Classifying Vulnerable C Functions using Text and Abstract Syntax Trees (AST)

Boden Chen
Professor Sriraam Natarajan
Computer Science
2025-05-13

Introduction

An initial analysis showed a significant imbalance in the dataset, with approximately 95% of the functions being non-vulnerable and a small percentage being vulnerable. This kind of imbalance is a known challenge in machine learning projects, as models can become biased towards the majority class. Due to constraints on the time available, particularly the time required to process and parse each function using a tool called `pycparser`, a decision was made to reduce the number of non-vulnerable functions processed. This was done instead of fully balancing the dataset through methods like oversampling, which would have increased the total number of samples requiring parsing. After this adjustment, the ratio of non-vulnerable to vulnerable functions was approximately 5:1.

During the data preparation phase, it was observed that a number of functions from the dataset could not be successfully parsed by `pycparser`. This meant these functions could not be used in the training or evaluation of the models, reducing the total available data. Why this happened could be due to various reasons, such as the specific syntax used in those C functions not being supported by the parser version, or errors within the function code itself.

Initial attempts to train machine learning models on this processed data showed poor performance, particularly in identifying the vulnerable functions. This is likely because the models were still heavily influenced by the large number of non-vulnerable examples and struggled to learn the patterns associated with the less frequent vulnerable class. To address this, a model parameter called class_weight was adjusted. Setting this parameter to 'balanced' typically instructs the model to give more importance to the minority class during training. After implementing this change, model performance improved significantly in classifying the vulnerable functions.

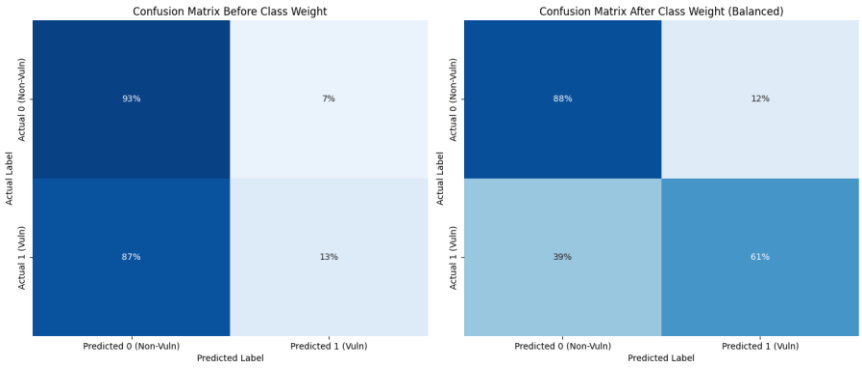


Fig 1. Initial results with a Logistic Regression model. All confusion matrices in this report will use % instead of count due to dataset imbalance.

In this specific problem of vulnerability detection, the consequence of a false positive (classifying a non-vulnerable function as vulnerable) might be considered less severe than a false negative (classifying a vulnerable function as non-vulnerable). Because of this aspect of the problem, giving even higher weight to the vulnerable class during training could be explored further to prioritize finding as many vulnerable functions as possible, even if it means increasing the number of false positives.

Given the significant imbalance in the data, using standard accuracy as the primary evaluation metric would be misleading. A model that simply classified everything as non-vulnerable would achieve a high accuracy score (around 83% with the 5:1 ratio) but would be useless for identifying vulnerabilities. Therefore, for the purpose of this report, the performance of the models will be assessed using the F1 score for the vulnerable class (label 1). This allows for a more accurate understanding of how well the models are actually identifying the vulnerabilities.

Feature Extraction

Manual Feature Extraction

For manual feature extraction, we used a combination of regex and calculations to return multiple number based results. A potential issue with this approach is that it depends heavily on our initial ideas about what features matter.

For feature selection, two methods were used: variance and mutual information. Variance is a simple measure that tells us how much a feature's values change across the different functions in our dataset. Mutual information is a more complex measure that tells us how much information a feature provides about the class label.

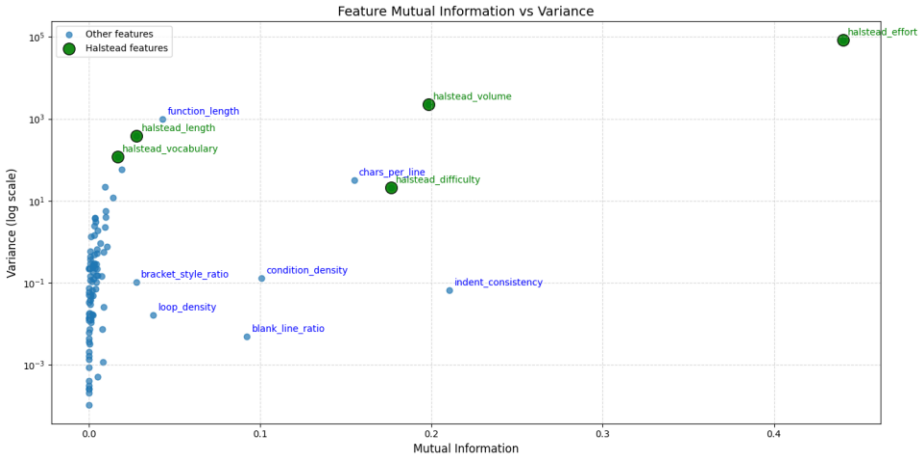


Fig 2. A plot of the variance and mutual information of extracted features. Note that variance is logarithmic.

Analyzing the extracted features, Halstead Effort proved particularly useful, exhibiting high mutual information. This metric, reflecting code complexity and the effort to understand/modify it, was expected to correlate with vulnerability, as more complex code can indicate potential errors. Conversely, simpler measures like function length and vocabulary size, despite their variability, showed low mutual information with the outcome.

A notable observation was that the 'volatile' keyword feature was zero for all functions in the cleaned dataset. This is likely because 'volatile' is uncommon in the types of C functions analyzed. As a feature that provides no distinction across samples, it was excluded as redundant.

Based on these findings, features with very low variance or very low mutual information were removed. Low variance features offer no discriminatory power, while low mutual information features lack a strong relationship with the target variable. The resulting filtered feature matrix was saved and prepared for combination with the TF-IDF matrix using hstack.

TF-IDF Vectorizer

In addition to manually created features, automated methods of TF-IDF vectorization were used extract features directly from the function code. TF-IDF stands for Term Frequency-Inverse Document Frequency, which gives numerical values to terms based on how often they appear in a single function and how rarely they appear across all functions. The idea is to identify terms that are important and specific to certain functions. Applying this method to code was challenging as it was difficult to capture the complexity and specific patterns within the code that might indicate vulnerability. It is hypothesized that this feature extraction is the most significant bottleneck in performance.

When experimenting with the hyperparameters, max_features (which limits the total number of unique terms) was found was found to have a direct correlation with performance and was therefore set to None. Other settings, such as ngram_range (which controls whether we look at single terms or sequences of terms) and the analyzer (which determines whether we treat code as sequences of

'words' or 'characters'), were harder to decide upon and depended on the two main ways to represent the code before vectorization. One was using the original string of the code, and the other was first converting the code into an Abstract Syntax Tree (AST), which is a representation of the code's structure. We then applied the vectorizer to either the original string or the AST representation.

A crucial step for reliable results was ensuring the dataset was split into training and testing sets before applying the vectorizer. Vectorizing the entire dataset first would lead to data leakage: information from the test set could influence the vectorizer's vocabulary and weights. This contamination, known as data leakage or cross-contamination, results in the model appearing to perform better than it would on truly unseen data.

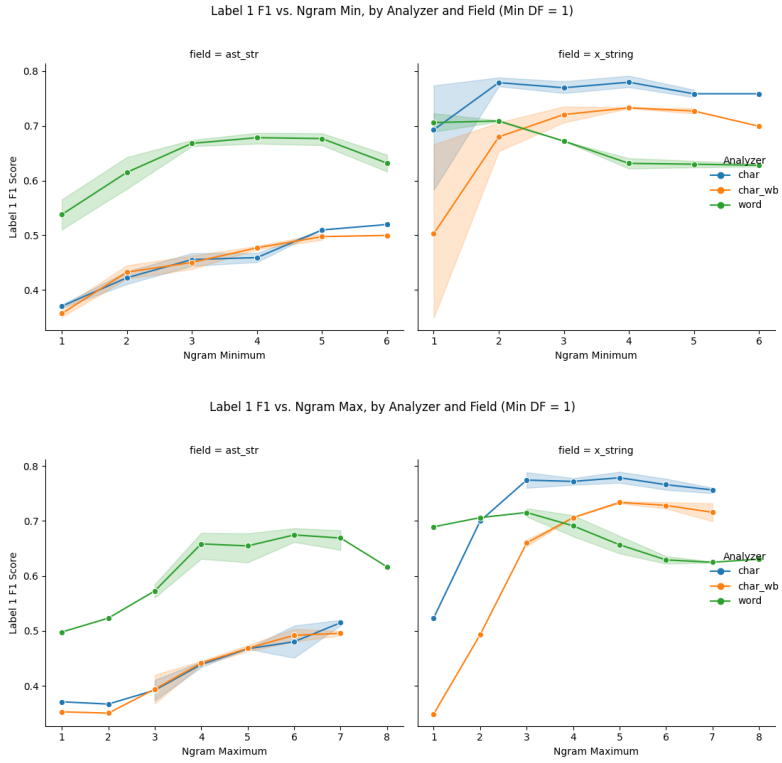


Fig 3. Representation of the ngram_range and performance based on the Label 1 F1 performance. This was measured through an initial SVM due to its speed of training.

When we used the AST representation with a 'word' analyzer, the resulting features performed notably better in initial tests compared to many others. This suggested that capturing the structural elements of the code through the AST was very helpful. However, we also found that vectorizing the original string using a 'char' analyzer (looking at character sequences) could sometimes achieve similar performance levels, although this approach generally required more memory to handle the large number of possible character sequences. The reason AST + 'word' might have performed better initially is likely due to the 'words' in an AST often representing meaningful programming constructs, while character sequences in the raw string might include a lot of less relevant combinations, although some useful patterns can still be found.

Ultimately, despite the promising initial performance of the AST representation with the 'word' analyzer and its potential to create features focused on keywords and structure with possibly lower memory usage, it was decided to proceed with vectorizing the original code string using the 'char' analyzer due to its performance increase. Merging the 2 matrices together was also attempted but resulting in negligible if not lower results. The other hyperparameters were optimized using the BayesSearchCV with a pipeline which will be explained in the next section.

Training

While the feature extraction and vectorization took the most amount of manual testing and time, the majority of training and optimization of hyperparameters was automated. This was due to the efficiency and performance of BayesSearchCV from the SciKit-Optimize library with a processing pipeline. Unlike other searches provided by SciKit, this library attempted to build a model that estimates the performance of hyperparameters and could decide which combination to try next. This repeats until it converges on an optimal combination of hyperparameters. BayesSearchCV worked incredibly well, increasing the detection capabilities of logistic regression especially.

SVM Algorithm

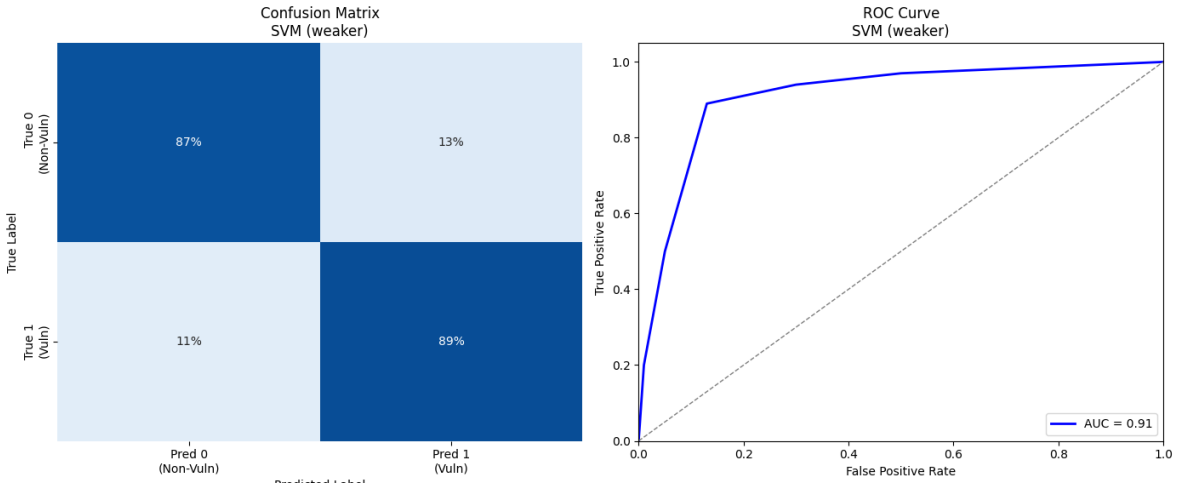


Fig 4. The final results of the optimized SVM. Label 1 F1 score was around 0.87.

The SVM works by finding a boundary or hyperplane that best separates the data points belonging to different classes and tends to not overfit on training data. In general, the SVM model showed good performance and speed on our dataset with an initial score of 0.79 and was therefore used as a simple measurement for testing initially.

An unexpected result was while the inclusion of the manually extracted features typically improved the performance of other models, it often made the performance of the SVM model worse. This suggests the SVM was not able to effectively use the information provided by the manual features in the same way other model types could. During the hyperparameter optimization process using BayesSearchCV, one parameter it could adjust was the relative weight given to the manually

extracted features compared to the TF-IDF features. The optimization process determined that the best performance for the SVM was achieved when the weight assigned to the manual features was very low, around 0.01, indicating the features were effectively ignored.

In the end, the hyperparameter optimization did not find significant changes other than the weight and the final performance was the lowest of all the models post-optimization. This is probably due to the SVM not being well-suited for the complex patterns which other algorithms excels at with the right parameters.

Logistic Regression

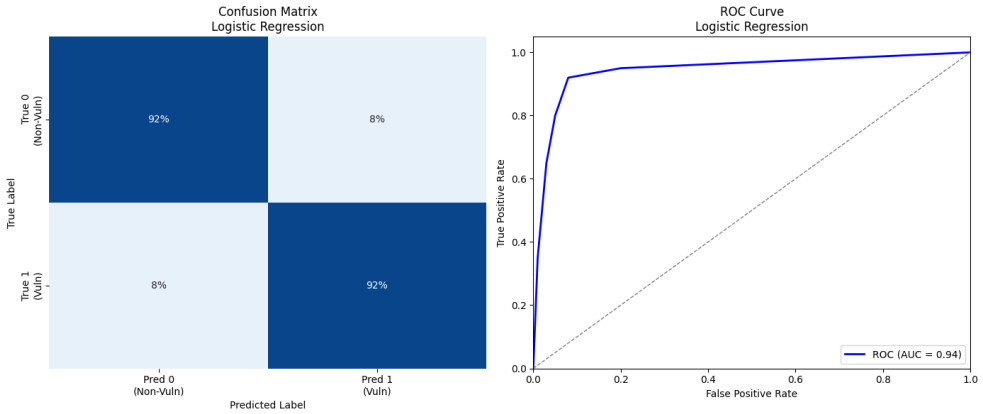


Fig 5. The final results of the optimized Logistic Regression. Label 1 F1 score was around 0.91.

While initially the performance of logistic regression was not as strong as the SVM and was disregarded, it showed the most significant performance increase by far after hyperparameter optimization using BayesSearchCV. We saw the F1 score for the vulnerable class (label 1), which is a measure that balances precision and recall, improve notably from an initial value of around 0.68 to approximately 0.91 after tuning. This substantial jump indicates that finding the right settings for the model's internal parameters was extremely important for its success on our dataset.

This large improvement from optimization is most likely due to the sensitivity of Logistic Regression, particularly when applied to potentially high-dimensional and complex feature sets like those derived from code using TF-IDF. Parameters related to regularization, which control how much the model is penalized for complexity to prevent it from overfitting to the training data, are often critical for Logistic Regression. The initial default settings for these parameters were likely not well-suited to balancing the need to capture the patterns in our specific feature set with the need to generalize well to unseen data. Most significantly, C was lowered possibly helping with the large dataset matrix.

Decision Tree

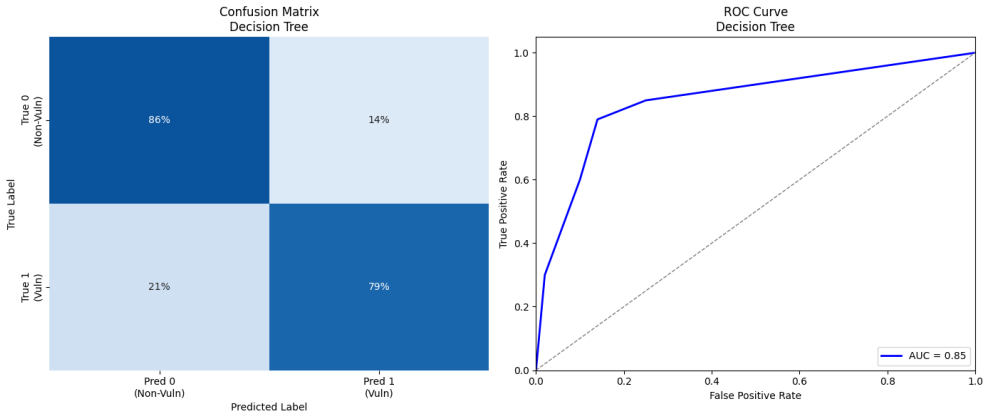


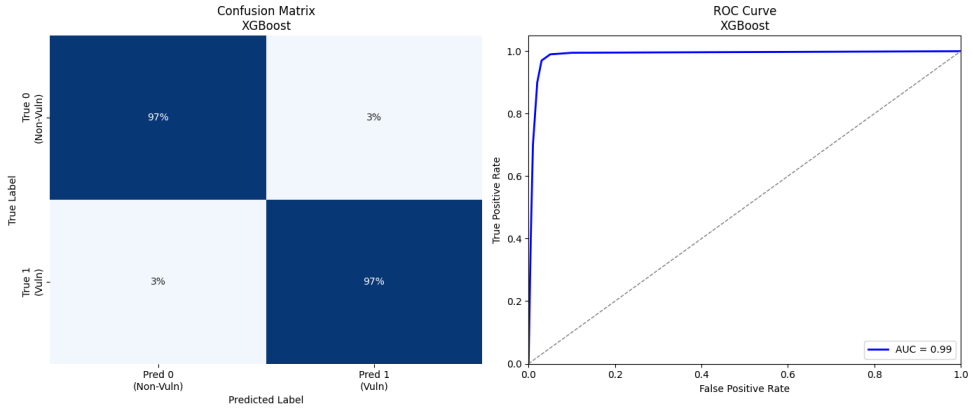
Fig 5. The **initial** results of the unoptimized Decision Tree. Label 1 F1 score was around 0.91.

This model was primarily used as a baseline to establish a starting level of performance for the ensemble methods, which is explained in the next. Despite our initial thoughts about its relative simplicity and its main role as a baseline for more complex approaches, the Decision Tree performed well on the classification task with an initial score of 0.82. This happened most likely because Decision Trees are effective at capturing non-linear relationships and interactions between different features, which are likely present in the features derived from code complexity and structure. The tree's ability to learn these complex rules allowed it to achieve a reasonable level of performance on its own. This led to the realization that non-linear models are likely better for this dataset. Hyperparameter optimization was not made on this as it was meant for the ensemble methods.

Random Forest

The Random Forest model, an ensemble method that combines multiple Decision Trees trained on different random subsets of the data through a technique called bagging. Despite the base Decision Tree model performing reasonably well on its own, the initial performance of the Random Forest was surprisingly bad. This poor initial result is possibly due to the bias-variance trade-off. Bagging methods like Random Forest are good at reducing variance by averaging the predictions of multiple models, which helps with overfitting. However, bagging does not significantly reduce the bias of the base models. The conclusion that bagging was not the correct choice and this model was disregarded.

Gradient Boosting



Based on a recommendation from Professor Natarajan, it was decided to explore Gradient Boosting models, which are another type of ensemble method. Gradient Boosting builds models sequentially, with each new model trying to improve the predictions by correcting the errors made by the previous ones. Initially, we planned to use the HistGradientBoostingClassifier available in the SciKit-Learn library. However, it requires the input feature data to be in a dense matrix format, meaning every value in the data grid must be explicitly stored. This became a problem because our TF-IDF feature matrix, created using the 'char' analyzer, was a sparse matrix with the majority being zero values. Attempting to convert this large, sparse matrix into a dense format required a significant amount of computer memory, which exceeded the available resources and caused memory issues, preventing us from using this specific implementation.

To overcome this limitation, we switched to using the XGBoost library, which also provides an implementation of Gradient Boosting. A key advantage of XGBoost is that it is designed to efficiently handle sparse input matrices directly without requiring them to be converted to a dense format. Using the XGBoost implementation of Gradient Boosting, we observed significantly better performance by a considerable margin. Notably, XGBoost was the only model that achieved an F1 score of over 90% for label 1 right from the initial, untuned experiments. Hyperparameter optimization further improved it by another 7%, possibly due to the multitude of hyperparameter available for XGBoost.

Conclusion

It is believed that the main bottleneck in this is in feature representation. Capturing the complex structure and behavior of code effectively remains challenging. While TF-IDF on character sequences performed well with certain models, exploring alternative ways to represent the code could be beneficial. Specifically, further investigation into using the Abstract Syntax Tree (AST) is warranted. Finding a more descriptive method to linearize or represent the AST. Overall, the most shocking results personally is the performance of XGBoost, showing that it highly suitable for this complex non-linear dataset. While this report was unable to obtain a >0.99 accuracy, it is believed there is a lot of possible improvements given more time.