Exploring Regular Expression Usage and Context in Python

ABSTRACT

Due to the popularity and pervasive use of regular expressions, researchers have created tools to support their creation, validation, and use. However, little is known about the context in which regular expressions are used, the features that are most common, and how behaviorally similar regular expressions are to one another. This information is critical to inform tool designers about how to best support developers working with regular expressions.

In this paper, we explore the context in which regular expressions are used through a combination of developer surveys and repository analysis. We survey 18 professional developers from a small software company about their regular expression usage and pain points. Our results indicate that developers frequently use regular expressions in their programming practices, often composing regular expressions at least weekly. Then, we analyzed nearly 4,000 open source Python projects from GitHub and extracted nearly 14,000 unique regular expression patterns that were used for analysis, focusing on how often distinct features are used. We also map the most common features used in regular expressions to those features supported by four common regular expression engines from industry and academia: brics, Hampi, RE2, and Rex. Using similarity analysis of regular expressions across projects, we identify six common behavioral clusters that describe how regular expressions are often used in practice. This is the first rigorous examination of regex usage and it provides empirical evidence to support design decisions by regex tool builders.

1. INTRODUCTION

Regular expressions (regexes) are an abstraction of keyword search that enables the identification of text using a pattern instead of an exact keyword. Regexes are commonly used for parsing text using a general purpose language like Python, validating content entered into web forms using Javascript, and searching text files for a particular pattern using tools like grep, vim or Eclipse.

Although regexes are powerful and versatile, they can be hard to understand, maintain, and debug, resulting in tens of thousands of bug reports [30].

Due in part to their common use across programming languages and how susceptible regexes are to error, many researchers and practitioners have developed tools to support more robust regex creation [30] or to allow visual debugging [6]. Other research has focused on learning regular expressions from text [4, 21], avoiding human composition altogether. Researchers have also explored applying regexes to test case generation [2, 15, 16, 31], as specifications for string constraint solvers [18,32] and using regexes as queries in a data mining framework [7]. Regexes are also employed in critical missions like mysql injection prevention [35] and network intrusion detection [26], or in more diverse applications like DNA sequencing alignment [3].

Regex researchers and tool designers must pick what features to include or exclude, which can be a difficult design decision. Supporting advanced features may be more expensive, taking more time and potentially making the project too complex and cumbersome to execute well. A selection of only the simplest of regex features limits the applicability or relevance of that work. Despite extensive research effort in the area of regex support, no research has been done about how regexes are used in practice and what features are essential for the most common use cases.

The goal of this work is to explore 1) the context in which developers use regular expressions, and 2) the features and similarities of regular expressions found in Python[1] *projects.*

First, we survey professional developers about how they use regexes. Second, we gather a sample of regexes from Python projects and analyze the frequency of feature inclusion in distinct patterns (e.g., kleene star: `*` and the end anchor: `$` are features). Third, we investigate what features are supported by four large projects that aim to support regex usage (brics [25], hampi [18], Rex [33], and RE2 [28]), and which features are not supported, but are frequently used by developers. Finally, we cluster regular expressions that appear in multiple projects by behavior, investigating high-level behavioral themes in these clusters.

Our results indicate that regexes are most frequently used in command line tools and IDEs. We observe that the capture group and endpoint anchor features are widely used by developers, but poorly supported in regex tools. Capturing

[1] Python is the fourth most common language on GitHub (after Java, Javascript and Ruby) and Python's regex pattern language is close enough to other regex flavors that our conclusions are likely to generalize.

the contents of brackets and searching for delimiter characters were some of the most apparent behavioral themes observed in the regex clusters created by our technique. The contributions of this work are:

- A survey of 18 professional software developers about their experience with regular expressions,
- An empirical analysis of regex feature usage in nearly 14,000 regular expressions in 3,898 open-source Python projects, mapping of those features to those supported by common regex tools and survey results showing the impact of not supporting various features,
- An approach for measuring behavioral similarity of regular expressions and qualitative analysis of the most common behaviorally similar clusters, and
- A discussion of opportunities for future work in supporting programmers in writing regular expressions.

2. RELATED WORK

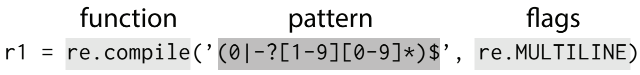
Regular expressions have been a focus point in a variety of research objectives. From the user perspective, tools have been developed to support more robust creation [30] or to allow visual debugging [6]. Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text [4,21].

Regarding applications, regular expressions have been used for test case generation [2,15,16,31], and as specifications for string constraint solvers [18,32]. Regexes are also employed in critical missions like mysql injection prevention [35] and network intrusion detection [26], or in more diverse applications like DNA sequencing alignment [3] or querying RDF data [1,20].

As a query language, lightweight regular expressions are pervasive in search. For example, some data mining frameworks use regular expressions as queries (e.g., [7]). Efforts have also been made to expedite the processing of regular expressions on large bodies of text [5].

Research tools like Hampi [18], and Rex [33], and commercial tools like brics [25] and RE2 [28], all support the use of regular expressions in various ways. Hampi was developed in academia and uses regular expressions as a specification language for a constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in applications such as test case generation [2,31]. Brics is an open-source package that creates automata from regular expressions for manipulation and evaluation. RE2 is an open-source tool created by Google to power code search with an efficient regex engine.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterns [22] and bug characterizations [12]. Exploring language feature usage by mining source code has been studied extensively for Smalltalk [8,9], JavaScript [29], and Java [14,17,23,27], and more specifically, Java generics [27] and Java reflection [23]. To our knowledge, this is the first work to mine and evaluate regular expression usages from existing software repositories. Related to mining work, regular expressions have been used to form queries in mining framework [7], but have not been the focus of the mining activities. Surveys have been used to measure adoption of various programming languages [13, 24], and been combined with software repository analysis [24], but have not focused on regular expressions.



```
r1 = re.compile('(0|-?[1-9][0-9]*)$', re.MULTILINE)
```

Figure 1: Example of one regex utilization

3. STUDY

To understand how programmers use regular expressions in Python projects, we scraped 3,898 Python projects from GitHub, and recorded regex usages for analysis. Throughout the rest of this paper, we employ the following terminology:

Utilization: A *utilization* occurs whenever a developer uses a regex in a project. We detect utilizations by recording all calls to the `re` module in Python. Within a particular file in a project, a utilization is composed of a function, a pattern, and 0 or more flags. Figure 1 presents an example of one regex utilization, with key components labeled. The function call is `re.compile`, `(0|-?[1-9][0-9]*)$` is the regex string, or pattern, and `re.MULTILINE` is an (optional) flag. This utilization will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with the `$` token matching at the end of each line because of the `re.MULTILINE` flag. Thought of another way, a regular expression utilization is one single invocation of the `re` library.

Pattern: A *pattern* is extracted from a utilization, as shown in Figure 1. In essence, it is a string, but more formally it is an ordered series of regular expression language feature tokens. The pattern in Figure 1 will match if it finds a zero at the end of a line, or a (possibly negative) integer at the end of a line (i.e., due to the `-?` sequence denoting zero or one instance of the `-`).

Note that because the vast majority of regex features are shared across most general programming languages (e.g., Java, C, C#, or Ruby), a Python pattern will (almost always) behave the same when used in other languages, whereas a utilization is not universal in the same way (i.e., it may not compile in other languages, even with small modifications to function and flag names). As an example, the `re.MULTILINE` flag, or similar, is present in Python, Java, and C#, but the Python `re.DOTALL` flag is not present in C# though it has an equivalent flag in Java.

In this work, we primarily focus on patterns since they are cross-cutting across languages and are the primary way of specifying the matching behavior for every utilization. Next, we describe the research questions and how the data set was collected and analyzed.

3.1 Research Questions

To understand the contexts in which regular expressions are used and feature usage, we perform a survey of developers and explore regular expressions found in Python projects. We aim to answer the following research questions:

RQ1: In what contexts do developers use regular expressions?

We designed and deployed a survey about when, why, and how often they use regular expressions. This was completed by 18 professional developers at a small software company.

RQ2: How is the `re` module used in Python projects?
We explore invocations of the `re` module in 3,898 Python projects.

RQ3: Which regular expression language features are most commonly used in Python?
We consider regex language features to be tokens that specify the matching behavior of a regex pattern, for example, the `+` in `ab+`. All studied features are listed and described in Section 3.3 with examples. We then map the feature coverage for four common regex support tools, brics, hampi, RE2 and Rex, and explore survey responses of feature usage for some of the less supported features.

RQ4: How behaviorally similar are regexes across projects?
As this is a first step in understanding behavioral overlap in regexes, we measure similarity between pairs of regexes by generating strings that match each and evaluating each other regex against those strings to build a similarity matrix. Then we use clustering to form behaviorally similar groupings.

Note that regular expressions in modern code libraries are not the same as regular languages in formal language theory. Some features of regular expression libraries, such as backreferences, provide expressiveness beyond regular languages. This allows a regular expression pattern to match, for example, repeat words, such as “cabcab”, using the pattern `(.+)\1`. However, building an automaton to recognize such a pattern is infeasible. Thus, performing a more formal analysis, such as a containment analysis, on the automata representing regular expressions found in practice, is not feasible. We use string matching as an alternative that allows regexes with such features to be considered in the behavioral analysis.

Next, we describe our survey, how the corpus of regex patterns was built, how features were analyzed, and how the clustering was performed.

3.2 Survey Design

To understand the context of when and how programmers use regular expressions, we designed a survey with 40 questions about regex usage and context.[2] This survey was deployed to 22 professional developers at Dwolla, a company that provides software for online and mobile payment management. Participation was voluntary and participants were entered in a lottery for a $50 gift card. The survey was completed by 18 participants (82% response rate) that identified as software developer/maintainers. The questions asked about regex usage frequency, languages, purposes, and the use of various language features, as explored in Section 4.1 and Section 4.3.

3.3 Regex Corpus

Using the GitHub API, we scraped 3,898 Python projects. We did so by dividing a range of about 8 million repo IDs for Python projects into 32 sections of equal size and scanning from the beginning of those segments until we ran out of memory. At that point, we felt we had enough data to do an analysis without further perfecting our mining techniques. We built the AST of each file in each project to find utilizations of the ’re’ module functions. In most projects, almost all regex utilizations are present in the most recent version of a project, but to be more thorough, we also scanned up to 19 earlier versions. The number 20 was chosen to try and maximize returns on computing resources invested after observing the scanning process in many hours of trial scans. Each project’s commit history was scanned at 20 evenly-spaced commits. If the project had fewer than 20 commits, then all commits were scanned. The most recent commit was always included, and the spacing between all other chosen commits was determined by dividing the remaining number of commits by 19 (rounding as needed). All regex utilizations were obtained, sans duplicates. Within a project, a duplicate utilization was marked when two versions of the same file have the same function, pattern and flags. In the end, we observed and recorded 53,894 non-duplicate regex utilizations in 3,898 projects.

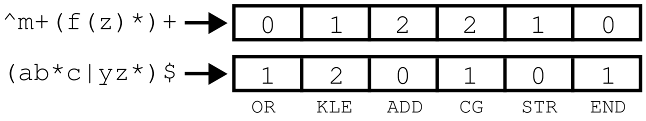


Figure 2: Two patterns parsed into feature vectors

In collecting the set of distinct patterns for analysis, we ignore the 12.7% of utilizations using flags, which can alter regex behavior. An additional 6.5% of utilizations contained patterns that could not be compiled because the pattern was non-static (e.g., used some runtime variable). The remaining 80.8% (43,525) of the utilizations were collapsed into 13,711 distinct pattern strings. Each of the pattern strings was preprocessed by removing Python quotes (‘`\\W`’ becomes `\\W`), unescaping escaped characters (`\\W` becomes `\W`) and parsing the resulting string using an ANTLR-based, open source PCRE parser[3]. This parser was unable to support 0.5% (73) of the patterns due to unsupported unicode characters. Another 0.2% (25) of the patterns used regex features that we chose to exclude because appeared very rarely (e.g., reference conditions).

The 13,597 distinct pattern strings that remain were each assigned a weight value equal to the number of distinct projects the pattern appeared in. We refer to this set of weighted, distinct pattern strings as the *corpus*.

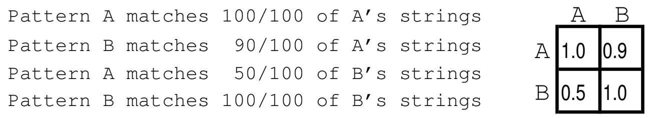
3.4 Analyzing Features

For each escaped pattern, the PCRE-parser produces a tree of feature tokens, which is converted to a vector by counting the number of each token present in the tree. For a simple example, consider the patterns in Figure 2. The pattern ‘`^m+(f(z)*)+`’ contains four different types of tokens. It contains the kleene star (KLE), which is specified using the asterisk ‘`*`’ character, additional repetition (ADD), which is specified using the plus ‘`+`’ character, capture groups (CG), which are specified using pairs of parenthesis ‘`(...)`’ characters, and the start anchor (STR), which is specified using the caret ‘`^`’ character at the beginning of a pattern.

Once all patterns were transformed into vectors, we examined each feature independently for all patterns, tracking

[2] https://github.com/softwarekitty/tour_de_source/blob/master/regex_usage_in_practice_survey.pdf

[3] https://github.com/bkiers/pcre-parser



```
Pattern A matches 100/100 of A's strings
Pattern B matches  90/100 of A's strings
Pattern A matches  50/100 of B's strings
Pattern B matches 100/100 of B's strings
```

$$\begin{array}{c|cc} & A & B \\ \hline A & 1.0 & 0.9 \\ B & 0.5 & 1.0 \end{array}$$

Figure 3: A similarity matrix created by counting strings matched

the number of patterns and projects that the each feature appears in at least once.

3.5 Clustering and Semantic Analysis

Our semantic analysis clusters regular expressions by their behavioral similarity. Consider two unspecified patterns A and B, a set mA of 100 strings that pattern A matches, and a set mB of 100 strings that pattern B matches. If pattern B matches 90 of the 100 strings in the set mA, then B is 90% similar to A. If pattern A only matches 50 of the strings in mB, then A is 50% similar to B. We use similarity scores to create a similarity matrix as shown in Figure 3. In row A, column B we see that B is 90% similar to A. In row B, column A, we see that A is 50% similar to B. Each pattern is always 100% similar to itself, by definition.

Once the similarity matrix is built, the values of cells reflected across the diagonal of the matrix are averaged to create a half-matrix of undirected similarity edges, as illustrated in Figure 4. This facilitates clustering using the Markov Clustering (MCL) algorithm[4]. We chose MCL because it offers a fast and tunable way to cluster items by similarity and it is particularly useful when the number of clusters is not known *a priori*.

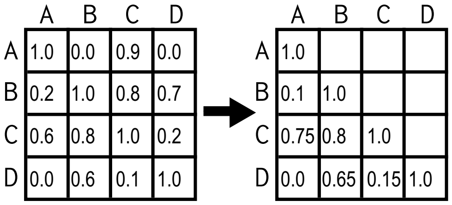
In the implementation, strings are generated for each pattern using Rex [33]. Rex generates matching strings by representing the regular expression as an automaton, and then passing that automation to a constraint solver that generates members for it[5]. If asked to produce more strings than the automation can provide, Rex will instead produce a list of all possible strings. Our goal is to generate 400 strings for each pattern to balance the runtime of the similarity analysis with the precision of the similarity calculations.

For clustering, we prune the similarity matrix to retain all similarity values greater than or equal to 0.75, setting the rest to zero, and then using MCL. We note that MCL can be tuned using many parameters, including inflation and filtering out all but the top-k edges for each node. After exploring the quality of the clusters using various tuning parameter combinations, the best clusters (by inspection) were found using an inflation value of 1.8 and k=83. The end result is clusters of highly semantically similar regular expressions. The top 100 clusters are categorized by inspection into six categories of behavior (see Section 4.4).

4. RESULTS

In this section, we present the results of each research question.

4.1 RQ1: In what contexts do developers use regular expressions?



$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 1.0 & 0.0 & 0.9 & 0.0 \\ B & 0.2 & 1.0 & 0.8 & 0.7 \\ C & 0.6 & 0.8 & 1.0 & 0.2 \\ D & 0.0 & 0.6 & 0.1 & 1.0 \end{array} \rightarrow \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 1.0 & & & \\ B & 0.1 & 1.0 & & \\ C & 0.75 & 0.8 & 1.0 & \\ D & 0.0 & 0.65 & 0.15 & 1.0 \end{array}$$

Figure 4: Creating a similarity graph from a similarity matrix

Table 1: Survey results for number of regexes composed per year by technical environment (RQ1)

Language/Environment	0	1-5	6-10	11-20	21-50	51+
General (e.g., Java)	1	6	5	3	1	2
Scripting (e.g., Perl)	5	4	3	3	2	1
Query (e.g., SQL)	15	2	0	0	1	0
Command line (e.g., grep)	2	5	3	2	0	6
Text editor (e.g., IntelliJ)	2	5	0	5	1	5

The survey was completed by 18 professional software developers with an average of nine years of programming experience ($\sigma = 4.28$). On average, survey participants report to compose 172 regexes per year ($\sigma = 250$) and compose regexes on average once per month, with 28% composing multiple regexes in a week and an additional 22% composing regexes once per week. That is, 50% of respondents uses regexes at least weekly. Table 1 shows how frequently participants compose regexes using each of several languages and technical environments. Six (33%) of the survey participants report to compose regexes using general purpose programming languages (e.g., Java, C, C#) 1-5 times per year and five (28%) do this 6-10 times per year. Regexes were rarely used in query languages like SQL, but for command line usage in tools such as grep, 6 (33%) participants use regexes 51+ times per year.

Table 2 shows how frequently, on average, the participants use regexes for various actives. Participants answered ques-

Table 2: Survey results for regex usage frequencies for activities, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1 (RQ1)

Activity	**Frequency**
Locating content within a file or files	4.4
Capturing parts of strings	4.3
Parsing user input	4.0
Counting lines that match a pattern	3.2
Counting substrings that match a pattern	3.2
Parsing generated text	3.0
Filtering collections (lists, tables, etc.)	3.0
Checking for a single character	1.7

[4] http://micans.org/mcl/

[5] http://research.microsoft.com/en-us/projects/rex/

tions using a 6-point likert scale including very frequently, frequently, occasionally, rarely, very rarely, and never. Assigning values from 1 to 6, where 6 is the most frequent, the responses were averaged across participants. Among the most common usages are capturing parts of a string and locating content within a file, with both occurring somewhere between occasionally and frequently.

Using a similar 7-point likert scale that includes 'always' as a seventh point, developers indicated that they test their regexes with the same frequency as they test their code (average response was 5.2, which is between frequently and very frequently). Half of the 18 developers indicate that they use external tools to test their regexes, and the other half indicated that they only use tests that they write themselves. Of the nine developers using tools, six mentioned some online composition aide such as `regex101.com` where a regex and input string are entered, and the input string is highlighted according to what is matched.

When asked an open ended question about pain points encountered with regular expressions, we observed three main categories. The most common, "hard to compose," was represented in 61% (11) responses. Next, 39% (7) developers responded that regexes are "hard to read" and 17% (3) indicated difficulties with "inconsistency across implementations," which manifest when using regexes in multiple languages. These responses do not sum to 18 as three developers provided overlapping answers.

Summary - RQ1.

Overall, regexes are used frequently with common usages including locating content within a file, capturing parts of strings, and parsing user input. The fact that all the surveyed developers compose regexes, and half of the developers use tools to test their regexes indicates the importance of tool development for regex. Developers complain about regexes being hard to read and hard to compose, and most of the tools that they indicate using are focused on composition.

4.2 RQ2: How is the `re` module used?

We explore regex utilizations and flags used in the scraped Python projects.

Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one regex utilization. To illustrate how saturated projects are with regexes, we measure utilizations per project, files scanned per project, files contained utilizations, and utilizations per file, as shown in Table 3.

Of projects containing at least one utilization, the average utilizations per project was 32 and the maximum was 1,427. The project with the most utilizations is a C# project[6] that maintains a collection of source code for 20 Python libraries, including larger libraries like `pip`, `celery` and `ipython`. These larger Python libraries contain many utilizations. From Table 3, we also see that each project had an average of 11 files containing any utilization, and each of these files had an average of 2 utilizations.

The number of projects that use each of the `re` functions is shown in Figure 5. The y-axis denotes the total utilizations, with a maximum of 53,894. The `re.compile` function encompasses 57.6% of all utilizations, presumably because each usage of those functions can accept a regex object compiled using `re.compile` as an argument.

Table 3: How saturated are projects with utilizations? (RQ2)

source	Q1	Avg	Med	Q3	Max
utilizations per project	2	32	5	19	1,427
files per project	2	53	6	21	5,963
utilizing files per project	1	11	2	6	541
utilizations per file	1	2	1	3	207

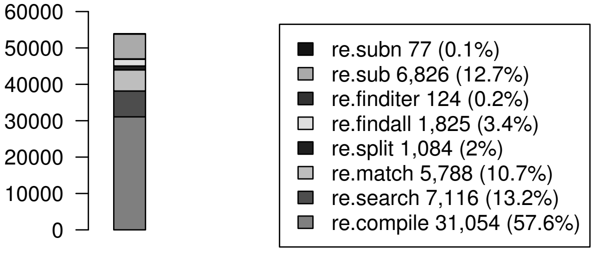


Figure 5: How often are `re` functions used? (RQ2)

When considering flag use, we excluded the default flag, which is built into the `re` module, and present internally whenever no flag is used. Of all utlizations, 87.3% had no flag, or explicitly specified the default flag. The debug flag, which causes the `re` regex engine to display extra information about its parsing process, was never observed. Figure 6 presents the number of projects in which each flag appears. Ignorecase (43.8%) and multiline (25.8%) were the most frequently used. Although multiple flags can be combined using a bitwise or, this was never observed.

Summary - RQ2.

Only about half of the projects sampled contained any utilizations. Most utilizations used the `re.compile` function to compile a regex object before actually using the regex to find a match. Most utilizations did not use a flag to modify matching behavior.

4.3 RQ3: Which regular expression language features are most commonly used in Python?

To measure feature usage, we count the number of usages of each feature per project and as a percent of all distinct regular expression patterns in the corpus.

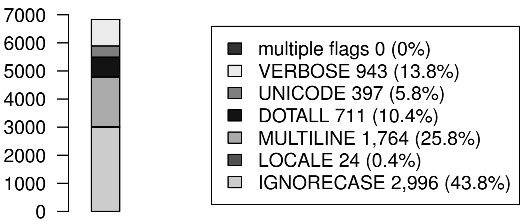


Figure 6: Which behavioral flags are used? (RQ2)

[6] `https://github.com/Ouroboros/Arianrhod`

Table 4: How frequently do features appear in projects? (RQ3)

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nProjects	% projects
1	ADD	one-or-more repetition	z+	●	●	●	●	6,003	44.1	1,204	73.2
2	CG	a capture group	(caught)	●	●	●	●	7,130	52.4	1,194	72.6
3	KLE	zero-or-more repetition	.*	●	●	●	●	6,017	44.3	1,099	66.8
4	CCC	custom character class	[aeiou]	●	●	●	●	4,468	32.9	1,026	62.4
5	ANY	any non-newline char	.	●	●	●	●	4,657	34.3	1,005	61.1
6	RNG	chars within a range	[a-z]	●	●	●	●	2,631	19.3	848	51.6
7	STR	start-of-line	^	○	●	●	●	3,563	26.2	846	51.4
8	END	end-of-line	$	○	●	●	●	3,169	23.3	827	50.3
9	NCCC	negated CCC	[^qwxf]	●	●	●	●	1,935	14.2	776	47.2
10	WSP	\t \n \r \v \f or space	\s	○	●	●	●	2,846	20.9	762	46.3
11	OR	logical or	a|b	●	●	●	●	2,102	15.5	708	43
12	DEC	any of: 0123456789	\d	○	●	●	●	2,297	16.9	692	42.1
13	WRD	[a-zA-Z0-9_]	\w	○	●	●	●	1,430	10.5	650	39.5
14	QST	zero-or-one repetition	z?	●	●	●	●	1,871	13.8	645	39.2
15	LZY	as few reps as possible	z+?	○	●	○	●	1,300	9.6	605	36.8
16	NCG	group without capturing	a(?:b)c	○	●	○	●	791	5.8	404	24.6
17	PNG	named capture group	(?P<name>x)	○	●	○	●	915	6.7	354	21.5
18	SNG	exactly n repetition	z{8}	●	●	●	●	581	4.3	340	20.7
19	NWSP	any non-whitespace	\S	○	●	●	●	484	3.6	270	16.4
20	DBB	$n \leq x \leq m$ repetition	z{3,8}	●	●	●	●	367	2.7	238	14.5
21	NLKA	sequence doesn't follow	a(?!yz)	○	○	○	○	131	1	183	11.1
22	WNW	word/non-word boundary	\b	○	○	○	●	248	1.8	166	10.1
23	NWRD	non-word chars	\W	○	●	●	●	94	0.7	165	10
24	LWB	at least n repetition	z{15,}	●	●	●	●	91	0.7	158	9.6
25	LKA	matching sequence follows	a(?=bc)	○	○	○	○	112	0.8	158	9.6
26	OPT	options wrapper	(?i)CasE	○	●	○	●	231	1.7	154	9.4
27	NLKB	sequence doesn't precede	(?<!x)yz	○	○	○	○	94	0.7	137	8.3
28	LKB	matching sequence precedes	(?<=a)bc	○	○	○	○	80	0.6	120	7.3
29	ENDZ	absolute end of string	\Z	○	○	○	●	89	0.7	90	5.5
30	BKR	match the i^{th} CG	\1	○	○	○	○	60	0.4	84	5.1
31	NDEC	any non-decimal	\D	○	●	●	●	36	0.3	58	3.5
32	BKRN	references PNG	\g<name>	○	●	○	○	17	0.1	28	1.7
33	VWSP	matches U+000B	\v	○	○	●	●	13	0.1	15	0.9
34	NWNW	negated WNW	\B	○	○	○	●	4	0	11	0.7

4.3.1 Feature Usage

Table 4 displays feature usage from the corpus and relates it to four major regex related projects. Only features appearing in at least 10 projects are listed. The first column, *rank*, lists the rank of a feature (relative to other features) in terms of the number of projects in which it appears. The next column, *code*, gives a succinct reference string for the feature, and is followed by a *description* column that provides a brief comment on what the feature does. The *example* column provides a short example of how the feature can be used. The next four columns, (i.e., *brics*, *hampi*, *Rex*, and *RE2*), map to the four major research projects chosen for our investigation (see Section 4.3.2). We indicate that a project supports a feature with the '●' symbol, and indicate that a project does not support the feature with the '○' symbol. The final four columns contain two pairs of usage statistics. The first pair contains the number and percent of *patterns* that a feature appears in, out of the 13,912 patterns that make up the corpus. The second pair of columns contain the number and percent of *projects* that a feature appears in out of the 1645 projects scanned that contain at least one utilization.

One notable omission from Table 4 is the literal feature, which is used to specify matching any specific character. An example pattern that contains only one literal token is the pattern `'a'`. This pattern only matches the lowercase letter 'a'. The literal feature was found in 97.7% of patterns. We consider the literal feature to be ubiquitous in all patterns, and necessary for any regex related tool to support, and so exclude it from Table 4 and the rest of the feature analysis.

The eight most commonly used features, ADD, CG, KLE, CCC, ANY, RNG, STR and END, appear in over half the projects. CG is more commonly used in patterns than the highest ranked feature (ADD) by a wide margin (over 8%), even though they appear in similar numbers of projects.

4.3.2 Feature Support in Regex Tools

While there are many regex tools available, in this work, we focus on the feature support for four tools, brics, hampi, Rex and RE2, which offer diversity across developers (i.e., Microsoft, Google, open source, and academia) and across applications. Further, as we wanted to perform a feature analysis, these four tools and their features are well-documented, allowing for easy comparison.

To create the tool mappings, we consulted documentation for each tool. For brics, we collected the set of supported features using the formal grammar[7]. For hampi, we manually inspected the set of regexes included in the `lib/regex-hampi/sampleRegex` file within the hampi repository[8] (this may have been an overestimation, as this included more features than specified by the formal grammar[9]). For RE2, we used the supported feature documentation[10]. For Rex, we collected the feature set empirically because we tried to parse all scraped patterns with Rex for the semantic analysis (Section 4.4), and Rex provides comprehensive error feedback for unsupported features.

Of the four projects selected for this analysis, RE2 supports the most studied features (28 features) followed by hampi (25 features), Rex (21 features), and brics (12 features). All projects support the 8 most commonly used features except brics, which does not support STR or END. No projects support the four look-around features LKA, NLKA, LKB and NLKB. RE2 and hampi support the LZY, NCG, PNG and OPT features, whereas brics and Rex do not.

4.3.3 Survey Results for Feature Usage

The pattern language for Python, which is used to compose regexes, supports default character classes like the ANY or dot character class: `.` meaning, 'any character except newline' (a full list of features and examples is in the first four columns of Table 4). It also supports three other default character classes: `\d`, `\w`, `\s` (and their negations). All of these default character classes can be simulated using the custom character class (CCC) feature, which can create semantically equivalent regexes. For example the decimal character class: `\d` is equivalent to a CCC containing all 10 digits: `\d` ≡ `[0123456789]` ≡ `[0-9]`. Other default character classes such as the word character class: `\w` may not be as intuitive to encode in a CCC: `[a-zA-Z0-9_]`.

Survey participants were asked if they use only CCC, use CCC more than default, use both equally, use default more than CCC or use only default. Results for this question are shown in Table 5, with 67% (12) indicating that they use default the most. Participants were also asked to explain their preferences. Participants who favored CCC mostly said something equivalent to "it is more explicit," whereas the participants who favored default character classes said, "it is less verbose" and "I like using built-in code."

Table 5: Survey results for preferences between custom character and default character classes (RQ3)

Preference	Frequency
use only CCC	1
use CCC more than default	5
use both equally	2
use default more than CCC	10
use only default	2

Table 6: Survey results for regex usage frequencies, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1 (RQ3)

Group	Code	Frequency
endpoint anchors	(STR, END)	4.4
capture groups	(CG)	4.2
word boundaries	(WNW)	3.5
lazy repetition	(LZY)	2.9
(neg) look-ahead/behind	(LKA, NLKA, LKB, NLKB)	2.5

TODO.MID: Maybe we are looking at this and saying: oh there is no clear favorite, that's too bad for us as researchers. But instead we can say that diversity is alive in the regex community and that maintaining both ways of doing this is important - tool builders should support both ways of doing things with char classes.

To further explore how frequently participants use various regex features, participants were asked five questions (on a 6-point likert scale) about how frequently they use specific related groups of features, chosen based on the tool feature support explored in Section 4.3.2. Results are shown in Table 6, indicating that lazy repetition and look-ahead features are rarely used and capture groups and endpoint anchors are occasionally to frequently used.

Summary - RQ3.

We found that the eight most common features are found in over 50% of the projects. We also identify brics as the project supporting the fewest features and RE2 as the project supporting the most features, and identify groups of features supported or not supported by the four regex projects.

From Table 4 we know that STR and END features are present in over half of the scanned projects containing utilizations. In our survey, over half (56%) of the respondents answered that they use endpoint anchors frequently or very

[7] `http://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html`

[8] `https://code.google.com/p/hampi/downloads/list`

[9] `http://people.csail.mit.edu/akiezun/hampi/Grammar.html`

[10] `https://re2.googlecode.com/hg/doc/syntax.html`

Table 7: Sample from an example cluster (RQ4)

index	pattern	nProjects	index	pattern	nProjects
1	':+'	8	5	'[:]'	6
2	'(:)'	8	6	'([^:]+):(.*)'	6
3	'(:+)'	8	7	'\s*:\s*'	4
4	'(:)(:*)'	8	8	'\:'	2

Table 8: Cluster categories and sizes (RQ4)

Category	Clusters	Patterns	Projects
Multi Matches	21	237	295
Specific Char	17	103	184
Anchored Patterns	20	85	141
Content of Parens	10	46	111
Two or More Chars	16	40	120
Code Search	15	27	92

frequently, and none of them claimed to never use them. The brics library does not support this feature, which is a missed opportunity for developers who could otherwise have used brics to model their regexes that use STR and END.

The LZY feature is present in over 36% of scanned projects with utilizations (Table 4), and yet was not supported by two of the four major regex projects we explored, brics and RE2. In our developer survey, 11% (2) of participants use this feature frequently and 6 (33%) use it occasionally, showing a modest impact on potential users.

When survey participants were asked if they prefer to always use numbered (BKR) or named (BKRN) back references, 66% (12) of survey participants said that they always use BKR, and the remaining 33% (6) said "it depends." No participants preferred named capture groups. BKR is present in 5% of scanned projects, while BKRN is present in only 1.7%, which corroborates our findings that numbered are generally preferred over named capture groups.

4.4 RQ4: How behaviorally similar are regexes across projects?

In clustering the regular expressions, we are interested in observing cross-project regex similarity and thus discarded patterns found in only one project. Further, since Rex does not support all the regex features present in the corpus, we only generated sets of strings for measuring similarity for 2,871 (21%) of the 13,597 patterns in the corpus. The impact is that 923 projects were excluded from the data set for the similarity analysis. Omitted features are indicated in Table 4, as described in Section 4.3.

From 2,871 distinct patterns, the MCL clustering technique identified 186 clusters with 2 or more patterns, and 2,042 clusters of size 1. Recall that only pairs of patterns with a similarity level of 0.75 were included in the matrix passed to MCL. The average size of clusters larger than size one was 4.5. Each pattern belongs to exactly one cluster.

Table 7 provides an example of a behavioral cluster containing 12 patterns (4 longer patterns omitted for brevity) with at least one pattern from this cluster present in 31 different projects. All patterns in this cluster share the literal ':' character. The smallest pattern, ':+', matches one or more colons. Generally, the smallest pattern in a cluster gives a good idea of what all the patterns within the cluster have in common. A shorter pattern will tend to have less extraneous behavior because it is specifying less behavior, yet, in order for the smallest pattern to be clustered, it had to match most of the strings created by Rex from other patterns within the cluster, and so we assume that the smallest pattern is a good representation of the cluster.

For the rest of this paper, a cluster will be represented by one of the shortest patterns it contains, followed by the number of projects any member of the cluster appears in, so the cluster in Table 7 will be represented as ':+'(31).

We manually mapped the top 100 largest clusters based on the number of projects into 6 behavioral categories (determined by inspection). The largest cluster was left out, as it was composed of patterns that trivially matched almost any string, like 'b*' and '^'. The remaining 99 clusters were all categorized. These clusters are briefly summarized in Table 8, showing the name of the category, the number of clusters it represents, number of patterns in those clusters, and the number of projects. The most common category is *Multi Matches*, which contains clusters that have alternate behaviors (e.g., matching a comma or a semicolon, as in ',|;'(18)). Each cluster was mapped to exactly one category. Next, we describe the categories, ordered by the number of projects the regex patterns map to.

4.4.1 Multiple Matching Alternatives

The patterns in these clusters match under a variety of conditions by using a character class or a disjunctive |. For example: '(\W)'(89) matches any alphanumeric character, '(\s)'(89) matches any whitespace character, '\d'(58) matches any numeric character, and ',|;'(18) matches a comma or semicolon. Most of these clusters are represented by patterns that use default character classes, as opposed to custom character classes. This provides further support for our survey results to the question, *Do you prefer to use custom character classes or default character classes more often?*, in which a majority of participants indicated they use the default classes more than custom. This category contains 21 clusters, each appearing in an average of 33 projects. These clusters have a combined total of 237 patterns, with at least one pattern present in 295 projects.

TODO.MID: With multiple matching alternatives, I think it's important to realize and admit freely that our clustering technique will be somewhat eager to cluster these types of things together as similar. These regexes will match if just one character in a string matches a (sometimes very wide) range of characters like [a-z_]. TODO.MID: Think about it - this regex will match all strings generated by Rex that have any lowercase characters, even if Rex just randomly inserted those characters.

TODO.MID: Also on a different note, I think that within this cluster, we could do some statistics about how many characters are in these 'multiple matching alternative' clusters. I'm suspecting a histogram with a bump at 2-3 and a bump at around 25-40.

4.4.2 Specific Character Must Match

Each cluster in this category requires one specific character to match, for example: `‘\n\s*’(42)` matches only if a newline is found, `‘:+’(31)` matches only if a colon is found, `‘%’(22)`, matches only if a percent sign is found and `‘}’(14)` matches only if a right curly brace is found. This behavior contrasts with the survey in Section 4.1 in which participants reported to very rarely or never use regexes to check for a single character (Table 2). This category contains 17 clusters, each appearing in an average of 17.1 projects. These clusters have a combined total of 103 patterns, with at least one pattern present in 184 projects.

TODO.MID: Again, this ’single character’ kind of fails to connect the dots with what we said moments ago about how we are representing clusters with the shortest regex in a cluster. That’s a good way to represent them, but we cannot turn around and say that a large number of regexes are one character long (which it sort of seems like we are saying). Again I think we could benefit from a histogram, this time of the actual number of characters in the regexes in these clusters. So for example, the cluster revolving around `‘:+’` **TODO.MID: also has** `\\s*:\\s*` **TODO.MID: and** `\\s*([^: ]*)\\s*:(.*)` **TODO.MID: in it. This should not lead the reader to believe that the cluster is composed of one-character regexes, but that this character was an important delimiter in a pretty large group of diverse patterns! What does the tool designer take away from this? I think that most of all, it reinforces the idea of finding flags in text. This is handled well for ASCII, but in the future one must consider the users who are using different flags/delimiter symbols from other languages. Are they able to perform this absolutely critical and fundamental feature? This is related to a phenomenon called ‘Text Segmentation’. The Unicode Standard Annex is devoted to managing text segmentation across languages. It mentions a large number of boundary characters that should be of interest to support, including characters that are exotic to American researchers like ‘U+060D’ the Arabic Date Separator, or ‘U+18806’ the Mongilian Soft Hyphen.**

4.4.3 *Anchored Patterns*

Each of the clusters uses at least one endpoint anchor to require matches to be absolutely positioned, for example: `‘(\w+)$’(35)` captures the word characters at the end of the input, `‘^\s’(16)` matches a whitespace at the beginning of the input, and `‘^-?\d+$’(17)` requires that the entire input is an (optionally negative) integer. This category contains 20 clusters, each appearing in an average of 15.4 projects. These clusters have a combined total of 85 patterns, with at least one pattern present in 141 projects.

TODO.MID: The thing I want to mention about anchored patterns (but have struggled to say in the past) is that they are the only way to guarantee that a character does not appear in a particular location by specifying what is allowed. Consider the regex `^[-_A-Za-z0-9]+$` **TODO.MID: which will fail to match if an undesirable character like ‘?’ appears anywhere in the input. In logic, there is a similar phenomenon. That is, ‘Always’ is true iff ‘Not Exists’ of the negation is true, and by requiring an entire input to always maintain some abstraction, you can indirectly specify the negation of another (inverse) abstraction. Even with only one anchor point, a regex like** `.*[0-9]$` **TODO.MID: is creating an ultimatum about the end being a digit. Without the endpoint anchors, I don’t see how one could specify absolutes about an input.**

4.4.4 *Content of Brackets and Parenthesis*

The clusters in this category center around finding a pair of characters that surround content, often also capturing that content. For example, `‘\(.*\)’(29)` matches when content is surrounded by parentheses and `‘".*"’(25)` matches when content is surrounded by double quotes. The cluster `‘<(.+)>’(23)` matches and captures content surrounded by angled brackets. This category contains 10 clusters, each appearing in an average of 18.4 projects. These clusters have a combined total of 46 patterns, with at least one pattern present in 111 projects.

4.4.5 *Two or More Characters in Sequence*

These clusters require several characters in a row to match some pattern, for example: `‘\d+\.\d+’(30)` requires one or more digits followed by a period character, followed by one or more digits. The cluster `‘  ’(17)` requires two spaces in a row, and `‘@[a-z]+’(9)` requires the at symbol followed by two or more lowercase characters, as in a twitter handle. This category contains 16 clusters, each appearing in an average of 13 projects. These clusters have a combined total of 40 patterns, with at least one pattern present in 120 projects.

TODO.MID: Again, it might be interesting to look at what particular sequences are looking like. I think I mention this again in the discussion, but should we put it here instead?

4.4.6 *Code Search and Variable Capturing*

These clusters center around a recognizable effort to parse source code or URLs and often capture variable values in it, for example: `‘^https?://’(23)` matches a web address, and `‘(.+)=(.+)’(9)` matches an assignment statement, capturing both the variable name and value. The cluster `‘\$\{([\w\-]+)\}’(11)` matches an evaluated string interpolation and captures the code to evaluate. This category contains 15 clusters, each appearing in an average of 11.7 projects. These clusters have a combined total of 27 patterns, with at least one pattern present in 92 projects.

Summary - RQ4.

When tool designers are considering what features to include, data about usage in practice is valuable. Semantic similarity clustering helps to discern these behaviors by looking beyond the structural details of specific patterns and seeing trends in actual matching behavior. We are also able to find out what features are being used in these behavioral trends so that we can make assertions about why certain features are important.

We used the behavior of individual patterns to form clusters, and identified six main categories that clusters belonged to. Overall, we see that many clusters are defined by the presence of particular tokens, such as the colon for the cluster in Table 7. These six categories define what users are doing with regexes at a high level: matching with alternatives, matching literal characters, matching with sequences, matching with endpoint anchors, parsing contents of brackets or braces, or searching and capturing code, and can be considered in conjunction with the self-described regex activities from the survey in Table 2 to be representative of common uses for regexes.

5. DISCUSSION

In this section, we discuss the implications of these empirical findings on tool designers and users of regex tools and opportunities for future work.

5.1 Implications For Tool Designers

The results in this work have several implications for tool designers who want to effectively support developers who use regular expressions.

5.1.1 Capturing Specific Content

The survey results from Section 4.1 indicate that capturing parts of strings is among the most frequent activities for which developers use regexes. The capture group (CG) feature is the most frequently used feature in terms of patterns (Table 4). The CG feature has two functions: 1) it allows logical grouping as would be expected by parenthesis, and 2) it allows retrieval of information in one logical grouping. As mentioned in Section 4.4, capturing content was a primary goal evident in several categories of cluster, in particular, the fourth-largest cluster will capture the content between brackets or parentheses (Section 4.4.4). Thus, we observe that the ability to capture some part of a match provides a powerful tool to programmers.

TODO.MID: In my experience, I have used capture groups in the find and replace functionality of a text editor to transform large, well-structured documents in many steps. I have also used them in java programs to get the critical data from some predictable text (like part of a number). Then I have also used them in shell scripts in a similar way - to pass the captured content into a program. It's worth noting the prevalence of `(.*)` **TODO.MID: also - capturing everything around some markers like** `(.*)\\(.*\\)` **TODO.MID: or** `(.*){(.*)}(.*)` **TODO.MID: or** `\\[(.*)\\]`. **TODO.MID: I think it's hard for a reader of our paper to understand this trend without seeing the large number of similar-themed actual regexes, and then our findings feel a little flat because of course capture groups are used to get information... At the same time, I think it's important to recognize that about half of the patterns I've seen could not be expressed without using CG to specify logical groups for repetition or as the end of any abstraction expressed using other features. One very interesting question for future research is how to tell when regexes are using CG for just capturing, just grouping or both. We do not have an approach but it's a point to be explored.**

Any non-trivial tool or research that hopes to be applicable to regex use in practice must treat the CG feature as especially important, and must support some way to reason about what information is retrieved by capture groups.

5.1.2 Counting Lines and Finding Flags

Text files containing one unit of information per line are common in a wide variety of applications (for example log and csv files). Out of the 13,912 patterns in the corpus, 3,444 (24%) contained ANY followed by KLE (i.e., `'.*'`), often at the end of the pattern. One reasonable explanation for this tendency to put `'.*'` at the end of a pattern is that users want to disregard all matches after the first match on a single line in order to count how many distinct lines the match occurs on. Survey participants indicated an average frequency of "Counting lines that match a pattern" and "Counting substrings that match a pattern" at 3.2 or rarely/occasionally.

Delimiters that separate items on one line like `'\t'` are also quite common. Although survey participants indicated an average frequency of 1.7 (very rarely or never) for "checking for a single character," we found that 17 of the top 100 clusters revolved around the presence of a single character (Section 4.4.2). The top ranked activity for developers is "Locating content within a file or files," and usually this content is located using some small set of characters that the user knows will flag that content. Looking closely at the *Specific Character Must Match* cluster, some of the characters being searched for were `\n`, `-` and `.`, which are all common delimiters in different scenarios. This emphasizes the need for regex tools to facilitate easy file parsing. **TODO.MID: The main difference between the Specific Character Must Match group and the multiple characters must match is that the first can subsume the second. I think it's important to defend this and expose that we understand that our simplification of clusters to their shortest representative combined with the behavioral clustering technique will drastically over-emphasize similarities revolving around single characters, so instead of listing this as a major group, or a major finding, we may need to admit that it is an emphasis that our technique introduces artificially. I think that in practice, the patterns being searched for usually revolve around more than just a single character, although maybe not a lot more than that. I think it is more telling that despite our technique's tendency to over-cluster things around a single character, there was still an entire category that required two or more characters, and so those specific pairs or triples or whatever in that cluster are extremely key character combinations. Let's see a few of those that seem to make sense:** ` {2,}` **TODO.MID: :two or more spaces,** `@[a-z]+` **TODO.MID: :a lowercase twitter handle or similar,** `\\$[()]` **TODO.MID: :a dollar followed by either paren,** `v[0-9]+.*` **TODO.MID: :the letter 'v' followed by some numbers, perhaps a version number.**

TODO.MID: In a completely unrelated train of thought, I have been considering how regexes are one major way, and perhaps the only practical way in the world of computer science to handle the invisible characters like ASCII 0-32. That's 33 characters like 'backspace' and 'end of medium' and 'vertical tab' that can appear in text and cause problems. I've heard them called 'Gremlins'. So what do people do? They write regexes to clean those out.

5.2 Opportunities For Future Work

Based on our findings, there are many opportunities for future work.

5.2.1 Context-Specific Regex Support

In some environments, such as command line or text editor, regexes are used extensively (Section 4.1), but these regular expressions do not persist. Thus, using a repository analysis for feature usage only illustrates part of how regexes are used in practice. Exploring how the feature usage differs between environments would help inform tool developers about how to best support regex usage in context, and is left for future work.

5.2.2 Refactoring Regexes

The survey showed that users want readability and find the lack of readable regexes to be a major pain point. This provides an opportunity to introduce refactoring transformations to enhance readability. As one opportunity, certain character classes that are logically equivalent can be expressed differently, for example, `\d` ≡ `[0123456789]` ≡ `[0-9]`. While `\d` is more succinct, `[0-9]` may be easier to read, so a refactoring for *default to custom character classes* could be introduced. Human studies are needed to evaluate the readability of various regex features in order to define and support appropriate regex refactorings for readability.

Another avenue of refactoring could be for performance. Various implementations of regex libraries may perform more efficiently with some features than others. An evaluation of regex feature implementation speeds would facilitate semantic transformations based on performance, similar to performance refactorings for LabVIEW [10, 11].

5.2.3 Developer Awareness of Best Practices

One category of 10 clusters in the top 100 contained regex patterns, *Content of Brackets and Parenthesis*, parses the contents of angle brackets. As the contents of angle brackets are usually unconstrained, regexes are a poor replacement for XML or HTML parsers. More research is needed into how regex users discover best practices and how aware they are of how regexes should and should not be used.

5.2.4 Library Support for Developers

Within standard programming languages, regular expressions libraries are very common, yet there are differences between languages in the features that they support. For example, Java supports possessive quantifiers like `'ab*+c'` (here the '+' is modifying the '*' to make it possessive) whereas Python does not. Differences among programming language implementations was identified as a pain point for using regular expressions by 17% of the survey participants. This provides an opportunity for future work that makes such differences between languages explicit or for new libraries that translate between regex utilizations in various languages.

5.2.5 Automated Regex Repair

Regular expression errors are common and have produced thousands of bug reports [30]. This provides an opportunity to introduce automated repair techniques to fix. Recent approaches to automated program repair rely on mutation operators to make small changes to source code and then re-run the test suite (e.g., [19, 34]). In regular expressions, it is likely that the broken regex is close, semantically, to the desired regex. Syntax changes can lead to big changes in semantics, so we hypothesize that using the semantic clusters identified in Section 4.4 to identify potential repair candidates could efficiently and effectively converge on a repair candidate.

6. THREATS TO VALIDITY

The threats to validity of this work stem primarily from the reliability of measures, instrumentation, selection bias, hypothesis guessing, and the representativeness of participants and projects.

Reliability of Measures: The validity of our survey results is dependent on the clarity of the questions. The authors went through several iterations of the survey and included examples for all the regex feature descriptions to improve understandability.

The similarity measure between regexes used in the cluster algorithm is computed empirically rather than analytically, and the more Rex-generated strings used to compute the similarity measure, the more likely it is to be accurate. Our experiments used 400 strings to balance performance and precision, but a higher number could lead to more cohesive clusters. However, regex patters that use any feature not supported by Rex were omitted from the cluster analysis.

Instrumentation: Regular expression patterns were clustered using strings generated by the Rex tool. We assume that the strings generated by Rex are reasonably diverse to help characterize the regular expression behavior. To mitigate this threat, Rex generated 400 strings per regular expression and we inspected strings randomly to ensure diversity.

Implementation errors are a risk for research involving repository analysis. To combat this, we have tested our code and made the repository publicly available[11].

Selection: We mined only 3,898 Python projects from GitHub, which is small in comparison with the over 100,000 available Python projects. The projects were mined using the GitHub API which sorts the projects by creation date. By using the API, the goal was to reduce any sampling bias introduced by the researchers.

We also did not scrape all commits in every project for regular expression utilizations, rather, we grabbed each project every 20 commits. It is possible that in between the scanned commits, a regular expression utilization was added and then removed, leading to fewer utilizations in our final data set.

Hypothesis guessing: The lead author was an employee of Dwolla when the survey was run, so participant behavior may have been affected by their relationship with the author. To combat this, the beginning of the survey stated, "As this is for research, please answer as accurately as possible, not guessing what is the desired answer and providing that."

Interaction of Selection and Treatment: Our survey participants were software developers from a small startup company. However, the participants may not be representative of all developers. Given that the average participant has nine years of development experience, their responses likely pull from a variety of experiences with regular expression usage.

Interaction of Setting and Treatment: We only explore regular expressions in Python projects so these results

[11] `https://github.com/softwarekitty/tour_de_source`

many be coupled with the activities performed using Python and not generalize to other languages. The regex usage context reported by survey participants, however, includes information on regex usage in a variety of settings and languages. Future work will replicate this study in other languages and the survey with a larger group of participants.

7. CONCLUSION

In this work, we have explored the contexts in which regular expressions are used as well as the features and behavioral similarities of regexes found in open source Python projects. In a survey of 18 professional developers, we find that 50% compose regular expressions at least weekly. The most common purposes are locating content within a file or capturing parts of strings. The most difficult parts about working with regular expressions were reported to be composing and reading them. In a study of regular expression usage in nearly 4,000 Python projects, we find that over 42% of projects contain a regular expression. We present an approach to measure behavioral similarity between regexes by generating strings that match one regex and pairwise testing the remaining regexes against it. This similarity measure is used to form cross-project behavioral clusters. In the top 100 largest clusters, we find that capturing the contents of brackets, searching for delimiter characters and matching alternative values were common behaviors. These results have implications for tool designers and for future research aimed at better supporting developers in using regular expressions.

Acknowledgment

Special thanks to the Dwolla developers for their survey participation. This work is supported in part by NSF SHF-1218265, NSF SHF-EAGER-1446932, and the Harpole-Pentair endowment at Iowa State University.

8. REFERENCES

[1] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending sparql with regular expression patterns (for querying rdf). *Web Semant.*, 7(2):57–73, Apr. 2009.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, Aug. 2013.

[3] A. Arslan. Multiple sequence alignment containing a sequence of regular expressions. In *Computational Intelligence in Bioinformatics and Computational Biology, 2005. CIBCB '05. Proceedings of the 2005 IEEE Symposium on*, pages 1–7, Nov 2005.

[4] R. Babbar and N. Singh. Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data*, AND '10, pages 43–50, New York, NY, USA, 2010. ACM.

[5] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, Nov. 1996.

[6] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf. Regviz: Visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 504–507, New York, NY, USA, 2014. ACM.

[7] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[8] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: The case of smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 23–32, New York, NY, USA, 2011. ACM.

[9] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How (and why) developers use the dynamic features of programming languages: The case of smalltalk. *Empirical Software Engineering*, 18(6):1156–1194, Dec. 2013.

[10] C. Chambers and C. Scaffidi. Smell-driven performance analysis for end-user programmers. In *Proc. of VLH/CC '13*, pages 159–166, 2013.

[11] C. Chambers and C. Scaffidi. Impact and utility of smell-driven performance tuning for end-user programmers. *Journal of Visual Languages & Computing*, 28:176–194, 2015. to appear.

[12] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91, New York, NY, USA, 2014. ACM.

[13] R. Dattero and S. D. Galup. Programming languages and gender. *Commun. ACM*, 47(1):99–102, Jan. 2004.

[14] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM.

[15] S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *Int. J. Softw. Tools Technol. Transf.*, 16(6):727–751, Nov. 2014.

[16] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.

[17] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[18] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and

context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, Feb. 2013.

[19] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[20] J. Lee, M.-D. Pham, J. Lee, W.-S. Han, H. Cho, H. Yu, and J.-H. Lee. Processing sparql queries with regular expressions in rdf databases. In *Proceedings of the ACM Fourth International Workshop on Data and Text Mining in Biomedical Informatics*, DTMBIO '10, pages 23–30, New York, NY, USA, 2010. ACM.

[21] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 21–30, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[22] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[23] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[24] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 1–18, New York, NY, USA, 2013. ACM.

[25] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[26] The Bro Network Security Monitor. https://www.bro.org/, May 2015.

[27] C. Parnin, C. Bird, and E. Murphy-Hill. Adoption and use of java generics. *Empirical Softw. Engg.*, 18(6):1047–1089, Dec. 2013.

[28] RE2. https://github.com/google/re2, May 2015.

[29] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[30] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.

[31] N. Tillmann, J. de Halleux, and T. Xie. Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 385–396, New York, NY, USA, 2014. ACM.

[32] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA, 2014. ACM.

[33] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, Washington, DC, USA, 2010. IEEE Computer Society.

[34] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM Research Highlight*, 53(5):109–116, May 2010.

[35] A. S. Yeole and B. B. Meshram. Analysis of different technique for detection of sql injection. In *Proceedings of the International Conference &#38; Workshop on Emerging Trends in Technology*, ICWET '11, pages 963–966, New York, NY, USA, 2011. ACM.