Evaluating the Quality of Drupal Software Modules

Benjamin Denham
School of Engineering, Computing & Mathematical Sciences
Auckland University of Technology
Auckland 1142, New Zealand
ben.denham@gmail.com

Russel Pears
School of Engineering, Computing & Mathematical Sciences
Auckland University of Technology
Auckland 1142, New Zealand
russel.pears@aut.ac.nz

Andy M. Connor
Colab
Auckland University of Technology
Auckland 1142, New Zealand
andrew.connor@aut.ac.nz

Evaluating software modules for inclusion in a Drupal website is a crucial and complex task that currently requires manual assessment of a number of module facets. This study applied data-mining techniques to identify quality-related metrics associated with highly popular and unpopular Drupal modules. The data-mining approach produced a set of important metrics and thresholds that highlight a strong relationship between the overall perceived reliability of a module and its popularity. Areas for future research into open-source software quality are presented, including a proposed module evaluation tool to aid developers in selecting high-quality modules.

Keywords: Software Quality; Data Mining; Drupal.

1. Introduction

Choosing appropriate tools and libraries is an important part of any software development project. This is particularly true for projects built with modular systems, such as the Drupal[a] content management system for websites. Many thousands of open-source and independently developed "modules" are available to extend the core functionality of Drupal, providing features such as e-commerce capabilities[b], calendar displays[c], and image carousels[d].

[a]https://drupal.org
[b]https://www.drupal.org/project/commerce
[c]https://www.drupal.org/project/calendar
[d]https://www.drupal.org/project/jcarousel

A common task in Drupal development is searching for modules to extend the functionality of a website. Whilst it is relatively straightforward to find a module for a particular use case, there are often unresolved upstream issues that either require substantial effort to remedy or necessitate the complete removal of module from the website. Furthermore, these issues may only become apparent after the module has been in use for some time, and has already become a critical part of the website's operations. Because this kind of risk is inherent in using any kind of third-party module, it is important to evaluate the quality of a Drupal module and its supporting community before deciding to use it in a website.

Drupal developers have many different opinions on the factors that should be considered when evaluating whether to use a particular Drupal module. However, the need for more robust measures of Drupal module quality is now more important than ever, as the Drupal Association (the organization that supports the Drupal community) has recently removed the restriction of only allowing peer-reviewed developers to fully publish modules [1].

While past studies have investigated measures of quality in software projects and analyzed factors common to popular open-source projects [2–4], there has been no formal investigations into Drupal modules.

For the purposes of this study, module popularity was chosen as a quantifiable measure to identify exemplar modules of high-quality. While popular modules may not necessarily have high software engineering quality, it is likely that they have been found to be of high utility to the websites that use them, and are therefore valued by the Drupal community. The popularity of a module also indicates the size of the community using and contributing to it (whether by providing software changes, documentation, or bug reports), which has been identified as an important prerequisite for high-quality open-source software [2]. Whilst popularity can be assumed to indicate high utility, it cannot be assumed that all modules with high utility are popular. Therefore, the analysis of popular modules provides insight into what features may be related to high utility and therefore gauge where there is potential risk in choosing a module that is not popular.

Therefore, this study aimed to discover quality-related metrics that are important to consider when deciding whether to add a Drupal module to a website. The research questions were as follows:

1. What quality-related metrics are common to popular Drupal modules?
2. What quality-related metrics are common to unpopular Drupal modules?

These questions were answered by collecting data related to the quality and popularity of Drupal modules, applying machine learning algorithms to construct models of the data, and analyzing those models to gain insights into which (if any) quality-related metrics are closely related to module popularity.

2. Literature Review

The analysis of software quality has received a large amount of attention from researchers. As at the time of writing, the current industry standard for software quality is defined by ISO25010 [5], which specifies a number of characteristics that software quality should be assessed according to: functional stability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. It is worth noting that all of these attributes can be evaluated (to some degree) outside of a specific usage context, making objective, generic comparison of software modules according to these characteristics a possibility. Past studies have also produced more specialized models for evaluating the quality of commercial, off-the-shelf software components [6] and open-source software projects [7], which include evaluation according to quantitative metrics.

Past surveys [8–10] have have documented a variety of ways in which data mining can be applied to help understand and improve software engineering practice. Applications have included predicting fault-prone modules based on source-code metrics [11], examining patterns in past source-code changes to provide recommendations to developers as they are making new changes [12], and aiding software project managers by estimating required development effort [13] and recommending developers with relevant expertise for tasks based on their past work [14]. The concept of "software intelligence" systems has also been proposed as a means of describing those systems that augment the software engineering process in a similar way to how "business intelligence" systems have improved business processes [15].

Of particular relevance are the previous studies that have evaluated the quality of open-source software on the basis of source-code quality metrics and the characteristics of project communities. Adewumi, Misra, Omoregbe, Crawford, and Soto [16] point out that much of the previous research has focused on creating models for software evaluation, and that data-mining and tool-based approaches require further exploration. Their review also notes that previously used source-code quality metrics can be aligned to the quality-characteristics outlined in the ISO25010 standard (primarily: maintainability, functional stability, and usability). However, it appears that much of the focus has been on evaluating larger, standalone pieces of open-source software, rather than modules for an extensible system (such as Drupal).

Other studies have used data mining techniques to evaluate factors contributing to the popularity of projects in open-source software ecosystems. Emanuel, Wardoyo, Istiyanto, and Mustofa [17] identified factors that were common to popular projects on Sourceforge (an open-source software publishing site) by performing 2-itemset association rule mining with the Weka data-mining toolkit. However, many of the factors they identified were not helpful for their goal of identifying how a developer could make their project more popular (for example, they identified that popular projects were more likely to target unspecialized end-users, but this is not an appropriate choice for all projects), arguably because the wide breadth of project types under analysis meant that detailed factors could not be considered. Cai and Zhu [18] focused on understanding how developer reputation relates to project popularity within the Ohloh social network of open-source developers and projects, although a limitation of this study appeared to be that developer reputation was based on aggregate project success, making the identified relationship somewhat cyclic. Some studies have made use of traditional software quality measures such as cyclomatic complexity and comment-to-code ratios [19–21], while others have focused on language-specific factors such as dependency hierarchies and abstract syntax tree (AST) node counts [22, 23].

Within the Drupal community, there has been an increasing focus on code quality evaluation in recent times. Static code analysis tools have been used within the Drupal community as part of the code-review process for many years [24], but recently discussion has turned towards following the lead of other open-source communities, and making code quality checks a more automated part of Drupal development [25–27]. Morrison [28] created tooling for quantifying the quality of Drupal modules, though no systematic study was performed using the tooling.

The sources reviewed above confirm that there is potential merit to data-mining approaches for understanding the quality and popularity of software modules, and that there is a need within the Drupal community for greater understanding of project metrics. However, the metrics to be considered in the analysis must be carefully selected to ensure that they will provide useful insights for informing the Drupal community. The reviewed studies list a variety of metrics relating to source-code quality and other project facets that are applicable to Drupal modules and are therefore candidates for analysis in this study. Furthermore, in order for the results of this study to be relatable to those of past research, the identified metrics should be aligned with the software quality factors of the ISO25010 standard.

3. Research Method

3.1. *Overview of Methodology*

To review, the objective of this research was to discover quality-related metrics that are important to consider when deciding whether to add a Drupal module to a website. The methodology used to achieve this objective was a typical data-mining approach comparable to the KDD, SEMMA, and CRISP-DM processes described by [29]. The steps involved in the methodology were as follows:

1. Identify features to include in the dataset based on a review of past studies and a survey of the Drupal community.
2. Construct and employ a tool to collect the dataset.
3. Manually inspect the dataset to remove or rectify incorrect or missing data. This process of "data cleaning" is considered to be an essential step for successful data-mining [30].
4. Apply different machine learning algorithms to determine an effective approach

for producing models from the data, performing additional data-cleaning as necessary.
5. Create and execute an experimental plan to optimize the models produced by the machine learning algorithms.
6. Analyze and interpret the models and their performance to answer the research questions above.

Some methodologies (such as CRISP-DM [29]) involve an iterative feedback loop where the results of analysis are used to drive further data collection, data cleaning, and algorithm application. For this study, iteration was constrained to step 4 of the methodology described above, where the initial results obtained by applying certain algorithms informed decisions for further transformations of the dataset (such as removing certain features or classes) and the application of different algorithms.

3.2. *Feature Identification*

The first step in the research process was to identify a feature-set of quality-related metrics to collect. Source-code and project metrics both needed to be considered. The full feature-set of source-code and project metrics (along with the data-sources from where they were obtained) are provided in Appendix A.

3.2.1. *Source-Code Metrics*

The source-code metrics used in prior studies were good candidates for the feature-set. Several tools were evaluated for extracting metrics from the PHP source-code of Drupal modules, including PHPMessDetector [31], phpmetrics [32], and phploc [33]. Two tools were selected for use. PHPDepend [34] was selected for its ability to generate more of the metrics from prior studies than any of the other tools. Secondly, PHPCodesniffer [35] was selected because of its wide use in the Drupal community through the Coder module [36] and PAReview.sh tool [24]. Three Drupal-specific code standards exist for use with PHPCodesniffer: `Drupal` and `DrupalPractice` [37], and also `DrupalSecure` [38]. All three standards were selected for testing module PHP source-code to generate code-style suggestions/notices that could be used as quality metrics.

3.2.2. *Drupal Project Metrics*

In order to gain a better understanding of how Drupal developers currently evaluate whether to add a module to a website, a small pilot survey of Drupal developers was performed. The two questions asked on the survey were:

1. What do you consider to be indicators of a Drupal module's quality?
2. What aspects do you consider when evaluating whether to add a particular module to a Drupal project?

Eleven survey responses were received, from which could be identified 46 unique factors considered by Drupal developers when evaluating a Drupal module for use. The responses to both questions were combined when evaluating the results because some respondents had answered both questions with a single response, and because there was a large degree of cross-over between the factors identified in the responses to both questions. Of the 46 unique factors, some were impractical to evaluate quantitatively (e.g. "Is the module configurable?"), and others were dependent on the context of a particular website (e.g. "Does the module meet all of my needs?"), making them impractical for general evaluation. However, many factors were identified that could be automatically derived from a Drupal module's codebase and project metadata, and would be suitable for quantitative analysis. Furthermore, there was alignment between these factors and the features analyzed in past studies, confirming their value for analysis in this study.

Some additional project metrics that were not identified by the survey were also selected for inclusion in the feature-set on the basis that they could easily be obtained from the information published about each module on Drupal.org.

3.2.3. *Classification Target*

The most essential metric for the proposed data mining process was the measure of popularity which all other metrics would be correlated against. The current number of Drupal websites with the module installed was selected for this purpose. One limitation of this metric (which is published for each module on Drupal.org) is that it is only able to account for websites with the "Update Status" module enabled, which includes a callback to Drupal.org to report module installations [39]. However, it is still the most accurate measure of module installation available. While the total number of module downloads is also published, the installation count is a better measure of current popularity, as it will not favour modules where developers have downloaded the module but decided to not use it or stop using it at a later point in time.

3.3. *Data Collection*

With the feature-set of project and source-code metrics identified, the next step was to collect the dataset for a number of Drupal modules.

The data collection was limited to modules available for version 7 of Drupal. While Drupal 8 is the latest version of Drupal, far fewer Drupal 8 modules exist at the time of writing (approximately 3000 Drupal 8 modules compared to approximately 13000 Drupal 7 modules [40]). Furthermore, many of the modules available for Drupal 8 are much more recently developed and less mature as projects, and would exhibit less of a distinction between popular and unpopular modules.

A tool was developed with the Python programming language to collect the selected features for all Drupal 7 modules with "Full Project" status on Drupal.org (excluding personal "sandbox" projects of Drupal.org users). The tool scraped project metrics from pages about each project on Drupal.org, and downloaded the latest stable release (or unstable release if no stable release existed) of each module to execute the PHPDepend and PHPCodesniffer tools on, which provided source-code metrics.

The Python tool was able to collect data for 12,945 modules out of the total 12,950 Drupal 7 modules available at the time the tool was run. The other five modules exhibited issues which made it difficult to extract data for them from Drupal.org. Other modules required specific tweaks in order for the scraping tool to work correctly, such as fixing broken PHP syntax or module archives (`.tar.gz` files) to allow the source-code analyses to run, and manually altering the character encoding expected by the scraping tool in some cases.

3.4. *Data Cleaning*

Once the dataset had been collected, a manual process of inspecting and evaluating the correctness of the data was carried out.

Firstly, Boolean values stored numerically were converted to nominal "True/False" values. This was done to ensure the applied machine learning algorithms would treat the features as only having two discrete possible values, as opposed to having many possible values on continuous numeric scale. This transformation was applied for the `automated_tests_status`, `available_for_next_core`, `security_covered`, and `recommended_release` features.

Secondly, the missing values in the collected dataset were analyzed. All source-code related features were missing for 6% of the dataset, due to the fact that there was no published release for those modules. Because modules without a published release are impractical to install and therefore rarely used, and because they made up a very small portion of the dataset, those records were removed from the dataset. For other features, values were missing because the scraping tool could not find values on project web-pages. Upon manual inspection of these cases, it became apparent that leaving the values as "missing" was not appropriate, as they were all numeric features where absence from the project web-pages implied the correct value was "zero" (for example, a missing installation count on the main project web-page only occurs in the case when there are zero reported installations of the module). Therefore, a value of zero was substituted for missing values for the following features: `average_translation_percent`, `description_length`, `issue_count`, `issue_new_count`, `issue_participants`, `install_count`, `install_download_ratio`, `growth_per_week`, `issue_response_rate`, `issue_response_hours`, `total_install_count`, and `total_growth_per_week`. Finally, there were yet other features with missing values because the data was either not available for the given module, or was not applicable for the module. Missing values were therefore left unaltered for the following features: `release_frequency_days` (missing for the 26% of modules with only a single release), `commit_frequency_days` (missing for the 5% of modules with only a single commit), `seconds_since_last_commit` (missing for 3 modules without a Git history), and `days_since_release` (missing on the project web-page for 1 module).

A flaw was also identified in the collection of the `available_for_next_core` feature. Some modules that were available for Drupal 7 have been included in the core of Drupal 8 itself, and therefore do not have a Drupal 8 version published on their project web-pages. These values were manually adjusted according to a list of Drupal 7 modules migrated to Drupal 8 core [41].

There were also some issues with the data collected from the "warning" notices reported by the PHPCodesniffer tool. Firstly, formatting issues in the CSV output of the tool meant that the names of some notices had been incorrectly scraped. As this only accounted for 812 of the total 4,901,256 notices, these incorrectly scraped notices were simply removed from the dataset. Secondly, the tool had reported `Drupal.InfoFiles.AutoAddedKeys.Project` and `Drupal.InfoFiles.AutoAddedKeys.Version` notices for most modules. The purpose of these notices is to discourage module developers from manually adding project and version information to module code, as this information is automatically added by the publishing process on Drupal.org. Because PHPCodesniffer was executed on published releases of modules that contained automatically added values, these notices were irrelevant, and were therefore removed from the dataset. Finally, because there were relatively few occurrences of each individual notice, it was theorized that stronger correlations might be observed if additional features were added for related groups of notices. For this reason, a new feature for each possible "level" in the notice namespace hierarchy was added to the dataset. For example, given the original `Drupal.ControlStructures.ElseIf` notice, new features were added for all notices starting with `Drupal.ControlStructures` and just `Drupal`.

A second run of the scraping tool was also required to amend and replace some values. The initial implementation for scraping the `has_documentation` feature incorrectly reported all modules as having documentation, so re-scraping was required after the implementation was corrected. It was also apparent that the `growth_per_week` could be either very high or very low for popular modules. Because this could indicate that popular modules are losing Drupal 7 installations simply because of migrations to Drupal 8, it was decided to re-scrape a `total_growth_per_week` that considered all installations of modules, not just those of the Drupal 7 versions. A `total_install_count` was also re-scraped to capture the total number of installations of all versions of the module (not just Drupal 7 versions) in order to ensure the installation count and growth rate were based on the same figures.

To reduce feature redundancy in the dataset, the `growth_per_week` and `install_count` were removed from the dataset in favour of the `total_growth_per_week` and `total_install_count` replacements. The cyclomatic complexity score (`ccn`) was also removed because of its similarity to the extended cyclomatic complexity measure (`ccn2`), which captures information about more branches in source-code and is more widely used in static source-code analysis tools [42].

Finally, as the purpose of the data mining was to analyze the distinctions between popular and unpopular modules, very recently published modules were removed from the dataset, as enough time would not have passed for their installation counts to show whether they were truly popular when compared with more mature modules. Modules published less than 730 days (∼2 years) from the date of data collection were removed, as this accounted for the tail of the module age distribution that were younger than the majority of modules (centred around a mean age of ∼4.5 years), as approximately illustrated in Figure 1. This resulted in the removal of 21% of the dataset.

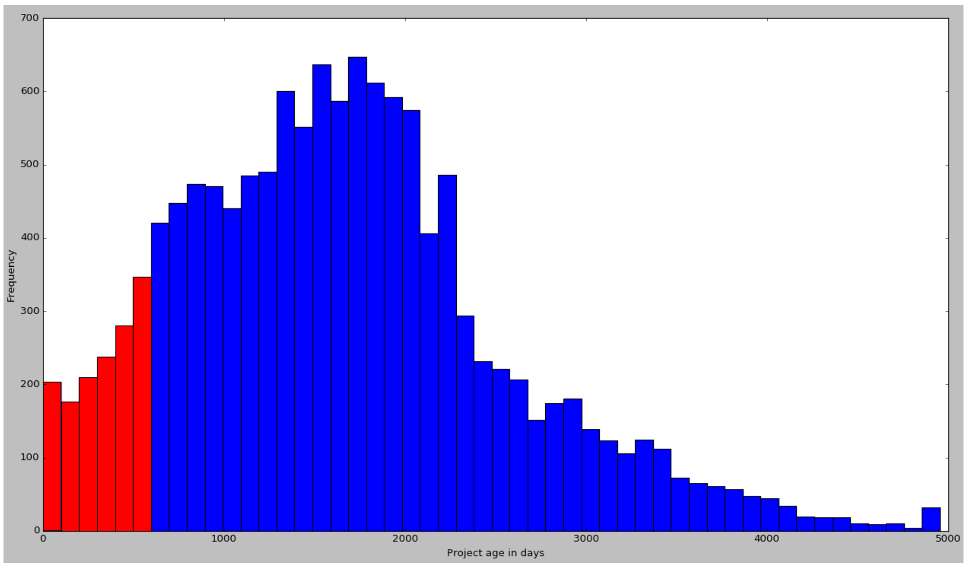


Fig. 1. Histogram of module project age, with those under 700 days old highlighted in red.

The final dataset produced after data cleaning included a total of 10,215 records.

3.5. *Exploratory Data Analysis*

3.5.1. *Supervised Machine Learning*

Unsupervised association rule mining with the Apriori algorithm [43] was initially used to analyze the dataset, but was unable to derive meaningful insights. Because of this, the focus of the research was shifted towards supervised machine learning algorithms, with the aim of producing models to predict module popularity according to the three classes in Table 1. This supervised learning approach has been used previously for the prediction of software project popularity [23].

A class from Table 1 was assigned to each module in the dataset according to the total number of reported installations. The `unpopular` and `popular` bin widths were selected to approximate what would generally be re-garded as the lowest and highest tiers of module popularity. Correlations of feature values to these two bins would represent meaningful insights.

Table 1. Manual installation count discretization

Label	Total installations	Bin size
unpopular	< 100	5414
moderate	>= 100 & < 10,000	4291
popular	>= 10,000	510

Initial tests with supervised learning models showed poor predictive performance. However, it was possible to substantially improve performance by removing modules with `moderate` popularity, which allowed the models to focus on identifying the key differences between highly popular and highly unpopular modules.

Furthermore, initial training of these models showed a heavy dependence on features that were directly related to module install counts, high issue queue activity (popular modules inherently have more active issue queues), or project age (an older module is more likely to have more installs). It was determined that identifying correlations between these features and module popularity would not be useful, as the aim of the research was to find insights into what previously unknown features of modules and their communities are related to module popularity. In order to force the models to be trained on other, more independent features, the following features were removed from the dataset: `committer_count`, `total_growth_per_week`, `install_download_ratio`, `issue_count`, `issue_new_count`, `issue_participants`, `issue_response_hours`, `issue_response_rate`, and `project_age_days`. `issue_open_ratio` was left in the dataset because it is not directly related to the number of issues or amount of overall issue queue activity. Appendix A notes which features were removed from the feature-set before the final classification models were trained.

3.5.2. *Target Classification Models*

Based on tests with several classification algorithms, it was decided that four different classification models should be trained to gain different kinds of insights into the data. All of these models could be trained using the Weka data mining workbench [44].

Two models would be built with the JRip classifier for generating classification rules. JRip is an implementation of the Repeated Incremental Pruning to Produce

Error Reduction (RIPPER) algorithm [45]. One model would produce a model starting with rules for the `popular` class, while the other would start with rules for the `unpopular` class. Both models would be trained on the dataset including modules with the `moderate` class. While including the `moderate` modules would decrease the overall predictive performance of the models, the best rules from the top of each model's rule list could be analyzed to provide insight into the most important factors for distinguishing `popular` and `unpopular` modules from all other modules.

Additionally, a J48 decision tree would be trained on the reduced dataset including only `popular` and `unpopular` modules. J48 is an implementation of the C4.5 algorithm for generating decision trees [46]. The decision tree would provide a model for distinguishing between the two classes, and potentially demonstrate which features were more or less important in the context of the values for other features.

A Naïve Bayes classifier would also be trained on the reduced dataset to show the relative importance of different features in distinguishing between the two classes.

3.6. *Model Optimization*

An experimental plan was produced for the optimization of the four classification models. The plan consisted of three steps to select optimal configurations for data balancing, feature selection, and classification algorithm parameters. The best configuration found in each step would be selected for application in subsequent steps.

For all experiments, the area under the precision-recall curve for the minority class (`popular`, which only accounts for approximately 5% of the dataset) was selected as the evaluation metric. Initial testing showed that predictive performance was good for the `moderate` and `unpopular` classes, but quite poor for the `popular` class. This is due to the fact that classification algorithms produce models that are inherently worse at predicting minority classes [47], and the focus of performance optimization therefore needs to be given to the minority class (though without comprising performance on other classes). Furthermore, optimizing the area under the precision-recall curve gives the greatest flexibility when choosing an appropriate predictive threshold for each of the final classification models in order to optimize the F-measure (a measure that represents a balance between precision and recall) for the minority class.

Data balancing techniques can be a useful way to improve classification algorithm performance on minority classes [48]. For each algorithm, a variety of data balancing techniques were applied, including variations of: over-sampling records for the minority class with the synthetic minority over-sampling technique (SMOTE), under-sampling records for the majority classes, and a combination of over-sampling and under-sampling by way of re-sampling. For efficiency, each technique was applied to a training dataset of two-thirds of the original data, and the target classification algorithm was trained to produce a model that was evaluated against the remaining third of the data (the test dataset).

Feature selection is another important step for improving the performance of classification algorithms, as the presence of irrelevant features has been shown to severely decrease their performance [49]. Various feature-selection techniques were tested, including information gain evaluation, gain ratio evaluation, feature correlation evaluation, and correlation-based feature subset selection. Given the large number of features (over 600 with the many PHPCodesniffer features), feature selection was limited to forward searches. The wrapper method of feature selection was not used, as the running time over the large set of features and training records was prohibitive. Once again, the same approach of using a training and test set with a two-third/one-third split was used to test feature selection techniques.

The parameters of the algorithms also needed to be tuned to achieve optimal performance. The appropriate parameters to be tuned for each algorithm and their purposes are listed in Table 2. Evaluation of the parameters was performed using 10 iterations (only 3 in the case of JRip, because of the prohibitive model-building time) of 10-fold cross-validation, where data balancing and feature selection were applied to the training fold while the model was evaluated on an untreated test fold.

Table 2. Classification algorithm parameters to tune

Algorithm	Parameter	Purpose
JRip	minNo	The minimum number of records a rule must apply to.
JRip	folds	The number of folds to split data into, where one fold is used for pruning rules and the rest for growing rules.
JRip	optimizations	The number of optimization runs to test randomized rule variants.
J48	minNumObj	The minimum number of records a tree leaf must apply to.
J48	confidenceFactor	The size of the confidence interval to use when estimating error rates for the purpose of pruning branches.
Naïve Bayes	useKernelEstimator	Whether to use kernel density estimation (rather than a normal distribution) for handling numeric features.
Naïve Bayes	useSupervisedDiscretization	Whether to use MDL-based supervised discretization for handling numeric features.

Additionally, in the case of J48 and Naïve Bayes, a threshold selector was applied to find the optimal predictive threshold to maximize the F-measure for the minority class (`popular`). This was not necessary for the JRip models, as the intention was to manually extract the best subset of generated rules for further evaluation.

3.7. *Final Models*

Experiments were run according to the optimization methodology described above to generate a set of optimal classification models. The critical performance statistics for the models (averaged over 10-fold cross-validation) are provided in Table 3, and the performance of each model is discussed below.

Table 3. Classification model performance statistics

Model	Accuracy	`popular` F-measure	`unpopular` F-measure
Popular classification rules	66.23%	0.378	0.737
Unpopular classification rules	54.22%	0.299	0.669
Naïve Bayes (feature selection first)	94.83%	0.690	0.972
Naïve Bayes (data-balancing first)	93.69%	0.641	0.965
Decision tree	93.11%	0.652	0.962

3.7.1. *Popular Classification Rules Model*

The optimal classification rules model for the `popular` class was produced by: using SMOTE to double the number of `popular` records, selecting the top fifteen features according to correlation evaluation, and configuring JRip to use 5 folds for the growing/pruning split, 10 optimization runs, and a minimum rule record count of 25.

This model has quite poor accuracy and F-measures, but it is of little importance because the focus of further analysis will only be the top rules for predicting the `popular` class.

3.7.2. *Unpopular Classification Rules Model*

The optimal classification rules model for the `unpopular` class was produced by: re-sampling the dataset to achieve an equal number of records for each class, selecting the top three features according to correlation evaluation, and configuring JRip to use 5 folds for the growing/pruning split, 10 optimization runs, and a minimum rule record count of 25. It is worth noting that several algorithm parameter configurations achieved the greatest area under the precision-recall curve for the `popular` class, as well as for the weighted average across all classes. The configuration above was selected on the basis of it also being the best performing configuration for the `popular` rules model.

Once again, while this model has quite poor accuracy and F-measures, it is of little importance because the focus of further analysis will only be the top rules for predicting the `unpopular` class. It is worth noting that even though the model is intended for use in predicting the `unpopular` class, the focus on the area under the precision-recall curve for the `popular` class is still justified as a performance metric because the focus must be on preventing confusion with the `popular` class.

3.7.3. *Naïve Bayes Model*

Because of the sensitivity of Naïve Bayes to choices in feature selection (as every feature contributes equally to the final model [50]), and because of the insignificant amount of time required to train additional Naïve Bayes models, it was decided to perform the experimental process with both possible orderings of the data balancing and feature selection steps. Additionally, once the best techniques for data balancing and feature selection had been identified, both orders of these operations were tested with each configuration of algorithm parameters in the final cross-validated set of experiments.

The best combination of data balancing and feature selection techniques was found by applying feature selection to the dataset before data balancing. However, the optimal Naïve Bayes model was produced by first under-sampling the `unpopular` class to achieve a class imbalance no greater than 50% of the `popular` class, then selecting the top eight features according to gain ratio evaluation, and finally using supervised discretization to handle numeric features. Additionally, a predictive threshold of 0.94 was chosen to achieve the greatest F-measure for the `popular` class.

This model achieved a very high overall accuracy, as well as a reasonably high F-measure for the minority class. This shows that the model is able to distinguish well between popular and unpopular modules based on the small number of features selected.

It is also worth noting that the inferior model produced by following the same experimental process that was used for the classification rule models (data balancing followed by feature selection) was still able to achieve a high level of accuracy (when using an optimized predictive threshold of 0.8486), as is demonstrated by the similarity of the performance statistics for both models in Table 3.

This shows that the drastic improvement in performance over that achieved by the classification rule models was due to the removal of the "moderately popular" modules from the dataset, and not simply because of the change in experimental process.

3.7.4. *Decision Tree Model*

The optimal decision tree model was produced by: re-sampling the dataset with a bias of 0.7 towards a uniform class distribution, selecting the top ten features according to information gain evaluation, and configuring J48 with a confidence factor of 0.25 and a minimum number of objects of 25. It is worth noting that using a confidence factor of 0.5 also achieved the greatest area under the precision-recall curve for the `popular` class, as well as the same weighted average across all classes. The confidence factor of 0.25 was selected on the basis of it being more conservative in its pruning, potentially leading to a simpler model. Additionally, a predictive threshold of 0.85 was chosen to achieve the greatest F-measure for the `popular` class.

The decision tree model achieved a similar accuracy to the Naïve Bayes model, although it did not achieve as high an F-measure on the minority class. However, it is still a useful addition, as the structure of the decision tree can provide different insights to those provided by the Naïve Bayes model.

4. Analysis of Classification Models

With the optimal classification models determined, the next step in the research process was to inspect the classification models to gain useful insights into the relationships between the analyzed metrics and module popularity.

4.1. *Classification Rule Models*

While the classification rule models did not achieve very high accuracy, they play an important role in the analysis of the dataset. Because they were trained on the entire dataset (including modules with "moderate" popularity), they produced rules that identify combinations of metrics that distinguish popular and unpopular modules from the rest of the dataset.

Table 4 shows the top rules for predicting unpopular and popular modules. Rules with confidence values less than 60% on the full dataset have not been included in the analysis. Note that the rules are intended to be applied in sequence for each class (e.g. the second rule for classifying popular modules is only applied to those not matched by the first). Also, note that the second and third rules for the "popular" class are essentially useless, as they make predictions based on maintainer counts that fall between integer values. These rules can only apply to the training records introduced by the SMOTE method of data balancing, so do not need to be considered when evaluating performance on the real dataset.

From the single rule for predicting unpopular modules, we can see that most unpopular modules have one or zero maintainers and no new commits for almost two years or more.

There are several metrics that are considered important by the model for predicting popular modules:

1. Having a version available for the next version of Drupal (as the dataset consisted of Drupal 7 modules, these would be Drupal 8 versions).
2. Good documentation: Having documentation pages and a module description of a reasonable length.
3. A relatively high proportion of translation work: more than 10% of all possible translations for the module. As there are currently 114 languages supported by Drupal, this would equate to roughly 11 full translations of the module (or many more partial translations).

Table 4. Best Performing Classification Rules

Consequent	Antecedent	Support	Confidence
unpopular	`(seconds_since_last_commit >= 58498871) and (maintainer_count <= 1)`	4825	69%
popular	`(maintainer_count >= 2.000363) and (available_for_next_core = True) and (roots_noc_ratio >= 0.00254) and (average_translation_percent >= 12.622163) and (roots_noc_ratio <= 0.31027)`	24	71%
popular	`(maintainer_count >= 2.000363) and (maintainer_count <= 2.999844)`	0	0%
popular	`(maintainer_count >= 3.003246) and (maintainer_count <= 3.982082)`	0	0%
popular	`(release_frequency_days >= 69.6) and (available_for_next_core = True) and (has_documentation = True) and (average_maintainer_commits >= 1146.24423) and (seconds_since_last_commit <= 11800320)`	74	80%
popular	`(release_frequency_days >= 69.6) and (available_for_next_core = True) and (has_documentation = True) and (description_length >= 1619.294735) and (average_translation_percent >= 10.657789) and (automated_tests_status = Enabled)`	38	68%

4. Having three or more maintainers. Furthermore, the maintainers of popular modules are likely to be very experienced members of the Drupal community, with an average of over 1000 commits on Drupal.org across the maintainers.
5. Having automated tests enabled.
6. An active code-base: with the latest commit fewer than ∼4.5 months ago.
7. A low release frequency: over ∼70 days between releases. This is probably just indicative of the fact that popular modules are typically older projects with many long-lived versions. However, it could indicate that popular modules have more stable releases. In any case, it seems that while popular modules are committed to more often, they do not necessarily have frequent releases.
8. The `roots_noc_ratio` is an interesting metric. It captures the proportion of classes in the module's code-base that are root classes (classes that do not inherit from a parent). It seems that popular modules have a ratio that is greater than 0% (likely indicating that they have at least some object-oriented code) but not more than about 30% (indicating that these modules make use of inheritance from either their own classes, or from classes in Drupal's core or other modules). Object-oriented code is not a large part of Drupal 7 module development, but it appears that typical object-oriented coding with class inheritance occurs more often in popular modules.

It is worth noting that none of these metrics are able to independently predict that a module is popular or unpopular, as the specified combinations of metrics must be present in order for each rule to apply.

Classification was very poor with the moderate class, suggesting it is quite difficult to distinguish between modules with less stark differences in popularity, perhaps because of the external factors affecting module popularity (E.g. If Facebook integration on websites is more than Twitter integration, then the Facebook modules will be more popular than Twitter modules, regardless of internal project/software metrics).

4.2. *Naïve Bayes Model*

Table 5 shows the probabilities (calculated according to the counts in the trained Naïve Bayes classifier) that a module is popular or unpopular. Note that the probabilities are not representative of the actual dataset, but of a balanced dataset which resulted in a classifier that was able to achieve better classification results on the minority class ("popular" modules). Also, note that because the classifier was trained without any of the "moderately popular" modules, these probabilities do not represent rules that will hold across the entire dataset, but rather show the factors that best differentiate between highly popular and highly unpopular modules.

Table 5. Class probabilities from Naïve Bayes classifier

Feature/value pair	Probability module is popular	Probability module is unpopular
`available_for_next_core = True`	89.39%	10.61%
`seconds_since_last_commit = (-inf-10595613]`	84.51%	15.49%
`maintainer_count = (2.5-inf)`	82.10%	17.90%
`automated_tests_status = Enabled`	79.94%	20.06%
`issue_open_ratio = (0.003352-0.497573]`	69.97%	30.03%
`seconds_since_last_commit = (10595613-59088360]`	59.62%	40.38%
`issue_open_ratio = (0.50173-0.664042]`	59.22%	40.78%
`average_translation_percent = (1.137024-inf)`	50.15%	49.85%
`maintainer_count = (1.5-2.5]`	48.94%	51.06%
`recommended_release = True`	48.69%	51.31%
`automated_tests_status = Disabled`	27.05%	72.95%
`available_for_next_core = False`	22.87%	77.13%
`maintainer_count = (-inf-1.5]`	15.71%	84.29%
`seconds_since_last_commit = (59088360-inf)`	12.35%	87.65%
`issue_open_ratio = (0.497573-0.50173]`	12.28%	87.72%
`issue_open_ratio = (0.664042-0.818075]`	10.77%	89.23%
`recommended_release = False`	4.03%	95.97%
`issue_open_ratio = (-inf-0.003352]`	3.19%	96.81%
`average_translation_percent = (-inf-1.137024]`	0.76%	99.24%
`issue_open_ratio = (0.818075-inf)`	0.61%	99.39%

There are a number of insights that can be drawn from this probability table:

1. If a module has a Drupal 8 version available, it is almost 90% likely that it is popular. However, if a Drupal 8 version is not available, it is only 77% likely to be unpopular, suggesting there are still a number of popular Drupal 7 modules without versions for Drupal 8.
2. A module with a commit in the last four months is likely to be popular, while a module without a commit in almost two years or more is likely to be unpopular.
3. A module with more than two maintainers is likely to be popular, while a module with one or no maintainers is likely to be unpopular.
4. A module with automated tests enabled is reasonably likely to be popular, while a module without automated tests is reasonably likely to be unpopular.
5. A module without a recommended release is very likely to be unpopular, but the presence of a recommended release does not strongly indicate that a module is popular.
6. A module with very little translation work ($< 1.14\%$) is almost certainly unpopular, but a higher amount of translation does not necessarily indicate the module is likely to be popular.
7. The ratio of open issues to closed or resolved issues appears to have a more complex relationship to popularity:
    - Modules with ratios very close to 0%, 50%, or 100% are more likely to be popular. This is probably because these extreme values are more common in unpopular modules with fewer issues. This is demonstrated in Figure 2, which shows a portion of the plot of `issue_count` against `issue_open_ratio`. Note the dense clusters of unpopular modules with very few issues and `issue_open_ratio` values near 0%, 50%, and 100%.
    - Importantly, a module is more likely to be popular if it has more closed issues than open issues, which would indicate that the module has an active issue base where issues are closed or resolved, and not simply left open.

4.3. *Decision Tree Model*

Figure 3 shows the trained J48 decision tree, with leaves predicting the minority "popular" class highlighted in green, and leaves where pruning occurred according to the threshold probability highlighted in red. The two numbers at each leaf node represent the number of modules in the dataset (excluding "moderately popular" modules) that are classified by each leaf, along with the number of these modules that were incorrectly classified. The percentage represents the success rate of the classification at that leaf.

Note that while the success rates of the "popular" leaves are relatively low, this is simply a product of the large class imbalance in the dataset, as there are many more "unpopular" modules which could be misclassified than there are "popular" modules.

The decision tree reflects many of the same insights into the features as the clas-

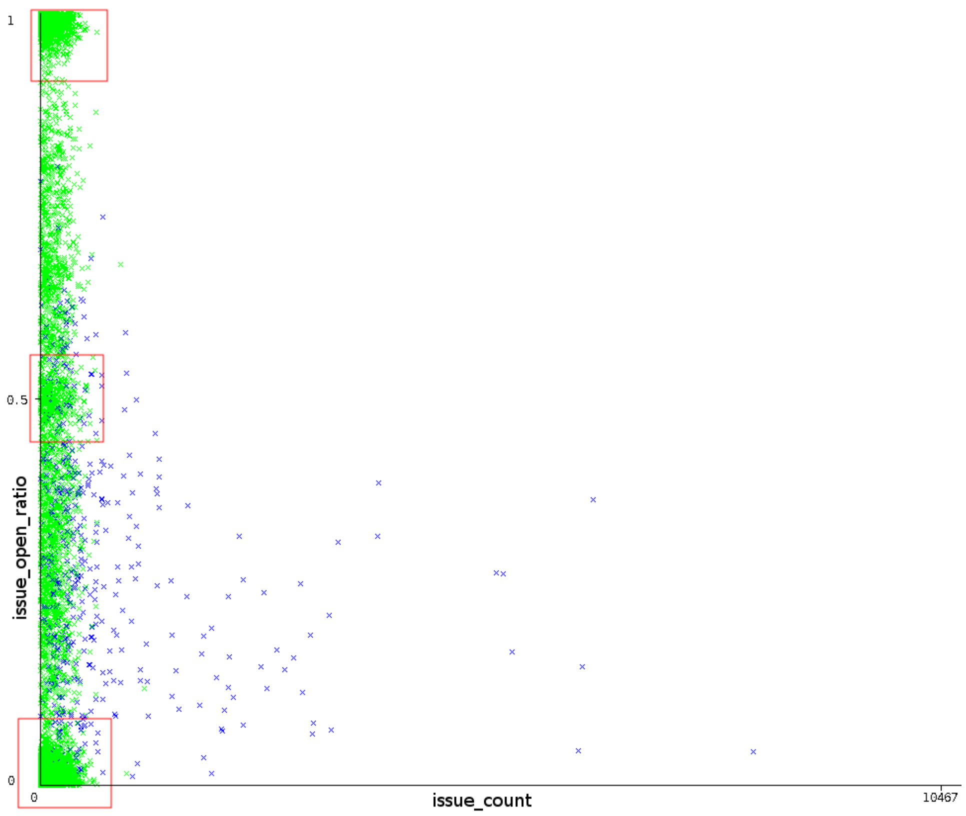


Fig. 2. Plot of open-issue ratios against issue counts. Unpopular modules marked with green; popular modules marked with blue. Jitter applied to aid visualization of dense clusters.

sification rule and Naïve Bayes models above, albeit with slightly different critical values for the features in order to best fit the data classified at each branch point.

However, there are two features present in the model that we have not seen in previous models: the number of days since the latest release (`days_since_release`), and the extended cyclomatic complexity number (`ccn2`). These features appear in the lower branches of the decision tree, so are not as important in the decision making process as other features, but the fact that they appear in multiple locations in the tree suggests they are not simply a result of over-fitting the dataset.

Based on the branches in the bottom left and bottom right of the tree, it can be seen that modules without a release in the last few years (807 and 1,708 days) or with a release in the last six months (169 days) are more likely to be popular. This most likely indicates that popular modules are likely to either have frequent releases, or very long-lived stable releases.

It can also be seen that the extended cyclomatic complexity is higher for popular modules. This indicates that popular modules tend to have more complex

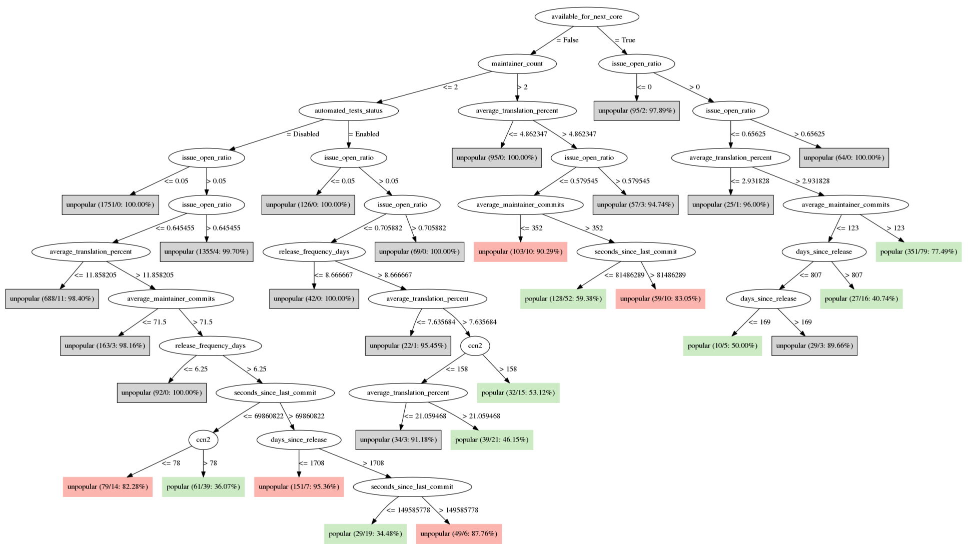


Fig. 3. The trained J48 decision tree

code-bases, most likely because they provide more complex functionality. Generally, maintaining a low cyclomatic complexity is recommended as good software engineering practice [51], but this finding suggests that it should not be used as an absolute measure of Drupal module code-quality. In essence, just because a module is more popular and has more experienced maintainers working on it, does not mean that the module's code can be simplified to achieve a lower cyclomatic complexity, as it depends on the complexity of the functionality of the module.

4.4. *Comparison of Models*

It is worth noting the metrics that feature in all three models above, as they are more likely to be strongly related to module popularity. These metrics are `available_for_next_core`, `maintainer_count`, `seconds_since_last_commit`, `average_translation_percent`, and `automated_tests_status`. Most notably, having two or more maintainers is a consistent indicator of popular modules across all models, and similarly, having one or no maintainers is a consistent indicator of unpopular modules.

It is also worth more closely analyzing metrics that are only used in one of the classification models. The `roots_noc_ratio`, `has_documentation`, and `description_length` metrics only appear in the classification rules for popular modules, suggesting that these metrics are better for distinguishing popular modules from all other modules, but not as strong as other features for distinguishing between only popular and unpopular modules. The `recommended_release` metric only appears in the Naïve Bayes model, where it features as the strongest indicator of a module being unpopular (in the case of a recommended release not being available). Finally, the `days_since_release` and `ccn2` metrics are only found in the decision tree model, and even then only in lower branches. This suggests that they are only strongly related to module popularity in the presence of certain combinations of the metrics that appear above them in the decision tree.

4.5. *Application of Models to Closely Related Modules*

A possible risk with the trained classifiers is that they simply predict some "confounding" external variable, which is in turn correlated with popular or unpopular modules. For example, if modules providing integration with third-party authentication services were consistently popular, then the classifiers may simply become sensitive to factors that are relevant to third-party authentication modules, but not necessarily related to the popularity of all kinds of modules.

This can be controlled for by observing the performance of the classifiers on a group of modules with similar functionality. Table 6 shows the predictions for a group of modules that all provide some mechanism for adding a content stream from a Facebook page to a Drupal website.

Table 6. Class predictions (with probabilities) for modules that provide Facebook feeds.

Module	Installs	Actual Class	Popular Rules	Unpopular Rules	Naïve Bayes	Decision Tree
`spider_facebook`	0	unpop	unpop 71.90%	unpop 60.60%	unpop 100.00%	unpop 99.20%
`fbapp`	9	unpop	unpop 71.90%	unpop 60.60%	unpop 100.00%	unpop 96.40%
`feeds_facebook`	618	mod	unpop 71.90%	mod 61.70%	unpop 100.00%	unpop 99.20%
`facebook_wall`	1094	mod	unpop 71.90%	unpop 60.60%	unpop 97.70%	unpop 96.40%
`facebook_boxes`	1991	mod	mod 56.80%	pop 59.70%	unpop 79.30%	pop[a] 55.10%
`facebook_pull`	2526	mod	mod 63.10%	pop 59.70%	unpop 52.00%	unpop 89.90%
`fb`	4687	mod	unpop 71.90%	unpop 60.60%	unpop 96.40%	unpop 92.90%
`fb_likebox`	14420	pop	mod 63.10%	pop 59.70%	pop 100.00%	pop 87.30%

Note: "unpop" = Unpopular, "mod" = Moderately Popular, "pop" = Popular

[a]Actually classified as "unpop" because of 85% predictive threshold.

It can be seen that the rules-based classifiers exhibit generally poor performance in determining the popularity of the "Facebook modules", but these classifiers have fairly poor performance in general. Still, there is a general trend of more "unpopular" predictions for those with lower install counts, and more "moderate" and "popular" predictions for those with higher install counts. Furthermore, the uncertainty in the decisions of these classifiers is reflected in their relatively low prediction probabilities.

The Naïve Bayes and decision tree classifiers are able to very accurately distinguish between the classes (100% accurate if the tuned threshold of 85% is taken into account for the decision tree classifier). Additionally, even though they were not trained to predict "moderate" modules, their probabilities for "unpopular" predictions decrease as the module popularity increases, suggesting the model is able to distinguish that those modules are less likely to be unpopular.

Based on these results, it is clear that the classifiers have indeed been successfully trained to distinguish between modules of different popularity, and not simply of different purpose.

5. Findings

Table 7 fulfills the original research objective of finding quality-related metrics that are common to popular or unpopular Drupal modules. The table summarizes the important metrics identified by the models described above. Table 7 also states which factor of the ISO25010 [5] software-quality model each metric is most related to, as well as whether the metric is a measurement of the module's code itself or the state of the project and its community.

Based on this summary, it appears that metrics related to the project itself and its community are more strongly connected to a module's popularity than metrics related to module source-code. Most of those project-related metrics appear to be measures of the activity and maturity of the community surrounding the module, which suggests they would be associated with the reliability of the module (i.e. to receive support or updates).

5.1. *Novelty of Results*

As no other similar study of Drupal modules was found in the literature, this appears to be the first recorded list of metrics correlated with Drupal module popularity. Of particular note are the metrics relating to the experience of module maintainers, the level of object-orientation in module source-code, the code complexity, and the amount of translation work, as these values are not presented on Drupal project web-pages. Furthermore, the thresholds for metrics relating to release and commit frequency, the number and experience of maintainers, the amount of translation work, the description page length, and the ratio of open issues provide an indication of expected values for these metrics in popular and unpopular modules.

Weber and Luo [23] found that in the case of Python projects on GitHub, source-code metrics were more relevant than author-related factors for predicting project popularity, though no attempt was made to train a model with a combination of these factors. This study indicates that the inverse may be the case for Drupal

Table 7. Important metrics for distinguishing between popular and unpopular modules

Related Class	Metric	ISO25010 Factor	Code or Project Metric?
Popular	Version available for next Drupal Core (8)	Portability	Project
Popular	More than 2 Maintainers	Reliability	Project
Popular	0.25% to ∼30% root-class/child-class ratio	Unknown	Code
Popular	Releases at least ∼70 days apart	Reliability	Project
Popular	Has documentation	Usability	Project
Popular	Average of more than ∼1100 commits on Drupal.org per maintainer	Reliability	Project
Popular	Average translation progress greater than ∼11%	Usability	Project
Popular	Latest commit less than ∼4 months ago	Reliability	Project
Popular	Description page more than ∼1600 characters long	Usability	Project
Popular	Has automated tests	Maintainability	Code
Popular	0.01% to 49% or 51% to 66% of issues open	Reliability	Project
Popular	Extended cyclomatic complexity score greater than ∼∼100	Unknown	Code
Popular	Less than ∼5 months or more than ∼3.3 years since last release	Reliability	Project
Unpopular	Latest commit more than ∼2 years ago	Reliability	Project
Unpopular	1 or no maintainers	Reliability	Project
Unpopular	No automated tests	Maintainability	Code
Unpopular	No version available for next Drupal Core (8)	Portability	Project
Unpopular	0%, ∼50%, or more than 66% of issues open	Reliability	Project
Unpopular	No recommended release	Reliability	Project
Unpopular	Average translation progress less than ∼1%	Usability	Project
Unpopular	Extended cyclomatic complexity score less than ∼∼100	Unknown	Code
Unpopular	More than ∼5 months but less than ∼3.3 years since last release	Reliability	Project

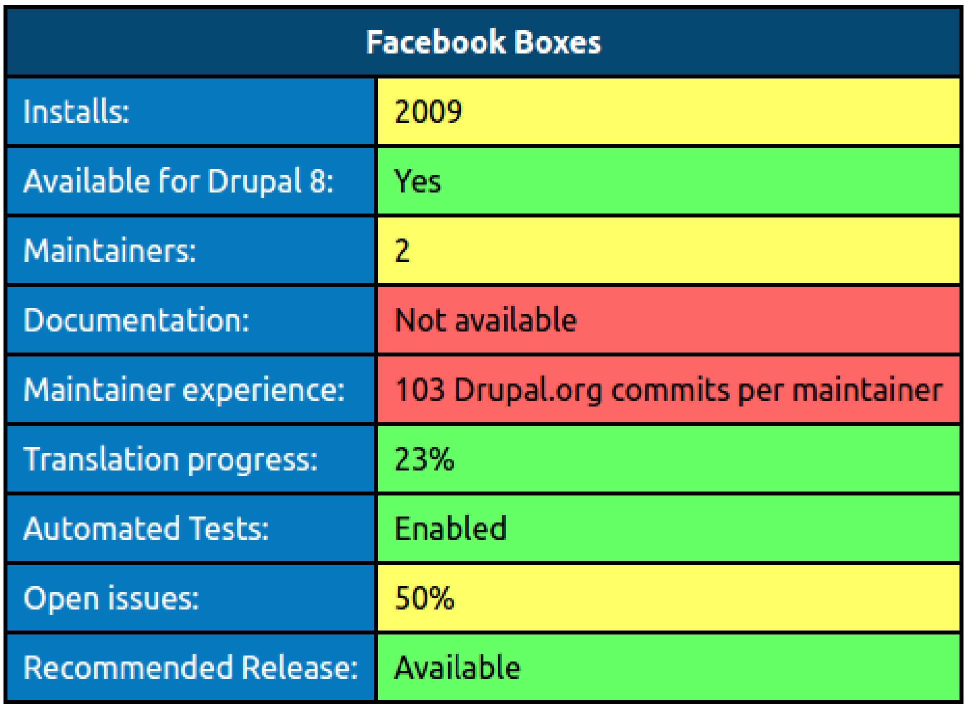
modules: source-code metrics appear to be less related to project popularity than project-related metrics (including the number of maintainers and their experience).

Emanuel et al. [17] found that projects with high download counts on the Source-Fourge open-source project hosting platform were more likely to have a “stable / production” development status. In the metrics identified by this study, only the absence of a “recommended release” was a useful metric for identifying unpopular Drupal modules. This suggests that while modules without a “recommended release” are likely to be unpopular, a module with a “recommended release” may be popular or unpopular.

The fact that maintainer experience was identified as an important metric in this study is consistent with the results reported by Cai and Zhu [18], who found that open-source projects with higher aggregate developer reputation on the Ohloh social network were more likely to be popular.

5.2. *Implications*

As the Drupal Association has recently removed the restriction of only allowing peer-reviewed developers to fully publish modules, there is a potential risk that many more low-quality modules could appear on Drupal.org. For this reason, the Drupal Association is looking for ways to present a clearer distinction between good and poor modules to Drupal website builders. The identified metrics are good candidates to be considered for presentation as measures of quality on module project pages. While many of these metrics are already presented in some form, they could potentially be grouped together and emphasized as project quality indicators. Additionally, the identified thresholds for the metrics could be used to create a scale for communicating module quality, such as a "red, amber, green" colour scale for metric values associated with unpopular, moderately popular, and popular modules (respectively). An example of how this could be applied to create a module evaluation tool is demonstrated in Figure 4. A tool such as this would help developers make informed decisions about whether to use a module, even when they have no prior knowledge of reasonable values for project metrics. Such an evaluation tool could also be useful for other open-source communities, as long as the included metrics were verified as being important indicators of project quality for the given context.



Facebook Boxes	
Installs:	2009
Available for Drupal 8:	Yes
Maintainers:	2
Documentation:	Not available
Maintainer experience:	103 Drupal.org commits per maintainer
Translation progress:	23%
Automated Tests:	Enabled
Open issues:	50%
Recommended Release:	Available

Fig. 4. Example of a module evaluation tool applied to the "Facebook Boxes" module

As automated code-quality checks are being considered by the Drupal Association and other open-source communities, it is worth noting that source-code quality metrics were not good distinguishers of module popularity; popular and unpopular modules could have a range of values for these metrics. The commonly held idea that popular open-source software projects with more experienced developers and more community attention have higher source-code quality appears to be incorrect. This does not necessarily mean that it is not worth running automated checks to remind developers to conform to coding standards and write maintainable code, but perhaps there is an argument to be made that a project with low source-code quality is still a worthy contribution to the open-source community, and may be useful for many thousands of users. At the very least, project-related metrics appear to be better indicators of project success than traditional software quality metrics.

6. Limitations

It is worth noting that the discovered metric/popularity relationships are not necessarily causal. For example, in the case of the majority of metrics associated with reliability: it is not known whether a module that becomes popular will become more reliable as its community grows, or if a module achieves popularity because of its reliability.

Furthermore, while the potential confounding factor of module purpose has been ruled out by applying the models to a group of modules with a similar purpose, it is possible that other confounding factors may affect module popularity. For example, this study did not take into account internal factors that were impractical to collect on a large scale, such as whether a module has any functional issues preventing its use, the extent and quality of documentation, and how easy the module is to install and configure. Also, external factors were not considered, such as the amount of promotion a module had received.

Additionally, while a popular module can safely be assumed to be of high quality in that it is useful for many Drupal websites, it is not necessarily the case that popular modules are exemplars of high software quality. In fact, the absence of a strong link between traditional software quality metrics and module popularity suggests that this may not be the case. Therefore, while the metrics identified by this study can be taken as indicators of modules that are useful to the community, they do not necessarily indicate that a module is well engineered.

7. Conclusions and Future Work

In summary, the data-mining process has identified a collection of metrics that can be used to distinguish between popular and unpopular Drupal modules. Some of these metrics are novel in that they are not currently presented on project webpages, and threshold values have been discovered for other metrics. An approach has also been proposed to allow Drupal developers to make informed choices based on the identified metrics and threshold values. Furthermore, this study has highlighted a strong relationship between module popularity and project-related metrics associated with software reliability, while revealing the absence of a strong relationship between module popularity and traditional software-quality metrics.

This study has also highlighted a number of areas for further research into the quality of open-source software.

Firstly, the presented module evaluation tool could be refined further and tested with Drupal website developers to evaluate whether they find it useful for deciding whether to use a module. An experimental trial could be set up to enable comparisons to be made. A comparison can be made between the decisions of more experienced developers with those made by both less experienced developers who did not use the tool as well as less experienced developers who did use the tool. Such an analysis would enable the utility of the tool to be ascertained. If the tool is found to be useful, similar tools could also be trialed in other open-source communities.

As the quantity and experience of module maintainers was shown to have a strong relationship with module popularity, further research could be performed to understand the relationship between module popularity and more detailed metrics about maintainers. A good example of this approach is presented by [18], where metrics such as maintainer reputation and the deviation of experience among maintainers were examined.

To understand how features in the dataset (including popularity) are associated with the software-engineering quality of modules, expert analysis of modules would need to be performed to qualitatively evaluate modules for software-engineering quality. Alternatively, it may be possible to determine quantitative measures for software-engineering quality through interviews with expert Drupal module developers. With measurements of Drupal software-engineering quality, a similar study to this could be performed to identify metrics associated with well or poorly engineered modules.

The dataset of Drupal 7 modules produced for this study could also be of use for further analysis by members of the Drupal community, such as in identifying typical values for software metrics to be included in any automated code-analysis implemented on Drupal.org.

Additionally, the approach taken in this study could be re-applied to study other open-source software projects, including Drupal Themes (which alter the appearance of Drupal websites) and modules for Drupal 8 (when the ecosystem has become more mature and made more data available).

To understand the nature of the relationships between software metrics and popularity in open-source projects (such as Drupal modules), the progression of metric values over time would need to be studied for a collection of projects. In particular, the project histories would need to be studied from an early point in their life-cycle, which could reveal whether projects become popular because of certain metric values, or if the metric values are a consequence of a project achieving popularity.

This study has shown that mining data about open-source software projects is a fruitful activity with the potential to aid the selection process for developers wishing to use them. The area also presents a number of opportunities for further research.

Appendix A. Dataset Features

Table 8: Full list of module metrics collected for the dataset

automated_tests_status	**Data type**: Boolean **Source**: Project page **Included in final models**: Yes Whether the module has automated testing by Drupal CI enabled on Drupal.org.
available_for_next_core	**Data type**: Boolean **Source**: Project page **Included in final models**: Yes Whether the module has a version available for the next major version of Drupal Core (as the scraped modules were all for Drupal 7, the presence of Drupal 8 modules was checked).
average_file_loc	**Data type**: Numeric **Source**: Source-files and PHPDepend report loc **Included in final models**: Yes The average lines of code per source-code file in the module.
average_maintainer_commits	**Data type**: Numeric **Source**: Project and user profile pages **Included in final models**: Yes The average number of commits each maintainer of the module has made to *any* project on Drupal.org. Included as a measure of the average experience of module maintainers.
average_npath	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The average npath complexity of all functions/methods in the module's source-code.
average_translation_percent	**Data type**: Numeric **Source**: Translations page **Included in final models**: Yes The average percentage of translations that exist for the module in different languages.
calls_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of method/function calls to the total lines of source-code.
ccn	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: No Cyclomatic complexity score.

Table 8: (Continued)

ccn2	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes Extended cyclomatic complexity score.
cloc_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of comment lines of source-code to the total lines of source-code.
clsa_noc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of abstract classes to the total lines of source-code.
commit_frequency_days	**Data type**: Numeric **Source**: Commits feed **Included in final models**: Yes The average number of days between the last (up to 10) Git commits to the project.
committer_count	**Data type**: Integer **Source**: Committers page **Included in final models**: No The total number of Drupal.org users who have committed code to the project.
days_since_release	**Data type**: Integer **Source**: Project page **Included in final models**: Yes The number of days since the latest stable release (or unstable release if no stable release exists) for Drupal 7.
dependencies_count	**Data type**: Integer **Source**: `.info` file **Included in final models**: Yes The number of other modules this module depends on.
description_images	**Data type**: Integer **Source**: Project page **Included in final models**: Yes The number of images in the project's description.
description_length	**Data type**: Integer **Source**: Project page **Included in final models**: Yes The number of text characters in the project's description.
development_status	**Data type**: Nominal[“Under active development”, “Maintenance fixes only”, “No further development”, “Obsolete”, “Unknown”] **Source**: Project page **Included in final models**: Yes The development status of the module advertised by the module's maintainers.

Table 8: (Continued)

eloc_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of "executable" lines of source-code to the total lines of source-code.
growth_per_week	**Data type**: Numeric **Source**: Usage page **Included in final models**: No The average increase or decrease in the install count of the Drupal 7 version of the module over the last (up to) 10 weeks.
has_documentation	**Data type**: Boolean **Source**: Project page **Included in final models**: Yes Whether the module has documentation pages linked to from its project page.
install_count	**Data type**: Integer **Source**: Usage page **Included in final models**: No The number of Drupal websites that currently report having the Drupal 7 version of the module installed.
install_download_ratio	**Data type**: Numeric **Source**: Project page **Included in final models**: No The ratio of the total number of installs for the module (across all Drupal versions) to the number of times the module has been downloaded from Drupal.org.
issue_count	**Data type**: Integer **Source**: Project page **Included in final models**: No The total number of issues in the module's issue queue.
issue_new_count	**Data type**: Integer **Source**: Project page **Included in final models**: No The current number of "new issues" as reported on the module's project page.
issue_open_ratio	**Data type**: Numeric **Source**: Project page **Included in final models**: Yes The ratio of open issues to the total number of issues in the project's issue queue.
issue_participants	**Data type**: Integer **Source**: Project page **Included in final models**: No The current number of issue queue participants as reported on the module's project page.

Table 8: (Continued)

issue_response_hours	**Data type**: Integer **Source**: Project page **Included in final models**: No The current number of hours until “1st response” in the issue queue as reported on the module’s project page.
issue_response_rate	**Data type**: Numeric **Source**: Project page **Included in final models**: No The current issue queue “response rate” as reported on the module’s project page.
lloc_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of “logical” lines of source-code to the total lines of source-code.
loc	**Data type**: Integer **Source**: PHPDepend report **Included in final models**: Yes The total lines of source-code.
maintainer_count	**Data type**: Integer **Source**: Project page **Included in final models**: Yes The number of maintainers that currently exist for the module.
maintenance_status	**Data type**: Nominal[“Seeking co-maintainer(s)”, “Actively maintained”, “Minimally maintained”, “Unsupported”, “Seeking new maintainer”, “Unknown”] **Source**: Project page **Included in final models**: Yes The maintenance status of the module advertised by the module’s maintainers.
noc_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of the number of object-oriented classes to the total lines of source-code.
nof_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of the number of functions to the total lines of source-code.
noi_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of the number of object-oriented interfaces to the total lines of source-code.
nom_loc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of the number of methods to the total lines of source-code.

Table 8: (Continued)

project_age_days	**Data type**: Integer **Source**: Project page **Included in final models**: No The number of days since the module was first published on Drupal.org
recommended_release	**Data type**: Boolean **Source**: Project page **Included in final models**: Yes Whether the module has a stable release for Drupal 7 recommended by the module's maintainers.
release_frequency_days	**Data type**: Numeric **Source**: Release feed **Included in final models**: Yes The average number of days between the last (up to 10) Drupal 7 releases of the module.
roots_noc_ratio	**Data type**: Numeric **Source**: PHPDepend report **Included in final models**: Yes The ratio of root classes (classes that do not inherit from another class) to the total number of classes in the source-code.
seconds_since_last_commit	**Data type**: Integer **Source**: Commits feed **Included in final models**: Yes The number of seconds since the last Git commit to the module.
security_covered	**Data type**: Boolean **Source**: Project page **Included in final models**: Yes Whether the module is covered by the Drupal Security Team.
total_growth_per_week	**Data type**: Numeric **Source**: Usage page **Included in final models**: No The average increase or decrease in the install count of *all* versions of the module over the last (up to) 10 weeks.
total_install_count	**Data type**: Boolean **Source**: Project page **Included in final models**: Yes The number of Drupal websites that currently report having *any* version of the module installed.
phpcs_ratio_total	**Data type**: Numeric **Source**: phpcs report **Included in final models**: Yes The ratio of the total number of phpcs notices to the total lines of source-code.
For each notice type in the PHPCodesniffer report:	

Table 8: (Continued)

phpcs_ratio_A.B.C.D.E	**Data type**: Numeric **Source**: phpcs report **Included in final models**: Yes The ratio of phpcs notices of type "A.B.C.D" to the total lines of source-code.
phpcs_ratio_level_0_A	**Data type**: Numeric **Source**: phpcs report **Included in final models**: Yes The ratio of phpcs notices starting with "A." to the total lines of source-code.
phpcs_ratio_level_N_PART	**Data type**: Numeric **Source**: phpcs report **Included in final models**: Yes The ratio of phpcs notices starting with "PART" to the total lines of source-code.

Table 9. Sources for module metrics

Project page	https://www.drupal.org/project/$MODULE
Commits feed	https://www.drupal.org/node/$MODULE_ID/commits/feed
User profile page	https://www.drupal.org/u/$USERNAME
Usage page	https://www.drupal.org/project/usage/$MODULE
Release feed	https://www.drupal.org/node/$MODULE_ID/release/feed ?api_version[0]=103
Translations page	https://localize.drupal.org/translate/projects/$MODULE
Commits feed	https://www.drupal.org/node/$MODULE_ID/commits/feed
Committers page	https://www.drupal.org/node/$MODULE_ID/committers
PHPDepend report	Output of running the PHPDepend static-code-analysis tool on the module's source-code
`.info` file	The `.info` file included in the source-code of the module
phpcs report	List of "code smell" notices reported by the PHP Codesniffer tool when evaluated against the Drupal, DrupalPractice, and DrupalSecure coding standards.

References

[1] Tim Lehnen. Goodbye project applications, hello security advisory opt-in. `https://www.drupal.org/drupalorg/blog/goodbye-project-applications-hello-security-advisory-opt-in`, March 2017.

[2] Mark Aberdour. Achieving quality in open-source software. *IEEE software*, 24(1), 2007.

[3] Ying Zhou and Joseph Davis. Open source software reliability model: an empirical approach. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM, 2005.

[4] Sang-Yong Tom Lee, Hee-Woong Kim, and Sumeet Gupta. Measuring open source software success. *Omega*, 37(2):426–438, 2009.

[5] International Organization for Standardization. ISO/IEC 25010:2011. Standard, Geneva, CH, March 2011.

[6] Sahra Sedigh-Ali, Arif Ghafoor, and Raymond A Paul. Metrics and models for cost and quality of component-based software. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 149–155. IEEE, 2003.

[7] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. The sqo-oss quality model: measurement based open source software evaluation. *Open source development, communities and quality*, pages 237–248, 2008.

[8] Quinn Taylor, Christophe Giraud-Carrier, and Charles D Knutson. Applications of data mining in software engineering. *International Journal of Data Analysis Techniques and Strategies*, 2(3):243–257, 2010.

[9] Xiaobing Sun, Xiangyue Liu, Bin Li, Yucong Duan, Hui Yang, and Jiajun Hu. Exploring topic models in software engineering data analysis: A survey. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016 17th IEEE/ACIS International Conference on*, pages 357–362. IEEE, 2016.

[10] Du Zhang and Jeffrey JP Tsai. Machine learning and software engineering. *Software Quality Journal*, 11(2):87–119, 2003.

[11] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2007.

[12] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[13] Gavin R Finnie, Gerhard E Wittig, and Jean-Marc Desharnais. A comparison of software effort estimation techniques: using function points with neural networks, case-based reasoning and regression models. *Journal of systems and software*, 39(3):281–289, 1997.

[14] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. Dretom: Developer recommendation based on topic models for bug resolution. In *Proceedings of the 8th international conference on predictive models in software engineering*, pages 19–28. ACM, 2012.

[15] Ahmed E Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 161–166. ACM, 2010.

[16] A. Adewumi, S. Misra, N. Omoregbe, B. Crawford, and R. Soto. A systematic literature review of open source software quality assessment models. *SpringerPlus*, 5(1), 2016.

[17] A. W. R. Emanuel, R. Wardoyo, J. E. Istiyanto, and K. Mustofa. Success factors of oss projects from sourceforge using datamining association rule. In *2010 International Conference on Distributed Frameworks for Multimedia Applications*, pages 1–8, Aug 2010.

[18] Yuanfeng Cai and Dan Zhu. Reputation in an open source software community: Antecedents and impacts. *Decision Support Systems*, 91:103 – 112, 2016.

[19] I. Stamelos, A. Oikonomou, L. Angelis, and G.L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[20] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Software defect prediction using relational association rule mining. *Information Sciences*, 264(Serious

Games):260 – 278, 2014.

[21] H. Barkmann, R. Lincke, and W. Löwe. Quantitative evaluation of software quality metrics in open-source projects. In *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, pages 1067–1072, Software Technology Group, School of Mathematics and Systems Engineering, Växjö University, 2009.

[22] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016*, pages 351–361, IBM T. J. Watson Research Center, 2016.

[23] S. Weber and J. Luo. What makes an open source code popular on git hub?. In *IEEE International Conference on Data Mining Workshops, ICDMW*, pages 851–855, Department of Computer Science, University of Rochester, 2015.

[24] Klaus Purer. Pareview.sh. `https://www.drupal.org/project/pareviewsh`, October 2011.

[25] Jospeh Purcell. Improving code quality with static analysis. `https://events.drupal.org/dublin2016/sessions/improving-code-quality-static-analysis`, September 2016.

[26] Tim Mallezie. Coverity scan for drupal core. `https://www.drupal.org/node/2844865`, January 2017.

[27] Tim Lehnen. [par phase 4] improve project discovery with strong project quality signals(automated) and incentivize peer code review(manual). `https://www.drupal.org/node/2861341`, March 2017.

[28] Dan Morrison. drupal-code-metrics. `https://github.com/dman-coders/drupal-code-metrics`, March 2015.

[29] A. Azevedo and M.F. Santos. Kdd, semma and crisp-dm: A parallel overview. In *MCCSIS'08 - IADIS Multi Conference on Computer Science and Information Systems; Proceedings of Informatics 2008 and Data Mining 2008*, pages 182–185, 2008. cited By 57.

[30] I. H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Getting to Know Your Data*, chapter 2, page 65. Cambridge, MA : Morgan Kaufmann Publisher, [2016], 2016.

[31] Manuel Pichler. Phpmd - php mess detector. `https://phpmd.org/`, April 2016.

[32] Jean-François Lépine. Phpmetrics. `http://www.phpmetrics.org/`, 2015.

[33] Sebastian Bergmann. Phploc. `https://github.com/sebastianbergmann/phploc`, May 2017.

[34] Manuel Pichler. Php depend. `https://pdepend.org/`, January 2017.

[35] Squiz Labs. Php codesniffer. `https://github.com/squizlabs/PHP_CodeSniffer`, May 2017.

[36] Doug Green. Coder. `https://www.drupal.org/project/coder`, December 2006.

[37] Mark W. Jarrell, Dan Feidt, Sandip Tekale, and William Turrell. Installing coder sniffer. `https://www.drupal.org/node/1419988`, March 2017.

[38] Ben Jeavons. Drupalsecure code sniffs. `https://www.drupal.org/sandbox/coltrane/1921926`, February 2013.

[39] Amit Goyal, Derek Wright, Chris Johnson, and Isaac Sukin. Popularity of modules. `https://www.drupal.org/node/329620`, September 2012.

[40] Drupal Association. Module project index. `https://www.drupal.org/project/project_module/index`.

[41] Albert Skibinski. Drupal 8: which drupal 7 modules went into core? `http://blog.merge.nl/2014/01/21/drupal-which-drupal-7-modules-went-into-drupal-core`, January 2014.

[42] Manuel Pichler. Cyclomatic complexity. `https://pdepend.org/documentation/software-metrics/cyclomatic-complexity.html`, January 2016.

[43] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, 1993.

[44] I. H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Appendix B: The Weka Workbench*, pages 553–571. Cambridge, MA : Morgan Kaufmann Publisher, [2016], 2016.

[45] William W. Cohen. Fast effective rule induction. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[46] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[47] I. H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Unbalanced Data*, pages 64–65. Cambridge, MA : Morgan Kaufmann Publisher, [2016], 2016.

[48] Nitesh V. Chawla. *Data Mining for Imbalanced Datasets: An Overview*, pages 853–867. Springer US, Boston, MA, 2005.

[49] I. H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Attribute Selection*, pages 288–289. Cambridge, MA : Morgan Kaufmann Publisher, [2016], 2016.

[50] I. H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Simple Probabilistic Modeling*, page 96. Cambridge, MA : Morgan Kaufmann Publisher, [2016], 2016.

[51] T.J. Mccabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.