that this method can help in the identification of authors of malicious programs, ghostwriting detection, software forensics and copyright investigation.

Mirza and Cosma explored the suitability of using code style in detecting plagiarism in the BlackBox dataset [11]. BlackBox is a project that collects data from users of the BlueJ which is a Java IDE [11]. Their study showed that code style is suitable for detecting plagiarism.

For evaluating software projects, Zou et al. explored how code style inconsistency can affect pull request integration in projects on Github [3]. By analyzing 117 public repositories, they concluded that code styles with specific criteria can influence both the acceptance of pull requests and the time required to merge a pull request.

Mi and Yu conducted a study on stylistic inconsistency in software projects [2]. They proposed a collection of stylistic metrics for C++ projects and used these metrics to analyze small-scale Github projects. By using hierarchical agglomerative clustering they showed that stylistic differences exist between source files in a project. They concluded that, using the degree of stylistic inconsistency as a basis, code comprehensibility and software maintainability could be improved in the future.

Several tools have been developed that can check stylistic inconsistencies and help programmers improve code style. Ala-Mutka et al. developed *style++* that helps students learn good C++ programming conventions [12]. Mäkelä et al. developed *Japroach* that checks whether Java programs have a particular style and if style related issues exist in them [13]. Ogura et al. developed *style-coordinator* to decrease inconsistency and improve code readability [1].

Table 1
Number of Participants in a Round

	2015	2016	2017
Qualification round	10744	11401	11342
Round 2	1650	1641	1824
Round 3	266	296	286
World Finals	22	20	21

3. Methodology

3.1. Dataset Description

The dataset for the study was made up of the solutions gathered from the Google Code Jam (GCJ) website [7]. GCJ is an annual programming contest held by Google. GCJ is selected because its data is publicly available and it can somewhat resemble programming exams in recruitment processes. Professional programmers, students and amateurs from all around the world participate in GCJ. Therefore, not only does the dataset consist of source code from varying sources, but they also solve the same problem which makes comparative study possible. The contest consists of seven rounds, each progressively harder than the previous. The rounds are: Qualification round, Round 1A, Round 1B, Round 1C, Round 2, Round 3 and World Finals. We consider the programmers who reached at least Round 3 as "good" programmers because participating in this round requires passing the previous rounds with a large number of accepted solutions.

Although GCJ accepts solutions in many programming languages, C++ was selected as the preferred language for evaluation as it is more prevalent among participants and has the highest number of submissions. Each problem of the contest has two validation sets: a small input set and a large input set. A solution for the large validation set is a valid solution for the small input set, but not vice versa. For our analysis, the solutions from the small input were taken as it had more submissions and it would also be redundant if both solutions were taken. A small number of solutions were rejected as the language encoding consisted of non-standard characters.

The solutions to the contests held in 2015, 2016, and 2017 are chosen for experimentation. However, we only include solutions from Qualification Round, Round 2 and Round 3 for our dataset. Round 1A, Round 1B, and Round 1C are excluded because participating in these rounds are optional and thus lacks submissions from all programmers. Solutions from the World Finals are excluded because the number of finalists is too small to take into consideration for analysis. Table 1 shows the number of participants in each round of the contests.

From the collected data, layout and lexical stylistic features were extracted based on [10]. Abstract syntax tree based features were omitted as these features are

Table 2
Feature Description of Dataset

Feature	Definition
numTabs/length	Number of tabs divided by file length in characters
numSpaces/length	Number of space characters divided by file length in characters
numEmptyLines/length	Number of empty lines divided by file length in characters
whiteSpaceRatio	Ratio between the number of whitespace characters (spaces, tabs, and newlines) and non-whitespace characters
newLineBeforeOpenBrace	Ratio between the number of code blocks preceded by a newline character and not preceded by a newline character
tabsLeadLines	Ratio between the number of lines preceded by a tab and not preceded by a tab
avgLineLength	Average length of each line
stdDevLineLength	Standard deviation of the lengths of each line
numkeyword/length	Number of occurrences of keyword divided by file length in characters, where keyword is one of if, else, else-if, for, while, do, break, continue, switch, case (10 different features)
numTernary/length	Number of ternary operators divided by file length in characters
numTokens/length	Number of word tokens divided by file length in characters
numUniqueTokens/length	Number of unique keywords used divided by file length in characters
numComments/length	Number of comments divided by file length in characters
numLineComments/length	Number of line comments divided by file length in characters
numBlockComments/length	Number of comments divided by file length in characters
numLiterals/length	Number of string, character, and numeric literals divided by file length in characters
numMacros/length	Number of preprocessor directives divided by file length in characters
nestingDepth	Highest depth of control statements and loops
numFunctions/length	Number of functions divided by file length in characters
avgParams	Average number of parameters of functions
stdDevNumParams	Standard deviation of the number of parameters of functions

not within the control of a programmer. Furthermore, term frequency based features were also excluded as they largely depend on the corpus being used. Following these criteria, 30 stylistic features were extracted. The features are listed in Table 2.

3.2. Approach

This section discusses the setups for exploring the effects of code style on classifying good programmers. Two methods were used for this purpose: (1) clustering techniques and (2) supervised machine learning algorithms.

3.2.1. Analyzing Using Clustering Techniques

Clustering is a method of partitioning objects into homogeneous groups on the basis of similarity among those objects [14]. t-SNE is one such algorithm that can discover the potential number of clusters in a dataset with high dimensions [15]. For each problem in the contests, t-SNE graphs were plotted with the intent of finding groups that conform to a particular style. Each data point in the plots represents a solution submitted by a programmer. The data points are labeled as:

- Red: reached World Finals
- Green: reached Round 3
- Light blue: other programmers

The plots provide an estimate for the number of clusters and the distribution of good programmers in the clusters.

To further validate the clustering provided by t-SNE, Hierarchical Agglomerative Clustering (HAC) with Ward linkage was performed and dendrograms were plotted. HAC is a clustering algorithm that treats every data point as a cluster and they are gradually merged to form a single cluster [16]. The number of clusters indicated by the dendrograms was matched with the number of clusters indicated by t-SNE before further analysis was performed.

To analyze the properties of the t-SNE clusters, solutions to each problem in the dataset were clustered using K-Means With K=number of clusters estimated by t-SNE. The solutions in the data were then labeled based on the cluster they belonged to. This labeled data was fitted to a Random Forest Classifier to obtain the feature importance of the tree. Based on the tree's feature importance, it was determined what style groups exist and whether good programmers belong to a specific style group.

3.2.2. Analyzing using Supervised Machine Learning Algorithms

Supervised learning is a method of training a model that can make predictions based on labeled data [17]. For predicting good programmers, the following models were

trained: Logistic Regression (LR), Support Vector Classifier (SVC), K-Nearest Neighbors (KNN), Decision Tree (DT) and Random Forest (RF). A Dummy classifier was also trained to act as a performance baseline for comparison [18]. The models were trained for each problem in the dataset. Table 1 shows that the number of participants in Round 3 is far less than the participants in the previous rounds. That is, the proportion of "good" programmers in the dataset is much lower in comparison to the other programmers. This makes the classification an imbalanced classification problem [19]. To balance the training data, the up-sampling technique SMOTE [20] was used prior to training the above mentioned models. Furthermore, Balanced Random Forest (BRF) and RUS Adaboost classifier (RUSAda) were also trained which performs under-sampling to balance training data [21]. For bias-free results, all trained models were K-fold cross-validated.

4. Experimental Analysis

4.1. Performance Evaluation

The clusters created for analysis were evaluated empirically. Although the analyzed dataset had labels and the results could be evaluated using a metric, this was not done, as evaluating clustering algorithms using labels is not recommended [22].

For the supervised algorithms recall, macro-F1 and Area Under Curve of ROC (AUC-ROC) were used to evaluate the models. Recall is the measure of the fraction of good programmers correctly identified as good programmers [23]. Recall is calculated as equation (1). F1 is an evaluation metric measured by combining precision and recall, and it is calculated as (3) [23]. Macro-F1 is the arithmetic mean of the per class F1 scores. It has been selected as an evaluation criterion because the training data was imbalanced, and macro-F1 is a good metric for imbalanced data [24]. AUC-ROC is the area under a ROC curve that allows comparison between models [23]. Apart from these evaluation metrics, balanced accuracy was also reported. Balanced accuracy is defined as the average of recall obtained on each class [25].

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad (1)$$

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \quad (2)$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (3)$$

Table 3
Feature importance of Clusters

Features	Importance
newLineBeforeOpenBrace	0.282
tabsLeadLines	0.163
numTabs/length	0.151
numSpaces/length	0.103

4.2. Results and Discussion

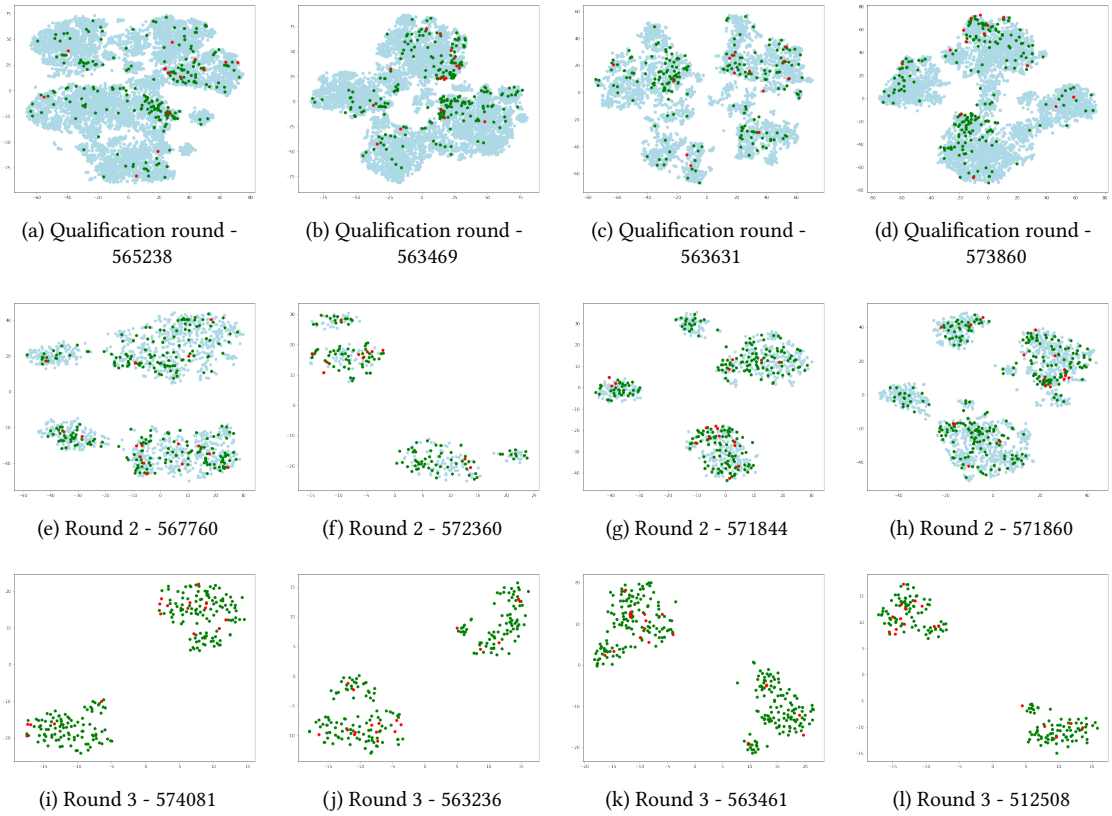
After analyzing the contest results from 2015, 2016, and 2017, it was discovered that they were similar. Therefore, only the results of one year (2016) are shown in this section.

Figure 2 shows the t-SNE clusters of all the problems in the dataset of the year 2016. The caption of each image shows the round and the problem number. From the graphs, it can be said that 4 stylistic clusters exist for each solution. The most important features of the clusters determined by a Random Forest Classifier are shown in Table 3. The importance of all other features was less than 0.05. It is seen that *newLineBeforeOpenBrace* and *tabsLeadLines* are the most prominent features in separating the clusters. A manual inspection of the codes also proved the findings to be true. The discovered clusters are formed around the following feature combinations:

- new line before opening braces, tabs lead lines
- no new line before opening braces, tabs lead lines
- new line before opening braces, whitespace lead lines
- no new line before opening braces, whitespace lead lines

Although style clusters were found, the good programmers were almost equally distributed among them. As a result, we cannot conclude that good programmers belong to a specific cluster.

The results of the supervised machine learning models are shown in Table 4. BRF, LR, SVC and RUSAda performed better than the dummy model, which indicates that some patterns can be identified by the models that can be used to predict good programmers. Also, BRF outperformed all models in terms of Recall, macro-F1 and AUC-ROC. While different studies have used code style for various aspects such as author identification and plagiarism detection, none of the studies have dealt with good programmer identification. Therefore, we cannot compare our results with those of existing studies. However, our results can inspire further research on the relationship between code style and the coding ability of programmers.



(a) Qualification round - 565238 (b) Qualification round - 563469 (c) Qualification round - 563631 (d) Qualification round - 573860

(e) Round 2 - 567760 (f) Round 2 - 572360 (g) Round 2 - 571844 (h) Round 2 - 571860

(i) Round 3 - 574081 (j) Round 3 - 563236 (k) Round 3 - 563461 (l) Round 3 - 512508

Figure 2: t-SNE Style Clusters of GCJ 2016

Table 4
Prediction Results of Supervised Learning Models

Model	Recall	macro-F1	AUC-ROC	Balanced Accuracy
BRF	0.650	0.511	0.695	0.645
LR	0.641	0.523	0.692	0.651
SVC	0.601	0.523	0.689	0.639
RUSAda	0.510	0.50	0.626	0.590
Dummy	0.485	0.412	0.499	0.489
KNN	0.469	0.494	0.593	0.565
DT	0.287	0.525	0.542	0.542
RF	0.185	0.539	0.664	0.537

5. Threats to Validity

This section presents aspects that may threaten the validity of the study:

- **Internal validity:** The result of our analysis largely depends on the stylistic features that were used. Using other stylistic features may affect the results. However, many existing studies [10, 26] have used these features for their analysis, so they can be relied upon.
- **External validity:** The analysis was done on the source files of the GCJ dataset. Therefore, the findings of this study may not be generally applicable to contests in other formats. Furthermore, as only C++ codes were selected for analysis, it cannot be said whether stylistic features of other programming languages will show similar results. Additionally, the criteria for defining a good programmer are subjective and could be defined in other ways depending on the context. In such contexts, our results can not be generalized.

To ensure the reliability of the study, the analysis results are made publicly available in Jupyter notebooks at github.com/rafed/GcjStyleAnalysis.

6. Conclusion

This paper explores whether code style can be used to identify good programmers. The study was conducted on C++ solutions from the Google Code Jam contest. Clustering techniques such as t-SNE and hierarchical agglomerative clustering were used to discover whether style clusters exist and if good programmers could be attributed to any of them. Although four style clusters were found, good programmers could not be associated with a particular cluster. However, supervised machine learning showed that stylistic attributes can be used to predict good programmers. Seven machine learning models were trained and evaluated using recall, macro-F1 and AUC-ROC. A Balanced Random Forest yielded the best results with 0.650 recall, 0.511 macro-F1 and 0.695 AUC-ROC. The results indicate that code style can be used as a measure to identify good programmers.

Future research will examine if defining style guidelines based on the coding style of skilled programmers enhances the quality of software. Additionally, it is possible to investigate how the current recruitment procedures might be efficiently linked with the prediction of good programmers utilizing code style. There is also potential for improving our results using other techniques.

References

[1] N. Ogura, S. Matsumoto, H. Hata, S. Kusumoto, Bring your own coding style, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 527–531.

[2] Q. Mi, J. Keung, Y. Yu, Measuring the stylistic inconsistency in software projects using hierarchical agglomerative clustering, in: Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, 2016, p. 5.

[3] W. Zou, J. Xuan, X. Xie, Z. Chen, B. Xu, How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects, Empirical Software Engineering (2019) 1–33.

[4] T. Erkkinen, Model style guidelines for production code generation, Technical Report, SAE Technical Paper, 2005.

[5] Pullrequest.com, 2022. URL: https://www.pullrequest.com/blog/create-a-programming-style-guide/.

[6] Google style guides, 2022. URL: https://google.github.io/styleguide/.

[7] Google, Past contests, google code jam, 2022. URL: https://code.google.com/codejam/past-contests.

[8] P. W. Oman, C. R. Cook, A programming style taxonomy, Journal of Systems and Software 15 (1991) 287–301.

[9] P. W. Oman, C. R. Cook, Typographic style is more than cosmetic, Communications of the ACM 33 (1990) 506–520.

[10] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, R. Greenstadt, De-anonymizing programmers via code stylometry, in: 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 255–270.

[11] O. M. Mirza, M. Joy, G. Cosma, Style analysis for source code plagiarism detection—an analysis of a dataset of student coursework, in: 2017 IEEE 17th international conference on advanced learning technologies (ICALT), IEEE, 2017, pp. 296–297.

[12] K. Ala-Mutka, T. Uimonen, H.-M. Jarvinen, Supporting students in c++ programming courses with automatic program style assessment, Journal of Information Technology Education: Research 3 (2004) 245–262.

[13] S. Mäkelä, V. Leppänen, Japroch: A tool for checking programming style, Kolin Kolistelut—Koli Calling 2004 (2004) 151.

[14] S. C. Johnson, Hierarchical clustering schemes, Psychometrika 32 (1967) 241–254.

[15] G. C. Linderman, S. Steinerberger, Clustering with t-sne, provably, SIAM Journal on Mathematics of Data Science 1 (2019) 313–332.

[16] K. Sasirekha, P. Baby, Agglomerative hierarchical clustering algorithm-a, International Journal of Scientific and Research Publications 83 (2013) 83.

[17] S. J. Russell, P. Norvig, Artificial intelligence: a modern approach, Malaysia; Pearson Education Limited,, 2016.

[18] Scikit-learn, Dummy classsifier, 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html.

[19] N. Japkowicz, S. Stephen, The class imbalance problem: A systematic study, Intelligent data analysis 6 (2002) 429–449.

[20] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: synthetic minority oversampling technique, Journal of artificial intelligence research 16 (2002) 321–357.

[21] Imbalanced-learn, Ensembled methods, 2022. URL: https://imbalanced-learn.readthedocs.io/en/stable/api.html#module-imblearn.ensemble.

[22] I. Färber, S. Günnemann, H.-P. Kriegel, P. Kröger, E. Müller, E. Schubert, T. Seidl, A. Zimek, On using class-labels in evaluation of clusterings, in: MultiClust: 1st international workshop on discovering, summarizing and using multiple clusterings held in conjunction with KDD, 2010, p. 1.

[23] D. M. Powers, Evaluation: from precision, recall and f-measure to roc, informedness, markedness

and correlation (2011).

[24] B. Wu, S. Lyu, B. Ghanem, Constrained submodular minimization for missing labels and class imbalance in multi-label learning, in: Thirtieth AAAI Conference on Artificial Intelligence, 2016.

[25] Balanced accuracy, 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.balanced_accuracy_score.html.

[26] M. Tereszkowski-Kaminski, S. Pastrana, J. Blasco, G. Suarez-Tangil, et al., Towards improving code stylometry analysis in underground forums, in: Proceedings on Privacy Enhancing Technologies (PETS), 2022.