Exploring Regular Expression Feature Usage in Practice and the Impact on Tool Design

Carl Chapman and Kathryn T. Stolee
Department of Computer Science
Iowa State University
{carl1978, kstolee}@iastate.edu

***Abstract*—Regular expressions are used frequently in programming languages for form validation, ad-hoc file searches, and simple parsing. Due to the popularity and pervasive use of regular expressions, researchers have created tools to support their creation, validation, and use. Each tool has made design decisions about which regular expression features to support, and these decisions impact the usefulness of the tools and their power. Yet, these decisions are often made with little information as there does not exist an empirical study of regular expression feature usage to inform these design decisions.**

In this paper, we explore regular expression feature usage, focusing on how often features are used and the diversity of regular expressions from syntactic and semantic perspectives. To do this, we analyzed 3,898 open source Python projects from GitHub. We also map the most common features used in regular expressions to those features supported by four common regex engines from industry and academia, brics, Hampi, Re2, and Rex. Our results indicate that the most commonly used regular expression features are also supported by popular research tools and that programmers frequently reinvent the wheel by writing identical or nearly identical regular expressions in different ways. We concluded by discussing the implications of omitting certain features in a tool's design and out like several directions of future work.

I. Introduction

Regular expressions (regexes) are an abstraction of keyword search that enables the identification of text using a pattern instead of an exact keyword. With regexes, there is a saying: 'now you have two problems'. A skilled programmer can quickly solve problems such as form validation and parsing text using regular expressions. Regular expression languages also enable a valuable text search/string specification technique used frequently within text editors (e.g., emacs), command line tools (e.g., grep, sed) and IDEs (e.g., the search feature in the Eclipse IDE). Although regexes are powerful and versatile, they can be hard to understand, maintain, and debug, resulting in tens of thousands of bug reports [16].

Due in part to their pervasive use across programming languages and how susceptible regexes are to error, many researchers and practitioners have developed tools to support more robust creation [16] or to allow visual debugging [5]. To remove the human in the loop, other research has focused on learning regular expressions from text [4], [11]. Beyond supporting regular expression usage, the applications of regular expressions in research include test case generation [2], [8], [9], [17], solvers for string constraints [10], [18], and as queries in a data mining framework [6]. Regexes are also employed in critical missions like mysql injection prevention [20] and network intrusion detection [14], or in more diverse applications like DNA sequencing alignment [3].

In writing tools to support regular expressions, tool designers make decisions about which features to support and which not to support. These decisions are sometimes made casually and may be dependent on the regular expressions the designers happen to have experience with, the designers have seen in the wild, or the complexity of the implementation. The goal of this work is to bring more context and information about regular expression feature usage so these design decisions can be better informed.

In fact, this paper emerges out a need to understand which features can be reasonably included in or excluded from a tool that supports regular expressions. In the absence of empirical research into how regular expressions are used in practice, this work emerged.

In this paper, to motivate the study of regular expressions in general, we explore how regular expressions are used in practice. For example, we measure how frequently regular expressions appear in projects, and after doing a feature analysis (e.g., kleene star, character classes, and capture groups are all features), we further measure how often such features appear in regular expressions and in projects. Then, we compare the features to those supported by four common regex support tools, brics [13], hampi [10], Rex [19], and RE2 [15]. We then explore the features not supported by these tools and, using a semantic analysis to cluster similar regular expressions, we explore the impact of omitting those features. Our results indicate that these tools support all of the top six most common features and that some of the omitted features, such as the lazy quantifier, are used in over 35% of projects containing regular expressions.

The contributions of this work are:

- An empirical analysis of the usage of regular expressions in 3,898 open-source Python projects
- A mapping of which features are omitted from common regular expression tools and the impact of ignoring those features
- A discussion on the semantic similarity of regular expressions in practice and identification of opportunities for future work in supporting programmers in writing regular expressions.

The rest of the paper is organized as follows. Section II motivates this work by discussing research in supporting programmers in the use, creation, and validation of regular expressions. Section III presents the research questions and study setup for exploring regular expressions in the wild. Results are in Section IV followed by a discussion in Section V and a conclusion in Section VII.

II. Background and Related Work

One common misconception is that all regular expression languages are *regular languages* which can be represented using deterministic finite automata (DFA), and so they are easy to model, easy to describe formally and execute in O(n) time. In fact, many regular expression matching engines run in exponential time in order to support useful features such as lazy quantifiers, capturing groups, look-aheads and back-references [1]. In a recent regular expression library, the RE2 projext [15], Russ Cox aimed to use DFAs as much as possible (maximizing speed) while supporting as many useful features as possible.

Within standard programming languages, such as Java or Python, there are even some differences in support. For example, **TODO: describe a difference between regex support in two standard languages**

Since regular expression languages vary somewhat in their syntax and feature set, researchers and tool designers have typically had to pick what features to include or exclude. Thus, researchers and tool designers face a difficult design decision: supporting advanced features is always more expensive, taking more time and potentially making the tool or research project too complex and cumbersome to execute well. A selection of only the simplest of regex features is common in research papers and automata libraries, but this limits the applicability/relevance of that work in the real world.

In this work, we perform a feature analysis on regular expressions used in the wild and compare that set to the features supported by four popular regular expression tools. Research tools like Hampi [10], and Rex [19], and commercial tools like brics [13] and RE2 [15], all use regular expressions for various task. Hampi was developed in academia and uses regular expressions as a specification language for a strong constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in several applications, such as test case generation [2], [17]. 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 for **TODO: describe it briefly**. While there are many regular expression tools available, in this work, we focus on the features support for these four tools, which offer diversity across developers (i.e., Microsoft, Google, open source, and academia) and across applications. Further, as the focus of this work is on tool designers and we wanted to perform a feature analysis, these four tools and their features are well-documented, allowing for easy comparison.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterms [12] and bug characterizations [7]. 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 [6], but have not been the focus of the mining activities.

III. 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 as described in Section III-B. Throughout the rest of this paper, we employ the following terminology:

Utilization: A *utilization* occurs whenever a developer uses a regex engine in a project. 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 called from the `re` module is `re.compile` does **TODO: describe this module**, the pattern observed (i.e., a regex string) is `'(0 |-?[1-9][0-9]*)$'` represents strings with **TODO: briefly describe the regex**, and the flag allows string matching over multiple lines. Thought of another way, a regular expression utilization is one single invocation of the `re` library in a project.

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

Fig. 1. example of one regex utilization

Pattern: A *pattern* is an ordered series of regular expression language feature tokens that can be used to find match start and end indices within an input string. Notice that because the vast majority of regex features are shared across most all-purpose languages, a Python pattern will (almost always) behave the same when used in Java, C#, Javascript, Ruby, etc, whereas a utilization is not universal in the same way (would not compile in other languages).

In this work, we primarily focus on patterns since these represent **TODO: explain why patterns**.

A. Research Questions

Our overall research goal is to understand how regular expressions and regular expression features are used in practice. We aim to answer the following research questions:

RQ1: How is the `re` module used in Python projects?

To address this research question, we measure how often any calls are made to the `re` module per file and per project in Python projects.

Furthermore, we measure the frequency of usage for calls to the 8 functions of the `re` module (`re.compile`, `re.search`, `re.match`, `re.split`, `re.findall`, `re.finditer`, `re.sub` and `re.subn`) in Python projects scraped from GitHub.

We also measure usage of the 8 flags (`re.DEFAULT`, `re.IGNORECASE`, `re.LOCALE`, `re.MULTILINE`, `re.DOTALL`, `re.UNICODE`, `re.VERBOSE` and `re.DEBUG`) of the `re` module.

Further, to provide context as to the overlap among regular expression strings used in Python, we explore the most common regex patterns across all utlizations.

RQ2: 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 III-B with examples.

To measure feature usage, we parse Python regular expression patterns using Bart Kiers' PCRE parser**TODO: add footnote or citation**, as described in Section III-B. We then count the number of usages of each feature per project, per file and as a percent of all distinct regular expression patterns.

RQ3: What is the impact of *not* supporting various regular expression features on tool designers and users?

To address, this question, we use semantic analysis to illustrate the impact of missing features on a tool's applicability by identifying what each feature (or group of features) is commonly used for.

At a high level, our semantic analysis clusters regular expressions by their behavioral similarity. Behavioral similarity is determined by a pairwise comparison among all patterns. Within each pair, a set of strings is generated for each regular expression and then tested against the other. The average percentage of matching regular expressions creates the similarity level.

TODO: Finish this example For example, consider the following two regular expressions, denoted A and B for reference.

```
state regex A
state regex B
```

For each regex, the following strings are generated:

A	B
s1	s2
s3	s4

Each string in the A column matches regex A, and each string in the B column matches regex B. When testing the strings in B against regex A, X/5 = 0.Y% match. When testing the strings in the A column against regex B, Z/5 = 0.W% match. Thus, the similarity between these two regular expressions is 0.U%.

Semantic analysis is accomplished by first establishing a similarity matrix between regexes. These strings are generated by Rex. We chose Rex to build matching strings because it supports the most features of any String-generation tool. To build the similarity matrix, we generated X strings per regular expression in an effort to balance the precision of the similarity metric (i.e., more strings lead to higher precision) with the speed of our analysis tool (i.e., more strings lead to longer runtimes). **TODO: what is X?**

Then clusters of regexes with similar behavior are discovered using Markov Clustering[1]. These clusters are used to see how programmers implement regular expressions that match similar strings and interpret what a feature is used for. We chose the mcl clustering tool because it offers a fast and tunable way to cluster items by similarity (without knowing the number of clusters in advance).

Next, we describe in greater detail how the corpus of regex patterns was built, how features were analyzed, and how the clustering was performed.

B. Building the Corpus

Github is a popular project hosting site containing over 100,000 Python projects. We used the GitHub API to page through all repositories, cloning projects that contain Python code. **TODO: How were these repositories sorted? by popularity? randomly?**

For each project, we used Astroid[2] to build the AST of each Python file and find *utilizations* of Python's `re` module. This ensured that all utlizations of the `re` module were captured for analysis.

Using git, each project was scanned at 20 evenly-spaced commits (or all commits if there were less than 20) in its history. Within one project, we define a duplicate *utilization* as a *utilization* having the same function, pattern and flags within the same file (same relative path). We ignored duplicate *utilizations* across project versions to protect against overcounting the same *utilization* as we rewind the project through its history. We observed and recorded 53,894 non-duplicate regex *utilizations* in 3,898 projects.

C. Extracting Patterns

As the focus of this study is regex features, our analysis focuses on the patterns found. Thus, we ignore the 12.7% of *utilizations* using flags that 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), or because of other unknown parsing failures.

The remaining 80.8% (43,525) *utilizations* were collapsed into 14,113 distinct pattern strings. The resulting set of patten strings were parsed using an ANTLR-based, open source PCRE parser released by Bart Kiers[3]. This parser was unable to support 0.5% (76) of the patterns due to unsupported unicode characters. Another 0.2% (27) of the patterns used regex features that we have chosen to exclude in this study[4]. **TODO: What? Which features did we choose to exclude? I'm lost** The 13,912 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*.

D. Analyzing Features

Given the corpus of regular expression patterns, we identified features by **TODO: describe this process**.

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

[2]https://bitbucket.org/logilab/astroid

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

[4]www.details.#thistopic

Once the feature set was established, we mapped the features from the corpus to those features supported by the four regular expression engines described in Section II: brics, Hampi, RE2, and Rex. Table IV shows this mapping. The first column, *rank*, lists the features in order of popularity, determined by the percentage of projects in which they appear. The next column, *code*, gives a succinct reference string for the feature followed by a description and example usage from the corpus. The mappings for each regex tool to the features are shown in the next four columns followed by usage statistics for the number and percent of patterns that the feature appears in, the number and percent of total files, and the number and percent of total projects.

To create the tool mappings, we consulted documentation for each of the selected regular expression engines. For brics, we collected the set of supported features using **TODO: make specific citations to the documents used for the mapping of features to tools for each tool. URLs are OK in this instance**

E. Clustering and Semantic Analysis

TODO: this section needs lots of work Markov clustering 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[5], the best clusters were found using an inflation value of 1.8 and k=83.

Note that the filteredCorpus is of size 9727, and at least one pattern from the fc can be found in 1375 of the original 3900 or whatever. Most patterns do not belong in a cluster (for example a very specific pattern like <title>[^<]*Revision \d+:), so after clustering is done only 2727 patterns are included, and only 999 projects have any of these patterns in them.

Again we used MCL to find clusters that aided a manual search for use cases strongly associated with particular features.

IV. Results

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

A. Frequency of Utilizations in Python Projects

1) Saturation of Projects with Utilizations: Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one utilization. For context about how saturated these projects were with utilizations, we consider how many utilizations were observed per project, how many files the average project scanned contained, how many of those files contained utilizations, and how many utilizations occurred per file in table I.

2) Usage Frequency of re *Module Functions and Flags:* As seen in Figure 2 The 'compile' function encompasses 57.6% of all utilizations, presumably because each usage of those functions could accept the compiled regex as an argument.

[5]www.details.#thistopic

TABLE I. How Saturated are Projects with Utilizations? (RQ1)

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

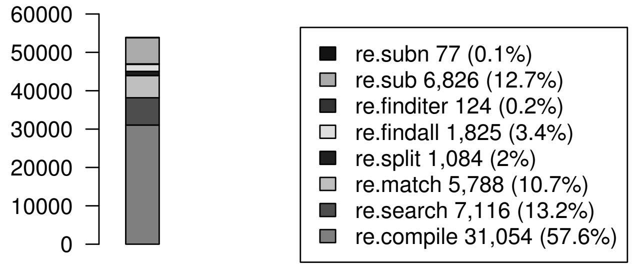

Fig. 2. How often are the 8 re functions used? (RQ1)

When considering flag use, we excluded non-behavioral flags (default and debug), which account for 87.3% of all utilizations.

As shown in figure 3, of all behavioral flags used, ignorecase (43.8%) and multiline (25.8%) were the most frequently used. It is also worth noting that although multiple flags can be combined using a bitwise or, this was never observed.

3) Most Frequently Observed Patterns: Table II (description of table contents)

B. Pattern Characteristics

Table III (description of table contents)

C. Frequency of Feature Usage

Literal tokens were found in 97.5% of patterns, and accounted for 70.6% of all tokens. We consider literal tokens to be ubiquitous in all utilizations, and necessary for any regex related tool, and so exclude them from the rest of the feature analysis. In table IV, we display a large body of information about feature usage and relate it to four major regex related projects.

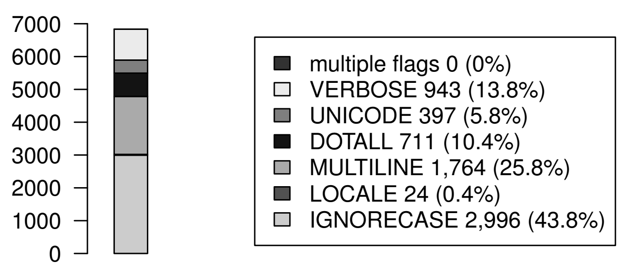

Fig. 3. Which behavioral flags are used? (RQ1)

TABLE IV. HOW FREQUENTLY DO FEATURES APPEAR IN PROJECTS, AND WHICH FEATURES ARE SUPPORTED BY FOUR MAJOR REGEX PROJECTS? (RQ2)

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nFiles	%files	nProjects	% projects
1	ADD	one-or-more repetition	z+	●	●	●	●	6,122	44	9,330	50.3	1,209	73.5
2	CG	a capture group	(caught)	●	●	●	●	7,248	52.1	9,759	52.6	1,197	72.8
3	KLE	zero-or-more repetition	.*	●	●	●	●	6,104	43.9	8,323	44.9	1,100	66.9
4	CCC	custom character class	[aeiou]	●	●	●	●	4,581	32.9	7,808	42.1	1,027	62.4
5	ANY	any non-newline char	.	●	●	●	●	4,708	33.8	6,394	34.5	1,006	61.2
6	RNG	chars within a range	[a-z]	●	●	●	●	2,698	19.4	5,196	28	849	51.6
7	STR	start-of-line	^	○	●	●	●	3,660	26.3	5,622	30.3	847	51.5
8	END	end-of-line	$	○	●	●	●	3,258	23.4	5,549	29.9	828	50.3
9	NCCC	negated CCC	[^qwxf]	●	●	●	●	1,970	14.2	4,027	21.7	777	47.2
10	WSP	\t \n \r \b \f or space	\s	○	●	●	●	2,908	20.9	4,812	25.9	764	46.4
11	OR	logical or	a|b	●	●	●	●	2,161	15.5	4,039	21.8	711	43.2
12	DEC	any of: 0123456789	\d	○	●	●	●	2,385	17.1	4,366	23.5	694	42.2
13	WRD	[a-zA-Z0-9_]	\w	○	●	●	●	1,457	10.5	3,004	16.2	652	39.6
14	QST	zero-or-one repetition	z?	●	●	●	●	1,922	13.8	3,821	20.6	647	39.3
15	LZY	as few reps as possible	z+?	○	●	○	●	1,318	9.5	2,291	12.4	606	36.8
16	NCG	group without capturing	a(?:b)c	○	●	○	●	813	5.8	1,748	9.4	404	24.6
17	PNG	named capture group	(?P<name>x)	○	●	○	●	934	6.7	1,517	8.2	354	21.5
18	SNG	exactly n repetition	z{8}	●	●	●	●	623	4.5	1,359	7.3	340	20.7
19	NWSP	any non-whitespace	\S	○	●	●	●	490	3.5	788	4.2	271	16.5
20	DBB	$n \leq x \leq m$ repetition	z{3,8}	●	●	●	●	384	2.8	692	3.7	242	14.7
21	NLKA	sequence doesn't follow	a(?!yz)	○	○	●	○	137	1	503	2.7	184	11.2
22	NWRD	non-word chars	\W	○	●	●	●	97	0.7	315	1.7	169	10.3
23	LWB	at least n repetition	z{15,}	●	●	●	●	97	0.7	337	1.8	167	10.2
24	WNW	word/non-word boundary	\b	○	○	○	●	248	1.8	438	2.4	166	10.1
25	LKA	matching sequence follows	a(?=bc)	○	○	○	○	114	0.8	360	1.9	159	9.7
26	OPT	options wrapper	(?i)CasE	○	●	○	●	232	1.7	378	2	154	9.4
27	NLKB	sequence doesn't precede	(?<!x)yz	○	○	○	○	102	0.7	321	1.7	139	8.4
28	LKB	matching sequence precedes	(?<=a)bc	○	○	○	○	82	0.6	262	1.4	120	7.3
29	ENDZ	absolute end of string	\Z	○	○	○	●	91	0.7	154	0.8	94	5.7
30	BKR	match the i^{th} CG	\1	○	○	○	○	60	0.4	129	0.7	84	5.1
31	NDEC	any non-decimal	\D	○	●	●	●	36	0.3	92	0.5	58	3.5
32	BKRN	references NCG	\g<name>	○	●	○	○	17	0.1	44	0.2	28	1.7
33	VWSP	matches U+000B	\v	○	○	●	●	13	0.1	16	0.1	15	0.9
34	NWNW	negated WNW	\B	○	○	○	●	4	0	11	0.1	11	0.7

D. Features Usage Analysis

1) Behavioral Clustering Results Overview: Our behavioral clustering technique found 952 clusters over 2727 patterns, with at least one cluster present in 999 of the 9727 projects that were compatible with Rex.

TODO: Need to know why MCL is behaving like this

Table V provides an example of a smaller behavioral cluster, representing 13 patterns, with at least one pattern from this cluster present in 100 different projects.

On first glance this cluster may seem to revolve around the '`\s*`' parts of these patterns, but actually this cluster was formed because each of these patterns has a comma literal, and other details did not interfere with matching the Rex-generated strings with commas in them.

It is not a coincidence that the smallest pattern in this cluster gives the best idea of what all the patterns within it have in common (the smallest pattern is just the single comma character, at index 1). All of the clusters we found follow this trend: the shortest pattern describes the rest of the pattern's behavior very well. In table VI, I show the top 10 clusters, ranked by the number of projects they appear in, using the shortest pattern from the cluster as an example. The cluster in Table V appears in the seventh row of Table VI.

2) Feature Groups Overview: Instead of analyzing every feature independently, we chose small groups of conceptually related features. For each of these groups, we selected all

TABLE II. TOP 10 PATTERNS BY NPROJECTS (RQ1)

<table>
<tr><th>pattern</th><th>nProjects</th></tr>
<tr><td>'\\s+'</td><td>181</td></tr>
<tr><td>'\\s'</td><td>78</td></tr>
<tr><td>'\\d+'</td><td>70</td></tr>
<tr><td>'[\\x80-\\xff]'</td><td>69</td></tr>
<tr><td>'\nmd5_data = {\n([^}]+)}'</td><td>69</td></tr>
<tr><td>'\\\\(.)'</td><td>67</td></tr>
<tr><td>'([\\\\"]|[^\\ -~])'</td><td>66</td></tr>
<tr><td>'(-?(?:0|[1-9]\\d*))(\\.\\d+)?([eE][-+]?\\d+)?'</td><td>61</td></tr>
<tr><td>'[^]]+?\\] +([0-9.]+): (\\w+) <-(\\w+)'</td><td>60</td></tr>
<tr><td>'.*rlen=([0-9]+)'</td><td>57</td></tr>
</table>

TABLE III. PATTERN CHARACTERISTICS (RQ1)

source	Q1	Avg	Med	Q3	Max
pattern weight	1	2	1	2	181
token count	8	24	14	24	31,349
distinct features	3	5	5	7	20
pattern length	13	37	22	38	43,255

clusters that had at least one of the features in at least one pattern within the cluster to form a 'feature group'-focused cluster set.

Table VII shows the total number of projects that contain at least one pattern from at least one cluster in the cluster set, and some selected clusters represented by the shortest string in the cluster. These clusters were selected not because of being within the largest number of projects, but because they illustrate some interesting usage of a feature that will be explored in detail later.

(note that for the ANY group, all but two of the top 30 clusters used '.*', but that .* as a pattern alone only appeared in 23 projects)

TABLE V. AN EXAMPLE CLUSTER (RQ3)

index	pattern	nProjects
1	`\s*,\s*'	54
2	`,'	30
3	`\s*,'	16
4	` ,\s*'	13
5	` *, *'	12
6	`,\S'	5
7	`,.*$'	3
8	`(\S+)\s*,\s*'	2
9	`,+'	1
10	`,\ ?'	1
10	`,\s*(\S)'	1
11	`\s*(,)\s*'	1
12	`\s*\,\s*'	1

TABLE VI. TOP 10 CLUSTERS BY NPROJECTS (RQ3)

rank	nProjects	nPatterns	example
1	227	31	`\s'
2	208	83	`\W'
3	193	87	`\d'
4	138	44	`[^!-~]'
5	122	54	`[a-zA-Z]'
6	114	31	`\\'
7	110	49	`\w'
8	100	13	`,'
9	91	32	`:'
10	78	14	`^\d+$'

tell them how the cluster groups in the first part are all drawn from a subset of the corpus limited by what Rex can support, whereas the cluster groups in the second part are all drawn from the complete corpus, but are not guaranteed to have behavioral similarity like in the first part.

V. DISCUSSION

A. *Frequency of* re *Module Usage*

...only 11.2% of the files observed had at least one regex usage. This indicates that regex usage may usually be concentrated in just a few files.

From table I, we see that on average each project had 2 files containing any regex usage, out of an average of 6 files. Each of the files that did have a regex usage had an average of 1 regex usages. Because we scanned 3,898 projects, we would expect to have seen 23,388 regex usages, which is lower than the actual 53,894 usages observed.

1) Saturation: Although 42.2% of the projects observed had at least one regex usage, only 11.2% of the files observed had at least one regex usage.

B. *DBB subsumes repetition, CCC subsumes character classes*

The DBB feature subsumes all other repetition features. Consider the equivalences shown in Figure 4

```
x* = x{0,MAX}     x? = x{0,1}      x{5,} = x{5,MAX}

x+ = x{1,MAX}     x{3} = x{3,3}    x{7,9} = x{7,9}
```

Fig. 4. How DBB Subsumes All Other Repetition Features

Similarly, the CCC feature subsumes NCCC, RNG, ANY, DEC, NDEC, WSP,NWSP, WRD, NWRD because each of these features is equivalent to a set of characters. We provide an example of how CCC subsumes DEC, WSP and WRD in Figure 5 (other equivalences not shown for brevity).

1) character classes are important: In replacing keyword search with an abstracted search, one of the most fundamental abstractions is that one element of a sequence can be one of several characters. This abstraction is realized in custom character classes.

TABLE VII. FEATURE GROUPS WITH SELECTED CLUSTER EXAMPLES (RQ3)

index	feature set	nProjects	% of Projects	selected cluster examples (nProjects for that cluster)
1	ADD,KLD,QST	970	97.1	'\W+'(208), '[A-Z]?[:;.A-Z]'(47), ':+'(91), 'https?://'(13)
2	CCC,NCCC,RNG	953	95.4	'[0-9]'(193), '[^!-~]'(122), '[aeiou]'(4), '^[a-f0-9]{40}$'(34)
3	CG	943	94.4	'coding[:=]\s*([-\w.]+)'(48), '<(.*)>'(63), '"(.*)"'(42), '\\(.)'(110s)
4	STR,END	807	80.8	'^\d+$'(78), '^\\s*$'(59), ',.*$'(100), '=.*$'(52), '^(.*)<(.*)>(.*)$'(63),
5	ANY	801	80.2	'\s.*'(277), '(\d+)(.*)'(193), '-.*'(74), '(.)([A-Z])'(47), '<.+>'(63)
6	WSP,NWSP	775	77.6	'\s'(277), '\S'(53), ':\s*'(91), ',\S'(100), '<\\S[^>]*>'(63)
7	OR	759	76.0	'((the|a|an)\\s+)?[0-9]+'(193), '([ ]+_)|(_[ ]+)|([ ]+)'(66), '<.*>|</.*>'(63)
8	DEC,NDEC	622	62.3	'\d'(193), '\D'(65), '\.\d+$'(14), '[^\w\d_]'(208), '(\D)[.]'(61)
9	WRD,NWRD	595	59.6	'\W'(208), '\w'(114), '[a-zA-Z]\w*'(138), '(\w*)=(\w*)'(52), '\\(\W)'(110)
10	DBB,LWB,SNG	459	45.9	'^[0-9]{1,5}$'(78), '\d{2}'(193), '[.]{2,}'(21), '^[0-9A-Za-z._-]{0,100}$'(27)

```
\d = [0123456789]

\s = [ \t\n\r\f\v]

\w = [abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_]
```

Fig. 5. How CCC Subsumes DEC, WSP and WRD

TABLE VIII. TOP 10 PATTERNS IN TOP 3 CLUSTERS (RQ3)

example	example	example
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'
'\s'	'\s'	'\s'

2) default character classes are widely used: The pattern language for Python and most major regex engines supports a few default character classes (and their negations) which we have described as the features ANY, DEC, WSP, WRD, NDEC, NWSP, NWRD. Throughout this analysis it was obvious that these default character classes were widely used. Specifically, \s, \d and \W were the top three behavioral clusters (as shown in Table VI). In Table VIII we show the top 10 patterns from these top three clusters.

One surprising result of our clustering is that behaviorally speaking, the negation of the word class was more heavily used (208 projects) than the word class itself (114 projects). One

The character class of all letters [a-zA-Z] appears so often that it is an excellent candidate for a new a character class.

The largest cluster using custom character classes (cluster N in Table VI whose shortest member is [^ -~]) has 44 patterns which create a more permissive word class. Inspection of the source code of several projects using this pattern indicates that these permissive word classes were typically used when trying to create system-friendly object names, which indicates that there is a demand for a word class that includes more characters (but not dashes, tildes or the first 9 unicode characters).

One obvious character class to consider is hexidecimal characters, and we see the pattern [a-fA-F0-9] appear many times.

For tools that do not support the default character classes, this is a significant obstacle for users trying to test the regexes that they already have.

3) use of repetition:

4) Anchors matter: The endpoint anchor features STR and END are the only way specify that an entire line has to match. Consider the following example, comparing the pattern '^\s*$' (found in 48 projects) to the pattern '\s*' (found in X projects) when looking for lines devoid of content. Without the endpoint anchors, the pattern matches every line, since there are always at least zero whitespace characters on every line. But with the endpoint anchors, only lines that contain nothing but whitespace will match, allowing the user to find all lines that don't have any content.

C. Opportunities for Future Work

1) New library feature for properties of a line:

2) Regexes need refactoring: We see the same features implemented many ways, and we don't know why. It might be that some methods of implementation are more understandable. However, if tool designers do not support that, refactoring may be needed.

example with anchors and .* in the middle which could be replaced by a comma?

3) Using 0-9 instead of
d: Out of the 9727 patterns acceptable to Rex, 1498 contained the range 0-9 within a character class, even though this is exactly equivalent to using \d within a character class (which appeared within a character class only 237 times). Why are users specifying digits using the RNG feature when a default character class already exists? Is it to aid readability, or does this represent an opportunity for refactoring?

4) dot-star at the end - refactoring?: One of the most ubiquitous sub-patterns, `'.*'` appeared in 1937 of the 9727 patterns acceptable by Rex, and appeared within the last four characters in 1161 of the 9727 patterns. Out of the top 30 clusters containing the ANY feature, 26 also had `'.*'` within the last four characters.

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, as illustrated in Figure 6.

```
fruitDiary = "Day1: Apples, Oranges, Apples, Peaches
              Day2: Kumquats, Berries, Pears, Peaches
              Day3: Oranges, Pears, Apples, Pears"
print(len(re.findall('Apples.*',fruitDiary))) # 2
```

Fig. 6. An Example of Using .* to Count Lines Containing 'Apples'

In many cases, this `'.*'` at the end of the string may not actually contribute any new behavior to the pattern, and may in fact be extraneous. Or users may be trying to bypass the whole-string nature of the `re.match` function, without realizing that they could instead use the `re.search` function. Consider the comparison shown in Figure 7

```
found = re.match('.*cde.*', 'abcdefg')   # found=true
found = re.match('cde', 'abcdefg')       # found=false
found = re.search('cde', 'abcdefg')      # found=true
```

Fig. 7. An Example of Using .* to get Search Behavior From The Match Function

Are programmers using the dot-star sub-pattern unnecessarily? More research is needed into this question to find out if these patterns are a candidate for refactoring.

Fun fact: while creating similarity matrix, row 5464 took 2 hours, or almost 1 second per cell avg, only suffering 18 timeouts (1.2 secs). What is this pesky pattern?

We do not assume that Python projects represent a perfect sample of regular expression usage in all environments, but to make the work of collecting data for the paper reasonable, we had to choose one language to focus on (we hope to compare results across languages in future work). Python is an attractive choice because the culture of Python programming makes it seem likely that someone would write the pattern directly in the function, not trying to over-complicate things with some extra Classes or functions. Other attractive choices are Perl (which probably has the most active regex community), javascript and ruby (which may emphasize web tasks like form validation), sql or a general purpose language like java or C#.

VI. Threats to Validity

The threats to validity of this work stem primarily from sampling bias, tool limitations, and language selection.

We mined only 3,898 Python projects from GitHub, which is very small in comparison with the over 100,000 available Python projects on that platform. The projects were mined using the GitHub API which sorts the projects by **TODO: how were they sorted?**. By using the API, the goal was to reduce any sampling bias introduced by the researchers.

We 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.

Our semantic clustering is dependent on Rex.

Generality - we only sample a subset of the total Python projects available, and from just one source.

In extracting patterns for analysis, we omit utilizations that contain flags since flags provide refinements on the functionality of the pattern as it is used in the project. Given that only 12.7% of all utilizations used flags, the impact on the clusters should be minimal.

Regular expression patterns were clustered using strings generated by the Rex tool. This introduces two threats to validity. First, we assume that the strings generated by Rex are reasonably diverse to help characterize the regular expression behavior. To mitigate this threat, Rex generated X strings per regular expression **TODO: what is X?**. Second, since the similarity between two regular expressions was calculated empirically using Rex rather than through analysis, we may have over-estimated the similarly. For this reason, in our clustering algorithm, we require a similarity level of Y so that we do not over-estimate the similarity between regular expressions **TODO: what is Y?**.

The final threat to validity comes from the fact that we only explore regular expressions in Python projects so these results many not generalize to other languages. Future work will replicate this study in other languages and compare the results.

VII. Conclusion

A. *Empirical Analysis of Regex Utilizations*

B. *Frequency of Feature Usage*

conclusion II

1) A Suggested Feature Implementation Priority List: One key consideration for Tool designers is what features are most important to implement in order of priority. We provide a prioritized list of feature groups to implement, based on the frequency of feature usage displayed in Table IV.

1 literals, sequences of tokens
2 CCC (and all subsumed features)
3 CG (without back-references)
4 DBB (and all subsumed features)
5 STR, END
6 OR
7 LZY
8 NCG
9 NLKA, LKA, NLKB, LKB
10 WNW, NWNW

11 ENDZ
12 BKR

C. How Features Are Used In Practice

Acknowledgment

This work is supported in part by NSF SHF-1218265, NSF SHF-EAGER-1446932, and the Harpole-Pentair endowment at Iowa State University.

References

[1] Msdn - matching behavior, 2015. `https://msdn.microsoft.com/en-us/library/0yzc2yb0.aspx`.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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