A Machine Learning Approach in Calculating Code Readability

Preston Ford, Rai Katsuragawa, Juneon Kim, YoungWoo Song, Jeffry Yoon
CS 270, Fall 2024
Department of Computer Science
Brigham Young University

Abstract

Software engineers are often expected to work together in a professional setting which means many programmers will often read code from another programmer. In this project, we use machine learning techniques to create an algorithm that would determine the readability of the provided code. We included the main criteria that are typically used in determining if provided code is readable or not. These features were generated with code that calculated these feature scores given a file containing the code. We used various learning methods like random forest and logistic regression. AIs using the decision tree model proved to be one of the best methods with a general accuracy rating of 67%. We believe that improving code readability is key in improving collaboration efficiency between software engineers.

1 Introduction

In the world of programming, code readability is a fundamental aspect of maintaining and enhancing software systems. Readable code facilitates effective collaboration among developers and teams. It also reduces the likelihood of bugs, and makes future code modifications more efficient. Thus, it would be beneficial for programmers if they could quantify the readability of their code and evaluate whether it is well-structured. However, assessing readability often relies on individual developers' experience and habits. Such subjectivity can lead to inconsistencies and loss of opportunities to improve code quality.

Recognizing this challenge, our project aims to use machine learning models to provide an objective and scalable approach to evaluating code readability. By scoring the readability of code, we hope that our models make sure the code of the software is not only functional but also maintainable in the long term. Just as a spell checker highlights errors in text, our readability models aim to pinpoint aspects in code that could be improved for better comprehension.

Through this approach, we aim to support developers and teams in producing cleaner, more maintainable codebases. By introducing machine learning into this process, we hope to standardize the evaluation of code readability and provide actionable insights to enhance the quality of software development projects.

2 Method

This project required us to collect the code samples, judge if the samples are valid, and evaluate and assign scores for each reasonable feature we decide. Furthermore, we needed to decide how we were going to compute the target value (readability score) due to its subjectivity. Then, we trained and tested the selected models on the generated dataset. We would then improve upon the model by either changing aspects of the features or changing the parameters of the models themselves.

2.1 Data Sources

First, we decided to collect code samples written in Python 3 to ensure uniformity in style convention adherence. Then, we collected data from open-source repositories like GitHub and coding platforms like LeetCode. We mainly use GitHub as our data source because it has a variety of code snippets and projects. To extract the data from GitHub, we used the API provided by GitHub to search repositories, issues, and files by keywords, languages, and other filters.

Leetcode provides a set of "Design" questions that have users implement complex classes or data structures. For example, Leetcode #355 "Design Twitter" involves implementing a news feed, tweet posting system, and follow/unfollow system. The questions we decided to use were "Design Twitter", "Find Median from Data Stream",

and “LRU Cache”, with about 75 samples of each. The samples from these questions often range from simple 10-line solutions to dense 70-line solutions, and implementations of the same question often vary wildly (unlike the simpler LeetCode questions outside the “Design” category).

In order to extract the code samples from LeetCode, we used a web scraper. We extracted code from Leetcode to complement more instances and provide more data variety.

2.2 Initial Feature Selection

For our first feature extraction, we determined that the five main aspects that are used when determining code readability subjectively would be suitable features for the initial testing and modeling. The five main aspects that are judged when subjectively determining code readability are:

1. **Comment Ratio:** This feature measures the readability of a code. This is calculated by the number of comment lines divided by the number of code lines. The intention is that the comment can indicate what a programmer intended to do and it leads to better code readability.
2. **Max Line Length:** The length of the longest line in the code. Lines that are too long can reduce readability and make the code harder to maintain. Keeping lines within a standard length (e.g., 80-120 character) improves clarity.
3. **Cyclomatic Complexity:** A software metric that measures the complexity of a program's source code by counting the number of linearly independent paths through it. The intention is that samples with a higher cyclomatic complexity tend to be less readable; as software design instruction often emphasizes the KISS principle (Keep It Simple, Stupid). Complexity is measured for each function in the source code and averaged, so that the models do not bias towards shorter code samples.
4. **Variable Name Quality:** This involves measuring and validating various properties of variable names: their compliance as Python names, linguistic validity, length, and lexical relatedness to one another. For linguistic validity, the “SpellChecker” library is used to verify whether the variable names exist in the dictionary. For lexical similarity, the “spaCy” library is used to compare each name and compute the average score. All these evaluations assume English variable names, as English is the universal language in programming.
5. **Style Guide Adherence:** This measures how well the code follows established style conventions. We decided to use PEP 8 as the style guide since it is the official style guide for Python code and is widely adopted by the Python community. It covers various aspects of code formatting, such as indentation, line length, naming conventions, and whitespace usage. In order to interpret the code samples and check their adherence to PEP 8 style guidelines, we used a Python library called "Pylint." It analyses code without actually running it and computes a numerical score.
6. **Readability (Target):** This is the categorical output class. To simplify the classification process, we manually grouped the readability scores into three categories: 0 (Bad), 1 (Decent), and 2 (Good). This adjustment aimed to reduce complexity and improve model performance by focusing on more distinct and meaningful classifications.

2.3 Selected Models

The models we initially started with are the Logistic Regression model and the Decision Tree model. We chose the Logistic Regression model to be the baseline model to see if there is a linear relationship between features and the readability score. The initial scores will be used to see how each feature contributes linearly to the readability. We chose the Decision Tree model as our second initial model to see if there is a nonlinear relationship between features and the readability score. Decision Tree would allow us to create splits based on the features which would reveal deeper insights.

3 Initial Results

Logistic Regression

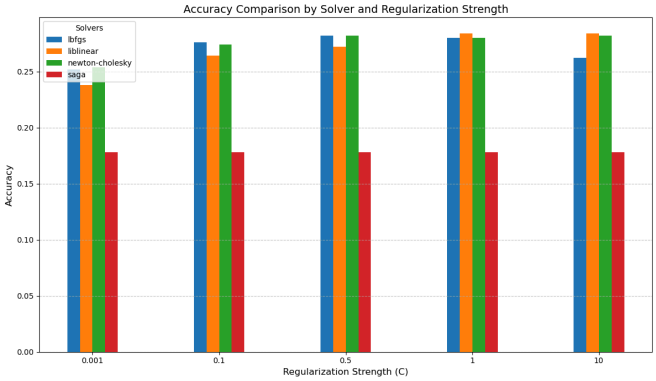

Fig. 1 - Accuracy vs. Solver and Regularization Strength

The graph above shows the initial results obtained when training a Logistic Regression model on the dataset. We tested different optimization algorithms and regularization values. Generally, as the regularization becomes weaker (i.e., the value of C increases), the accuracy improves, meaning that allowing the model to adapt more to the training data and become more flexible helps it capture more complex relationships between features and target values. Furthermore, except for the "saga," all solver algorithms have a chance to compute the highest accuracy, and there is no clear optimal value for the solver hyperparameter, which tells us that the problem is not overly complex, and all solvers are equally capable of converging to a good enough solution. However, the best accuracy achieved is about 27%, which indicates that the model struggled to capture the underlying patterns in the dataset. This suggests that the model had difficulty finding linear separation, even though the data itself might be simple.

Decision Tree

In the initial round of model building with the Decision Tree Classifier (clf = DecisionTreeClassifier()), the model achieved an accuracy of 35%, which is significantly lower than expected and suggests that the model is not capturing the relationships between the features and the target variable well. This performance is worse than random chance, indicating that the decision tree may be overly simplistic or prone to underfitting. This result highlights the need for further optimization, including hyperparameter tuning (e.g., max_depth, min_samples_split, min_samples_leaf) to improve the model's ability to generalize and perform better on unseen data.

Naive Bayes

	precision	recall	f1-score	support
Very Bad	0.33	0.20	0.25	10
Bad	0.00	0.00	0.00	7
Decent	0.27	0.32	0.29	22
Good	0.36	0.30	0.33	27
Very Good	0.52	0.50	0.51	34

Fig. 2 - Initial Naive Bayes Classification Report

The results above are derived from The Gaussian Naive Bayes model, used on a test set of size 100. Most notably, the precision and recall of the “Bad” class is 0; signifying that the model cannot correctly identify a code sample that is better than “Very Bad” and worse than “Decent”. The overall accuracy for this run was just 34%.

4 Model and Feature Improvement

One of the things we noticed with the code was that some of the files had a lot of duplicated code. The duplicated code would cause the readability score to skew too much to one side as repeated data would increase the weight on some features while decreasing the impact of other features.

The Comment Ratio feature was also adjusted to become categorical, i.e. “Very bad”, “Good”, etc rather than continuous. The ratio that described good readability was often unreliable, as code samples with large segments of commented code should be considered poor quality but instead skewed our models into classifying it as good.

For a number of instances, we found that a complexity score could not be determined due to syntax errors or unicode symbols that had bad compatibility with the Radon Library. In order to mitigate this issue, the mean of the feature was used for the instances where the complexity score could not be determined.

We also adjusted the code that determined the score for the variable name quality feature to also include the names of functions. We decided to do this because having a good naming convention for functions is also important for good readable code.

We discovered that decreasing the number of output classes from 5 to just 3 (Bad, Decent, and Good) greatly improved accuracy, as the models no longer have to distinguish between extremities like Good and Very Good, which often had specific reasons as to what differentiated code from having good readability and very good readability.

Additionally, we adjusted how the output class was assigned to each data instance. We first tried to automate the process with code that would calculate some score which would be sorted into bins that represented the categories of the output classes. But we found that the automated scores were very inconsistent so each data instance was manually assigned an output class instead.

We also decided to train the data set with the random forest classifier model. We chose this model because out of the initial models, the decision tree seemed to be consistently better than the other models. Because the random forest classifier model was designed based on the algorithm of the decision tree, we decided to add it to our list of models to see how well it would perform.

Another improvement that we made was replacing LeetCode samples with PyPI samples. We did this because we found that the LeetCode samples did not provide enough variance for the models to train on while the PyPI samples that we extracted did provide the variance needed to better improve the accuracy.

4.1 Data Instances

Sample	Style Guide	Var Names	Compl exity	Max Line	Comm ents	Score
Github #135	0	0.367	15.545	132	Med	Bad
Github #26	4.92	0.628	0.364	103	Med	Good

Fig. 3 - Data Instance Example

5 Final Results

Logistic Regression

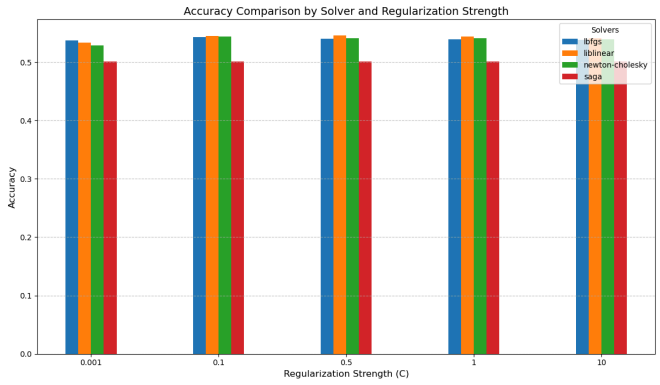

Fig. 4 - Accuracy vs. Solver and Regularization Strength

After the improvement, the Logistic Regression model was able to achieve an accuracy of about 55%. As the regularization strength (C) increases, accuracy tends to stabilize or improve across solvers, indicating that a weaker regularization allows the model to better fit the training data by being more flexible. Additionally, the difference between the 'saga' algorithm and others decreased because we used the larger dataset. However, other algorithms still showed better performance, and especially "liblinear" and "newton-cholesky" indicated good efficiency.

Solver	Average Number of Iterations for All Classes
lbfgs	[391.2]
liblinear	[11.4 16.8 25.]
newton-cholesky	[7.2 8. 7.]
saga	[500.]

Fig. 5 - Average Number of Iterations

As we can see from the graph, there is an accuracy ceiling, suggesting that the data might lack strong predictive signals or linear separability for the Logistic Regression model.

Decision Tree

We utilized a Decision Tree Classifier to analyze readability categories within the dataset. To optimize the model's performance, we performed Grid Search Cross-Validation (GridSearchCV) using a range of hyperparameters: max_depth (3, 5, 7, None), min_samples_split (2, 5, 10), min_samples_leaf (1, 2, 4), and the criterion ('gini' and 'entropy'). The model was evaluated using 5-fold cross-validation with accuracy as the scoring metric. The best parameters identified were max_depth = 3, min_samples_split = 2, min_samples_leaf = 2, and criterion = 'gini'. The result is a cross-validation accuracy of 67%.

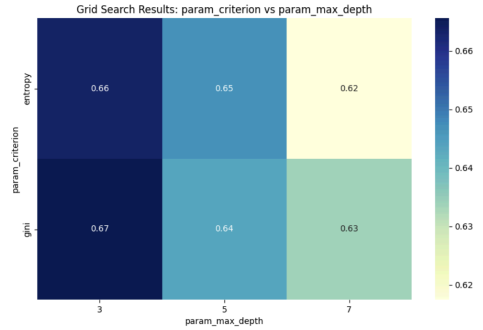

Fig. 6 - Criterion vs Max Depth

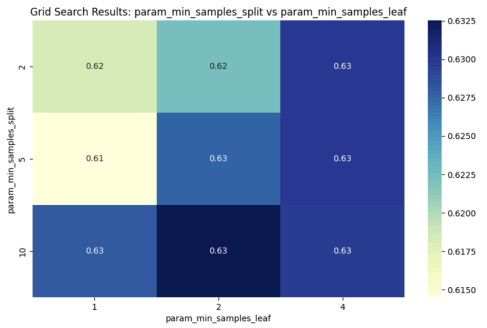

Fig. 7 - Minimum Split vs Minimum Leaf

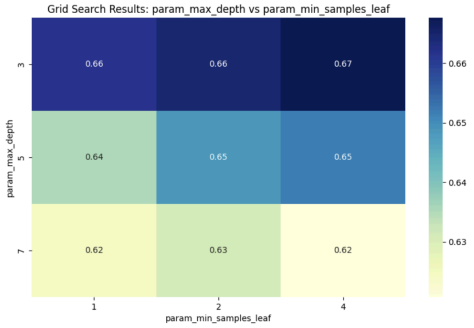

Fig. 8 - Max Depth vs Min Leaf

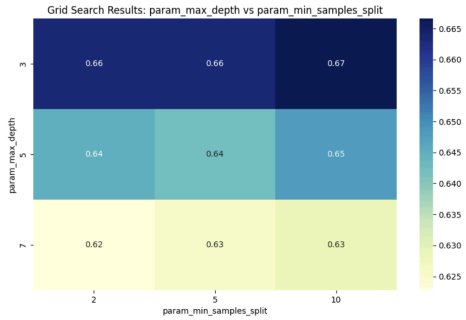

Fig. 9 - Max Depth vs Minimum Split

Naive Bayes

The Gaussian Naive Bayes model was used on a dataset with 500 instances and a train/test split of 80/20. Its accuracy score with three output classes was 60% with the following report:

	precision	recall	f1-score	support
Bad	0.47	0.41	0.44	17
Decent	0.39	0.55	0.45	22
Good	0.76	0.67	0.71	61

Fig. 10 - Final Naive Bayes Classification Report

From this data we can learn that the “Good” class had the most true positives, followed by “Bad” and “Decent”. This implies that it is easier to classify between a binary good or bad than it is to classify an in-between.

Random Forest Classifier

After the initial experiments, we added the Random Forest Classifier to our list of models and performed extensive hyperparameter tuning to evaluate its performance. This model was tested across three datasets: data with only Github python files, data with only PyPI python files, and data that combined those two files. For each dataset, we experimented with different hyperparameter combinations, varying the number of estimators (n_estimators), the maximum tree depth (max_depth), and the minimum samples required to split a node (min_samples_split). The performance of the Random Forest Classifier was visualized using three graphs, one for each dataset,, which illustrate the impact of different hyperparameter settings on accuracy.

The Random Forest Classifier showed varying performance across the three datasets. With the Github dataset, it achieved a peak accuracy of 47% using n_estimators=100, max_depth=None, and min_samples_split=2. The accuracy improved to 51% with the PyPI dataset, using n_estimators=300, max_depth=10, and min_samples_split=5. The highest accuracy of 52% was achieved with the combined total dataset, using n_estimators=200, max_depth=15, and min_samples_split=2, where the larger dataset size enhanced generalization and reduced overfitting.

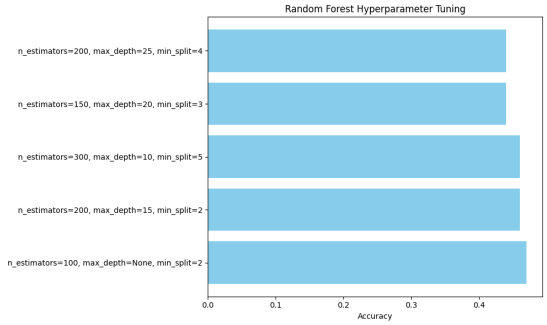

Fig. 11 - 500 GitHub dataset

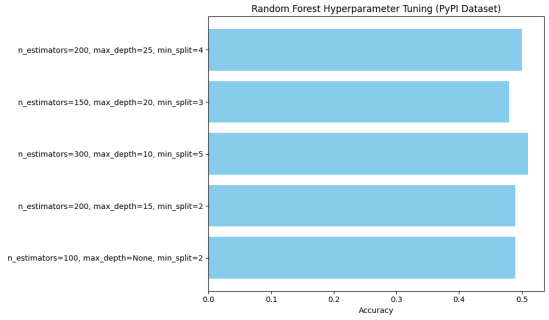

Fig. 12 - 500 PyPI dataset

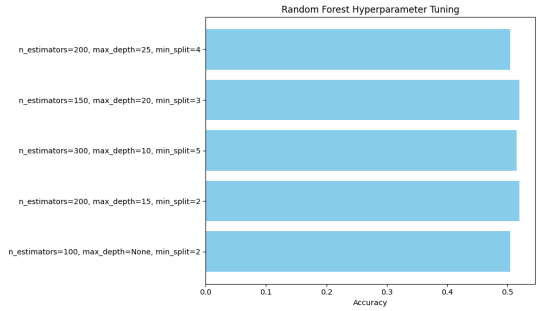

Fig. 13 - 1000 total dataset

Overall, the Random Forest Classifier demonstrated its capability to handle non-linear relationships in the data and showed a little bit better performance compared to some of the initial models. The three graphs illustrate the influence of hyperparameters and dataset size on model accuracy.

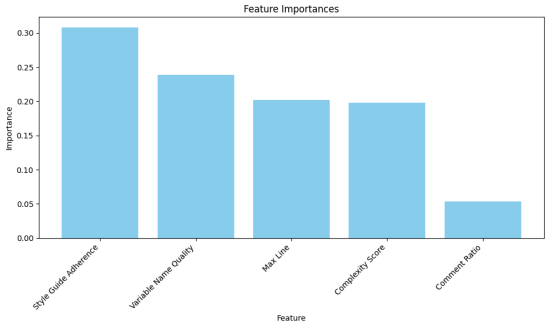

Fig. 14 - Feature Importances

The figure illustrates the feature importances determined by the Random Forest Classifier trained on the combined dataset. “Style Guide Adherence” emerged as the most influential feature, followed by “Variable Name Quality,” “Max Line,” and “Complexity Score.” The “Comment Ratio” had the least impact on predicting code readability, highlighting its comparatively lower significance in the model’s decision-making process.

6 Conclusions

In an attempt to devise a model that can accurately predict code readability, a subjective metric used to measure how well a human being can parse and understand code, we found some success. Although our initial accuracies and model performance was poor, we were able to modify our features and simplify our target classes such that accuracy reached 67%. While not as high as we had hoped, this score tells us that even a subjective matter of code readability can be predicted using simple features that can be obtained rather easily.

There were difficulties in collecting data instances, particularly from GitHub. Python 2 and Python 3 files are not easily distinguished as they use the same file extensions. Additional preprocessing was used to convert such files using the 2to3 utility, however several were unable to be converted. This left several instances with missing features (particularly cyclomatic complexity and style guide adherence). A modified script to better distinguish these files and get samples of Python 3 was created to mitigate this issue.

LeetCode samples were more easily collected via web scraping, but came with the limitation that we can only find many samples of a few questions, rather than few samples from many questions. This led to instances with similar feature values among samples of the same question, as implementations of the same questions were more similar than we had hoped. In particular, many LeetCode questions auto-generate comments describing the question itself inside the editor that gets left in the submission. These kinds of comments do not contribute to readability, but can get picked up by the features.

The greatest challenge was the inherent subjectivity of the target metric itself: code readability. Readability is influenced by individual preferences, experiences, and biases, making it difficult to define objectively. While we found success in addressing this by using features such as cyclomatic complexity and style guide adherence, they cannot perfectly capture the nuances of human preference.

For further details and access to the code used in this research, please visit our GitHub repository at [Code Readability Project] (https://github.com/easyasme/Code_Readability_Project.git).

7 Future Work

It would be hard to conclude that the current models would be able to bring about the desired outcome of this project which is to identify how one could improve the readability of their code. Although the accuracy of all the models sit around the 60 to 70% mark, it would be ideal if we could bring the accuracy above the 80% mark. There are various steps that can be taken to achieve this number.

For one, we can increase the number of features that the models can work on. If the models had more relative features to work on, then it would improve the accuracy of the model. Speculative speaking, the accuracy would improve greatly in our case because there are only five features that the current models are able to work with.

Another option would be to improve on the features themselves. The current features measure the logistic aspects of each observed criteria that is used when determining the readability of the code. For example, the max line length feature records the longest line of code found in the code. But this feature does not take into account whether the longest line of code was long because it was the developer's choice or if it was out of their control.

Similarly, the complexity feature cannot be used to determine if a code sample has many for-loops, while-loops, if-statements, etc. out of necessity to solve a complex problem, or because of suboptimal code that does not have a straightforward logical flow when it should.

These additional aspects of these features could be solved in one of either two ways. The first way would be to improve the code that extracts the feature score so that it includes more of the subjective nature of these criteria. The second would be the aforementioned improvement: adding more features. The additional features could represent the subjective aspect of these features. The main issue with both of these improvements would be detecting these aspects. If one wanted to use code to detect the subjective nature of these features, then one would have to implement some sort of pattern recognition software. If developing code to detect these aspects is not an option, then one would have to manually look at each line of code and determine the subjective aspects of these features. But given time, these improvements could certainly be made in order to improve the accuracy rating of the machine learning models.