Program authorship attribution using machine learning methods

Kaituo Li and Yi Lu
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{kaituo,yilu}@cs.umass.edu

Mei Duanmu
Department of Mathematics
University of Massachusetts, Amherst
Amherst, MA 01003
duanmu@math.umass.edu

December 14, 2012

Abstract

Program authorship attribution has emerged as an important challenge. Past work demonstrated effective ways to find program authors by considering stylistic features. Although such prior techniques attempt to distinguish between programs written by a small set of authors with fair accuracy, their techniques suffer from small training data and redundant or irrelevant data in the training data. This work introduces the combination of stylish and textual features and feature selection to enhance our ability to tackle the training data issues. The combined technique increases the amount of features for modeling programs. The feature selection eliminate redundant and irrelevant features. We demonstrate that the the combined technique and feature selection result in practical benefit. In a corpus of 812 C and C++ source files comprising 8.65 MB of code by 9 different authors, we achieve higher accuracy (91.87%) than past techniques.

1 Introduction

Program authorship attribution is to classify programmers based on the ***consistent programmer hypothesis*** [5] that programmers are unique and that this uniqueness can be observed in the code they write. Program authorship attribution finds a set of characteristics of a programmer's program in terms of what remain constant for a significant portion of the programs that the programmers write. Furthermore, program authorship attribution can be used for in many applications such as plagiarism detection, malicious code analysis, author tracking, quality control, ownership disputes, etc.[9].

Despite highly desirable, training data issues can cause program authorship attribution techniques to miss the possibility of identifying true program authors. One issue is the amount of code for training is small. A small amount of code might not be sufficient to build a discriminating model for a programmer's coding characteristics. Another issue comes from the redundancy or irrelevances of the training data. For example, if a programmer has reused code written by others, the model we build cannot capture the programmer's coding characteristics accurately. For another example, it is simple to see that with programs altered by code formatters and pretty printers, the available features to characterize an author must decrease. In our implementation we address these challenges in two ways:

- We combine stylish and textual features. Specifically, we employ our style analysis system that scans programs, detect stylish features, such as blank lines or keywords with whitespace only on the left hand side, and increments the variables for each detected stylish features. The stylish feature values are then computed based on the values of these variables. Furthermore, we extract all the comments of a program file's code and use R's n-gram analysis to tokenize, or split the text of each comment into a series of n-grams. For example, the comment "drawing the cursor" can be subdivided into three 1-grams (drawing, the, and cursor), two 2-grams (drawing the, and the cursor), and one 3-gram (drawing the cursor). We then compute the frequency with which every n-gram in the program corpus appeared for each program file of the analysis. The frequency values resulting from this process indicate how often a particular n-gram appeared in a program file written by a particular programmer. Additionally, when we go through comments, we record those common grammar and spelling issues of a particular programmer. For example, if we find that a programmer have a lot of subject-verb inconsistency issues in their comments, we set the subject-verb inconsistency issues feature variable to be 1 for the programmer. By combining stylish and textual features, we get more information available to build a discriminating model for a programmer's coding characteristics: even in small examples we can produce many features, which we can increase the accuracy of the model we build.

- We do not fully use all features we collect, but instead keeping a subset of features that minimize error rate by removing many redundant or irrelevant features. Redundant features are those which provide no more information than the currently selected features, and irrelevant features provide no useful information in any context [1]. We performs feature selection to help select the most relevant features in model construction, which involves computing information gain (it measures the expected reduction in entropy or the uncertainty associated with a random feature) and ranking all features via information gain of each feature. By removing redundant and irrelevant features, we ensure that the error introduced by reusing code and code formatting are minimized.

2 Related Work

Literary document authorship attribution performs an authorship search by computing stylometric features such as bag of words and letter n-grams[3, 6]. Program authorship attribution is the analogy of literary document authorship attribution in the domain of software programs. Different from literary document authorship attribution, we are given a multitude of text from detailed specification in the form of comments, program statements, and meta-data or other details of program headers. This gives us ways to take advantage of such text from detailed specification through specification-specific features (e.g. proportions of operators with whitespace on both sides[7]).

Previous program authorship attribution can be divided into two categories: source-code based approach and binary-code based approach. Source-code based approach extracts feature data at source level. Spafford and Weeber[10] provides some features that could be used to identify program authors. Nevertheless, they neither offer classification methods to identify authors nor evaluate their method with any real micro- and macro-benchmarks. Based on this paper, Macdonell etc.[7] uses 26 stylish features extracted from 351 C++ source files by 7 different authors. They make the classification using feed-forward neural networks, multiple discriminant analysis, and case-based reasoning with accuracy from 81.1% to 88.0%. Our approach shares commonality with Macdonell etc.'s approaches in that both use stylish features. However, the salient novelty and difference of our approach is our novel way of using additional text features such as n-grams analysis on comments and spelling and grammar errors of comments to make the author characteristics more distinct and thereby achieves better accuracy (91.87%).

[1] http://en.wikipedia.org/wiki/Feature_selection

We also performs feature selection to help select the most relevant features in model construction, where Macdonell etc.'s approach would apply all features for training and testing. In addition, we use more diverse and larger real data benchmark for evaluation. We uses a corpus of 812 C and C++ source files (.c, .cc, .cpp, .h) comprising 8.65 MB of code by 9 different authors.

Binary-code based approach extracts feature data at program binary level. The challenge here is that much information available at the source code level is missing at the binary level such as indentation and formatting style. Rosenblum etc.[8] shows that stylistic features can be extracted from binary programs. Based on binary code representation on instruction-level and structural characteristics, they pick out features using instruction sequence idioms and interprocedural control flow graphs. They apply support vector machine on these features, achieving 51% accuracy on their benchmark. The low accuracy is understandable as much less information can be used as features. While our approach can only identify program authors at the source code level, it is interesting to see whether new features can be found to characterize a programmer style at the binary code level as well.

3 Our Approach

3.1 Data

3.1.1 Source Code

Our source code comes from open source c and c++ projects downloaded from *SourceForge*[2]. We selected 9 projects that totally have 812 valid source files (.c,.h,.cc,.cpp) written by 9 authors (exclude library files and files written in other languages). One project may be written by multiple authors. The number of authors of one project varies from 1 to 6. The number of valid source files of one project varies from 5 to 125. Details are listed in Table 1

Table 1: Data Source

# authors	#projects	#files	average#projects per author	average #files per author
9	9	812	2	90.2

3.1.2 Data Preparation

Each file (.cc,.c,.h,cpp) serves as a data unit. We labeled each file with an author. The labelling process was done manually, and if one file contains multiple authors, we simply ignore it. In the following section, we will introduce feature construction and selection process.

3.2 Feature Construction and Selection

When extracting features for classification, we consider two kinds of features, textual and stylish features. Textual features refer to the features extracted from textual part of the code. The comments of all the code are extracted as textual part. We separated the textual features into two parts, spelling and grammar, and N-gram sequence.

[2]http://sourceforge.net/

3.2.1 Text Features

N-gram sequence
An n-gram is a continuous sequence of n words of comments. Generally, different authors would like to use different combination of words. So we use the counts of n-gram sequence as features. For example,in the sentence 'Who is the author of the code'. Unigram will be 'who', 'is', 'the', 'author', 'of', 'the', 'code'. 2-gram sequences are 'who is', 'is the','the author','author of','of the','the code'. 3-gram sequences are 'who is the', 'is the author', 'the author of', 'author of the', 'of the code'. In our experiments, we chose unigram, 2-gram,3-gram,4-gram,and 5-gram. For each of them, we chose top 200 n-gram counts as features. We do the classification based on these textual features with different classifiers.
Spelling and Grammar
Many programmers have difficulty in writing correct sentences without any spelling or grammar mistakes. Misspelled words inside comments may be quite telling if the misspelling is consistent. Likewise, same grammar mistakes inside comments or print statements also provide a hint that these program may come from the same author.

To catch the characteristics of misspelling and grammatical mistakes, one idea is to categorize them. To achieve this goal, we use the library from *LanguageTool*[3], an Open Source proofreading software for English and other languages. In this tool, it divides misspelling and grammar mistakes into 24 categories (see Appendix 1). The counts of mistakes in each categroy serve as 24 features

3.2.2 Stylish Features

Different programmers have different styles of writing code. For example, how often does a programmer use keywords (such as if and while) with whitespace only on the left side? How often does a programmer use operators (such as "=" and "+") with whitespace on either side? These stylish features can help to identify the author of a program. In Appendix 2, we list 27 stylish features used in our experiments.

3.2.3 Feature Selection

Mutual information measures the information that two variables share: it measures how much knowing one of the variables reduces uncertainty about the other.

Feature selection based on mutual information of the features is a common procedure in document classification [8]. In our case, feature t is ranked by mutual information with the author class variable a, or formally

$$MI(t) = \sum P(a)P(t|a) \log \frac{P(t|a)}{P(t)} \qquad (1)$$

where A is the set of all possible author classes, and probabilities are computed using maximum likelihood estimation.

3.3 Methodology

Weka 3.7, a collection of machine learning algorithms for data mining tasks, are used for the analysis. The following classifiers are used: different classifier in LIBLINEAR, SVMs using different kernels in LIBSVM, MultilayerPerceptron, Naive Bayes, multinomial Naive Bayes, normalized Gaussian radial

[3]http://www.languagetool.org/

basisbasis function network (RBFNetwork), John Platt's sequential minimal optimization algorithm for training a support vector classifier (SMO), C4.5 decision tree (J48), decision tree based on the ID3 algorithm (JD3), logistic model trees (LMT), tree that considers K randomly chosen attributes at each node (RandomTree), and forest of random trees (RandomForest) . RandomForest classifier exhibited superior accuracy to other classifiers we have tried in terms of overall accuracy. Furthermore, RandomForest performs much faster with a calculation time of minutes compared to hours for classifiers like LMT and RBFNetwork.

Due to space limitation, we skip here the full details. We only introduce four methods: L2-loss linear SVM in LIBLINEAR, SVM with linear kernel in LIBSVM, Naive Bayes, and Random Forest.

3.3.1 Libsvm

Libsvm [2] is a library for Support Vector Machines(SVMs). Libsvm is L1-loss SVM.

L1-loss SVM solves the following primal problems:

$$min_{\omega}\frac{1}{2}\omega^T\omega + C\sum_{i=1}^{l}(max(0, 1 - y_i\omega^T x_i))^2 \tag{2}$$

The dual form is

$$min_{\alpha}\frac{1}{2}\alpha^T\hat{Q}\alpha - e^T\alpha$$

subject to $0 \leqslant \alpha_i \leqslant C$. $\hat{Q} = Q + D$, D is a diagonal matrix. $D_{ii} = 0$,$Q_{ij} = y_iy_jx_i^Tx_j$.

3.3.2 Liblinear

Liblinear [4]is optimized to deal with linear classification (i.e. no kernels necessary) and it is efficient for training large-scale problems. We use Libliner with L2-loss SVM(dual). Given training vectors $x_i \in R^n$, $i = 1, ..., l$ in two class, and a vector $y \in R^l$ such that $y_i = \{1, -1\}$, a linear classifier generates a weight vector ω as the model. The decision function is

$$sgn(\omega^T x) \tag{3}$$

We use Liblinear with L2-loss SVM (dual). L2-loss SVM (dual) solves the following primal problem:

$$min_{\omega}\frac{1}{2}\omega^T\omega + C\sum_{i=1}^{l}(max(0, 1 - y_i\omega^T x_i))^2 \tag{4}$$

The dual form is

$$min_{\alpha}\frac{1}{2}\alpha^T\hat{Q}\alpha - e^T\alpha \tag{5}$$

subject to $0 \leqslant \alpha_i \leqslant \infty$. $\hat{Q} = Q + D$. D is a diagonal matrix. $D_{ii} = 1/(2C)$,$Q_{ij} = y_iy_jx_i^Tx_j$.

Different from Libsvm, Liblinear do not use the loss term, so the results of Liblinear and libsvm are similar, but not exactly the same.

3.3.3 Naive Bayes

Naive Bayes belongs to Bayes classification. Naive Bayes assumes all input features are conditionally independent. Training step for Naive Bayes is easy and fast, and testing is straightforward.

In the learning step, for each label a_k, estimate $P(A = a_k)$ with instances in the training set. For each attribute value x_{ij} of each attribute x_i, estimate the conditionally probability $P(X_i = x_{ij}|A = a_k)$ with observations in training set.

In the testing step, for unknown instance X, look up tables to assign the label a^* to X if $\hat{P}(a_1|a^*) \cdot \cdots \cdot \hat{P}(a_n|a^*) > \hat{P}(a_1|a) \cdot \cdots \cdot \hat{P}(a_n|a)\hat{P}(a)$, $a \neq a^*$. In this way, we can maximum a posterior and find optimal label for the testing instances.

3.3.4 Random Forest

Random Forest [1]is an ensemble classifier which consists of many decision trees and outputs the class that is the mode of the classes. It is a substantial modification of bagging. It builds a large collection of de-correlated trees, and then averages them.

It is a very accurate learning algorithm.In addition, it runs efficiently for large datasets.

4 Experiments and Results

As described in section 3.1, our data sets contain 812 valid data units, and involve totally 9 authors. After labelling the data, we extracted our features with the procedures in section 3.2.

To find the best classifier, we experimented with several methods include LibSVM, LibLinear, Navie Bayes and Random Forest as described in section 3.3.To compare the classification accuracies of textual features and stylish features, we conducted several groups of experiments. The results are shown in the sections below. We used a 7-fold cross-validation to evaluate accuracy of each classifier. In addition, we did a group of experiments to evaluate the effect of feature selection.

4.1 Textual Features

At the beginning, we did the classification using textual features only. Results are shown in Table 2 and Fig. 1. In our experiments, We not only included the n-gram, but also included the combination of them. The combination 1-n gram are consist of 1 gram sequence, 2 gram sequence, until n gram sequences. For example, 1-2 gram includes all the 1 gram sequence and 2 gram sequence.

From the result of 1-gram to 5-gram, we found that 2-gram sequences and 3-gram sequences are best among them. The results shows that with the increasing of n , the classification accuracies increased until a moderate size (n = 2 or n = 3 here), then decreased. Another observation is the results from 2-gram and 3-gram, 84.73% is pretty good.

We also conclude that a combination of n-gram features will generally increase the classification accuracies, which can be observed from that 1-n gram is generally better than the n-gram.

Among these classifiers, Liblinear with L2-loss dual has the best results, which is 88.18%. SVM with linear kernel also performs very well here. Ranodm Forest is slightly better than SVM with linear kernel. The results of Naive Bayes are not well, and the reason may be that naive Bayes assumes independence among features and our features may not independent.

In conclusion,the textual features alone can make a good classification given that the best result is 88.18%.

In addition, there is a limitation that textual features can be absent in a program. For example, there are not comments and output in a program, especially when the program is short.

Table 2: Textual Part

Classifier	1-gram	2-gram	3-gram	4-gram	5-gram	1-2 gram	1-3 gram	1-4 gram	1-5 gram
Liblinear(L2-loss dual)	83.99	84.73	84.73	79.93	73.28	88.18	88.05	88.18	87.93
SVM(linear)	81.28	84.48	83.87	78.94	72.66	85.10	84.98	84.98	85.47
Naive Bayes	43.84	65.89	68.10	62.81	59.98	53.33	56.65	59.24	70.81
Random Forest	85.47	86.47	86.08	84.61	84.61	85.96	85.22	86.58	87.44

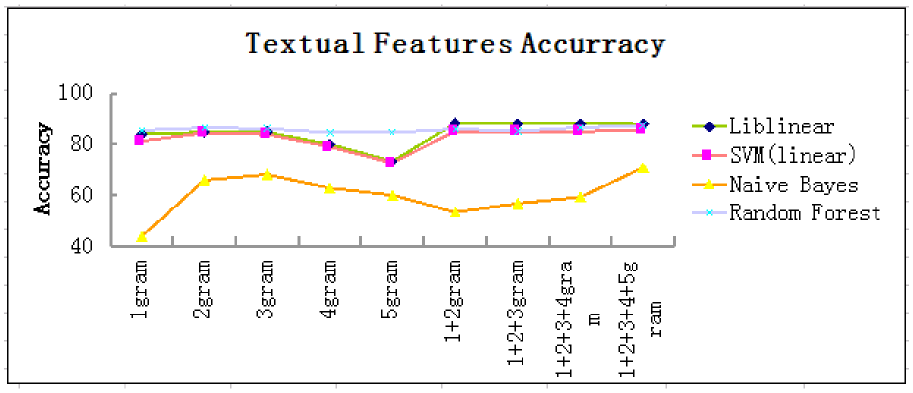

Figure 1: Textual Part

4.2 Stylish Features

Table 3 shows the result of stylish features. We can see that the classfication accuracy is not well, the best result is Random Forest, 71.7%.

Table 3: Experiment Results Using Stylistic Features

Classifier	Accuracy[%]
Liblinear(L2-loss dual)	63.1
SVM(linear)	62.4
Naive Bayes	56.7
Random Forest	71.7

4.3 Textual&Stylish Features

In this section, we combined textual features and stylish features to see the classification accuracy. Results are shown in Table 4 and Fig. 2.

Compared with the results with previous sections, we found that the combination improved the accuracies. Here the best result is 91.87%, which is better than 88.18%.

The best result of Liblinear(L2-loss dual) is 1-5 gram. For SVM with linear kernel, the best result comes from 1-2 gram. Random Forest reached best performance at 4-gram, however, it decreased as more features added. The results of Liblinear, LibSVM and Random Forest are all very good, and Libliner is the best.

Though stylish features alone don't make a good classification, after integrating it with stylish features, it help to improve the classification accuracies. So stylish features are also important in authorship attribution, especially when textual features are absent in the program.

Table 4: Textual and Stylish Part

Classifier	1-gram	2-gram	3-gram	4-gram	5-gram	1-2 gram	1-3 gram	1-4 gram	1-5 gram
Liblinear(L2-loss dual)	89.41	89.53	89.41	88.30	87.07	91.13	91.13	91.13	91.87
SVM(linear)	87.93	87.68	87.93	86.21	85.59	89.66	89.66	89.66	89.66
Naive Bayes	64.41	77.46	76.97	76.35	72.29	68.23	70.44	71.67	78.45
Random Forest	87.81	88.92	89.04	89.04	84.61	86.95	88.55	86.08	87.44

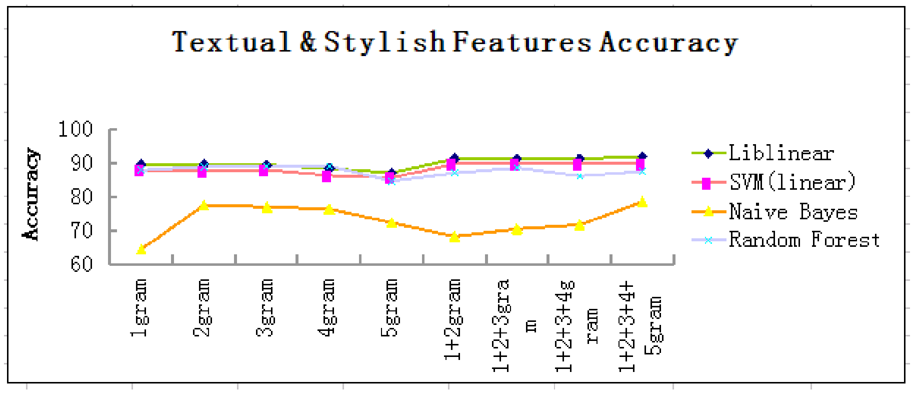

Figure 2: Textual and Stylish Part

4.4 Selected Features

There are totally 1052 features in our expermiments. We performed feature selection as the procedures outlined in section 3.2.3.

As shown in table 5, after ranking the features according to the mutual information, the dominat features are texutal features. Actually, no stylish feature appeared in top 100. But all the 27 stylish features are among top 600.

Table 5: Top Features

Features	top 50	top 100	top 200	top 300	top 400	top 500	top 600
#Textual Features	50	100	196	292	387	484	573
#Stylish Features	0	0	4	8	13	16	27

We also conducted a group of experiments using the selected features. The results indicated that in top 600 features, we got the best result, that is 91.87% using Liblinear with L2-loss dual.

In the Liblinear(L2-loss dual), the experiment of top 600 features had highest accuracy, but no further improvement with the increase of features. For SVM with linear kernel, it had best performance with top 800 and top 900 features, but it decreased with more features added. Random Forest reached it best performance with top 400 features, however, it decrease with more features added. Naive Bayes also decreased with more features after reach best performance top 400 features.

The results from Libliner, SVM(linear kernel) and Random Forest all had good results, and Libliner is the best.

Table 6: Selected Feature Accuracy

Classifier	50	100	200	300	400	500	600	700	800	900	1000	1052
Liblinear(L2-loss dual)	69.83	66.50	85.59	89.41	91.01	91.38	91.87	91.87	91.87	91.87	91.87	91.87
SVM(linear)	68.97	69.09	83.99	88.30	88.79	89.66	89.41	89.41	89.78	89.78	89.66	89.66
Naive Bayes	61.33	61.82	68.72	79.56	80.30	79.06	78.45	78.45	78.57	78.57	78.57	78.45
Random Forest	76.60	77.22	87.93	88.18	89.29	88.18	87.81	88.42	86.82	89.29	88.42	87.44

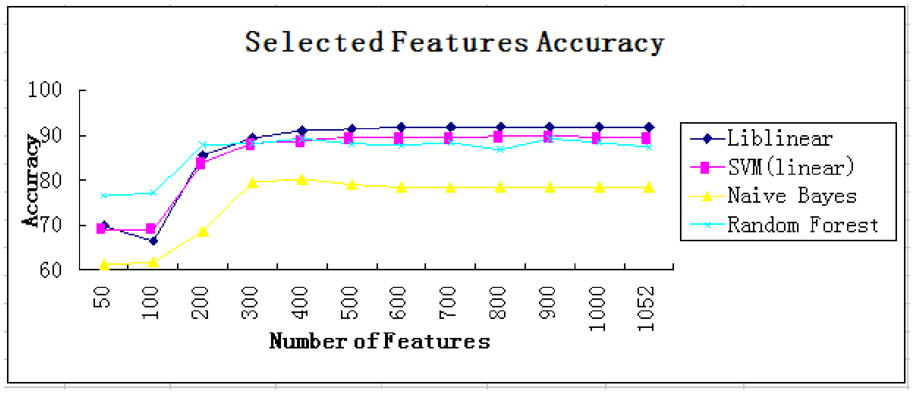

Figure 3: Selected Feature Accuracy

5 Conclusion

We presented a program authorship attribution approach that combines stylish and textual features. Our approach also contains a step of feature selection by removing redundant or irrelevant features. Our approach can help derive more features, while ensuring these features are relevant. We have demonstrated our approach can achieve high authorship attribution accuracies with a subset of features for use in model construction.

References

[1] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[2] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[3] Joachim Diederich, Jörg Kindermann, Edda Leopold, and Gerhard Paass. Authorship attribution with support vector machines. *Applied Intelligence*, 19(1-2):109–123, May 2003.

[4] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[5] J.H. Hayes and J. Offutt. Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability*, 20(4):329–356, 2009.

[6] Moshe Koppel, Jonathan Schler, Shlomo Argamon, and Eran Messeri. Authorship attribution with thousands of candidate authors. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 659–660, New York, NY, USA, 2006. ACM.

[7] S.G. Macdonell, A.R. Gray, G. MacLennan, and P.J. Sallis. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *Neural Information Processing, 1999. Proceedings. ICONIP '99. 6th International Conference on*, volume 1, pages 66 –71 vol.1, 1999.

[8] N. Rosenblum, X. Zhu, and B. Miller. Who wrote this code? identifying the authors of program binaries. *Computer Security–ESORICS 2011*, pages 172–189, 2011.

[9] N.E. Rosenblum. *The provenance hierarchy of computer programs*. PhD thesis, UNIVERSITY OF WISCONSIN, 2012.

[10] E.H. Spafford and S.A. Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12(6):585–595, 1993.

Appendix 1

Rules in LanguageTool:

1. "HAVE_PART_AGREEMENT",
2. "PHRASE_REPETITION",
3. "COMMA_PARENTHESIS_WHITESPACE",

4. "DOUBLE_PUNCTUATION",
5. "NON3PRS_VERB",
6. "MASS_AGREEMENT",
7. "COMP_THAN",
8. "TO_NON_BASE",
9. "DT_DT",
10. "WHITESPACE_RULE",
11. "EN_UNPAIRED_BRACKETS",
12. "EN_A_VS_AN",
13. "THREE_NN",
14. "UPPERCASE_SENTENCE_START",
15. "EN_QUOTES",
16. "GENERAL_XX",
17. "ENGLISH_WORD_REPEAT_BEGINNING_RULE",
18. "MORFOLOGIK_RULE_EN_US",
19. "HE_VERB_AGR",
20. "ENGLISH_WORD_REPEAT_RULE",
21. "POSSESIVE_APOSTROPHE",
22. "ALL_OF_THE",
23. "CD_NN",
24. "BEEN_PART_AGREEMENT"

Appendix 2

1. Proportions of lines that are blank.
2. Proportions of keywords with whitespace on both sides.
3. Proportions of keywords with whitespace only on the left side.
4. Proportions of keywords with whitespace only on the right side.
5. Proportions of keywords with whitespace on neither side.
6. Proportions of keywords with operators on both sides.

7. Proportions of keywords with operators only on the left side.
8. Proportions of keywords with operators only on the right side.
9. Proportions of keywords with operators on neither side.
10. Average number of characters per non-comment line.
11. Average number of letters that are upper case per non-comment line.
12. Proportions of lines that contain only line comments.
13. Proportions of lines that contain only close comment (/*).
14. Proportions of lines that start with start comment (/*).
15. Proportions of lines that begin with opening bracket ().
16. Proportions of lines that contain only opening bracket.
17. Proportions of lines that contain only closing bracket ().
18. Proportions of keyword ”new”.
19. Proportions of keyword ”const”.
20. Proportions of keyword ”for”.
21. Proportions of keyword ”if”.
22. Proportions of keyword ”static”.
23. Proportions of keyword ”struct”.
24. Proportions of keyword ”switch”.
25. Proportions of keyword ”while”.
26. Proportions of keyword ”else”.
27. Average McCabe’s cyclomatic complexity number per method. If there is no method in a source file, the cyclomatic complexity number is 0.