Stylometric approach to code de-anonymization

Stefan Solarski
ESG Data Science
Ludwig-Maximilians-Universität
s.solarski@campus.lmu.de

Dušan Urošević
ESG Data Science
Ludwig-Maximilians-Universität
dusan.urosevic@campus.lmu.de

***Abstract*—De-anonymizing the authors of open source code presents a serious privacy and security threat to the otherwise anonymous contributors. At the same time, being able to identify a programmer using their code could help authorities find and prosecute authors with malicious intent. Our work is focused on the idea of de-anonymizing source code and identifying the programmer according to their coding style. We build several types of stylistic features by analysing raw text data as well as the code's abstract syntax tree. Using data from a programming competition, we are able to detect the author of the code with 95% accuracy. In this report we outline our findings regarding the usefulness of different feature types, classification models and their ability to generalize. We found that combining a random forest classifier with mutual information based feature selection, allows for accurate results in determining code authorship as well as fast training and inference.**

***Index Terms*—anonymization, authorship, stylometry, security**

I. Introduction

Can programmers leave fingerprints in their source code? It is possible that each programmer may have a distinct coding style, such as a preference for spaces instead of tabs, while loops instead of for loops, or deeply nested code. This might lead to important privacy and security implications, especially for those contributing to open-source projects, who might be trying to conceal their identity. Furthermore, code attribution can be useful in forensic contexts, such as the detection of ghostwriting and copyright disputes, or in discovering the identity of malware authors. Code de-anonymization is a computer security technique that has become increasingly prominent in recent years due to its ability to reveal the identities of anonymous users. By applying sophisticated data analysis and modeling techniques, code de-anonymization can effectively uncover the information that was originally hidden behind seemingly anonymous user profiles.

There are varying opinions about the ethics of using this method. Identifying contributors to some malicious piece of software might be of great benefit to government agencies and proactively protect online users and their data security. However, these modern methods of detecting authorship may also be used by mischievous actors to reveal anonymous users, who do not wish to share their identity. Our goal is to go over the idea of code stylometry, show that it is possible to detect stylistic differences between programmers, and even identify authors with very high success rate.

II. Motivation

We are looking at this problem from the point of view of a data scientist who wants to determine the programmer behind anonymous source code. We assume that they have access to code snippets from all possible authors and can build a classification model to select the most probable one. Now, by only using the source code itself as input data, they want to extract the stylistic features of the code and then de-anonymize the author. Here we mention a few possible scenarios that sparked our interest for this project.

Programmer de-anonymization works by analyzing the style and structure of a piece of code and comparing it to known coding styles from other programmers. For example, if an analyst was trying to determine who created Bitcoin's source code, they could compare samples from multiple programmers against the original version of Bitcoin's code and see which one shared the highest resemblance. This is a closed-world machine learning task as it assumes that all potential authors have been identified and there is no chance that another unknown author exists.

Software forensics is an important and often overlooked aspect of computer security. It involves the analysis of code to identify the author of a particular program, usually with the aim of attributing malicious software to its source. While traditional methods such as code similarity metrics are useful for this purpose, they are not always effective due to changing code structures or obfuscation techniques used by malicious actors. In such cases, programmer de-anonymization techniques based on stylometry can be used to narrow down a set of possible suspects.

Ghostwriting detection is another situation where the technique of de-anonymization can be helpful. In this scenario, we have a suspicious piece of code and one or more candidate code snippets that it may have been plagiarized from. Here, code stylometry can be applied to identify which piece of code most closely matches the suspicious sample, enabling us to identify individuals who might be involved in plagiarism or ghostwriting activities.

We will demonstrate that de-anonymization can work by extracting stylistic features using the abstract syntax trees. Additional use cases, as well as robustness to possible adversarial manipulation and other extensions are discussed in the last chapter.

III. Code stylometry

The goal of our project is to create a model that determines the most likely author of a source code file based on its contents. We take the Code stylometry approach, meaning that we capture the users' coding habits (style) through a feature set. These features can vary greatly in their complexity, including anything from the number of white spaces present in the code to the users preference for particular control structures and the way in which they are nested in one another. The questions that we answer are:

- which stylistic features are relevant for achieving good predictive performance
- does performance differ depending on the authors level of coding expertise
- how well can a model trained on one set of coding problems generalize to new problems

We first give an overview of the preprocessing steps that we are required to perform prior to feature extraction - tokenization and parsing. Then, we outline the three types of features that our model makes use of - layout, lexical and syntactic. Next, we perform feature selection and train two types of classification models. Finally, we compare the results of these models under different settings. The theoretical background for our work was first presented in Caliskan-Islam et al. [1] and our implementation is available at [2].

A. Feature engineering

Earlier methods for source code de-anonymization were based on standard natural language processing (NLP) techniques such as byte and N-gram frequencies. Although these models can capture certain aspects of an author's coding style, such as naming conventions and keyword preferences, they are language agnostic and thus do not capture habits that can only be understood with the additional context of a programming language's constructs. An example feature that can not be captured through N-grams is the author's preference for nesting code or using few/many operations in a single instruction. In order to model these types of features we tokenize and parse the original source code. The former provides us with additional information about individual tokens - such as whether they represent variable/function names, operations etc. while the latter captures the way in which the entire source code is structured in the form of an Abstract Syntax Tree (AST).

B. Abstract syntax Tree (AST)

The abstract syntax tree represents the output of a language-specific parser and forms the basis of our syntactic feature extraction. Figure 1 shows an example Java source code file and its equivalent AST. Note that the AST contains a node for each language specific construct present in the source code, for example, class and function definition as well as a function call. The leaf nodes represent the ”atomic” elements of the language such as function/variable names, literals, operations, and Java specific keywords. The inner nodes represent how these elements are combined to form complex expressions and statements. Performing only tokenization instead of parsing would result in the loss of these nodes. In our implementation, we use the javalang [3] library for Python in order to tokenize and parse Java code. Once we have obtained the tokens and the AST we proceed with the calculation of features [4] presented in the following section.

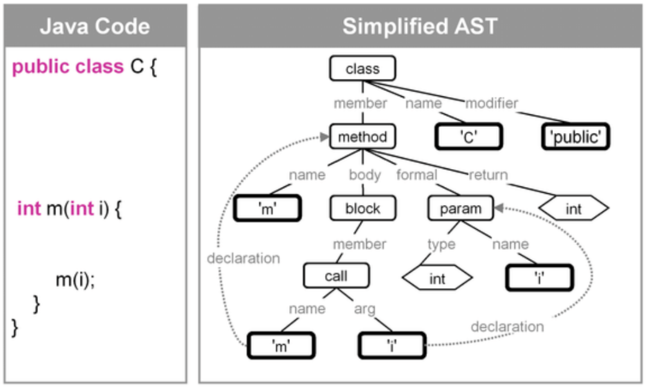

Fig. 1. Java Code and Abstract Syntax Tree

C. Types of features

In this subsection, we give an overview of the three types of features used by our classification models - layout, lexical and syntactic.

*1) **Layout features**:* correspond to previous NLP approaches where we treat the source code as regular text. Due to being language agnostic, they have the benefit of being applicable to any source code file, however, because of this they are also severely limited in their predictive performance. This can be seen more clearly in the Results and Insights section. For our model we used the following layout features:

- number of tabs, spaces, and empty lines in the code
- the proportion of the entire code that is white spaces
- presence of new lines before open braces
- average number of tabs per line

*2) **Lexical features**:* require tokenization of the source code, however they do not need the entire AST. These features are language dependent and represent coding habits such as the frequency with which a user organises their code in functions or their preference for a particular Java keyword. The lexical features that our model uses are:

Static

- proportion of Java keywords in the code
- relative frequency of tokens, literals, functions, and ternary expressions
- mean and deviation of line length
- mean and deviation of function parameters

Dynamic

- relative frequency of each Java keyword separately
- relative frequency of each unigram in the source code

Here dynamic features are those that can have different lengths depending on the code that they are extracted from. For instance, the frequency of unigrams feature will count the number of appearances of only those unigrams that are present in at least one of the source code files.

3) ***Syntactic features:*** describe the structure found in the abstract syntax tree. These features are invariant to the source code's layout and comments. For each source code, we create a separate AST from which we extract the following features:

Static

- maximum node depth in the AST

Dynamic

- relative frequency of node bigrams in the AST
- relative frequency of each node type in the AST

One particularly informative feature is the bigram term frequency which represents the number of occurrences of a parent-child relationship between different types of nodes in the AST. For example, how often we encounter a function call within a for-loop. For every source code file, we calculate all the aforementioned features and we combine them in a single feature vector. These vectors are then used as the input to our classification models, which return the most likely author of the code.

D. Feature selection - Mutual Information

The feature extraction procedure presented in the previous section produces a large feature set. While the layout features are all static, some of the lexical and syntactic features are dynamic, which means that their number increases with the size and diversity of the training set used. Using only 250 authors with 9 files each we produced about 7500 features. Many of these, such as the unigram and bigram frequencies, are very sparse. The extreme sparsity carries the risk of overfitting as well as limiting the number of useful splits in tree based classifiers. In order to counter this, as well as achieve good scalability, we introduce a selection criterion based on which we filter out the uninformative features. To determine how informative a feature is we use the mutual information (MI) criterion. [5] The mutual information criterion represents the decrease in entropy in the target value (author) as a result of fixing the value of a single feature. In other words, the amount of information the feature provides about the target value, on average. It is calculated as:

$$MI(Y, F_i) = H(Y) - H(Y|F_i)$$

where Y is the target variable, F_i is the i-th feature and H is Shannon entropy. The features with larger MI values have a greater influence on overall predictive performance and thus should be kept in the feature set.

By using it on a variable per variable basis we implicitly assume independence between the features with respect to their impact on the class label i.e. we ignore interaction terms. This conservative approach to feature selection guarantees that we only keep features for which we are certain an effect on the target variable exists.

IV. Implementation

In this section we will go over the original dataset that we used, the alterations made to it for our experiments, and introduce our choice of classification algorithms.

A. Data and Application

To train our models we are using data [6] from the programming competition *Google Code Jam* [7]. The dataset contains user submitted solutions from 2008 onward. The tasks are algorithmic problems typical in competitive programming. For our exploration, we chose to focus on code written in Java based on our familiarity with the language as well as the resources available for parsing Java code in Python. For the year 2009, the data consists of over 12 thousand source code files, written by almost 2 thousand unique users. In total there are 26 tasks split into 7 rounds. From this data, we construct a series of closed-world tasks. This means that during inference the only possible authors are those that have previously been observed during the model's training. Given that we are using submissions for a single year, we know that all the coders were solving similar tasks. Because of this, we can assume that the authors' code had the same purpose, and thus only varied by the style of coding. Undoubtedly, open-world problems, where we do not know who the author might even be, are more difficult to solve. However, given that our models can be adapted to solving open-world tasks by the introduction of a thresholding procedure (where none of the authors having a high likelihood signifies a new author), this does not invalidate the idea of code stylometry and de-anonymizing based on the author's style of solving problems.

B. Experimental setup

In order to answer the previously presented questions about feature relevance, the influence of authors' proficiency in coding, and the possibility of generalization in code authorship we construct a series of training and feature sets to evaluate the respective effects.

For determining feature relevance we use different combinations of feature types and feature set sizes during model training. These are:

- Layout features
- Layout features (extracted from C++ code)
- Layout + Lexical features
- Layout + Syntactic features
- Full feature set - keeping N features with the highest MI

The C++ code and its layout features were extracted for the purposes of performing a sanity check. Given its similarity to Java, as well as both languages' high representation in the dataset, we expect similar performance when using only the layout features. Next, in order to estimate the importance of each feature type we combine the lexical and syntactic features with the layout features and observe the resulting predictive performance. Note, that the language specific features do not have to be isolated from the layout features as they address different aspects of an author's coding style. Lastly, we extract the full feature set and observe how the predictive performance changes as we increase the number of features that we use - N. Feature selection is performed using the previously discussed mutual information criterion.

The next question we want to answer is whether programmers with higher levels of expertise are easier to identify compared to the average coder. Although the relation of a file's unique `problem_id` and `round_id` to the problem's difficulty isn't clearly defined, a workaround exists. The competition is organized in such a way that difficult exercises can only be attempted in higher levels (later rounds) of the competition. Based on this we can identify which problems had a higher difficulty based on the number of correct and attempted solutions submitted. Proficient authors are identified based on the fact that they have completed at least one of the difficult problems. To get a quantitative measure of the difference in performance we compare:

- The full feature set - extracted from random authors
- The full feature set - extracted from proficient authors

In order to evaluate our models' ability to generalize to code that solves previously unseen tasks we use the data from 2010 as a validation dataset. We only form predictions for those users who have previously been observed in our training data (competitors from Code Jam 2009). Unfortunately, this requires us to discard new dynamic features that were not observed in the 2009 data. Additionally, there is no guarantee that authors with the same `username` across different years of the competition are the same person. However, these same concerns would be present if we were to use data from other sources such as GitHub with the additional concerns of not knowing if the tasks were truly different from the ones in the training set, as well as a smaller validation dataset.

C. Classification models

We decided on using two different types of ensemble algorithms, random forest as a representative of bagging, and CatBoost as a representative of boosting classifier.

1) Random Forest Classifier: was mainly chosen because of its overall great performance and few hyperparameters that need to be extensively tuned. It is an ensemble learner, built by combining decision trees. Each tree is created from a random training sample of size N, where N is the size of the dataset. Additionally, to control the correlation between the decision trees, we sub-sample the features F. Commonly, a sub-sample of size $(log(F) + 1)$ features is created, without replacement. During classification each one of our test examples is run through the trained decision trees, making binary decisions at each node until a leaf node is reached. Once all the trees have their individual solution, we take that as their 'vote' and aggregate all the votes to get the result of the random forest classifier. This method is popular because it works with both categorical and numerical features, the fact that it requires very little pre-processing, and its solid performance with high dimensional and sparse data. [8]

2) CatBoost Classifier: is a third-party library that provides an efficient implementation of the gradient boosting algorithm. [9] Gradient boost is also an ensemble classifier that can use decision trees as building blocks. The idea of boosting is to iteratively combine weak learners, which are boosted to become one strong learner. We start by fitting a simple initial model to the data. The second model is fitted to accurately predict the cases where the first model performed poorly. Then, each following model's attempts to correct the shortcomings of the combined boosted ensemble model thus far. [10] The name *gradient* boosting comes from the fact that the objective of the model is to minimize the loss by adding weak learners (in our case trees) using a gradient descent procedure. [11] The main benefits of CatBoost over regular gradient boosting is the computational speed improvements, as well as its ability to work with categorical input variables.

V. Results and Insights

In this section, we give an overview of our results in each experiment. Most of our insights are drawn from validation accuracy on the different data sets. In order to make sure that our results are not distorted by biased and unbalanced train-validations splits, we analyzed our results by using stratified k-fold cross-validation. The idea behind using stratified cross-validation is to make sure that the properties of the training and test sets are identical across authors. We set k to 3 and averaged the results across the validation sets. By doing this we are expecting to get more representative results.

A. Feature performance

We create a sample of 2250 source code files (250 authors, 9 files each). Then, for each of these files, we perform tokenization and parsing. The resulting tokens and AST allow us to extract the different feature types. Next, we construct for each of the previously mentioned feature combinations a separate feature set. Based on the difference in the performance of the models trained on different sets we draw insights about the importance of different feature types. Figure 2 illustrates the validation accuracy for each feature combination using the top 1500 most informative features. We show a side-by-side comparison of performance achieved with a random forest classifier as well as CatBoost. Additionally, for the full feature set, we observe the proportion of information captured by each feature type as the number of total features is increased from 10 to 1500.

1) Joint performance: Using the best features from the full feature set we achieved 95% and 94% accuracy on random forest and CatBoost respectively. We note that there are no significant changes in performance by increasing the number of features beyond 750. As we can see in figure 3, after the first 1500 features we begin to see a drop-off in validation performance, likely due to overfitting. To get better insight into the importance of each feature type, we plot their contributions to the set of most informative features. We notice in figure 4 that the first 500 features account for most of the information gain and that after 500 the additional information gained from a single feature quickly approaches 0.

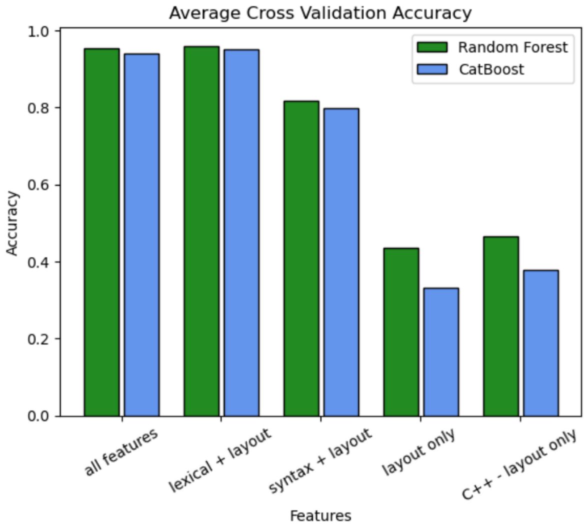

Fig. 2. Validation accuracy

2) Layout: Although the smallest feature set, layout features allow us to compare performance across languages, without any preprocessing of the code. As expected the performance using only the layout features is the lowest across feature sets. Using the random forest classifier we achieve a cross-validation accuracy of 43.5% on Java and 46.5% on C++. However, with the CatBoost algorithm, the accuracy was only 33.15% on Java and 38% on C++ code.

3) Lexical: The combination of lexical and layout features gives the best validation accuracy when the large feature set is used (1500 features). The random forest gets 96% while CatBoost is at around 95%. Figure 5 shows that lexical features only become the most relevant for large feature sets (1500 and above). We believe that this is tied to the number of authors in the training data. With 1500 lexical features and 250 unique authors the model should be able to learn ~ 6 user-specific features (for example variable names that are rarely used or unique text in print statements). While learning these types of details doesn't cause overfitting it is certainly not scalable to datasets with many unique authors.

4) Syntactic: Syntax features were consistently the most informative for medium sized feature sets. For a large number of syntactic features, the average validation accuracy is 82% for the random forest and 79% for the CatBoost algorithm. The reason for the drop in performance, compared to the full set and lexical feature set, could be due to the fact that many of the syntactic features such as Node bigrams are mandatory structures that result from Javas syntax. Because of this, the less informative syntax features are likely acting as noise variables.

B. Performance on difficult problems

In our attempt to find out whether expert programmers are easier to de-anonymize, we first had to identify the best programmers from the data set. We decided to define difficult problems as problems that had less than 30 correct solutions.

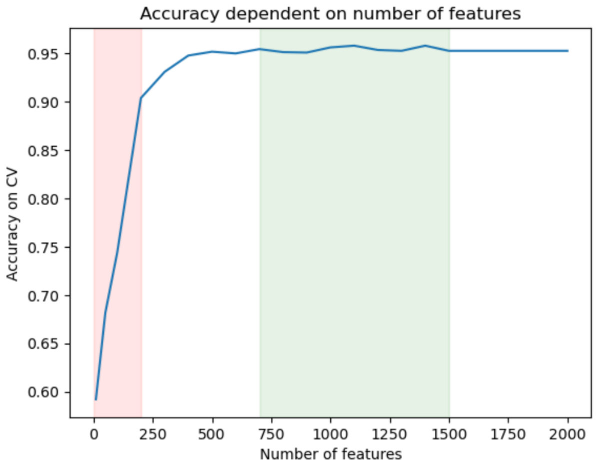

Fig. 3. Accuracy per number of features

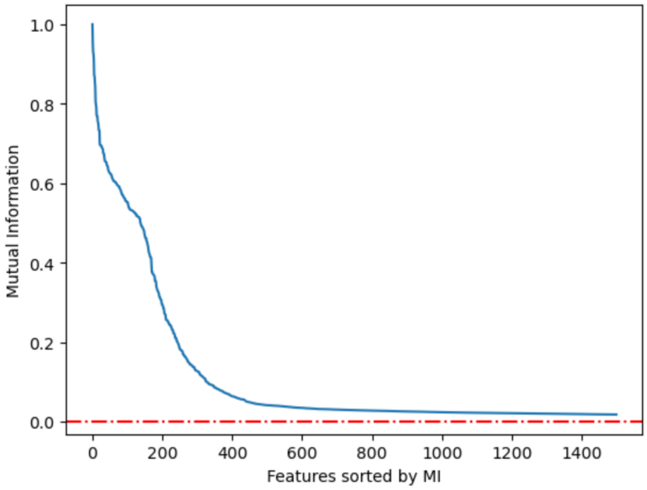

Fig. 4. Mutual information per feature

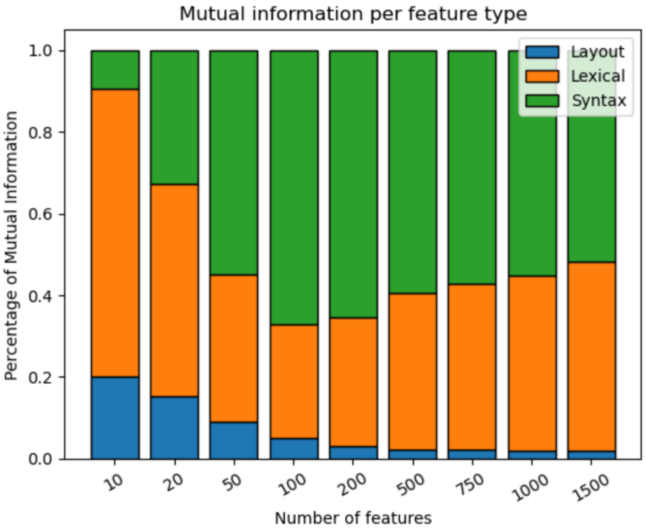

Fig. 5. Proportion of Mutual information.

We found exactly 14 such problems, which were solved by a total of 74 unique users. Then we sampled only those users that have at least 9 source code files, to be able to compare to our generalization results. From this dataset, we extracted a total of 2800 features, which were lowered to 1500 using mutual information. The final result using random forest on classifying the expert user is 97%. While this number is slightly higher than our baseline of 95%, testing on 65 random users with 9 files each gave us an even higher average result of 98%. Therefore, we could not conclude that programmers that have solved more difficult problems are easier to classify because of their higher level of coding. Instead, the increase in accuracy is most likely a result of the small sample.

C. Generalisation to unseen problems

To see if our model can generalize to code for previously unseen problems we train our classification models using the entire 2009 dataset and observe the accuracy in predicting authors for code from the 2010 competition (around 1700 source code files). The scores are 81% for random forest and 77% for CatBoost. Unfortunately, we can't guarantee that the same username across different years signifies the same person. As a result, we don't know what portion of the drop in performance can be attributed to the model overfitting to specifics of the tasks seen during training. It is possible that the models are simply failing to recognize new authors that are using old usernames.

VI. Conclusion

Code stylometry, as a method of code de-anonymization, can be applied in many different fields, from plagiarism detection in the academic setting to identifying individuals behind malicious software. Of particular interest, for data security experts, is the risk of infringement on the privacy of open source developers, who would prefer to remain anonymous. With our feature sets and classification models we were able to achieve up to 95% accuracy on a dataset of 250 programmers with 9 source code files per programmer. This is a significant improvement in performance compared to using only the language agnostic layout features (43.5%). Further analysis of the MI criterion reveals that lexical and syntactic features have the greatest contribution to the predictive performance of our models. Although lexical features achieve better performance on their own, syntactic features have the advantage of having better scalability to large datasets and capturing more complex stylistic choices. We show that generalization to new tasks is possible by evaluating the performance of our models on previously unseen problems with an accuracy of about 80%. Lastly, based on our results, we conclude that there is no significant improvement in performance when limiting the classification problem to highly skilled programmers.

A. Limitations

The model by design can not identify new authors. The closest it can get to solving an open world problem, on its own, is to detect that the author is not from the training set based on low probability scores. For an online learning system, our model would have to be coupled with a clustering procedure that would identify new users over time based on their code stylometry. Also, we assume that all code is given in the original format and that it has not been obfuscated in any way by the author [12]. For instance, using common formaters available in most IDEs would eliminate the information contained in the layout features.

B. Extensions

In order to get more reliable estimates of our models' generalization performance, a new dataset would have to be created. To improve upon the Code Jam dataset, the identity of the author of each file would have to be guaranteed and the corpus would need to be diversified by adding code that is not from the realm of competitive programming. Other extensions would include making it possible to identify the author across different programming languages and based on code snippets instead of entire source code files. The latter can be easily achieved for languages where a fuzzy parser implementation is available. These parsers are able to ignore individual tokens, which for a standard parser could cause a syntax error, in order to construct a partial AST. Applying these parsers to code snippets would allow us to extract the same feature sets as before. Knowledge transfer between programming languages poses a much harder problem. One approach would be to construct language specific lexical and syntactic feature sets. On top of being inefficient, this approach would also require an additional system for identifying the language being used. Alternatively, we could attempt to form a correspondence between concepts in different languages. However, it is unclear if this is even possible for languages from different paradigms (for example between functional Haskell and procedural C).

References

[1] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, p. 255–270.

[2] S. Solarski and D. Urosevic, "Data security project github repository," February 2023. [Online]. Available: https://github.com/dusan0098/DS_Code_Deanonymisation

[3] C. Thunes, "Javalang - pure python java parser and tools," 2023. [Online]. Available: https://github.com/c2nes/javalang

[4] Y. Rebryk, "De-anonymizing programmers via code stylometry." [Online]. Available: https://github.com/rebryk/code_stylometry

[5] A. Zhu, "Select features for machine learning model with mutual information," June 2021. [Online]. Available: https://towardsdatascience.com/select-features-for-machine-learning-model-with-mutual-information-534fe387d5c8

[6] "Kaggle - google code jam dataset," 2022. [Online]. Available: https://www.kaggle.com/datasets/jur1cek/gcj-dataset

[7] "Google code jam competition," 2022. [Online]. Available: https://codingcompetitions.withgoogle.com/codejam/about

[8] J. Brownlee, "Understanding random forest," June 2019. [Online]. Available: https://towardsdatascience.com/understanding-random-forest-58381e0602d2

[9] "Catboost github," 2023. [Online]. Available: https://github.com/catboost

[10] J. Brownlee, "A gentle introduction to the gradient boosting algorithm for machine learning," August 2020. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/8

[11] J. Hoare, “Gradient boosting explained – the coolest kid on the machine learning block.” [Online]. Available: https://www.displayr.com/gradient-boosting-the-coolest-kid-on-the-machine-learning-block/

[12] M. Appeltauer and G. Kniesel-Wünsche, “Towards concrete syntax patterns for logic-based transformation rules,” *Electronic Notes in Theoretical Computer Science*, vol. 219, pp. 113–132, 11 2008.