LASCAD: Language-Agnostic Software Categorization and Similar Application Detection

Doaa Altarawy[a,*], Hossameldin Shahin[a], Ayat Mohammed[b], Na Meng[a,*]

[a] *Computer Science Department, Virginia Tech, Blacksburg, VA.*
[b] *Texas Advanced Computing Center, Austin, TX.*

Abstract

Categorizing software and detecting similar programs are useful for various purposes including expertise sharing, program comprehension, and rapid prototyping. However, existing categorization and similar software detection tools are not sufficient. Some tools only handle applications written in certain languages or belonging to specific domains like Java or Android. Other tools require significant configuration effort due to their sensitivity to parameter settings, and may produce excessively large numbers of categories. In this paper, we present a more usable and reliable approach of Language-Agnostic Software Categorization and similar Application Detection (LASCAD). Our approach applies Latent Dirichlet Allocation (LDA) and hierarchical clustering to programs' source code in order to reveal which applications implement similar functionalities. LASCAD is easier to use in cases when no domain-specific tool is available or when users want to find similar software across programming languages.

To evaluate LASCAD's capability of categorizing software, we used three labeled data sets: two sets from prior work and one larger set that we created with 103 applications implemented in 19 different languages. By comparing LASCAD with prior approaches on these data sets, we found LASCAD to be more usable and outperform existing tools. To evaluate LASCAD's capability of similar application detection, we reused our 103-application data set and a newly created unlabeled data set of 5,220 applications. The relevance scores of the Top-1 retrieved applications within these two data sets were, separately, 70% and 71%. Overall, LASCAD effectively categorizes and detects similar programs across languages.

Keywords: Software categorization, similar applications detection, topic modeling, LDA, source code analysis.

1. Introduction

As more projects are open sourced to facilitate communication and collaboration, effectively categorizing and detecting similar software becomes crucially important to assist skill learning, expertise sharing, program comprehension, and rapid prototyping Kontogiannis (1993); Michail and Notkin (1999); Liu et al. (2006); Sager et al. (2006); Schuler et al. (2007); McMillan et al. (2012b). Specifically, there are two major scenarios in which automatic software categorization and similar software search are useful:

- Cross-platform software migration. When software requirements and execution environments change developers may migrate their applications to new software or hardware platforms (e.g., from Windows OS to Linux, or from Android phones to iPhones). As a result, the software, components, or libraries that work in the original environment may not work in the new context. Developers have to search for or build alternative software solutions to replace the not-working software. For example, if developers would like to build an iPhone version for their Android app, they may want to find iOS apps similar to their Android app, and learn (1) how other similar apps are implemented for iOS, and (2) what libraries or technologies they can use that provide similar functionalities as their original code.

- Software upgrading. Developers sometimes reimplement software with a different programming language to support more features or to improve

*Corresponding author
[1]daltarawy@vt.edu (D. Altarawy)
[2]hshahin@vt.edu (H. Shahin)
[3]maaayat@vt.edu (A. Mohammed)
[4]nm8247@cs.vt.edu (N. Meng)

software quality. For instance, with C, MLBibTex reimplements BibTex (originally implemented in Pascal) to support the extra multilingual features Hufflen (2004). When such reimplementation information is not well documented, automatic software categorization and similar application detection can assist users to discover a useful reimplementation software, and to further decide whether to upgrade software and benefit from the reimplementation.

As of 2017, GitHub has 57 million repositories. Despite the availability of millions of open source projects, the GitHub showcases (now known as Collections) has less than a 1000 manually labeled applications for the users to explore. Manually classifying and detecting similar software is time-consuming and infeasible for large source code search engines. Added to the challenge, not all projects have documentation or a detailed readme file. Thus, more automatic categorization and similar software detection approaches are needed to utilize the wide availability of open source projects and to improve software showcases and functionality-based search engines.

Existing automatic categorization and similar software detection approaches Linares-Vásquez et al. (2016); McMillan et al. (2012a); Bajracharya et al. (2010); Tian et al. (2009); Kawaguchi et al. (2006); Michail and Notkin (1999) are limited for a variety of reasons. CodeWeb identifies similar classes, functions, and relations based on name matching. It is sensitive to the naming style of different software Michail and Notkin (1999). SSI Bajracharya et al. (2010), CLAN McMillan et al. (2012a) and CLANdroid Linares-Vásquez et al. (2016) detect similar Java or Android applications based on language-specific and/or domain-specific features, such as APIs, permission configurations, and sensor usage. Nevertheless, these approaches are limited to the domains for which they are designed. They are not helpful to detect similar software across languages. MUDABlue Kawaguchi et al. (2006) and LACT Tian et al. (2009) categorize software by extracting words from source code, and by applying Information Retrieval (IR) techniques, such as Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA), to cluster programs containing similar or relevant words. However, the effectiveness of both tools is sensitive to parameter settings, which are data-dependent and thus difficult to tune. In addition, MUDABlue and LACT can produce unbounded numbers of categories and cannot be used to produce a desired number of classes. In addition, some cases can require generating a specific (or at least bounded) number of categories such as for the purpose of visualization (limited space) or software browsing catalogs[5].

In this paper, we present LASCAD—a more usable and reliable approach for language-agnostic software categorization and similar application detection using only the source code. Although developers may build similar software differently (e.g., using various languages and following different coding styles), it is believed that identifiers used in code and words mentioned in comments are defined or used in meaningful ways that can indicate similar program semantics Tian et al. (2009); Kawaguchi et al. (2006). Therefore, we rely on *terms* (i.e., identifiers and words) used in source code to classify and detect similar software.

Specifically, we designed and implemented LASCAD that combines LDA with hierarchical clustering to categorize and detect similar software. Although LDA is the most widely used topic modeling method in software engineering research Chen et al. (2015), its parameter for specifying the number of latent topics has been notoriously difficult to tune Binkley et al. (2014); Grant et al. (2013); Panichella et al. (2013); Chen et al. (2015). Different from prior work based on LDA, LASCAD leverages hierarchical clustering to eliminate the need for tuning this specific parameter and to reduce developers' manual effort of parameter tuning.

LASCAD contains three phases. Given a set of open source applications, LASCAD first extracts terms from the source code of each software, and preprocesses terms by removing English stop words and programming language keywords, splitting identifiers, and removing most and least frequent terms. In Phase II, LASCAD uses LDA to identify latent topics of similar or relevant terms in each application. It leverages hierarchical clustering to recursively merge similar topics until getting a desired number of categories. By associating software with the categories, LASCAD establishes an application database with applications categorized based on the latent topics in their terms. In Phase III, to detect applications similar to a query application, LASCAD extracts latent topics from the query application, and then searches its database for programs with similar topic distribution using Jensen-Shannon Divergence similarity Lin (1991).

LASCAD's implementation is available at `https://github.com/doaa-altarawy/LASCAD`.

We used three data sets to evaluate LASCAD's capability of categorizing software. The first is MUDABlue's

[5]https://github.com/showcases

category-labeled data of 41 C programs; the second is LACT's labeled data set of 43 programs in 6 languages; and the third is our newly created labeled data of 103 programs in 19 different languages. We experimented with these data sets to compare LASCAD with two prior tools: MUDABlue and LACT, and found that LASCAD outperformed other tools in three aspects. First, LASCAD obtained higher F-scores (i.e., the harmonic mean of precision and recall) and produced better numbers of categories. Second, LASCAD is easier to use without requiring developers to tune the LDA parameter—number of topics. Third, LASCAD produces a bounded number of categories while allowing users to directly control its number. On the other hand, prior tools produced an undesirably large number of categories with no way to specify it). Although previous methods attempt to automatically determine the number of categories, they produce many non-meaningful categories Kawaguchi et al. (2006) which does not actually make the number of categories successfully automated.

To evaluate LASCAD's similar software search capability, we reused the third data set mentioned above and created another data set of 5,220 unlabeled open source applications implemented in 17 different languages. For the unlabeled data set, we randomly chose 38 software applications as queries and used LASCAD to search for similar applications for each of them within the whole dataset of 5,220 applications. Then, we manually inspected the top retrieved results for each query to check their relevance. For the other data set with ground truth labels, we used all of the 103 applications as queries. After inspecting the top retrieved applications of LASCAD, we found that 71-70% of Top-1 and 64% of Top-5 results were relevant to queries. In addition, the correctly retrieved similar applications are cross languages.

Finally, we conducted two case studies with the applications that LASCAD did not correctly classify or retrieve as similar software. Our case studies can shed light and help in future design and evaluation of automatic approaches for software classification and similar application detection.

In summary, this paper makes the following contributions.

- We developed LASCAD, a more usable and reliable language-agnostic approach to categorize software and to detect applications with similar functionalities. Only source code is used and no other language- or domain-specific information is required. Our tool is particularly useful when cross language is needed or when no domain-specific tool for software classification and detection (such as Java or Android tools) exists.

- We are the first to design and implement an algorithm, combining LDA with hierarchical clustering, for software categorization that eliminates the need to tune number of latent topics—a non-intuitive and well-known hard-to-tune parameter in LDA.

- With three different data sets, we conducted a comprehensive evaluation of three automatic software categorization approaches. Our evaluation showed that LASCAD outperformed prior tools, and worked stably well even when varying the number of categories.

- Unlike previous tools, LASCAD gives the user direct control over number of desired categories and avoids the large and non-meaningful number of categories produced by previous categorization tools. In future work, we plan to make this parameter optional by investigating appropriate machine learning methods that can correctly learn number of categories.

- We created two data sets that can be used as benchmarks for source code analysis. The first data set has manually labeled categories and can be used in software categorization. It has 103 projects belonging to 6 categories and implemented in 19 languages. The second data set contains 5,220 unlabeled projects written in 17 languages, which can be used as a pool for finding similar applications across languages.

- We conducted two case studies with the applications that LASCAD did not correctly classify or retrieve as similar software. Our case studies can shed light on future research directions to design and evaluate automatic approaches for software classification and similar application detection.

In the following part of this paper, we will first introduce background knowledge in Section 2, including LDA, hierarchical clustering, and metrics to measure categorization effectiveness. In Section 3, we will discuss our approach in detail. Section 4 will expound on all our experiments to evaluate LASCAD's capabilities to categorize software and detect similar software. Section 5 presents related work. Section 6 explains threats to validity, and Section 7 concludes the paper.

2. Background

This section first introduces the two techniques used in our approach: LDA and hierarchical clustering. It then defines four metrics to evaluate LASCAD's categorization effectiveness. Finally, it defines two notations frequently used in the paper.

2.1. LDA Topic Modeling

Topic modeling Wallach (2006) is a natural language processing method to discover the abstract "topics" in a collection of documents. Each topic is a collection of relevant words. There are various topic modeling methods, such as LDA Blei et al. (2003) and LSI Deerwester et al. (1990). Among various topic modeling methods, we chose to use LDA because it has been widely used, and has shown more advantages than other methods Chen et al. (2015).

LDA is a generative statistical model that identifies a set of latent topics in a collection of documents. In LDA, each topic has possibilities of generating various words, and a word can be generated from multiple topics with different probabilities. This gives LDA the flexibility of considering the same word in different contexts. Moreover, a document can be represented as a mixture of several topics Blei and Lafferty (2009).

LDA takes two inputs: the *number of latent topics* and a *document-term* matrix D. For a collection of documents $\{d_1, d_2, \ldots, d_n\}$, we can create the matrix D by extracting terms from documents, and by assigning each cell D_{ij} with the number of occurrence of term j in document d_i. LDA outputs another two matrices: a *document-topic* matrix to describe the likelihood of each document belonging to each topic, and a *topic-word* matrix to describe the possibilities of each topic generating various words.

2.2. Hierarchical Clustering

Clustering or cluster analysis Everitt et al. (2009) is the task of grouping a set of objects so that similar objects are put in the same group. Hierarchical clustering Zaki and Meira Jr (2014) is a cluster analysis method which seeks to build a hierarchy of clusters. To conduct hierarchical clustering, users often specify one parameter: *number of clusters*, which we call cat_{num}.

There are mainly two approaches to perform hierarchical clustering: *agglomerative* and *divisive*. Given the objects to cluster, the *agglomerative* (bottom-up) approach initiates an independent cluster for each object, compares clusters pair-by-pair, and merges the most similar ones into larger clusters. In comparison, the *divisive* (top-down) approach starts with the largest cluster and splits the cluster into smaller ones recursively. In our approach, we take the agglomerative approach.

Specifically, the agglomerative algorithm works as follows: Given N objects to cluster, it initializes a cluster for each object, getting N clusters: $\{C_1, C_2, \ldots, C_N\}$. Then it creates an $N \times N$ distance matrix by computing the distance between every two clusters. The more similar two objects are to each other, the smaller distance they have. Next, in each round of cluster merging, the algorithm looks for any cluster pair with the minimum distance, such as (C_i, C_j) where $i, j \in [1, N]$. Then it merges the two clusters into a bigger one C', removes the original two clusters, and updates the distance matrix accordingly. This merging process continues until the desired number of clusters are acquired.

2.3. Categorization Effectiveness Metrics

We use four metrics to measure categorization effectiveness: precision, recall, F-score, and logDiff.

2.3.1. Precision

As defined in prior work Kawaguchi et al. (2006), given a classification approach A (such as LASCAD, MUDABlue, or LACT), *precision* describes how precise A's categorization is compared with an *ideal* categorization. Formally,

$$precision = \frac{\sum_{s \in S} precision_{soft}(s)}{|S|} \tag{1}$$

$$precision_{soft}(s) = \frac{|C_A(s) \cap C_{Ideal}(s)|}{|C_A(s)|} \tag{2}$$

S represents the set of applications under categorization, while s represents an arbitrary application in the set. In Formula (1), *precision* is the mean of $precision_{soft}$ among all applications, where $precision_{soft}$ is called *soft precision*. Given an application s, soft precision compares ideal category labels $C_{Ideal}(s)$ with A's category labels $C_A(s)$, and decides what percentage of assigned labels by A are correct.

2.3.2. Recall

As defined in prior work Kawaguchi et al. (2006), *recall* describes what percentage of ideal category labels are correctly identified by A. Formally,

$$recall = \frac{\sum_{s \in S} recall_{soft}(s)}{|S|} \tag{3}$$

$$recall_{soft}(s) = \frac{|C_A(s) \cap C_{Ideal}(s)|}{|C_{Ideal}(s)|} \tag{4}$$

In Formula (3), *recall* is the mean of $recall_{soft}$ among all applications, where $recall_{soft}$ means *soft recall*. Given an application s, soft recall compares the set of ideal category labels $C_{Ideal}(s)$ with A's category labels $C_A(s)$, and decides what percentage of ideal labels are identified by A.

2.3.3. F-score

It computes the harmonic mean of precision and recall to weight them evenly. Mathematically,

$$\text{F-score} = \frac{2 * precision * recall}{precision + recall} \tag{5}$$

Suppose given an application s_1, $C_{Ideal}(s_1) = \{L_1, L_2\}$, and $C_A(s_1) = \{L_1\}$. Then $precision_{soft} = \frac{1}{1} = 100\%$, because A did not wrongly assign any label to s_1. Meanwhile, $recall_{soft}(s_1) = \frac{1}{2} = 50\%$, because A only identified L_1, but missed the other correct label L_2. Overall, F-score $= \frac{2*100\%*50\%}{100\%+50\%} = 67\%$.

2.3.4. Relative difference of category number (relDiff)

This metric is defined to measure the difference between the *identified* number of categories and the *ideal* number of categories. If the number of identified categories is too small compared to the ideal number, many dissimilar applications are wrongly clustered together. On the other hand, if the number of identified categories is much larger than the ideal number, many similar applications are scattered in different categories and become hard to interpret. Formally, given a set of applications,

$$\text{relDiff} = \frac{|\text{\# of identified categories - \# of ideal categories}|}{\text{\# of ideal categories}}. \tag{6}$$

RelDiff's range is $[0, +\infty)$. When relDiff=0, the number of identified categories is equal to the ideal number. The closer relDiff is to 0, the better number of identified categories we obtain.

2.4. Notations

We use the following notations to facilitate explanation:

- *t_num* represents LDA's parameter: the *number of latent topics*. By default, we set it to 50 in Lascad.
- *cat_num* is Lascad's parameter: the *number of desired software categories*. It is set to 20 in our evaluation.

3. Approach

There are three phases in Lascad. As shown in Fig. 1, given a set of software applications, Lascad first preprocesses the data to prepare for the categorization and similar application detection (Section 3.1) which is a common Information Retrieval step. In Phase II, given *cat_num*, Lascad applies LDA and hierarchical clustering to classify software into a desired number of categories (Section 3.2). In Phase III, based on the LDA results, Lascad retrieves software similar to a given query application (Section 3.3).

3.1. Phase I: Source Code Preprocessing

Given a collection of documents, source code preprocessing takes three steps to extract and refine terms, and finally, outputs a *document-term* matrix.

Extracting terms. For each software application, Lascad identifies source code files to extract identifiers in code and words in comments. These extracted terms compose the initial corpus generated from applications. Documentations and HTML files are excluded by Lascad, because they do not always exist in every application, and may bias topic modeling if they contain a lot of natural language descriptions.

Refining terms based on language features. We refine the corpus by removing language-specific terms and by splitting synthesized terms. We remove English-language stop words like "in", "are", "at", "the", etc., because they are unimportant. We also remove programming language-specific keywords, such as "class", "for", "if", and "while", because keywords vary with languages. Specifically, we created a list of keywords for the most common programming languages and removed those keywords from the corpus. Note that it is not necessary to remove keywords for each and every newly added language because most frequent keywords overlap across several languages.

Developers define identifiers differently. To mitigate the influence of coding styles on Lascad's effectiveness, we split identifier names in camel case and snake case into simpler terms. For instance, a camel case identifier "methodName" is split to "method" and "name", while a snake case identifier "method_name" is split to "method" and "name". Even though these two identifiers are different, their normalized representations are the same.

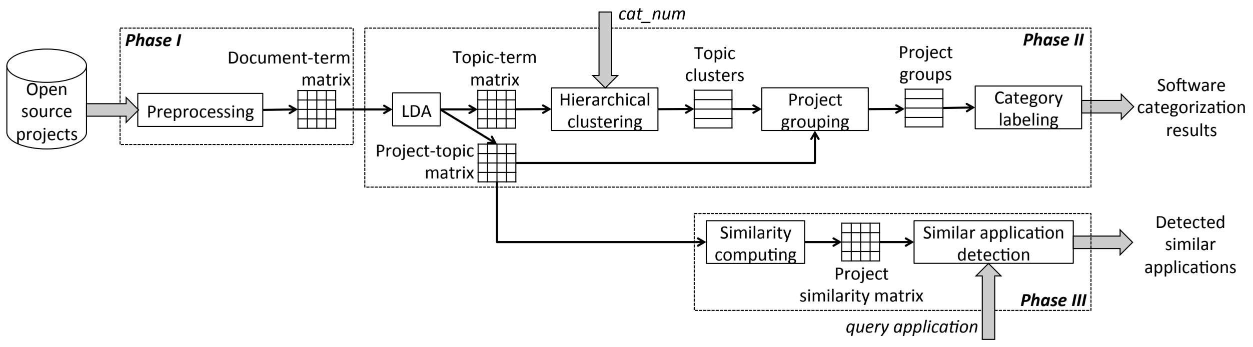

Figure 1: LASCAD overview

Removing overly common and overly rare terms. As any LDA-based approach does, LASCAD removes the most frequent and the least frequent terms from the corpus, because these terms may not reflect the program's real semantics, but can confuse LDA. Formally, suppose among m documents, a term t occurs in k documents. Then the document frequency of term t is $df = \frac{k}{m}$, where df describes how frequently a term occurs in a collection of documents. Similar to prior work Chen et al. (2015), LASCAD removes any term whose df is either above 0.8 or below 0.2.

3.2. Phase II: Software Categorization

To categorize software based on the corpus extracted from source code, we take four steps. Given the desired number of categories *cat_num*, we first use LDA to extract topics from the corpus. Then we perform hierarchical clustering to group topics until getting *cat_num* groups. Next, we assign projects to clusters based on the *project-topic* matrix and the generated clustering of topics. Finally, we assign a category label to each group of projects.

Step 1. After taking in the document-term matrix output from Phase I, LDA discovers latent topics, and produces two matrices: the topic-term (TT) matrix and the project-topic (PT) matrix. By default, we configured LDA to identify 50 latent topics in the given corpus, because our experiment in Section 4.2 showed that LASCAD was not very sensitive to the topic number parameter, and we could make the parameter transparent to users by setting a default value.

Step 2. Given a TT matrix created by Step 1, hierarchical clustering groups similar topics recursively until getting *cat_num* clusters, where *cat_num* has the default value 20. Specifically, the TT matrix represents each topic with a vector $L = [l_1, l_2, \ldots, l_m]$, where m is the number of extracted terms, and l_i ($i \in [1, m]$) represents the likelihood of term t_i belonging to the topic. To cluster topics, we initially consider every topic as an independent cluster, and then compare every two topics for the cosine similarity as

$$Cos\_Sim_{ij} = \frac{L_i \cdot Lj}{||L_i||\ ||L_j||} = \frac{\sum_{k=1}^{m} l_{ik} l_{jk}}{\sqrt{\sum_{k=1}^{m} l_{ik}^2}\sqrt{\sum_{k=1}^{m} l_{jk}^2}}. \quad (7)$$

Once we identify the most similar two topics or clusters, we merge them into a larger cluster, remove the original two clusters, and calculate the centroid vector for the new cluster using

$$L_{cen} = [\frac{l_{i1} + l_{j1}}{2}, \frac{l_{i2} + l_{j2}}{2}, \ldots, \frac{l_{im} + l_{jm}}{2}]. \quad (8)$$

For the newly created cluster, we then calculate its similarity with other clusters and find the next two closest clusters to group together. As visualized in Fig. 2, suppose given 6 topics, we set $cat\_num = 2$. Clusters are recursively merged in a bottom-up way. In each round, two clusters with the maximum similarity or minimum distance (i.e. $1 - Cos\_Sim$) are chosen and merged until we get 2 topic clusters: (Topic 0, 3, 4) and (Topic 1, 2, 5). To facilitate later explanation, we formally represent the created topic clusters as $Clu = \{cls_1, cls_2, \ldots, cls_{cat\_num}\}$.

Step 3. We group projects based on the project-topic (PT) matrix produced by Step 1, and the topic clusters Clu from Step 2. Intuitively, if two projects belong to the topics within the same cluster, the projects are put into the same software group. In this way, we get *cat_num* software classes, formally represented as $Cls = \{cls_1, cls_2, \ldots, cls_{cat\_num}\}$.

With more detail, the PT matrix represents each project as a vector $S = [s_1, s_2, \ldots, s_{t\_num}]$, where s_i

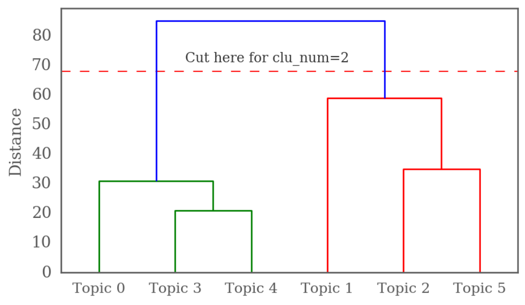

Figure 2: Clustering similar topics in a bottom-up way

($i \in [1, t\_num]$) shows the likelihood of the project belonging to the i^{th} topic. A project may belong to multiple topics, while a topic belongs to exactly one cluster. Therefore, LASCAD computes the project-cluster relevance matrix M as follows:

$$M_{ij} = \sum_{k=1}^{t\_num} s_{ik} b_{kj}, \text{ where}$$
$$b_{kj} = \begin{cases} 0, \text{ if } k^{th} \text{ topic does not belong to } cls_j, \text{ or} \\ 1, \text{ if } k^{th} \text{ topic belongs to } cls_j \end{cases} \tag{9}$$

Intuitively, to compute the relevance of i^{th} project to j^{th} cluster, we identify all topics inside the cluster, and sum up the project's likelihoods for those topics. Since the sum is not guaranteed to be within [0, 1], we further normalize the values for each project. If the normalized relevance value between the i^{th} project and the j^{th} cluster is above a relevance threshold r_th, it is classified into the corresponding j^{th} software group cls_j; otherwise, it is not. If a project has multiple relevance values above r_th, it means that the project belongs to multiple groups simultaneously. By default, we set $r\_th = 0.1$ to identify as many categories as possible for each application.

Step 4. With software grouped based on topic clusters, we created a category label for each group to complete software categorization. In prior work Kawaguchi et al. (2006); Tian et al. (2009), researchers read all projects in each group and then assigned category labels manually. We can take the same approach. However, as shown in Section 4.2, since we used software applications with *known* category information to evaluate LASCAD's categorization effectiveness, we managed to leverage the applications' labels to automatically name clusters. In particular, if a group contains three applications labeled as "Text Editor", and one application labeled as "Web Framework", LASCAD labels the group with the majority category "Text Editor".

In future, we would like to investigate two alternative approaches to automatically label groups if the true labels are unknown. First, in each software group cls_j, we will identify the projects which are most relevant to the topic cluster cls_j, and parse out the most frequent terms from the projects to label the group. Second, we will use the most frequent terms in each topic of cluster cls_j to name the software group cls_j.

3.3. Phase III: Detecting Similar Applications

In the application pool, when users select an application to search for similar applications, LASCAD reuses the project-topic PT matrix computed in Phase II to calculate the similarity between projects. Specifically, in the PT matrix, each project corresponds to a vector $S = [s_1, s_2, \ldots, s_{t\_num}]$, which is considered a probability distribution of generating the project from these topics. LASCAD computes the similarity between every two projects based on Jensen-Shannon Divergence Lin (1991), a metric used to measure the similarity between two probability distributions. Finally, when users choose an application in the pool, we query the application's precomputed similarity scores with all other applications, rank those applications accordingly, and return the top results in descending order by similarity scores.

3.4. Implementation

Our tool is implemented in Python. We leveraged the NLTK Bird et al. (2009) natural language processing library, applied Scikit-learn Buitinck et al. (2013) for LDA modeling and hierarchical clustering implementations, and used Pandas McKinney (2011) and Scipy Jones et al. (2001) to process and analyze data.

We ran LASCAD for the 103 projects on a machine with an Intel Core i7 processor. The preprocessing for all open source projects took around 2 hours, the LDA algorithm extracted topics in 34 minutes, while the hierarchical clustering to create software categories took only 3 seconds.

4. Evaluation

In this section, we first present the four data sets used in our experiments (Section 4.1). Then we discuss our evaluations of LASCAD for software categorization (Section 4.2, 4.3, and 4.4) and similar application detection (Section 3.3). Finally, we discuss our case studies to understand why LASCAD failed to categorize some applications or wrongly suggested similar applications (Section 4.6).

Table 1: The category-labeled data set of 103 projects

	Machine Learning (26)	Data Visualization (22)	Game Engine (20)	Web Framework (16)	Text Editor (12)	Web Game (7)
JavaScript (33)	2	16	-	4	5	6
C++ (15)	7	-	7	-	1	-
Python (14)	6	2	-	4	2	-
Java (8)	5	-	2	1	-	-
Ruby (5)	2	-	-	3	-	-
C# (4)	-	-	4	-	-	-
PHP (4)	-	-	-	4	-	-
CoffeeScript (3)	-	1	-	-	2	-
C (3)	-	-	2	-	1	-
HTML (3)	-	2	1	-	-	-
Objective-C (2)	1	1	-	-	-	-
TypeScript (2)	-	-	2	-	-	-
CSS (1)	-	-	-	-	-	1
Clojure (1)	-	-	-	-	1	-
R (1)	1	-	-	-	-	-
Go (1)	1	-	-	-	-	-
Scala (1)	1	-	-	-	-	-
D (1)	-	-	1	-	-	-
ActionScript (1)	-	-	1	-	-	-

4.1. Data Sets

There are four data sets used in our experiments: one labeled set borrowed from prior work MUDABlue Kawaguchi et al. (2006), one labeled set from prior work LACT Tian et al. (2009), one newly created labeled data set, and one created unlabeled data set. The first three data sets were used for software categorization evaluation, while the last two sets were used for similar software detection evaluation.

Although there is a publicly available data set with manual classification of the most popular 5,000 GitHub repositories Borges and Valente (2017), we chose not to use the data set in our evaluation. The reason is that their categorization is too broad while our labels are functionality based. The category labels like "Application software" and "Software tools" do not mention or indicate any software functionality and can contain diverse applications.

The MUDABlue labeled data set includes 41 C programs selected from SourceForge by the MUDABlue authors. These programs belong to 13 SourceForge categories provided by LACT Tian et al. (2009): xterm, Gnome, Conversion, Board Games, Artificial Intelligence, Database Engines, Turn Based Strategy, Text Editors, Software Development, Internet, Compilers, Interpreters, and Cross Compilers. Each application can belong to multiple categories.

To fairly compare with prior approaches MUDABlue and LACT, we need to execute these tools and LASCAD on the same data set. Since MUDABlue is not available (after contacting the authors) and the paper Kawaguchi et al. (2006) does not include enough technical detail for us to reimplement the tool, we decided to reuse MUDABlue's data set, and to compare our results against MUDABlue's results that are reported in their paper.

The LACT labeled data set includes 43 programs implemented in 6 languages. These programs belong to six categories: Game, Editor, Database, Terminal, E-mail, and Chat. Although LACT is not available (after contacting the authors), we were able to reimplement it according to their paper Tian et al. (2009), and experimented with LACT and LASCAD on the same data sets.

The New labeled data set was created by us to contain 103 open source projects in 19 different languages. To avoid any bias in favor of our approach when labeling applications, we did not label applications ourselves. Instead, we collected labeled data from GitHub Showcases GitHub (2016)—a website organizing popular repositories by categories. The GitHub developers manually labeled some projects with category information, and then grouped those projects based on the labels. For our experiment, we randomly selected the following six categories of software: Data Visualization, Game Engines, Machine Learning, Text Editor, Web Framework, and Web Games. We got 103 labeled applications in this way, and present them in Table 1. In the table, JavaScript is the mostly used language, while Machine Learning contains the most applications. Each application belongs to only one category.

New unlabeled data set was built by us to contain 5,220 projects implemented in 17 languages. To evaluate LASCAD's effectiveness of detecting similar applications, we need a large number of applications to prepare LASCAD's application pool. We want to ensure that for any query application, LASCAD can retrieve a sufficient number of relevant applications from the pool. To prepare the data set, we sampled GitHub projects using nine keyword queries: Web Framework, Text Editor, Compiler, Machine Learning, Chatting, Database, Game Engine, Mobile App, and Visualization. For each query, we leveraged the GitHub APIs GitHub (2018) to download the top 1,000 retrieved projects obtaining 9,000 projects in total as the initial raw data. Notice that not all these sampled projects implement the functionalities indicated by the keywords. We intentionally sampled projects in this way to create a noisy data set, which contain applications relevant or irrelevant to certain functionality focuses. To facilitate our evaluation, we further refined the raw data with three filters:

1. Each included programming language should correspond to at least 40 programs. We believe that if a language is not well represented by a reasonable number of applications in the pool, it is not quite useful to evaluate the language-agnostic search effectiveness. Therefore, if an included language covered fewer than 40 programs, we removed the programs from the raw data.
2. The storage size of each included project is at least 250 KB. Alternatively, we can also filter out small projects based on the lines of code (LOC) they contain.
3. Each included project has received at least 10 stars. A poorly maintained project may lack comments and have confusing identifiers. In GitHub, users star the projects they appreciate or keep track of GitHub (2017), making the number of stars a good indicator of project quality. Therefore, we set a threshold (i.e. 10) to the number of stars to filter out possibly low-quality projects.

After refining the raw data, we derive 5,220 projects which correspond to 17 languages.

4.2. Software Categorization Effectiveness

Fig. 3 includes six heat maps to show the categorization results of LASCAD on the 103-application labeled data set. A heat map graphically visualizes a matrix by representing each digital value with a color Wilkinson and Friendly (2009). The values are within [0, 1]. The greater a value is, the darker its corresponding cell is colored. In Fig. 3, every heat map corresponds to one known Showcase (or *ideal*) category. The rows in each heat map are the projects belonging to the ideal category, while the columns are the 20 identified categories. The more relevant the i^{th} project is to an identified category cls_j, the darker color cell (i, j) has.

Each heat map has 2-3 darker columns marked with **red circles**, indicating the mapping relationship between the identified categories and the ideal ones. For instance, in the heat map of Text Editor, categories 11 and 12 (C11 and C12) have more cells darkened, meaning that both identified categories characterize the common semantics of text editors. In comparison, the heat map of Web Framework has C11, C12, and C13 darkened. It means that some of these applications have text editor components (e.g., Derby Derby (2018)) as their C11 and C12 cells are dark, while C13 mainly captures the web framework characteristics.

For the 103 projects, LASCAD categorized software with 67% precision, 85% recall, 75% F-score, and 2.33 relDiff. Our case study in Section 4.6 will further discuss our investigation about why LASCAD could not always correctly categorize software.

Finding 1: LASCAD *categorized software with 67% precision, 85% recall, 75% F-score, and 2.33 relDiff. For each ideal category,* LASCAD *identified two or three categories, with each of which specially characterizing certain functionality semantics.*

4.3. Categorization Sensitivity to Parameter Settings

One well known challenge for LDA-based approaches is to tune the LDA parameter *t_num* Binkley et al. (2014); Grant et al. (2013). By default, we set *t_num* = 50 in LASCAD to eliminate users' effort to tune the hard-to-use parameter. LASCAD has one parameter for the user to choose which is the number of categories: *cat_num*. Number of categories *cat_num* is different from number of latent topics of LDA *t_num* in two aspects:

1. *cat_num* is a high level parameter corresponds to an estimated upper bound on the desired categories. In contrast, *t_num* is number of latent (hidden or abstract) topics in the data which is not easy to guess or estimate according to the literature.
2. *t_num* is known to be a difficult to tune parameter because the accuracy of the results is sensitive to its value. On the other hand, we show that LASCAD is not sensitive to *cat_num* thus easier to guess without compromising accuracy. For instance in Section 4.2, we set the guess for *cat_num* to 20 (as an upper bound) while the true number of categories is 6 and still got better performance than other tools.

To investigate LASCAD's sensitivity to these parameters, we conducted the following two experiments.

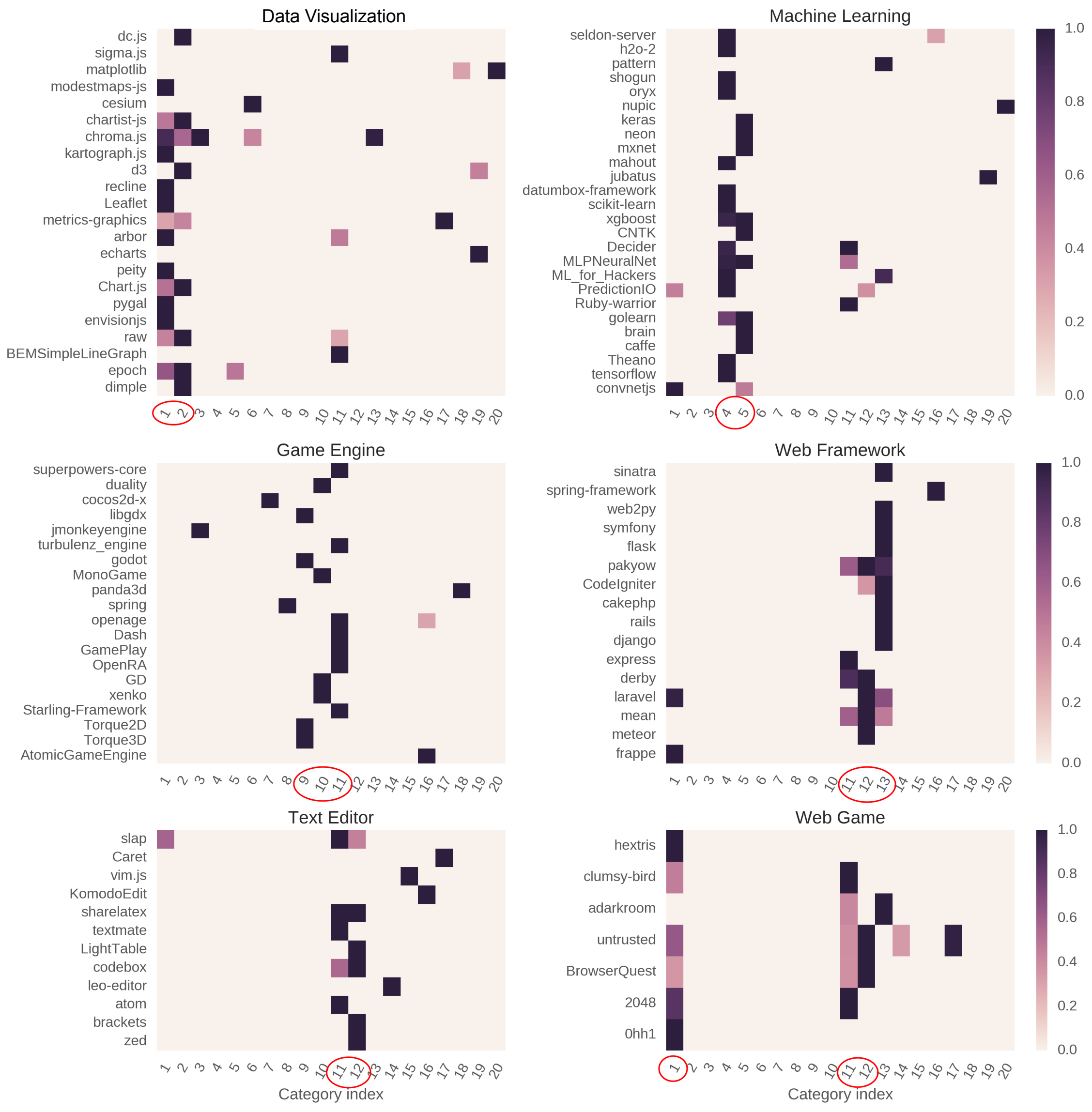

Figure 3: Heat maps of Lascad's categorization results

In the first experiment, as shown in Fig. 4, we fixed *cat_num* to 20, changed *t_num* from 20 to 100 with 5 increment, and checked how overall effectiveness varied. The relDiff remained unchanged as *cat_num* was fixed to 20, so we only compared F-scores. We found that the F-score did not vary greatly with *t_num*. The mean value was 71%, while the standard deviation was as low as 2.9%. This indicates that Lascad is not sensitive to *t_num* when we fixed *cat_num*. Therefore, it is reasonable to set a default *t_num* value and make this hard-to-configure parameter transparent to users. The insensitivity may be due to our usage of hierarchical clustering, which considerably reduces the impact of *t_num* on the categorization effectiveness.

For our second experiment, we fixed *t_num* to 50, and changed *cat_num* from 5 to 50, with 5 increment. We did not try any number greater than 50. As there are 103 applications, $cat\_num = 50$ indicates that applications may be distributed among so many categories that interpreting the meaning of each identified category is challenging. Therefore, for N applications to classify, we intentionally set $\lfloor \frac{N}{2} \rfloor$ as the upper bound of our exploration for *cat_num*. As shown in Figure 5, the F-score increases significantly with *cat_num* when $cat\_num \leq 15$. When $cat\_num > 15$, F-score becomes

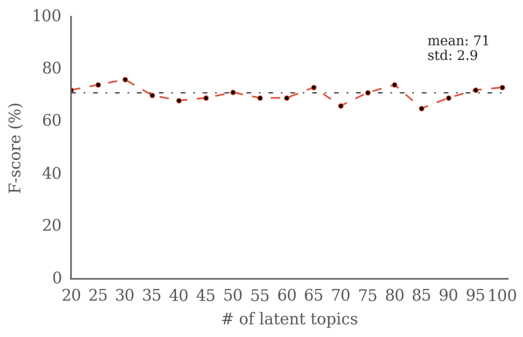

Figure 4: LASCAD's F-scores with *cat_num* = 20 and *t_num* varying from 20 to 100

stable. For generality, we chose *cat_num* = 20 as the default setting in all our following experiments, because the setting achieves a reasonable trade-off between F-score and relDiff in this experiment.

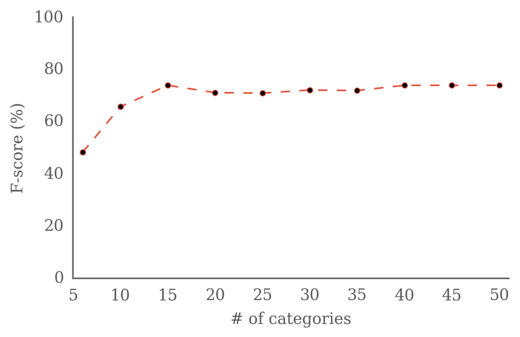

Figure 5: LASCAD's F-scores with *t_num* = 50 and *cat_num* varying from 6 to 50

Finding 2: *LASCAD's categorization capability is not sensitive to t_num, and only varies considerably when cat_num ≤ 15. We thus set t_num = 50 and cat_num = 20 by default for generality.*

4.4. Comparison With Prior Categorization Tools

We conducted several experiments to compare LASCAD with two prior approaches: MUDABlue Kawaguchi et al. (2006) and LACT Tian et al. (2009), because both tools leverage topic modeling to categorize software based on source code. Neither tool is available (we contacted the authors, but were unable to obtain the tools), so we tried to reimplement them. However, the MUDABlue paper lacks implementation details, so we could not reimplement the tool, and simply *reused MUDABlue's results reported in the paper*. Additionally, the LACT paper contains enough technical details. We *reimplemented the tool, used the best parameter setting mentioned in the paper, and executed LACT on the same benchmarks as* LASCAD. LACT requires users to specify *t_num*, which we set as 40 based on their paper.

Table 2 shows the comparison results on MUDABlue's data set. We bolded LASCAD's results. According to the table, LASCAD worked best by obtaining the highest F-score and the relDiff closest to 0. Notice that although MUDABlue's reported F-score is close to LASCAD's (72% vs. 74%), it is doubtful if the two approaches have comparable F-scores because MUDABlue's authors manually labeled clusters while LASCAD labels them automatically. Thus, the original MUDABlue evaluation was subject to human bias. Second, *MUDABlue classified 41 programs (of 6 actual categories) into 40 categories*, which is counterintuitive. According to the paper Kawaguchi et al. (2006), 14 out of the 40 categories identified by MUDABlue were difficult to interpret, meaning that MUDABlue produced meaningless categories. Third, all programs in their data set were implemented in one programming language. Therefore, they do not evaluate MUDABlue's capability of categorizing software across languages.

Table 2: Tool comparison based on MUDABlue's 41 C programs of 13 ideal categories

Tool	# of categories	Precision	Recall	F-score	RelDiff
MudaBlue	40	-	-	72%	5.67
LACT	23	76%	65%	70%	2.83
LASCAD	**20**	**83%**	**67%**	**74%**	**2.33**

Table 3: Tool comparison based on LACT's 43 programs of 6 ideal categories

Tool	# of categories	Precision	Recall	F-score	RelDiff
LACT	25	56%	72%	64%	3.17
LASCAD	**20**	**64%**	**72%**	**68%**	**2.33**

Table 4: Tool comparison based on our 103 applications of 6 ideal categories

Tool	# of categories	Precision	Recall	F-score	RelDiff
LACT	38	57%	91%	70%	5.33
LASCAD	**20**	**67%**	**85%**	**75%**	**2.33**

Table 3 shows the comparison results between LACT and LASCAD based on LACT's data set. We also bolded LASCAD's results. LASCAD outperformed LACT for both F-score and relDiff. There are three parameters to tune in LACT but only one parameter in LASCAD. Nevertheless, Lascad performed better with fewer parameters and less parameter tuning effort. Additionally, we also conducted a similar comparison experiment on the 103-application labeled data set, and observed similar results. As shown in Table 4, Lascad obtained both better F-score and better relDiff.

Table 5: Previous tool LACT's categorization results on the 103-application data set with *t_num* changing from 20 to 100. Pay attention to the great variability in the column **# of categories**.

t_num	# of categories	Precision	Recall	F-score	RelDiff
20	20	51%	86%	64%	2.33
30	27	54%	94%	68%	3.50
40	35	55%	93%	69%	4.83
50	38	57%	91%	70%	5.33
60	50	63%	88%	73%	7.33
70	47	60%	86%	71%	6.83
80	49	59%	89%	71%	7.17
90	52	58%	82%	68%	7.67
100	54	64%	85%	73%	8.00

Furthermore, we show that the previous tool LACT is sensitive to *t_num* (which has also been shown in their paper Tian et al. (2009)). Using the 103-application data set, we changed LACT's *t_num* from 20 to 100 with 10 increment, and checked how the overall effectiveness varied. As shown in Table 5, as *t_num* increased, the number of generated categories varied a lot. With *t_num* = 20, LACT identified 20 categories—same as the default value of *cat_num* in Lascad, but obtained a much lower F-score (64%). With *t_num* = 60 or 100, LACT achieved its highest F-score (73%), a slightly lower score than Lascad's 75%, but generated too many categories (50 and 54) for the data set of 6 ideal categories.

Table 6: Comparing Lascad and LACT's categorization results on the 103-application data set with approximately similar *cat_num*.

LACT		LASCAD	
# of categories	F-score	# of categories	F-score
20	64%	20	75%
27	68%	30	75%
35	69%	35	72%
38	70%	40	76%
47	71%	45	74%
50	73%	50	76%
Avg.	**68.67%**		**74.67%**

Finally, in Table 6, we compare the F-scores of Lascad and LACT when they produced the same number of categories. Since we cannot directly control LACT's *cat_num*, we choose the closest possible generated *cat_num* in LACT to compare it with LASCAD. According to the table, with 20, 35, and 50 categories produced, Lascad's F-scores are 75%, 72%, and 76%; however, LACT's F-scores are 64%, 69%, and 75%. Lascad's F-scores are always higher than LACT's. More importantly, LACT does not allow users to directly control the number of generated categories. Instead, users can only manipulate the number of topics (*t_num*) to indirectly influence the number of produced categories. In comparison, Lascad is better because it enables users from directly controlling the number of categories to identify.

Finding 3: Lascad *categorized software stably better than prior approaches on different data sets. It allows users to flexibly control the number of generated categories, without producing overwhelming numbers of categories as previous tools do.*

4.5. Evaluation of Similar Application Detection

To assess Lascad's similar software detection capability, we defined a metric *relevance* to measure how relevant retrieved applications are to a given query. With n query applications $\{q_1, q_2, \ldots, q_n\}$ provided, Lascad ranks all applications in the pool for each query based on how similar each application is to the query. The higher the score is, the higher rank an application gets. Using the Top m retrieved applications $\{a_1, a_2, \ldots, a_m\}$, we calculated the query relevance r_i as

$$r_i = \frac{\sum_{j=1}^{m} b_j}{m}, \text{ where } b_j = \begin{cases} 1, & \text{if } a_j \text{ is similar, or} \\ 0, & \text{if } a_j \text{ is not similar} \end{cases} \quad (10)$$

Correspondingly, the overall relevance for the n queries is

$$relevance = \frac{\sum_{i=1}^{n} r_i}{n} \quad (11)$$

We used the unlabeled data set of 5,220 projects to initialize Lascad's application pool, and randomly selected 38 applications from the pool as queries. Then we manually inspected the Top-1 and Top-5 retrieved applications for each query. We read the source code and relevant documents to decide whether each inspected application had the same major functionality as the query. In this way, we found that the relevance of the Top-1 and Top-5 applications was 71% and 64% respectively. Furthermore, those relevant applications belong to various programming languages demonstrating Lascad's capability of finding similar applications across languages using only source code.

We also used the 103-application labeled data set to conduct a similar experiment. We prepared Lascad's pool with these applications, and tried each of them as a query to search for similar applications. For the Top-1 and Top-5 ranked applications of each query, we compared their category labels with that of the query. If the labels were identical, the retrieved application is considered relevant. In this way, we calculated the relevance of Top-1 and Top-5 applications as 70% and 64% respectively.

Comparison Experiments.

We performed three alternative experiments to find similar software to a query application to compare with LASCAD using our 5,220 large data set. In the first experiment, we conducted a random search were the results of the query are chosen at random. In this case, the relevance of the Top-1 (as well as any Top-i) result was approximately %11.

The second experiment is a full text search using the title and the description of applications. We used the applications title as the query and searched the other projects title and description using full text search. The relevance of the Top-1 results was %8.3. Clearly, this approach is ineffective because the title of similar applications does not usually appear in the projects description.

In the third experiment, we used the projects readme files in an approach similar to RepoPal Zhang et al. (2017). That is, using topic modeling on readme files rather than on source code. We have not used RepoPal because, in addition to readme files, it uses Stars history which is specific to GitHub. First, we extracted the readme files of the 5,220 projects using GitHub API. There were 654 repositories without a readme file. Next, we processed the readme files into terms and ran LDA to generate the document-topic matrix. Finally, we applied similarity search on the LDA output of the readme files as described in section 3.3. The relevance of the Top-1 and Top-5 results were %23 and %19 respectively.

> **Finding 4:** LASCAD *effectively detected similar software on different data sets. The relevance of Top-1 and Top-5 retrieved applications was 70-71% and 64% respectively.*

4.6. Case Studies

To understand why LASCAD did not work well in some cases, we conducted a case study to manually examine the applications which were wrongly classified by LASCAD. We also performed another case study to manually check the applications that were wrongly retrieved as similar applications to a query.

Case Study I. We found three reasons to explain why LASCAD did not correctly classify all applications.

First, the ground truth categories were not exclusive to each other. For instance, some Web Game applications (e.g., *clumsy-bird* Leão (2017)) contained certain Data Visualization features (e.g., rendering scenes) or Game Engine features (e.g., game logic), so LASCAD assigned multiple category labels to those Web Games. Since *the ground truth data only had one category label for each application*, we strictly considered all extra labels as wrong ones, which can underestimate LASCAD's categorization precision.

Second, some ground truth labels provided by GitHub showcases were not complete. For instance, although the ground truth label of *ruby-warrior* Bates (2012) was "Machine Learning", the application's website described the program as "Game written in Ruby for learning Ruby and artificial intelligence". Therefore, the label "Game Engine" created by LASCAD also precisely summarized the software, even though the label does not match the ground truth. Since the ground truth labels are incomplete, we might underestimate LASCAD's categorization effectiveness.

Third, LASCAD *could not differentiate between some applications that shared latent topics but had divergent functionality focuses*. For instance, LASCAD wrongly put some Game Engine applications and Text Editor applications into the same category, because all these programs supported similar interactions with users. This observation indicates that in future, we also need a more fine-tuned approach for term extractions for better software categorization.

> **Finding 5:** *The generated category labels by* LASCAD *may be different from the oracle labels for three reasons. First, the oracle labels are incomplete. Second, the oracle categorization contains some incorrect labels. Third, few applications may share latent topics and features but actually implement different functionalities.*

Case Study II. We identified two reasons that can explain why LASCAD wrongly suggested applications for certain queries.

First, LASCAD *did not work well for query applications that contained few lines of code.* For instance, *map-chat* Cohen (2017) was a small location-based chat tool with only six program source files, and LASCAD was unable to effectively extract topics or suggest relevant applications for this query. As a result, it wrongly suggested other irrelevant applications such as Web Frameworks, Text Editors, and Mobile apps.

Second, LASCAD *could not detect the differences between applications that had similar topics but different implementation focuses.* For instance, *django* Django (2017) is a high-level Web Framework application that implements templates, authentication, database, servers, clients, and connections. With this application as a query, LASCAD wrongly suggested *ibid* Google (2018)—a multi-protocol general-purpose chat robot, because *ibid* also includes some implementation of authentication, database, templates, and connections. The extracted topics were similar, although the two applications actually implemented different program logic. Despite that this case is considered a false positive search result, the two applications are actually functionally similar but differ in their goals. In future, we can leverage program analysis to collect more program context information, to better model the relationship between topics for each program, and thus to improve similar software detection.

Finding 6: LASCAD *may incorrectly retrieve applications dissimilar to a query application for two reasons. First, the query application has a small codebase. Second,* LASCAD *does not differentiate between applications with similar functionalities but different implementation focuses.*

5. Related Work

This section describes related work on similar application detection, software categorization, code search, and LDA parameter settings.

5.1. Similar Application Detection

There are various similar software detection approaches Linares-Vásquez et al. (2016); McMillan et al. (2012a); Bajracharya et al. (2010); Tian et al. (2009); Kawaguchi et al. (2006); Michail and Notkin (1999). For instance, Google Play has a "Similar" feature to recommend applications similar to a given Android app. To decide similarity, it leverages metrics like human-labeled app category information, app name, app description, and target country/language. Although these metrics are helpful to scope similar applications, they are not effective to filter irrelevant applications, and always produce false alarms. When approaches are built on top of such imprecise application suggestion, the approach effectiveness also suffers Lu et al. (2015).

CodeWeb relies on name matching to identify similar classes, functions, and relationships in different libraries Michail and Notkin (1999). Since it does not check any implementation detail of program entities, it can mismatch significantly different classes with accidentally similar names. SSI Bajracharya et al. (2010) and CLAN McMillan et al. (2012a) rely on API usage to find similar applications such as Java APIs. The basic assumption is that similar applications may invoke the same library APIs similarly. However, such approaches are not designed to recommend applications across languages or work for other languages because of the dependence on specific libraries APIs. CLANDroid detects similar applications in a specific domain—Android Linares-Vásquez et al. (2016). It is not directly applicable to programs in other domains.

RepoPal detects similar GitHub repositories based on the similarity of projects' readme files, and repositories starred by the same users within the same period of time Zhang et al. (2017). Nevertheless, RepoPal can only detect similar applications implemented in the same language, only in GitHub, and with suitable readme files. In comparison, LASCAD automatically detects similar software across languages, from any repository, and using only source code. Although several tools are proposed for domain-specific similar application detection and are successful within their domain, they are not directly applicable to other or across domains. SimilarTech is an interesting approach for finding analogical application across languages that uses tags from Stack Overflow questions Chen et al. (2016b); Chen and Xing (2016a).

5.2. Software Categorization

Approaches were proposed to automatically categorize software McMillan et al. (2011); Kawaguchi et al. (2006); Tian et al. (2009). For instance, McMillan et al. extracted JDK API invocations as features to train a machine learning model for automatic software categorization McMillan et al. (2011). MUDABlue Kawaguchi et al. (2006) and LACT Tian et al. (2009) apply topic modeling methods to categorize software based on the textual relevance of terms extracted from source code. Although these categorization approaches can be extended to detect similar applications, such capability extension has not been fully investigated. McMillian et al. once leveraged MUDABlue to detect similar Java programs McMillan et al. (2012a), and found the search relevance was as low as 33%.

They did not explore the cross-language similar software detection effectiveness of categorization-based approaches.

Compared with MUDABlue and LACT, LASCAD leverages a novel approach to combine LDA with hierarchical clustering for software categorization and similar software detection. LASCAD is more usable by requiring less configuration effort while providing better categorization results. It allows users to manipulate the number of software categories to generate. More importantly, LASCAD does not produce overwhelming numbers of categories.

Both LASCAD and LACT starts with LDA but proceed differently afterwards. LACT is a simple idea but with two difficult to tune parameters. In addition, it produces unpredictable and uncontrollable number of categories. LACT starts by LDA without any guidance of how to choose number of latent topics (which is known to be difficult to tune). Next, LACT does one pass to merge topics whose similarity is above a threshold (yet another difficult to choose parameter). Thus, the results can have a quadratic number of categories. On the other hand, LASCAD is a simple but an effective approach. First we perform a properly processed LDA with a fixed number of latent topics. Next, LASCAD iteratively merges the most similar pairs of topics into new topics. This process continues until the desired number of categories is obtained. The results include the desired number of categorization with a high f-score at the same time. More importantly, there is no parameter tuning for the LDAs number of latent topics, nor data-dependent thresholds.

5.3. Code Search

There are various commercial and open source search engines built to retrieve code snippets or projects relevant to the keyword queries, such as Sourcegraph Sourcegraph (2017), Google Code Search Google (2015), and Sourcerer Baldi et al. (2008). GitHub and other repository hosting services also provide search engines for developers to find projects by topic. However, none of them is able to retrieve applications implementing the similar functionality as a query application based on source code.

5.4. LDA Parameter Settings

LDA is well known to be sensitive to the selected number of latent topics, and tuning this parameter is challenging Binkley et al. (2014); Grant et al. (2013); Panichella et al. (2013); Chen et al. (2015). For instance, Binkley et al. conducted an empirical study to understand how LDA parameter settings affect source code analysis results Binkley et al. (2014). They concluded that there is no universal best parameter setting because the appropriate setting depends on the problems being solved and the input corpus. Grant et al. proposed various heuristics to estimate an appropriate number of latent topics in source code analysis, such as using 300 or fewer latent topics when analyzing projects with $20,000$ or fewer lines of code Grant et al. (2013). However, such heuristics are not applicable to data sets of variable-sized projects. This motivated us to design and implement an approach to make this specific parameter transparent, and to eliminate the need for user configuration for the parameter.

6. Threats to Validity

To evaluate LASCAD's similar software detection capability on the unlabeled data set, and since there is no ground truth for applications relevance, we manually checked the similarity of the retrieved applications to the query applications using our best knowledge. We require both the query and the retrieved application to strictly implement the same main functionality in order for them to be considered relevant. To better decide applications relevance, a user study can be conducted in the future to recruit developers, ask them to separately compare applications, and then leverage their majority judgments to evaluate LASCAD.

Also in the evaluation of LASCAD for similar application detection, we randomly chose 38 applications as queries to search for similar software in the whole pool of 5220 unlabeled applications. For each query, we manually checked the Top-10 retrieved applications, inspecting in total $38 + 38 * 10 = 418$ applications. We were unable to further increase the number of queries due to the long manual effort needed for result verification. However, using the 103 already labeled applications, we were able to use the whole 103 projects as queries (achieving similar relevance as the former data set).

We reimplemented LACT for tool comparison because the original tool is not available even after contacting the authors. Our reimplementation may not exactly reproduce the original tool and our parameter configuration may not be optimal. However, we tried our best to reimplement the tool following the descriptions in the paper and used the best parameter values reported by the paper. Furthermore, the results produced by our reimplementation match those reported in their paper on their data set.

In our approach, we eliminated the need for developers to manually configure the hard-to-tune parameter: number of latent topics of LDA, *t_num*, by using an algorithm that utilizes both LDA and hierarchical clustering, and we showed in Section 4.3 the LASCAD is not sensitive to that parameter. Thus, we set a default value to *t_num* in LASCAD to save users' configuration effort. Although such default value is not guaranteed to achieve optimal performance, perfectly configuring parameters has been a challenging problem for IR researchers. Even though our default setting is not optimal, it did not affect LASCAD's effectiveness to categorize and detect similar software.

We also mitigate the issue of unbounded number of categories (and also many non-meaningful categories) produced by previous tools by using an agglomerative clustering approach which requires guessing number of clusters. Although our experiments showed that LASCAD is not sensitive to this parameter, and the user can safely choose an upper bound, it is useful in the future to investigate approaches that can automatically find the correct number of categories without producing non-meaningful clusters.

Within the three labeled data sets, only one dataset assigns multiple category labels per application as the ground true. For the showcases dataset, we avoided modifying the ground truth labels and left it as presented in GitHub to be objective, although an application may actually belong to multiple categories simultaneously. With such datasets containing partial category ground truth, we may underestimate the effectiveness of LASCAD. In the future, we plan to create a data set that contains applications with multiple known category labels. Chen et al. built an approach to mine for analogical libraries across programming languages Chen and Xing (2016b); Chen et al. (2016a). With Chen's approach, we may automatically detect software that can have multiple category labels, and efficiently build a better and larger data set.

7. Conclusion

This paper presents LASCAD, an approach to categorize software and detect similar applications. Our tool is easy to implement and to use, language-agnostic and is solely based on the software source code. LASCAD can be particularly useful when no domain-specific tool exists or when cross-language software categorization and detection capabilities are needed.

Furthermore, by combining LDA with hierarchical clustering and the proper data processing, LASCAD is less sensitive to the number of latent topics of the LDA, *t_num*, which is known to be difficult to tune Binkley et al. (2014); Grant et al. (2013). Therefore unlike prior LDA-based approaches, we were able to set a default value to *t_num* and make the parameter transparent to users which makes LASCAD more usable.

LASCAD mitigated another problem in previous tools by producing a bounded number of categories. Although previous tools claimed to automatically generate number of categories, in practice they generate an excessively large and hard to interpret number of categories. According to their paper Kawaguchi et al. (2006), 14 out of the 40 categories identified by MUDABlue were difficult to interpret in a dataset of only 41 applications and 6 actual categories. LASCAD allows the user to specify an upper bound on the desired number of categories, and we showed that LASCAD is not sensitive to this parameter permitting a rough estimate from the user. Although an actual automated approach would be useful, in some cases users may need a specific (or bounded) number of categories for the purpose of visualization or software showcases in app stores or web catalogs.

We evaluated LASCAD for software categorization using three labeled data sets, with two data sets from the literature, and one data set we built. For all three data sets, LASCAD consistently outperformed prior tools by achieving higher F-score accuracy and obtaining lower relDiff values without showing sensitivity to parameter settings. Our investigation with the 103-application data set also revealed that LASCAD was less sensitive to parameter settings than LACT, showing that LASCAD was easier to configure. We also evaluated LASCAD for similar application detection using two data sets. For the unlabeled data set of 5,220 applications and given 38 queries, the relevance of the Top-1 and Top-5 applications was 71% and 64% respectively, whereas the relevance of the top-1 results of the search based on readme files was only 23%.

In summary, LASCAD demonstrates great *usability* and *reliability* when categorizing and detecting similar software regardless of the application domains or implementation languages of software. With LASCAD's better effectiveness than existing tools, we envision our tool will further facilitate expertise sharing, program comprehension, rapid prototyping, and plagiarism detection.

LASCAD, and similar source code based categorization and application search, can play a big role in transforming source code search engines such as GitHub. Huge number of open source projects can be automatically categorized into groups regardless if they have documentation or not. Each group can be inspected and labeled based on a sample of the projects it has. This approach can build large showcases or label projects for better functionality-based search.

By providing high-quality similar software, LASCAD will also boost relevant research areas such as automatic program repair Sidiroglou-Douskos et al. (2015) and security vulnerability detection Lu et al. (2015), which compare similar software for anomalies and reveal potential software defects.

This paper is our first step to improve existing software categorization and similar software search techniques by integrating LDA with hierarchical clustering. As the next step, we aim to combine the keywords of the identified latent topics with program static analysis to further establish finer-grained mappings between source code files of different applications. In this way, we can facilitate program comprehension by helping developers more precisely locate the similar code implementation they are looking for, and by highlighting the distinct implementation between similar software.

8. Acknowledgements

This work was supported by the National Science Foundation [grant number CCF-1565827], and Office of Naval Research [grant number ONR-N00014-17-1-2498].

References

Bajracharya, S. K., Ossher, J., Lopes, C. V., 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Baldi, P. F., Lopes, C. V., Linstead, E. J., Bajracharya, S. K., 2008. A theory of aspects as latent topics. SIGPLAN Not.

Bates, R., 2012. ruby-warrior: Game written in Ruby for learning Ruby and artificial intelligence. `https://github.com/ryanb/ruby-warrior`.

Binkley, D., Heinz, D., Lawrie, D., Overfelt, J., 2014. Understanding lda in source code analysis. In: Proceedings of the 22Nd International Conference on Program Comprehension. ACM, pp. 26–36.

Bird, S., Klein, E., Loper, E., 2009. Natural language processing with Python. " O'Reilly Media, Inc.".

Blei, D., Lafferty, J., 2009. Text mining: Classification, clustering, and applications. chapter Topic Models, Chapman & Hall/CRC.

Blei, D. M., Ng, A. Y., Jordan, M. I., 2003. Latent dirichlet allocation. the Journal of machine Learning research 3, 993–1022.

Borges, H. S., Valente, M. T., 2017. Application Domain of 5,000 GitHub Repositories. `https://zenodo.org/record/804474#.WlZwVsbMw6i`.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., et al., 2013. Api design for machine learning software: experiences from the scikit-learn project. arXiv preprint arXiv:1309.0238.

Chen, C., Gao, S., Xing, Z., March 2016a. Mining analogical libraries in q a discussions – incorporating relational and categorical knowledge into word embedding. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 1. pp. 338–348.

Chen, C., Gao, S., Xing, Z., 2016b. Mining analogical libraries in q&a discussions -incorporating relational and categorical knowledge into word embedding. In: 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 338–348.

Chen, C., Xing, Z., 2016a. Similartech: Automatically recommend analogical libraries across different programming languages. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 834–839.

Chen, C., Xing, Z., 2016b. Similartech: Automatically recommend analogical libraries across different programming languages. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016. ACM, New York, NY, USA, pp. 834–839.
URL `http://doi.acm.org/10.1145/2970276.2970290`

Chen, T.-H., Thomas, S. W., Hassan, A. E., 2015. A survey on the use of topic models when mining software repositories. Empirical Software Engineering, 1–77.

Cohen, I., 2017. map-chat: A super simple location based chat. `https://github.com/idoco/map-chat`.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., Harshman, R., 1990. Indexing by latent semantic analysis. Journal of the American society for information science 41 (6), 391.

Derby, 2018. Derby. `http://derbyjs.com/docs/derby-0.6/components`.

Django, 2017. django: The Web framework for perfectionists with deadlines. `https://github.com/django/django`.

Everitt, B. S., Landau, S., Leese, M., 2009. Cluster Analysis. Wiley Publishing.

GitHub, 2016. Showcases. `https://github.com/showcases`.

GitHub, 2017. Notifications & Stars. `https://github.com/blog/1204-notifications-stars`.

GitHub, 2018. GitHub API v3. `https://developer.github.com/v3/`.

Google, 2015. codesearch: Fast, indexed regexp search over large file trees. `https://github.com/google/codesearch`.

Google, 2018. ibid: Ibid is a multi-protocol general purpose chat bot written in Python. Bugs tracked on launchpad. `https://github.com/ibid/ibid`.

Grant, S., Cordy, J. R., Skillicorn, D. B., 2013. Using heuristics to estimate an appropriate number of latent topics in source code analysis. Science of Computer Programming 78 (9), 1663–1678.

Hufflen, J.-M., 2004. MlBibTeX: Beyond LaTeX. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 203–215.

Jones, E., Oliphant, T., Peterson, P., et al., 2001. SciPy: Open source scientific tools for Python. [Online; accessed 2016-10-14].
URL `http://www.scipy.org/`

Kawaguchi, S., Garg, P. K., Matsushita, M., Inoue, K., 2006. Mudablue: An automatic categorization system for open source repositories. Journal of Systems and Software 79 (7), 939–953.

Kontogiannis, K., 1993. Program representation and behavioural matching for localizing similar code fragments. In: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1. CASCON '93. IBM Press, pp. 194–205.
URL `http://dl.acm.org/citation.cfm?id=962289.962307`

Leão, E., 2017. clumsy-bird: A MelonJS port of the famous Flappy Bird Game. `https://github.com/ellisonleao/clumsy-bird`.

Lin, J., 1991. Divergence measures based on the shannon entropy. IEEE Transactions on Information theory 37 (1), 145–151.

Linares-Vásquez, M., Holtzhauer, A., Poshyvanyk, D., 2016. On automatically detecting similar android apps. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). IEEE, pp. 1–10.

Liu, C., Chen, C., Han, J., Yu, P. S., 2006. Gplag: Detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '06. ACM, New York, NY, USA, pp. 872–881.
URL `http://doi.acm.org/10.1145/1150402.1150522`

Lu, K., Li, Z., Kemerlis, V. P., Wu, Z., Lu, L., Zheng, C., Qian, Z., Lee, W., Jiang, G., 2015. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In: 22nd Annual Network and Distributed System Security Symposium.

McKinney, W., 2011. pandas: a foundational python library for data analysis and statistics. Python for High Performance and Scientific Computing, 1–9.

McMillan, C., Grechanik, M., Poshyvanyk, D., 2012a. Detecting similar software applications. In: Software Engineering (ICSE), 2012 34th International Conference on. IEEE, pp. 364–374.

McMillan, C., Hariri, N., Poshyvanyk, D., Cleland-Huang, J., Mobasher, B., 2012b. Recommending source code for use in rapid software prototypes. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, pp. 848–858.

McMillan, C., Linares-Vasquez, M., Poshyvanyk, D., Grechanik, M., 2011. Categorizing software applications for maintenance. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, pp. 343–352.

Michail, A., Notkin, D., 1999. Assessing software libraries by browsing similar classes, functions and relationships. In: Proceedings of the 21st International Conference on Software Engineering. ICSE '99. ACM, New York, NY, USA, pp. 463–472.
URL `http://doi.acm.org/10.1145/302405.302678`

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A., 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 522–531.

Sager, T., Bernstein, A., Pinzger, M., Kiefer, C., 2006. Detecting similar java classes using tree algorithms. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. MSR '06. ACM, New York, NY, USA, pp. 65–71.
URL `http://doi.acm.org/10.1145/1137983.1138000`

Schuler, D., Dallmeier, V., Lindig, C., 2007. A dynamic birthmark for java. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. ASE '07. ACM, New York, NY, USA, pp. 274–283.
URL `http://doi.acm.org/10.1145/1321631.1321672`

Sidiroglou-Douskos, S., Lahtinen, E., Long, F., Rinard, M., 2015. Automatic error elimination by horizontal code transfer across multiple applications. SIGPLAN Not.

Sourcegraph, 2017. Sourcegraph. `https://sourcegraph.com`.

Tian, K., Revelle, M., Poshyvanyk, D., 2009. Using latent dirichlet allocation for automatic categorization of software. In: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on. IEEE, pp. 163–166.

Wallach, H. M., 2006. Topic modeling: Beyond bag-of-words. In: Proceedings of the 23rd International Conference on Machine Learning.

Wilkinson, L., Friendly, M., 2009. The history of the cluster heat map. The American Statistician.

Zaki, M. J., Meira Jr, W., 2014. Data mining and analysis: fundamental concepts and algorithms. Cambridge University Press.

Zhang, Y., Lo, D., Kochhar, P. S., Xia, X., Li, Q., Sun, J., Feb 2017. Detecting similar repositories on github. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 13–23.