Investigating the Popularity of Open Source Code

ABSTRACT

The rise of social networks for software development has attached a notion of popularity to open source projects. This work attempts to extract knowledge from the differences between popular and unpopular Python projects on GitHub. Projects were mined for a variety of features that measure language utilization, documentation, and code volume. These features were used to train a classifier which predicted current popularity well (F-score = .8). Notably, these features outperformed measures of author popularity (F-score = .7). However, these features did not strongly predict future growth in popularity. Popular projects have far more collaborators, suggesting that these features could be useful as a measure of not only popularity, but of code quality.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; H.2.8 [**Database management**]: Database Applications—*Data Mining*; I.5.4 [**Pattern Recognition**]: Applications

General Terms

Algorithm, Experimentation, Application, Measurement

Keywords

Social media retrieval, open source software, popularity prediction, Python, GitHub

1. INTRODUCTION

This work exclusively analyzes Python projects on GitHub, due to our domain knowledge with both the Python open source community and GitHub. GitHub is an increasingly popular destination, with 3.5 million users and 6 million repositories as of April 2013 [11].

GitHub stars were used in our popularity measurement, which are similar to a 'like' on Facebook. Project forks – which are created when a user would like to contribute – were also considered for this purpose. However, many forks end with no action, so stars were decided to be preferable.

Specifically, popularity was measured with GitHub star velocity. This is the average growth of stars over time, calculated as the current number of stars divided by project age.

The idea of classifying a body of work to its popularity is not new. Music, YouTube videos, Twitter messages, and news articles are just a few of the domains examined [7, 10, 4, 14]. However, such work typically focuses on classification accuracy, not information retrieval. Most of this work employs black-box learning models (such as SVMs).

Previous work in our domain seems focused on software maintenance issues [13, 8]. More relevant is a Kaggle competition which required participants to classify code to certain open-source projects [5]. We were unable to find work that focused on the popularity of code, motivating us to investigate what knowledge could be extracted.

2. DATA

GitHub allows users to declare the language of their project. If left unspecified, GitHub will classify the project using its own open source tool [2]. This classification result was assumed to be the ground truth; we have not observed any misclassifications.

GitHub does not provide historical data. Therefore, our data collection represents snapshots of GitHub projects at a certain date and time.

GitHub provided a list of all Python projects present in November 2012 (about 300 thousand). We then used GitHub's API to gather repository metadata, such as stars, date of creation and author username. We performed this process twice: once in November 2012, and once in March 2013. These two collections will be referred to as the fall snapshot and winter snapshot, respectively. The winter snapshot was collected only for later attempts at future popularity prediction; the fall snapshot is being referred to unless otherwise mentioned.

Popularity is very unevenly distributed: 72% of all projects have 0 or 1 stars. Only 5% of projects have more than 10 stars.

When choosing our samples, projects created in the previous month, as well as those with less than 5 stars or star velocity below 1 star per month were excluded because such data is unreliable. This left about 13 thousand projects for evaluation.

Two classes, low popularity and high popularity, were then artificially created. This choice was motivated by the extremely uneven distribution of popularity: setting the cutoff too high would not allow for sufficient high popularity training samples. The star velocity of the 1000th most popular project was chosen as a cutoff, which represented a star velocity of slightly more than 1 star per week.

The full GitHub repository histories of the top 1000 and bottom 1000 projects by star velocity were then collected as training data for the two classes.

3. FEATURES

The choice to analyze exclusively Python projects allowed for very targeted features. 38 features were considered in total. Twenty of these measured the relative occurrence of various Python abstract syntax tree (AST) nodes [12]. The remaining 18 features evaluated a range of metrics, including project volume, documentation volume, presence of supporting files, and standard library usage. Code volume was measured by AST nodes, not lines of code.

Standard library usage (as a set of module names that are likely to be imported) was the only non-numeric feature; it was vectorized with a binary one-hot coding. N-grams were built for n<4. Due to Python's dynamic nature, we cannot tell if a module would be imported if code were run. Therefore, we included all modules found in an import statement anywhere in the project. Modules from outside the standard library were collected but not used in training, as they were not used frequently enough across projects.

Python style guide adherence was planned as a feature, but the normal methods to compute it were too slow to be practical [15].

A complete list of features is included in our code, available on GitHub.

4. CURRENT POPULARITY

These features were used to train a random forest classifier for prediction of current popularity [1]. This classifier was chosen due to its ability to be introspected, as well as its performance with noisy and redundant data.

Note, however, that classifier performance was not our primary goal. Rather, this task supported information retrieval in two ways. First, obtaining reasonable performance would support our hypothesis that differences exist between the classes. Second, information from the white-box model could be used guide our investigation of class differences.

Project scikit-learn (version 0.13.1) provided the random forest classifier implementation [9]. A grid search found that using 200 estimators with fully-built trees would improve performance. 5-fold stratified cross validation saw strong performance, as shown in Table 1 [3]. These results suggest that popularity is well captured by our in-project features.

Table 1: Performance on Current Popularity Task

	low popularity	high popularity
precision	.766	.799
recall	.803	.758

Table 2: Author Metadata Performance

	low popularity	high popularity
precision	.673	.695
recall	.687	.660

All in-project features were included in training and evaluation. Due to the large number of features, feature selection was attempted, including various univariate measures, recursive elimination, and L1-based selection. To our surprise, none of these methods greatly improved performance. In addition, the various methods yielded quite different final sets of features. That performance did not differ greatly across different feature subsets suggests that the popularity signal is highly redundant across our features.

We also attempted to utilize the greater number of unpopular projects in training. Random undersampling, oversampling and asymmetric bagging all resulted in worse performance than our manual undersampling [3, 6].

To provide perspective for our results, we also trained the same classifier on a separate feature set: author metadata (number of followers, number following, number of projects). As author metadata contains direct measures of popularity, we expected strong performance. However, the classifier trained on the in-project features outperformed the classifier trained on author metadata, as shown in Table 2.

We did not train a classifier on the union of the two feature sets. Again, our goal was not classifier performance, but rather investigation of the class differences.

4.1 Feature Performance

Decision trees allow estimation of feature importance from rank in the tree, and averaging rank over a random forest allows for a reduced-variance feature importance measure. Table 3 shows the most distinguishing features by this measure.

Most features were deemed unimportant, which clearly displays the redundancy in the data.

Table 3: Feature Performance

Feature	Relative Importance
readme size	.1465
with statement usage	.1430
TravisCI configuration size	.0640
class definitions	.0390
comment ratio	.0330

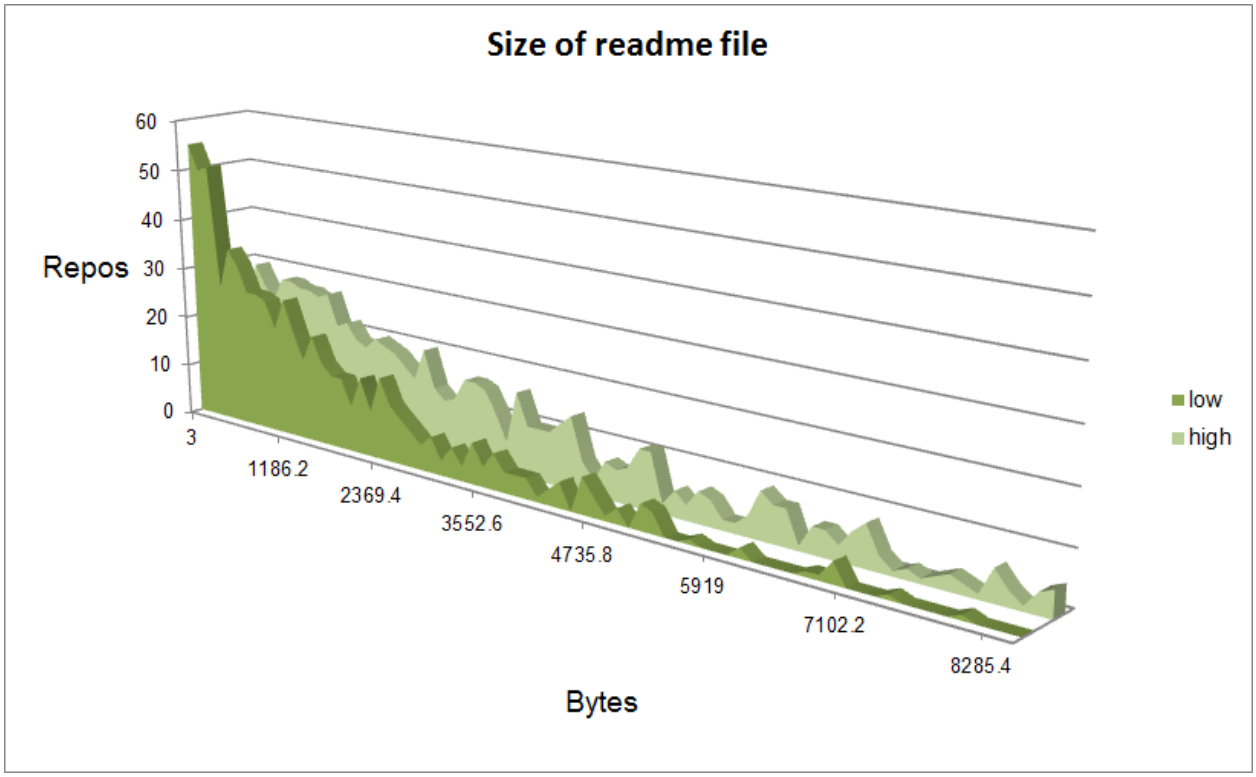

Figure 1: High popularity projects have larger README files (outliers omitted)

The top three features warrant discussion. On GitHub, the README file is presented automatically to visitors of the project. Upon investigation, popular projects were found to have larger READMEs (median 2 kilobytes vs 500 bytes). Also, 95% of popular projects have nonempty READMEs, compared to only 65% of unpopular projects. Figure 1 shows these trends, while omitting a high popularity long tail and low popularity spike at 0 bytes.

This is a somewhat obvious result. We suspect that since popular projects have more users, they feel more need to present information to them. Interestingly, various forms of internal documentation (such as comments) were also more frequent in popular projects, though not deemed to be as powerful of a signal.

More interesting is the second best feature. The *with statement* is a Python construct best employed when two related operations need to be executed symmetrically: common examples are the opening and closing of a file, or the locking and unlocking of a mutex. The feature is syntactic sugar, but considered best practice in appropriate situations.

We found that popular projects used the *with statement* more frequently. This is shown in Figure 2, which again omits a high popularity long tail and low popularity spike at 0. The reasons for this are likely a combination of factors. More users contribute to popular code: the median number of forks in the high popularity sample is 22, but only 1 in the low popularity sample. Thus, we suspect that more popular projects tend to be of higher quality. Of course, we cannot rule out that popular projects may simply deal with appropriate scenarios for this construct more often than unpopular code.

Table 4: Difference in Stdlib Module Usage

module name	factor	high popularity	low popularity
`functools`	4.66	.27	.058
`collections`	4.08	.31	.076
`json`	3.70	.37	.1
`imp`	2.76	.16	.058
`errno`	2.86	.18	.063
`uuid`	2.75	.14	.051
`warnings`	2.44	.22	.09
`__future__`	2.38	.31	.13
`operator`	2.25	.2	.089
`base64`	2.27	.25	.11

The third feature captures use of TravisCI, a hosted continuous integration service popular in the open source community. It is commonly configured to run a test suite automatically after code changes. 22% of high popularity projects have non-empty TravisCI configuration files, compared with less than 1% of low popularity projects.

Note that the imported modules feature could not be properly included in this calculation due to its vectorization as many separate features. Upon removing it from consideration, F-score dropped by .05. Table 4 shows the largest relative differences (excluding modules used in less than 5% of projects).

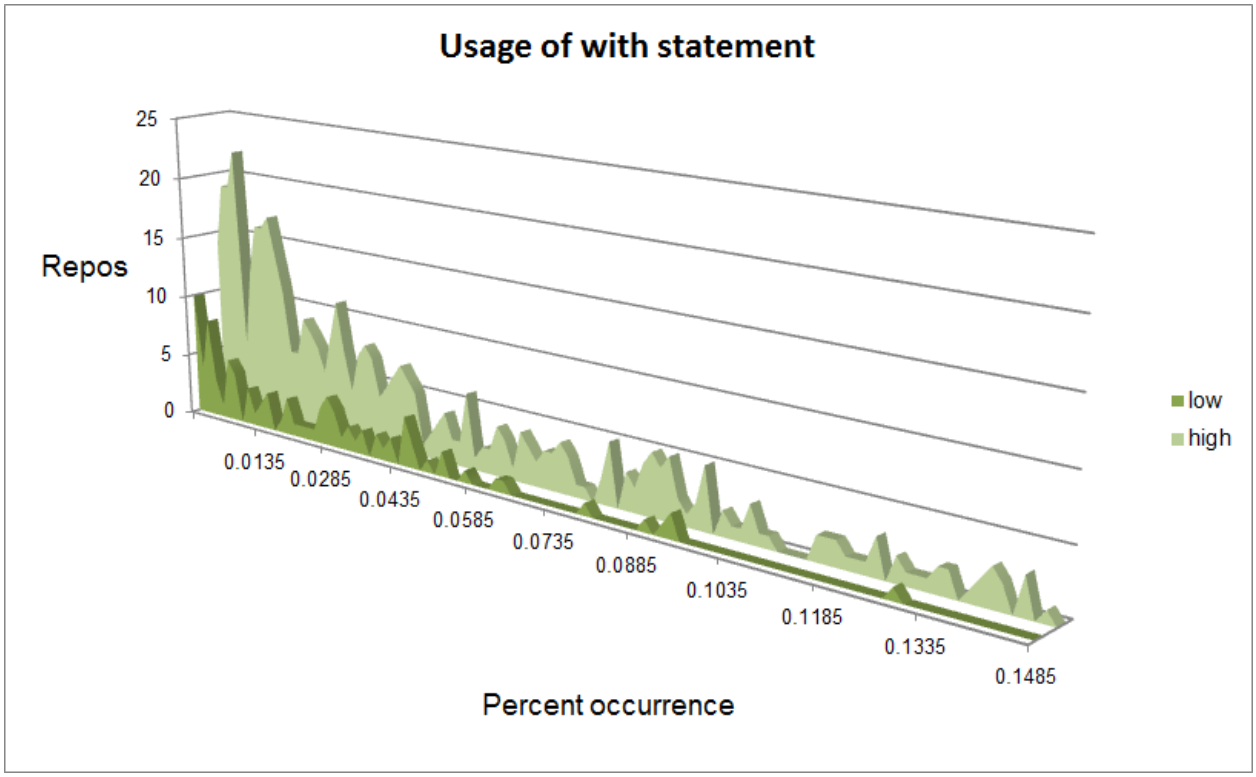

Figure 2: High popularity projects use the *with statement* more often (outliers omitted)

The difference in usage for the first two modules is striking. Like the *with statement*, the `functools` and `collections` modules can often simplify code, but are never necessary. The `operator` module, further down the list, has a similar purpose. We believe these usage differences are evidence of a code quality separation between the two classes.

The `json` and `base64` modules' usage does not seem as interesting: it is reasonable that highly popular projects would need to be more flexible with inputs and outputs.

The `imp`, `errno` and `uuid` modules are similar to each other: each solves a very specific technical problem that is unlikely to occur in basic projects. They are advanced tools, and it is not surprising to see them in the larger and more sophisticated popular projects.

The `warnings` and `__future__` modules both solve maintenance problems related to deprecation and backwards compatibility. Dealing with these issues could be considered the cost of success for popular projects.

5. FUTURE POPULARITY

Motivated by the strong performance of our current popularity classification, we also attempted to predict the future popularity of projects. Given the data from the fall snapshot, we attempted regression over the star velocity between the fall and winter snapshot periods. A scikit-learn random forest regressor with the same parameters as before was used.

Our performance for this task was weak: the mean r^2 after 5-fold cross validation was close to 0. However, we did outperform linear extrapolation, which had an r^2 of -.32.

Employing feature selection techniques and other estimators did not improve performance.

6. CONCLUSION AND FUTURE WORK

By analyzing Python projects on GitHub with machine learning techniques, we found that in-code features strongly signal current popularity, even more so than author metadata features. When investigating performant features, we found both unsurprising – popular projects have more documentation – and surprising – popular projects use the *with statement* more frequently – differences between popular and unpopular projects.

Of course, we cannot assume more than correlation between popularity and our features; merely adding *with statements* is not likely to increase the popularity of a project. However, we suggest that some features – especially those measuring AST node occurrence – may be useful as a measure of code quality, as popular projects tend to have far more contributors than unpopular projects. The most obvious practical application would be in developer tools: code could be compared to baseline usage values for popular projects, and over or under utilization of node types detected.

We used many features, but most were relatively simple. Specifically, we did not employ temporal features, nor any that exploit social graphs between projects. Either of these

may be worth consideration in the future.

Our sample size was limited by the number of popular projects. Expanding data collection to include other similar websites (SourceForge, Google Code, BitBucket, CodePlex, etc) could improve confidence in our results.

Comparable analysis could also be performed for other languages.

Our code and data is available on GitHub under a liberal open source license.

7. REFERENCES

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

[2] GitHub. Linguist. `https://github.com/github/linguist`.

[3] J. Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., third edition, 2012.

[4] L. Hong, O. Dan, and B. D. Davison. Predicting popular messages in twitter. In *Proceedings of the 20th international conference companion on World wide web*, WWW '11, pages 57–58, New York, NY, USA, 2011. ACM.

[5] Kaggle. EMC Israel data science challenge. `http://www.kaggle.com/c/emc-data-science`, Sept. 2012.

[6] Y. Liu, D. Xu, I. W. Tsang, and J. Luo. Textual query of personal photos facilitated by large-scale web data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(5):1022–1036, May 2011.

[7] Y. Ni, R. Santos-Rodríguez, M. McVicar, and T. De Bie. Hit song science once again a science? In *4th International Workshop on Machine Learning and Music: Learning from Musical Structure*, 2013.

[8] K. Nishizono, S. Morisaki, R. Vivanco, and K. Matsumoto. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 473–481, 2011.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] H. Pinto, J. M. Almeida, and M. A. Gonçalves. Using early view patterns to predict the popularity of youtube videos. In *Proceedings of the sixth ACM international conference on Web search and data mining*, WSDM '13, pages 365–374, New York, NY, USA, 2013. ACM.

[11] T. Preston-Werner. Five years. `https://github.com/blog/1470-five-years`.

[12] Python Software Foundation. ast module documentation. `http://docs.python.org/2/library/ast.html`.

[13] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 303–312, 2011.

[14] A. Tatar, P. Antoniadis, M. de Amorim, and S. Fdida. Ranking news articles based on popularity prediction. In *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*, pages 106–110, 2012.

[15] G. van Rossum. Style guide for python code. *Python Enhancement Proposals*, 2001.