Automatic Labeling of Software Components and their Evolution using Log-Likelihood Ratio of Word Frequencies in Source Code

Adrian Kuhn
Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch

Abstract

As more and more open-source software components become available on the internet we need automatic ways to label and compare them. For example, a developer who searches for reusable software must be able to quickly gain an understanding of retrieved components. This understanding cannot be gained at the level of source code due to the semantic gap between source code and the domain model. In this paper we present a lexical approach that uses the log-likelihood ratios of word frequencies to automatically provide labels for software components. We present a prototype implementation of our labeling/comparison algorithm and provide examples of its application. In particular, we apply the approach to detect trends in the evolution of a software system.

1. Introduction

In recent years, software vocabulary has been proven to be a valuable source for software analysis, often including the retrieval of labels (*e.g.* [4, 8, 11]). However, labeling software is not without pitfalls. The distribution of words in software corpora follows the same power-law as word frequencies in natural-language text [13]. Most of the text is made up of a small set of common terms, whereas content-bearing words are rare. Analysis of software vocabulary must deal with the reality of rare terms, thus statistical tests that assume normal distribution are not applicable. For example, textual comparison based on directly counting term frequencies is subject to overestimation when the frequencies involved are very small.

For text analysis the use of log-likelihood ratio improves the statistical results. Likelihood tests do not depend on assumptions of normal distribution, instead they use the asymptotic distribution of binomial likelihood [6]. Using log-likelihood ratios allows comparisons to be made between the significance of occurrences of both common and rare terms.

In this paper we present a lexical approach that uses the log-likelihood ratio of word frequencies to automatically retrieve labels from source code. The approach can be applied i) to compare components with each other, ii) to compare a component against a normative corpus, and iii) to compare different revisions of the same component. We present a prototype implementation and give examples of its application. In particular, we apply the approach to detect trends in the evolution of the JUnit software system.

The remainder of this paper is structured as follows: In Section 2 we provide the mathematical background of log-likelihood ratios. In Section 3 we present the application of our approach in two examples. In Section 4 we discuss related work. In Section 5 we conclude.

2. Log-Likelihood in a Nutshell

This section explains how log-likelihood ratio is applied to analyse word frequencies. The explanations are kept as concise as possible. We provide the general background and just enough details such that a programmer may implement the algorithm. Please refer to Ted Dunning's work [6] for more background.

The idea behind log-likelihood ratio is to compare two statistical hypotheses, of which one is a subspace of the other. Given two text corpora, we compare the hypothesis that both corpora have the same distribution of term frequencies with the "hypothesis" given by the actual term frequencies. Because we know that terms are not equally distributed over source code, we use binomial likelihood

$$L(p, k, n) = p^k (1 - p)^{n-k}$$

with $p = \frac{k}{n}$, where k is the term frequency (*i.e.* number of occurrences) and n the size of the corpus. Taking the logarithm of the likelihood ratio gives

$$-2\log\lambda = 2\big[\log L(p_1,k_1,n_1) + \log L(p_2,k_2,n_2) - \log L(p,k_1,n_1) - \log L(p,k_2,n_2)\big]$$

with $p = \frac{k_1+k_2}{n_1+n_2}$. The higher the value of $-2log\lambda$ the more significant is the difference between the term frequencies in of both text corpora. By multiplying the $-2log\lambda$ value with the signum of $p_1 - p_2$ we can further distinguish between terms specific to the first corpus and terms specific to the second corpus. Terms that are equally frequent in both corpora have a $-2log\lambda$ value close to zero and thus fall in between.

Example. Let C_1 be the corpus of a software project with size $n_1 = 10^6$, where the words `'rare'`, `'medium'`, and `'common'` appear respectively 1, 100, and 1×10^4 times; and let C_2 be the corpus of one of the project's classes with size $n_2 = 1000$, where each word appears 10 times. Then the log-likelihood ratio values are

	p_1	p_2	$-2log\lambda$	χ^2
`rare`	10^{-6}	10^{-2}	131.58	9.08
`medium`	10^{-4}	10^{-2}	71.45	0.89
`common`	10^{-2}	10^{-2}	0.00	0.00

The column χ^2 lists the value of Pearson's chi-square test, which assumes normal distribution. As we can see, there is an overestimation when the frequencies involved are very small. Therefore, text analysis should use log-likelihood ratios to compare the occurrences of common and rare terms [6].

3. Applications

In this section we present two example applications of log-likelihood ratio for software analysis. There are two main types of corpus comparison: comparison of a sample corpus to a larger corpus, and comparison of a two equally sized corpora. In the first case, we refer to the large corpus as a *normative* corpus since it provides as norm against which we compare.

Applications of these comparisons are

- *Providing labels for components.* Comparing a component's vocabulary with a large normative corpus (as *e.g.* Sourceforge, Github, or Sourcerer [3]), we obtain labels that describe the component. In the same way, we can compare a class's vocabulary against the containing project.
- *Comparing components to each other.* Comparing two components, we obtain labels to describe their differences as well as commonalities. This is applicable at any level of granularity, from the level of projects down to the level of methods.

`java.io`		`java.text`		`java.util`	
read	521.99	pattern	228.92	iterator	306.91
write	481.93	format	209.40	entry	301.90
skip	154.61	digits	183.24	next	237.82
close	113.41	FIELD	167.58	E	222.33
mark	111.47	instance	127.16	contains	187.69
println	99.66	fraction	104.98	sub	166.49
UTF	85.27	integer	102.77	of	165.57
flush	80.96	index	93.43	K	154.07
desc	69.19	run	90.99	T	154.30
TC	68.88	currency	91.55	key	145.10
prim	61.48	decimal	86.34	all	145.74
char	61.28	contract	84.92	V	142.15
buf	60.15	separator	72.11	remove	128.59
stream	56.86	grouping	62.26	last	128.09
fields	52.38	parse	56.93	map	115.68
bytes	47.99	collation	56.60	clear	114.03
...		...		...	
border	-28.35	UI	-23.07	create	-62.67
set	-33.89	border	-22.77	listener	-62.42
remove	-37.49	property	-24.23	action	-63.83
listener	-39.79	remove	-30.11	UI	-64.68
accessible	-40.97	accessible	-32.91	border	-63.83
paint	-44.90	listener	-31.97	accessible	-92.25
value	-59.10	type	-32.18	paint	-101.11
get	-64.38	paint	-36.07	get	-164.14

Table 1. Labels retrieved for three Java packages using the full Java 6.0 API as normative corpus.

- *Documenting the history of a component.* Comparing subsequent revisions of the same component, we obtain labels to describe the evolution of that component. (Using multinominal distribution we could even compare all revisions at once, although such results are harder to interpret [6].)

We implemented $-2log\lambda$ analysis in a Java prototype which is available on the HAPAX website[1] under AGPL license. In the remainder of this section we present results obtained with that prototype.

3.1. Labeling the Java API

In this example, we compare the packages `java.io` and `java.text` and `java.util` with the normative corpus of the full Java 6.0 API. We use the Java Compiler (JSR 199) to parse the byte-code of the full Java API and then extract the vocabulary of all public and protected elements. We extract the names of packages, classes (including interfaces, annotations, and enums), fields, methods and type parameters. We split the extracted names by camel-case to accomodate to the Java naming convention.

Results are shown in Table 1. For each package we list the most specific words and the least specific words. All three packages are characterized by not covering UI code, in addition `java.io` and `java.util` have obviously substantially fewer `get`-accessors than is usual for the Java API. The remaining findings offer no fur-

1 `http://smallwiki.unibe.ch/adriankuhn/hapax`

ther surprises, except maybe for the uppercase letters in `java.util` which are generic type parameters; obviously the majority of the Java 6.0 API makes less use of generic types than the collection framework.

3.2. The Evolution of JUnit

In this example, we report on the vocabulary trends in the history of the `JUnit`[2] project. We use a collection of 14 release distributions of JUnit and parse the source code of each release. We compare the vocabulary of each two subsequent releases and report on the most significant changes in the vocabulary. We extract all words, including comments; split by camel-case, and exclude English stopwords but not Java keywords.

Results are shown in Table 2. For each release we list the top removed terms and the top added terms, as well as the $-2log\lambda$ value of the top-most term. Large $-2log\lambda$ values indicate substantial changes.

The top 7 change trends (*i.e.* $-2log\lambda \geqslant 100.0$) in the history of JUnit are as follows. In 3.2 removal of `MoneyBag` example and introduction of graphical UI; in 4.0 removal of graphical UI and introduction of annotation processing; in 4.2 removal of HTML tags from Javadoc comments; in 4.4 introduction of theory matchers and `hamcrest` framework; in 4.5 introduction of blocks and statements. We manually verified these findings with the release notes of JUnit and found that the findings are appropriate.

JUnit	$2log\lambda$	Top-10 terms (with $-2log\lambda \geqslant 10.0$)
3	-8.21	
	54.11	count, writer, wrapper
3.2	-382.80	money, CHF, assert, case, USD, equals, test, fmb, result, currency
	114.21	tree, model, constraints, combo, reload, swing, icon, pane, browser, text
3.4	-19.48	stack, util, button, mouse
	15.73	preferences, base, zip, data, awtui
3.5	-38.78	param, reload, constraints
	69.34	view, collector, context, left, cancel, values, selector, views, icons, display
3.6	-1.20	
	8.72	
3.7	-8.25	
	2.79	
3.8	-13.30	deprecated
	23.40	printer, boo, lines
4.0	-349.34	constraints, grid, bag, set, label, panel, path, icon, model, button
	350.47	description, code, nbsp, org, annotation, notifier, method, request, runner, br
4.1	-1.43	
	61.90	link, param, check
4.2	-288.53	nbsp, br
	20.03	link, builder, pre, li
4.3.1	-8.91	
	53.36	array, actuals, expecteds, multi, dimensional, arrays, values, javadoc
4.4	-34.32	introspector, code, todo, multi, javadoc, dimensional, array, runner, test, fix
	151.98	matcher, theories, experimental, hamcrest, matchers, theory, potential, item, supplier, parameter
4.5	-30.11	theory, theories, date, result, static, validator, pointer, string, assert, experimental
	124.28	statement, model, builder, assignment, block, errors, unassigned, evaluate, describable, statements

Table 2. Evolution of JUnit: for each release we list the removed and the added words, large $2log\lambda$ values indicate more significant changes.

4. Related work

The present work is related to Jonathan Feinberg's comparison of inaugural addresses [7]. Feinberg analysed the inaugural address of Mr. President Barack Obama and his predecessors in office. For each inaugural addresses he provides a pair of WORDLE[3] word clouds. One cloud consists of words that are specific to the address, and the other cloud consists of words that are missing in the address. Font size is used to represent frequency of a word and saturation to represent the log-likelihood ratio. The color blue is used in the left cloud to represent likely terms, and red is used in the right cloud to represent unlikely terms.

Anslow *et. al.* [1] visualized the evolution of words in class names in Java version 1.1 and Java version 6.0. They illustrated the history in a combined word cloud that contains terms from both versions. Each word is printed twice, font size represents word frequency and color the corpus. As such they compared word counts, which assumes normal distribution and is thus not as sound as using log-likelihood ratios.

Kawaguchi *et. al.* [10] presented MUDABLUE, a tool that provides labels for projects. They used Sourceforge as normative corpus and applied Latent Semantic Indexing (LSI) at the level of projects. They categorized projects and described the categories with the most similar terms. However, the probabilistic model of LSI does not match observed data: LSI assumes that words and documents form a joint Gaussian model, while a Poisson distribution has been observed [9].

Lexical information of source code has been further proven useful for various tasks in software engineering (*e.g.* [2, 16, 17]). Many of these approaches apply Latent Semantic Indexing and inverse-document frequency weighting, which are well-accpeted techniques in Information Retrieval but are according to Dunning "only justified on very sketchy grounds [6]."

Baldi *et. al.* [5] present a theory of aspects (the programming language feature) as latent topics. They apply Latent Dirichlet Analysis (LDA) to detect topic

2 `http://www.junit.org`

3 `http://www.wordle.net`

distributions that are possible candidates for aspect-oriented programming. They present the retrieved topics as a list of the 5 most likely words. The model of LDA assumes that each topic is associated with a multinomial distribution over words, and each document is associated with a multinomial distribution over topics; their approach is thus sound.

5. Conclusion

We presented log-likelihood ratio as a technique to label software components. Using log-likelihood ratios allows comparisons to be made between the significance of occurrences of both common and rare terms. We presented how to use the log-likelihood ratio of word frequencies to retrieve labels, and how to compare a component against a normative corpus. In addition, we proposed to use log-likelihood ratios to characterize the evolution of a software component. We presented the results of two example applications, one using the Java API as case study, the other using 14 releases of JUnit as case study. By comparing subsequent revisions of JUnit to each other we were able to characterize substantial changes in the history of JUnit.

Acknowledgments: We thank Oscar Nierstrasz for his feedback on this paper. We thank Dominique Matter for his help with the parameters of log-likelihood ratios. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

References

[1] C. Anslow, J. Noble, S. Marshall, and E. Tempero. Visualizing the word structure of java class names. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 777–778, New York, NY, USA, 2008. ACM.

[2] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.

[3] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of 1st Intl. Workshop on Search-driven Development, Users, Interfaces, Tools, and Evaluation (SUITE'09)*, page Too appear, 2009.

[4] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 543–562, New York, NY, USA, 2008. ACM.

[5] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 543–562, New York, NY, USA, 2008. ACM.

[6] T. Dunning. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1):61–74, March 1993.

[7] J. Feinberg. Comparison of inaugural addresses. Website (retrieved March 4, 2009), Feb. 2009. Available at: `http://www.research.ibm.com/visual/inaugurals`.

[8] E. Hoest and B. Ostvold. The java programmer's phrase book. In *Proceedings of 1st Int. Conf. on Software Language Engineering*, pages 1–10, 2008.

[9] T. Hofmann. Probabilistic latent semantic analysis. In *In Proc. of Uncertainty in Artificial Intelligence, UAI99*, pages 289–296, 1999.

[10] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 184–193, 2004.

[11] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, Mar. 2007.

[12] A. Kuhn, P. Loretan, and O. Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 209–218, Los Alamitos CA, Oct. 2008. IEEE Computer Society Press.

[13] E. Linstead, L. Huges, C. Lopes, and P. Baldi. Exploring java software vocabulary: A search and mining perspective. In *Proceedings of 1st Intl. Workshop on Search-driven Development, Users, Interfaces, Tools, and Evaluation (SUITE'09)*, page Too appear, 2009.

[14] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 125–135, May 2003.

[15] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.

[16] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[17] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, Feb. 2009.

[18] P. Rayson and R. Garside. Comparing corpora using frequency profiling. In *Proceedings of the Workshop on Comparing Corpora*, pages 1–6, Oct. 2000.