# Building Classifiers to Identify Split Files

Pam Green, Peter C.R. Lane, Austen Rainer, and Sven-Bodo Scholz

University of Hertfordshire, School of Computer Science,
College Lane, Hatfield, Herts, AL10 9AB
{p.d.green,p.c.lane,a.w.rainer,s.scholz}@herts.ac.uk

**Abstract.** We apply machine-learning techniques to help automate the process of mining the version history of software projects. Analysis of version histories is important in the study of software evolution. One of the associated problems is tracing program elements which have changed or moved as the result of file restructuring. As an initial application, we have developed classifiers to identify one such type of file change, 'split files'. Our process involves extracting features through syntactic analysis of the original source code, and then training and evaluating classifiers against a set of data assessed by visual inspection. We analysed 266K files from 84 open-source projects, filtering out a set of candidate files for which our classifiers achieve either 89% overall accuracy, or a false positive rate of 5%.

**Key words:** classification, ensemble learning, feature selection, text mining, source code analysis, software evolution

## 1 Introduction

Mining software repositories for information about the evolution of software code is an active field of research. The large volume of open-source code available provides a rich supply of material for data mining. Software engineers study the mined information to gain greater understanding of the software development process, and the maintenance and design of software systems. The studies include measuring software qualities in the evolving system [1–3]; analysing past transactions to provide recommendations to users [4, 5]; discovering patterns in source code to predict faults [6, 7]; or using change patterns to identify defect-causing changes [8–10], co-changing files [11, 12], or to predict refactorings [13].

As software evolves in response to changing requirements, to upgraded platforms, or for fault correction, it tends to become more complex [14]. Periodic restructuring is therefore required to simplify the code. This may take the form of moving, renaming, merging, splitting or recombining files. However, restructuring often occurs under time or cost constraints which may leave documentation incomplete or absent, leading to difficulty in tracing program elements. For example, a file which is renamed, or merged with another file, will seem to 'disappear'. It is important to be able to follow these changes, both to aid subsequent maintainers of the system and to provide continuity between elements in version histories. Godfrey and Zou [15, p.1] state that “detecting where merges and splits have occurred can help software maintainers to better understand the change history of a software system” and “that having an accurate evolutionary history that takes structural changes into account is also of aid to the research community”.

One way to identify or predict such changes is to study those which have taken place in other projects, and from these infer a model or typical set of criteria for identifying changes from other properties of the software project. However, there are no datasets of this type readily available in the public domain.

There are several challenges when mining software repositories. The first is the large volume of source code that may be considered. A moderately sized project may contain 10K-100K lines of code, and this can be multiplied by the number of development versions through which it evolves. Coupled with this volume is the relative scarcity of the items of interest. The number of times a particular type of complex change occurs in a single project’s lifetime will be small. Finally, there is the challenge of identifying and recognising these changes [16], a task which is frequently manually intensive [17, p.1], [18, p.5].

The aim of this study was to explore whether data-mining and machine-learning techniques could automate part of the identification process, and so facilitate the analysis of large repositories of source code. In particular, we are interested in techniques which analyse the source code within the projects, and do not employ information-rich features. To test the validity of our approach, we focus on split files in this work. Split files are those files from which blocks of code are removed and placed elsewhere in the system, either in a new file or in a more suitable location, such as a utility file.

Our procedure involved two separate stages: filtering candidate split files and constructing features to build classifiers. We employed Ferret [19–21], a text-based tool for comparing source code files, during the filtering stage. Filtering involved detecting both candidate split files and possible target files for the extracted code. Ferret computes the Jaccard coefficient of similarity between pairs of files, based on token trigrams. This measure yields a value between 0 (no similarity) to 1 (identical). The advantage of Ferret is its high level of efficiency, with processing time approximately linear in the amount of input code.

Having pinpointed the candidate files and related target files, we constructed features based on the files and the relationship between them, using values output by Ferret and by Duplo [22]. Duplo is an open-source text-based tool which detects copied blocks in software by matching hashed lines of code. The features were used to build classifiers to determine whether the candidate files had been split. We inspected every group of files visually to label the candidates. The Weka [23] system provided the infrastructure and algorithms for conducting the machine learning experiments.

To summarise, the process is: first, filter out obvious non-split files, leaving a set of candidate files which may or may not be split files; second, construct a set of features based on the text of the source code; and third, train a classifer to determine whether a candidate file is or is not a split file. Two outcomes of this work are: first, a feature construction and classification tool, which can be used to identify split files in new code; and second, an automated method for marking up datasets, so generating information which would be of use in software evolution research.

We define split files in Section 2, the method used for identifying these files is described in Section 3, the outcome in Section 4 and Section 5 concludes.

## 2 Defining Split Files

We use the terms below to refer to the different types of files in our analysis. The *original* file is the file in release $n$ which may be split. The *amended* file has the same name as the original file, but follows in release $n + 1$. The *new* file or files are those files created in release $n + 1$ which are related to the original file.

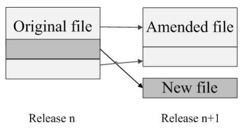

**Fig. 1.** Code extracted to reduce file complexity

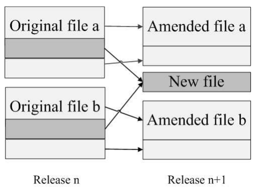

**Fig. 2.** Common code extracted from 2 files

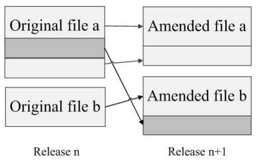

**Fig. 3.** Code relocated to a more suitable file

Figures 1 to 3 exemplify what we mean by a split file. In Figure 1, the original file has been shaded to indicate three blocks in the code. The following release contains two files: the amended file having two of the original file's blocks, with the third block in a new file. Figure 2 depicts common code, the shaded block, similarly removed from two original files. Blocks of code may be moved from one file to another, as shown in Figure 3. These situations can arise in typical refactorings, such as Class Move or Extraction [24].

## 3 Experiment: Identifying Split Files

The aim of this experiment was to identify and extract examples of split files, from the version history of a variety of software projects. We focused on split files as a syntactic form of change which may be identifiable using machine-learning techniques, without requiring semantically-based features. There were two parts to this experiment. First, similarity scores were computed between pairs of files and used to select groups of comparisons which identify possible, or candidate, split files. Second, information relating to these groups was gathered, and features constructed for use in classifying the candidate files.

## 3.1 Candidate File Selection

Eighty-four projects were selected from the Sourceforge repository [25] for analysis. Selection was based on the maturity and C code content of the projects. The C programming language was chosen for this study because of its popularity and overlap with C++ and Java. However, the method described here is text-based, and is therefore language independent. The projects contained a total of 69,425K lines of code in 266,687 files, an average of 826K lines and 22 releases per project. For each release of each project, all C source code files were selected.

It is not uncommon for each file in a project to begin with a large identical comment block, which inflates the similarity score between files, particularly when the code element of the file is relatively small. To remove biases of this type, every file was stripped of comments.

For every pair of files from consecutive releases within each project, similarity scores were computed using the Ferret tool. These scores provide information about both the changes to a file between one release and the next, and the similarity between a file in one release and all other files in the next release.

If a file has been split, there are three likely indicators: the file will become smaller; the two versions of the file will not be highly similar; and either a new and similar file will appear, as in Figures 1 and 2, or an existing file will become more similar, as in Figure 3. When there are no other changes to the files, this pattern of indicators will be clear. However, these indications are usually obscured by other changes made to the code between releases, by multi-way splits, or by a combination of these two factors.

For our experiments, we needed to manually construct a classified set of files. File similarity and size were used to select those files which had changed sufficiently to indicate possible splitting. Potential target files for extracted code were identified using the similarity scores. Each candidate split file has a related amended file and one or more target files which may be new or existing files. This group of related files is called the *comparison group*. Apart from the selection of the parameter values, this process is fully automated and generates a set of comparison groups.

Given the lengthy process of manually checking these groups of files, the filter parameters were narrowed to produce a dataset of manageable size. Comparison groups with at least one new file were selected for this initial investigation. Every comparison group was examined to establish whether it contained a split file. The experimental dataset comprised 151 candidate split files, 104 positive examples (files that had been split) and 47 negative examples.

## 3.2 Feature Set and Classification

The features constructed to enable classification contained information about the files in a comparison group and their relationship with each other. Both Ferret and Duplo were used to match the original file with various combinations of other files in the comparison group. By matching token trigrams wherever they occur, Ferret is useful for detecting rearranged or inexact copies in the code.

Duplo provides a complementary function by locating blocks of identical lines of code. Ferret delivers output in two forms, giving both metrics and a report showing the location of duplicated trigrams. The metrics, which are trigram counts and similarity scores, were used directly. The reports were analysed to provide information about the distribution of the duplicated trigrams. Most of the features were based on the output from Ferret, the remainder being derived from the report provided by Duplo. For example, the number of duplicated blocks containing at least $n$ lines of at least $m$ characters, or the ratio of copied lines to total lines in a file.

Many of the features were intended to provide answers to the type of question that someone scanning the files might ask. Questions such as "is the duplicated code in this file in large enough blocks to be significant?" determined by using Duplo to find the proportion of the shared code in blocks of a minimum size; or "is all of the code from the original file present in the amended and new files?", which is addressed by using Ferret to compare a concatenation of the new and amended files with the original file. Another such question is "what proportion of the original file appears in another file?", answered by measuring containment, which is calculated by dividing the number of trigrams shared by the two files by the number of trigrams in the original file. In total, we constructed 136 features.

Lastly, for each machine learning algorithm, to test generalisation, 100 selections of 66% of the dataset were used to build models, leaving the remaining data for testing. The mean of each of the false negative and the false positive classifications in the test sets was recorded. All of the suitable base classifiers available in Weka were run in this way. The better performing of these classifiers were selected for use with all of the homogenous meta-classifiers. The heterogenous meta-classifiers were run with various combinations of base classifiers.

## 4 Results

The ten classifiers with the fewest false positive classifications are shown on the left of Table 1. The figures given are: the mean number of false negatives, the mean number of false positives, the false positive rate, and the overall classification rate. Shown on the right of the table are the seven best base classifiers, and the three ensemble classifiers which improved on the rate achieved by SMO [26], the best base classifier.

Minimising false positives is important when building a set of positive examples. The best algorithm in this case is Conjunctive Rule, misclassifying a mean of 0.85 instances per run, 5.32% of 15.98, the mean number of negative instances in a test set. This comes with the cost of missing a mean of 8.47 positive instances, meaning that 96.94% of the positive classifications (0.85 fp + 26.89 tp) are correct.

The grading algorithm [27] combining locally weighted learning [28], a support-vector machine with sequential minimal optimisation [26] and voted perceptron [29] classifiers provides the best mean overall classification rate, 88.76%.

**Table 1.** Results: On the left are the 10 classifiers which give the fewest false positives. On the right, the 7 base classifiers with fewest total misclassifications and the 3 ensembles which outperform them. The figures show false negatives and positives, false positive rate and overall classification rate. ∗ Pre-pruned. † Algorithms used in heterogenous meta-classifiers are abbreviated as follows: L stands for LWL, Locally Weighted Learner; P for PART; S for SMO; and V for Voted Perceptron.

| **Ranked by number of false positives** | | | | | **Ranked by overall classification rate** | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Classifier | FN | FP | FP (%) | % correct | Classifier | FN | FP | FP (%) | % correct |
| ∗**Conjunctive Rule** | 8.47 | 0.85 | **5.32** | 81.85 | †**Grading L,S,V** | 1.95 | 3.82 | 23.90 | **88.76** |
| LWL | 7.36 | 0.96 | 6.01 | 83.79 | †Voted avg. L,S | 2.50 | 3.57 | 22.34 | 88.18 |
| NBUpdateable | 13.61 | 1.54 | 9.64 | 70.49 | †MultiScheme L,S,V | 2.58 | 3.56 | 22.28 | 88.04 |
| †Voted avg. P,S | 4.21 | 2.66 | 16.65 | 86.62 | SMO | 2.58 | 3.57 | 22.34 | 88.02 |
| Bayes Network | 5.50 | 2.91 | 18.21 | 83.62 | Random Forest(100) | 3.50 | 3.76 | 23.53 | 85.86 |
| REPTree | 5.99 | 3.21 | 20.09 | 82.08 | Logistic Model Trees | 3.85 | 3.73 | 23.34 | 85.24 |
| Random Forest(100 trees) | 4.09 | 3.34 | 20.90 | 85.53 | Alternating Decn.Tree | 3.93 | 4.05 | 25.34 | 84.46 |
| Bagged Simple Logistic | 3.60 | 3.52 | 22.03 | 86.13 | LWL | 7.36 | 0.96 | 6.01 | 83.79 |
| Bagged ADTree | 4.09 | 3.52 | 22.03 | 85.18 | Bayes Network | 5.50 | 2.91 | 18.21 | 83.62 |
| Multilayer Perceptron | 4.04 | 3.54 | 22.15 | 85.24 | J48 graft | 4.33 | 4.15 | 25.97 | 83.48 |

## 5 Conclusion

We have described our current work on constructing classifiers to locate files of a specific type within evolving software projects. Our process uses syntactic features, calculated from the source code, as the basis for classifier construction. After analysing 266,687 files containing 69,425K lines of code from 84 projects, we have developed classifiers to locate split files with an overall accuracy of 89%, or a false-positive rate of 5%. Our best overall classifier is an ensemble technique, combining the best base classifiers according to overall score, false negative score, and false positive score respectively.

We aim to continue this work by developing related techniques for extracting syntactic features and by detecting other restructured files, such as merged or recombined files, using a similar approach. This will provide models for use by software engineers needing to track such changes, and help contribute to the development of automated techniques to analyse the evolution of software code.

## References

1. Gyimothy, T., Ferenc, R., Siket, I.: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In: IEEE Transactions on Software Engineering, 31(10), pp. 897–910, IEEE Press, Piscataway, NJ, USA. (2005)
2. Meyers, T.M., Binkley, D.: An Empirical Study of Slice-Based Cohesion and Coupling Metrics. In: ACM Transactions on Software Maintenance, pp. 25–51, ACM, New York, NY, USA. (2007)
3. Stewart, K.J., Darcy, D.P., Daniel, S.L.: Observations on Patterns of Development in Open Source Software Projects. In: 5th Workshop on Open Source Software Engineering, pp. 1–5, ACM, New York, NY, USA. (2005)

4. Cubranic, D., Murphy, G.C., Singer, J., Booth, K.S.: Hipikat: A Project Memory for Software Development. In: IEEE Transactions on Software Engineering, 31(6), pp. 446–465, IEEE Computer Society, Los Alamitos, CA, USA. (2005)
5. Zimmermann, T., Peter Weissgerber, P., Diehl, S., Zeller, A.: Mining Version Histories to Guide Software Changes. In: ICSE '04: 26th International Conference on Software Engineering, pp. 563–572, IEEE Computer Society, Washington DC, USA. (2004)
6. Livshits, B., Zimmermann, T.: Dynamine: Finding common error patterns by mining software revision histories. In: 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 296–305, ACM, New York, NY, USA. (2005)
7. Schröter, A., Zimmermann, T., Zeller, A.: Predicting component failures at design time. In: ISESE '06: ACM/IEEE International Symposium on Empirical Software Engineering, pp. 18–20, ACM, New York, NY, USA. (2006)
8. Aversano, L. Cerulo, L., Grosso, C.D.: Learning from Bug-Introducing Changes to Prevent Fault Prone Code. In: IWPSE '07: 9th International Workshop on Principles of Software Evolution, pp. 19–26, ACM, New York, NY, USA. (2007)
9. Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J.: Automatic Identification of Bug-Introducing Changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 81–90, IEEE Computer Society, Washington, DC, USA. (2006)
10. Śliwerski, J., Zimmermann, T., Zeller, A.: When do Changes Induce Fixes? In: MSR '05: 2005 International Workshop on Mining Software Repositories, pp. 1–5, ACM, New York, NY, USA. (2005)
11. Antoniol, G., Rollo, V.F., Venturi, G.: Detecting Groups of Co-changing Files in CVS Repositories. In: 8th International Workshop on Principles of Software Evolution (IWPSE 2005), pp. 23–32, IEEE Computer Society, Washington DC, USA. (2005)
12. Ying, A.T.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C.: Predicting Source Code Changes by Mining Change History. In: IEEE Transactions on Software Engineering, 30(9), pp. 574–586, IEEE Press, Piscataway, NJ, USA. (2004)
13. Ratzinger, J., Sigmund, T., Vorburger, P., Gall, H.: Mining Software Evolution to Predict Refactoring. In: ESEM '07: 1st International Symposium on Empirical Software Engineering and Measurement, pp. 354–363, IEEE Computer Society, Washington, DC, USA. (2007)
14. Lehman, M.M., Belady, L.A. editors.: Program Evolution: Processes of Software Change. Academic Press Professional, Inc., San Diego, CA, USA, (1985)
15. Godfrey, M.W., Zou, L.: Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. In: IEEE Transactions on Software Engineering, 31(2) pp. 166–181, IEEE Computer Society, Los Alamitos, CA, USA. (2005)
16. Walenstein, A., Jyoti, N., Li, J., Yang, Y., Lakhotia, A.: Problems Creating Task-Relevant Clone Detection Reference Data. In: Working Conference on Reverse Engineering. IEEE Computer Society, Los Alamitos, CA, USA. (2003)
17. Maletic, J., Marcus, A.: Supporting Program Comprehension Using Semantic and Structural Information. In: 23rd International Conference on Software Engineering, pp. 103–112, IEEE Computer Society, Washington, DC, USA. (2001)
18. Weissgerber, P., Diehl, S.: Identifying Refactorings from Source-Code Changes. In: ASE '06: 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 231–240, IEEE Computer Society, Washington, DC, USA. (2006)

19. Lyon, C.M., Barrett, R., Malcolm, J.A.: A Theoretical Basis to the Automated Detection of Copying Between Texts, and its Practical Implementation in the Ferret Plagiarism and Collusion Detector. In: JISC(UK) Conference on Plagiarism: Prevention, Practice and Policies Conference, pp. 15–21, Northumbria University Press, Newcastle, UK. (2004)
20. Lyon, C.M., Malcolm, J.A, Dickerson, R.G.: Detecting Short Passages of Similar Text in Large Document Collections. In: 2001 Conference on Empirical Methods in Natural Language Processing, pp. 118–125, SIGDAT, Special Interest Group of the ACL. (2001)
21. Rainer, A.W., Lane, P.C.R., Malcolm, J.A., Scholz, S-B.: Using N-grams to Rapidly Characterise the Evolution of Software Code. In: 4th International ERCIM Workshop on Software Evolution and Evolvability. IEEE. (2008)
22. Ammann, C.M.: Duplo - Duplicate code detection tool. `http://duplo.giants.ch/`
23. Witten, I.H., Frank, E.: Data mining: Practical Machine Learning Tools and Techniques. Morgan Kaufman, San Francisco, CA, USA. (2000)
24. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA, USA. (1999)
25. Sourceforge: open-source software repository. `http://sourceforge.net/`
26. Platt, J.: Fast Training of Support Vector Machines Using Sequential Minimal Optimization. In: B. Schoelkopf and C. Burges and A. Smola, editors, Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge, MA, USA. (1999)
27. Seewald, A.K., Fürnkranz, J.: An evaluation of Grading Classifiers. In: 4th International Conference on Advances in Intelligent Data Analysis, pp. 115–124, Springer-Verlag, London, UK. (2001)
28. Atkeson, C.G., Moore, A.W., Schall, S.: Locally Weighted Learning. In: Artificial Intelligence Review. 11(1-5), pp. 11–73, Kluwer Academic Publishers, Norwell, MA, USA. (1997)
29. Freund, Y., Schapire, R.E.: Large Margin Classification Using the Perceptron Algorithm. In: 11th Annual Conference on Computational Learning Theory, pp. 209–217, ACM, New York, USA. (1998)