September 2021

Software Input Pattern and Test Coverage using Computational Linguistics on Structured Data

Yifan Gao

Software Input Pattern and Test Coverage using Computational Linguistics on Structured Data

ABSTRACT

This disclosure describes computational linguistics techniques for software input patterns and test coverage. Structured input data which can have arbitrary and evolving schema, obtained from production software and from testbeds, are tokenized using tree traversal to generate vocabulary, unigram statistics, and bags of words (BoW). BoWs are subjected to statistical analysis to programmatically and intelligently discover software usage patterns in production, to identify test coverage, and to flag gaps in testing.

KEYWORDS

- Software-under-test (SUT)
- Software usage pattern
- Test coverage
- Unstructured data
- Tree traversal
- Computational linguistics
- Natural language processing
- Tokenization
- Bag-of-words (BoW)
- Latent Dirichlet Allocation (LDA)
- Pairwise statistics
- Unigram
- Topic modeling
- n-gram

BACKGROUND

Input and configuration coverage of software by diffing production logs against test logs has been described in the literature [1]. In particular, structured data relating to the inputs and configurations consumed by the software is traversed by treating it as a tree; by emitting single leaves of the tree; and by aggregating, counting, and diffing production leaves against test-environment leaves.

However, by decomposing a structured log entry into leaves, all dimensions are treated as independent, such that questions such as “should foo_endpoint=’e2’ AND bar_mode=’legacy’ be tested together?” remain unanswered (foo_endpoint and bar_mode being inputs or configuration variables). On the other hand, joining fields without due consideration to their semantics (meanings) results in inconsequential (noisy) coverage gaps, especially in a high-dimensional (tens or hundreds of thousands of dimensions) feature space. It is therefore desirable to determine clusters of input/configuration space where multiple dimensions (foo_endpoint, bar_mode, etc.) co-occur at high frequency. Such co-occurrences can indicate dominant usage patterns for the software, and hence point to clusters of the input/configuration space that are well-suited for concentrated testing.

DESCRIPTION

This disclosure describes computational linguistics techniques for software usage pattern and test coverage. Per the techniques, structured input data (which can have arbitrary and evolving schema) is tokenized using tree traversal to programmatically and intelligently discover software usage patterns in production, to identify test coverage, and to flag gaps in testing.

To find software usage patterns and identify test coverage gaps, analysis of structured data with arbitrary and evolving schema is treated as a computational linguistics or natural language processing (NLP) problem. Although the structured data that appears in software configuration and input might seem unsuited to NLP techniques (which work on unstructured data), the following observations hold:

- In automatic test coverage analysis, if the meaning of each field is not taken as a priori known, then the semantics of each field can be discovered automatically by the NLP pipeline, rather than being ad hoc specified by humans.

- For many test-coverage use cases, NLP techniques such as bag-of-words (BoW) are applicable because insights can be obtained even when some topological structure of the input is ignored.
- A structured data entry (e.g., a protobuf or JSON message), organized as a tree, can be transformed into an unordered collection of leaves by performing tree traversal [1]. Such transformation can be regarded as equivalent to tokenization performed in NLP, where a human-language document (or text) is transformed into a relatively unstructured bag-of-words.

NLP concept	Test-coverage equivalent
Document	An instance of multi-dimensional input to a software-under-test logged as a structured data entry, e.g., in protobuf or JSON format.
Term, leaf, or unigram	For finding software usage patterns, term is the atomic (e.g., smallest) unit that is treated as a feature dimension of structured data. For example, a term can be a protobuf field value, metadata about the length or presence/null of a field, etc. Term is synonymous with the “leaf” of [1] and with “unigram.” Although unigram, term, and leaf are used interchangeably, unigram is preferred since it emphasizes atomicity.
Corpus	A collection of documents gathered for a specific environment, e.g., production or testbed, during a specific period.
Topic	A weighted mixture of unigrams that tend to appear together; and whose cohort indicates a specific latent usage pattern of the software.
Stop word (e.g., “a”, “an”, “the”, etc. in English)	A unigram that appears so frequently that its occurrence doesn’t carry much information.
Rare word	A unigram that occurs in isolation with low frequency and doesn’t form meaning patterns.

Fig. 1: A correspondence between NLP concepts and their test-coverage equivalents

Fig. 1 establishes a correspondence between NLP concepts and their test-coverage equivalents, such that NLP can be leveraged for the purposes of test-coverage analysis. The techniques herein leverage the analogies drawn in Fig. 1 to collect data from production and from the testbed, to treat each as a separate corpus, and to perform computational linguistics analysis to evaluate test coverage.

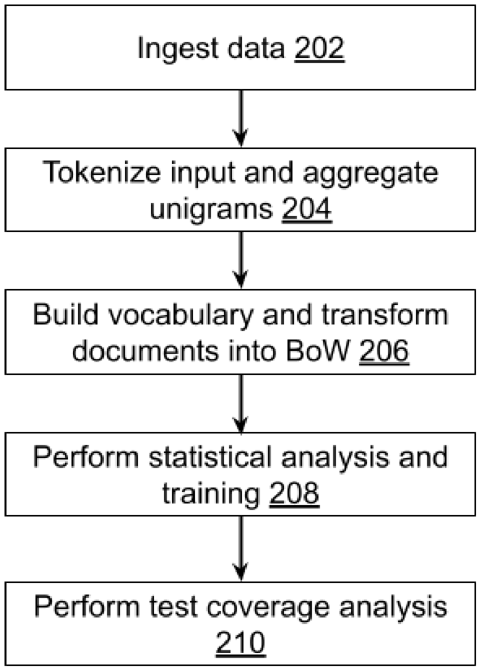
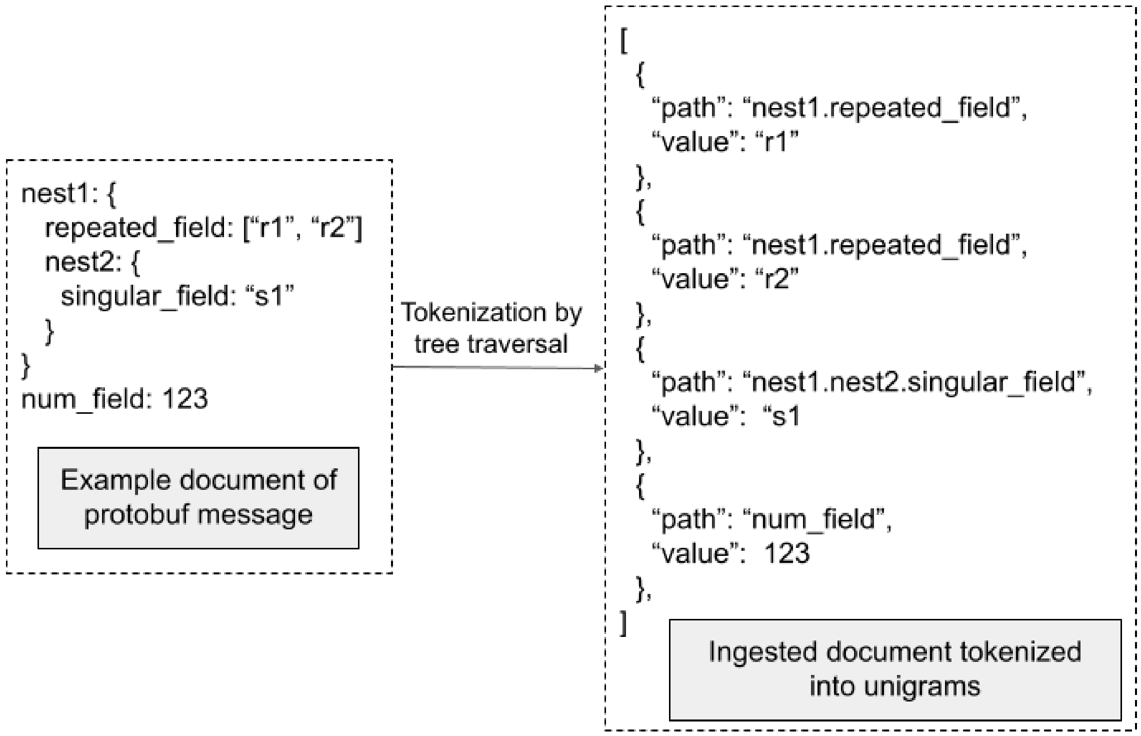

Ingest data 202
Tokenize input and aggregate unigrams 204
Build vocabulary and transform documents into BoW 206
Perform statistical analysis and training 208
Perform test coverage analysis 210

Fig. 2: Test coverage and analysis of structured Data using computational linguistics

Fig. 2 illustrates test coverage and analysis of structured data using computational linguistics, explained in greater detail below. Computation can be carried out by creating data-pipelines using frameworks such as Google Flume or Apache Beam.

Ingest data (202): A structured log of test data, e.g., from the production software or from the testbed, is collected. Each log entry can be a protobuf, JSON, or other structured message, and, as explained before in the NLP↔testing concept-equivalence table, can be considered as equivalent to a document for the purposes of test analysis. Such collection can be performed separately for production and testing environments. The set of collections for an environment (e.g., production, staging, etc.) is called that environment's corpus.

Tokenize input and aggregate unigrams (204): Using tree traversal, the input (which is structured data in the form of a protobuf or JSON message) is tokenized, e.g., mapped into an unordered collection of leaves. In the language of natural language processing, the pipeline maps each document to a collection of unigrams.



```
nest1: {
  repeated_field: ["r1", "r2"]
  nest2: {
    singular_field: "s1"
  }
}
num_field: 123
```

```
[
  {
    "path": "nest1.repeated_field",
    "value": "r1"
  },
  {
    "path": "nest1.repeated_field",
    "value": "r2"
  },
  {
    "path": "nest1.nest2.singular_field",
    "value":  "s1
  },
  {
    "path": "num_field",
    "value":  123
  },
]
```

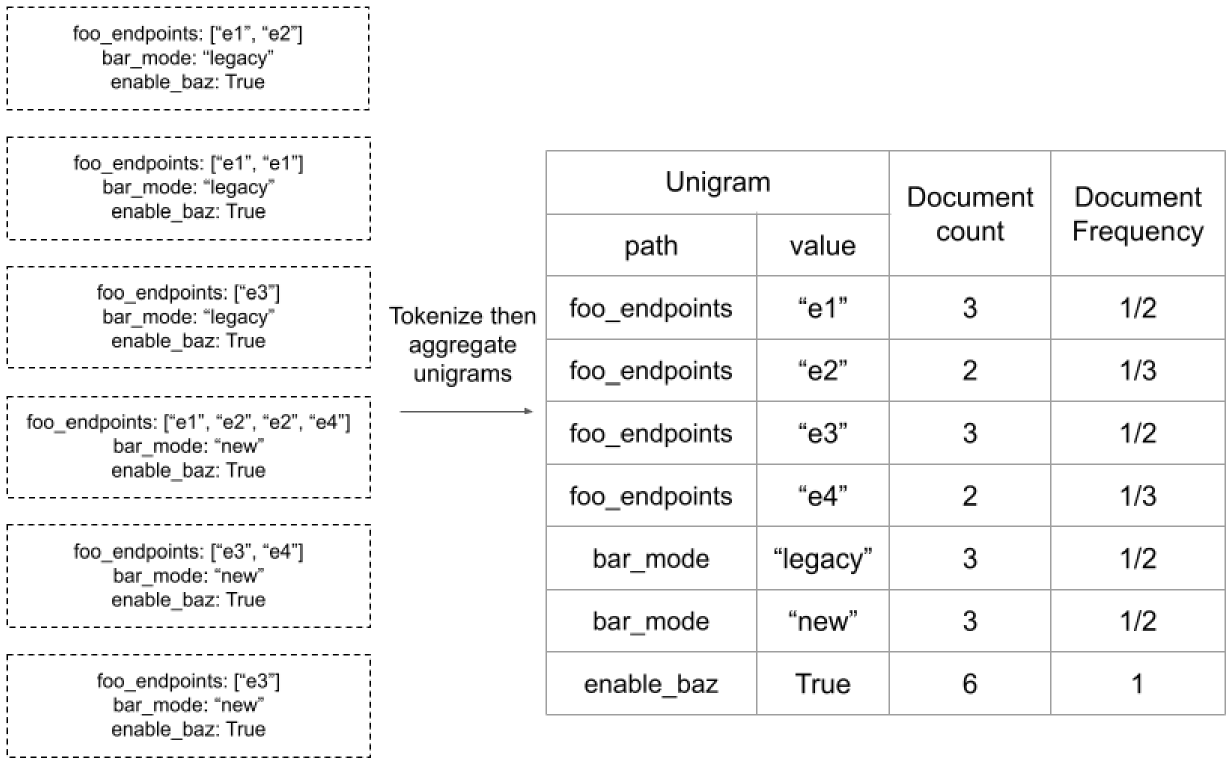
Example document of protobuf message
Tokenization by tree traversal
Ingested document tokenized into unigrams

Fig. 3: Emitting leaves of a tree by traversing it. In the terminology of natural language processing, an ingested document is tokenized into an unordered collection of unigrams

Fig. 3 illustrates an example of tokenization via tree traversal. A tree of structured data (e.g., a protobuf or JSON message or document) is traversed to generate leaves. As illustrated, a path through the tree (e.g., nest1→nest2→singular_field) and the value at the end of the path (s1) are flattened into leaves that comprise (path, value) pairs, e.g., {path:

"nest1.nest2.singular_field"; value: "s1"}. The flattening of a tree into a vector of leaves is known as tokenization. Leaves are also referred to as unigrams.

Using unigrams as aggregation keys, the pipeline performs counting in each corpus to obtain basic statistics such as frequency and term frequency — inverse document frequency (tf-idf). For production data, differential privacy can be used to ensure that the aggregated result is anonymous even if the input includes user data. Statistics thus obtained, with some diffing and postprocessing, can lend insights into the test coverage [1].



```
foo_endpoints: ["e1", "e2"]
bar_mode: "legacy"
enable_baz: True
```

```
foo_endpoints: ["e1", "e1"]
bar_mode: "legacy"
enable_baz: True
```

```
foo_endpoints: ["e3"]
bar_mode: "legacy"
enable_baz: True
```

```
foo_endpoints: ["e1", "e2", "e2", "e4"]
bar_mode: "new"
enable_baz: True
```

```
foo_endpoints: ["e3", "e4"]
bar_mode: "new"
enable_baz: True
```

```
foo_endpoints: ["e3"]
bar_mode: "new"
enable_baz: True
```

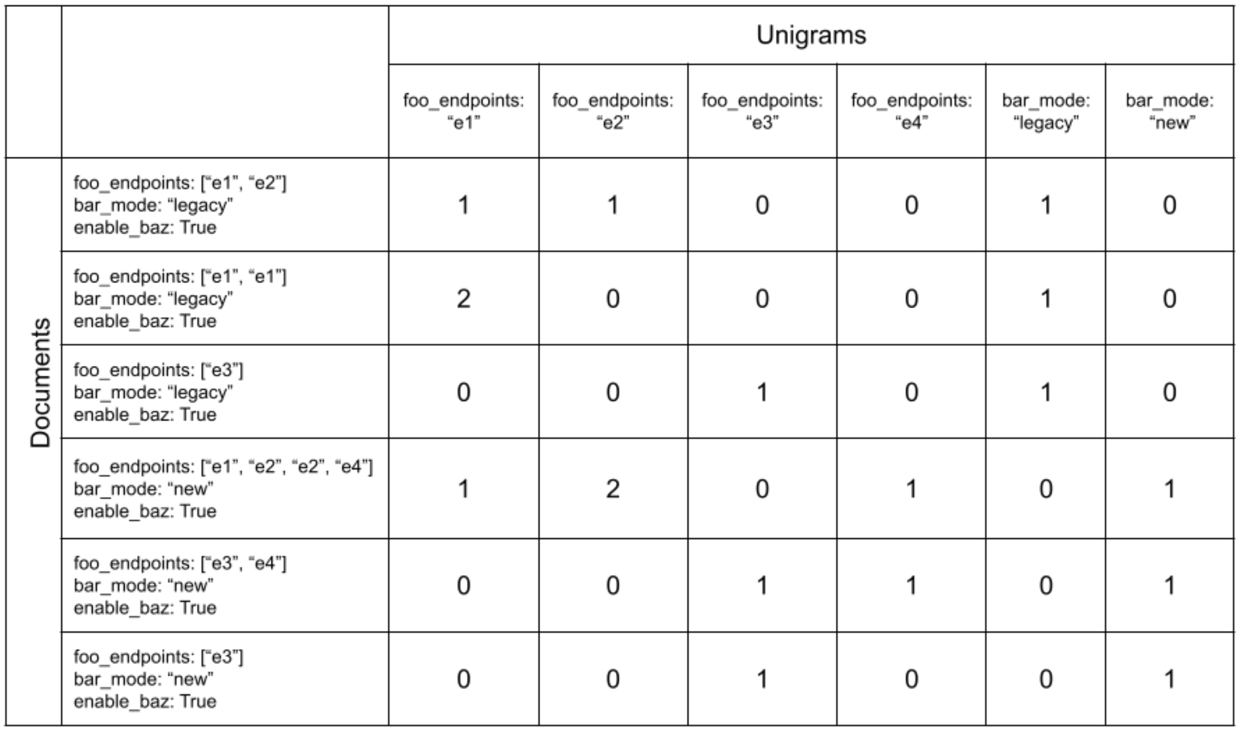
Unigram		Document count	Document Frequency
path	value		
foo_endpoints	"e1"	3	1/2
foo_endpoints	"e2"	2	1/3
foo_endpoints	"e3"	3	1/2
foo_endpoints	"e4"	2	1/3
bar_mode	"legacy"	3	1/2
bar_mode	"new"	3	1/2
enable_baz	True	6	1

Fig. 4: Aggregating unigrams to obtain their statistics

Fig. 4 illustrates aggregating unigrams to obtain their statistics, e.g., histograms. In this example, a document, e.g., comprising {foo_endpoints: ["e1", "e2"]; bar_mode: "legacy"; enable_baz: true}, is generated by an end-user of production software and can represent the inputs and configurations used by that user. Tokenization (tree-traversal to obtain unigrams) followed by aggregation determines unigram statistics. In the example of Fig. 4, in the corpus of six documents, foo_endpoints takes on the value “e4” two times out of a total of six documents, bar_mode takes on the value “legacy” three times out of six; etc. Such statistics for independent unigrams, when obtained for both production as well as testing corpuses, can reveal gaps in test coverage [1].

Build vocabulary and transform documents into BoW (206): Using unigrams and their statistics, the pipeline can build the vocabulary $\{m_i \mid i = 1, 2, 3, \dots V\}$, also known as feature dictionary, where each word m_i (also known as feature) is a unigram. Whether a unigram is to be opted in or discarded from the vocabulary depends on frequency criteria, vocabulary-size criteria, human-curated lists, etc. Upon applying the NLP concepts of stop word and rare word to test coverage (illustrated in Fig. 1) to the six-document corpus of Fig. 4, enable_baz=true is identified as a stop word due to its 100% document frequency. It is therefore excluded from the vocabulary. On the other hand, consider a corpus of ten million similar documents with each unigram of Fig. 4 appearing at least a hundred thousand times. Each document also includes a timestamp of its creation. For such a scenario, the frequencies of the unigrams listed in Fig. 4 are much higher than each timestamp value; thus, the unigrams of Fig. 4 (excluding enable_baz=true) are deemed useful, and the timestamps discarded as rare words.

With the vocabulary thus obtained, the pipeline processes and trims each document, such that it contains only unigrams in the vocabulary. Each original document is thus transformed into a histogram of unigrams, also known as bag-of-words (BoW).



		Unigrams					
		foo_endpoints: “e1”	foo_endpoints: “e2”	foo_endpoints: “e3”	foo_endpoints: “e4”	bar_mode: “legacy”	bar_mode: “new”
Documents	foo_endpoints: [“e1”, “e2”] bar_mode: “legacy” enable_baz: True	1	1	0	0	1	0
	foo_endpoints: [“e1”, “e1”] bar_mode: “legacy” enable_baz: True	2	0	0	0	1	0
	foo_endpoints: [“e3”] bar_mode: “legacy” enable_baz: True	0	0	1	0	1	0
	foo_endpoints: [“e1”, “e2”, “e2”, “e4”] bar_mode: “new” enable_baz: True	1	2	0	1	0	1
	foo_endpoints: [“e3”, “e4”] bar_mode: “new” enable_baz: True	0	0	1	1	0	1
	foo_endpoints: [“e3”] bar_mode: “new” enable_baz: True	0	0	1	0	0	1

Fig. 5: Transforming the documents of Fig. 4 into a BoW. Note that enable_baz=true **is a stop word and is excluded from the vocabulary**

Fig. 5 illustrates a bag-of-words obtained from transforming the documents of Fig. 4. The described techniques consider documents as comprising a single type of tree-structured input read from a single data source. However, data pieces from multiple sources can be usefully designated as belonging to one document. For example, if a cloud-computing service provider separately logs the configurations of its virtual machines and the configurations of the physical hosts that run the virtual machines, it can be useful to join the two logs such that the described techniques discover relations among fields across the two sources. Such a combined document is relatively easy to obtain, e.g., through a simple join of the bag-of-words generated for the two sources. In this manner, fine distinctions between multi-source and single-source documents have little bearing.

Perform statistical analysis and training (208): Having tokenized each document via tree traversal, the pipeline performs statistical analysis and machine learning, e.g., latent Dirichlet allocation (LDA), principal component analysis (PCA), pairwise statistics, neural networks, etc. Some examples of statistical analysis include:

Pairwise statistics, such as joint probability, conditional probability, pointwise mutual information, etc., can be used to feed into unsupervised training (e.g., topic modeling), or they can be used to directly discover test coverage gaps (further explained below) involving bigrams (two unigrams).

- $P(i, j)$, the joint probability of observing unigrams i and j simultaneously in a document, is estimated using the corresponding frequency, i.e., count of observations in corpus normalized by corpus cardinality. For test coverage, non-zero $P(i, \mathrm{j})$ does not necessarily mean that the pair (i, j) mandates testing, because $P(i, j)$ can be non-zero even when i and j are probabilistically independent. A high conditional probability or pointwise mutual information, on the other hand, indicates an above-chance correlation between i and j, and indicates that they are to be tested together.
- $P(i \mid j) = P(i, j) / P(j)$ is the conditional probability of observing unigram i in a document, given that j has been observed.
- Pointwise mutual information (PMI) or normalized pointwise mutual information (NPMI) between unigrams i and j is defined as

$$\mathrm{pmi}(i; j) = \log[\, P(i \mid j) / P(i)\,]$$

$$\mathrm{npmi}(i; j) = -\,\mathrm{pmi}(i; j) / \log[P(i, j)].$$

Topic Modeling can be done based on the data and preprocessed documents obtained thus far. In one example, the pipeline can use LDA to perform unsupervised learning to obtain the topic model, where a topic is a probabilistic distribution over unigrams. The cohort of dominant unigrams in a topic reflects an aspect of the software usage pattern. Topic modeling can automatically detect a handful or hundreds of dominant unigrams out of a trimmed vocabulary of tens of thousands.

Higher quality topic models can be obtained by leveraging side information [4], e.g., in the form of a similarity graph, in addition to the document-term matrix. The similarity graph encodes the pairwise relationship between unigrams, which can be used to influence the training model and can be obtained using information such as normalized pointwise mutual information (NPMI), relative field distance in the protobuf schema tree, information from the source-code text and its modification history, human-curated configurations, etc. Although obtaining the topic model enables the determination of software usage patterns and test-coverage gaps, topic modeling can be used for other purposes, e.g., for the generation of test data mimicking production patterns.

Perform test coverage analysis (210): Test coverage analysis can be done using unigrams, bigrams, or n-grams, as follows. Unigram coverage analysis can be as described in [1]; bigram and n-gram coverage analysis are described below.

Bigram coverage analysis: A bigram is an unordered pair of 2 unigrams. Bigram coverage analysis is based on pairwise statistics. It complements n-gram analysis based on topic modeling and is simpler and cheaper to execute. For the unigram pair $\{i, j\}$, if the conditional probability estimated using production corpus is higher than some critical value P_c, which is close to 1, e.g.,

$$\mathrm{P}(i \mid j) \geq P_c ,$$

then there is high confidence that in production the unigram *i* appears whenever the unigram *j* appears. If, in the testbed, unigram *j* is observed but unigram *i* is not in the same document, then the testbed does not cover production usage, i.e., there is a test gap.

Example: A production corpus is made of documents listed in Fig. 4, each observed thousands of times. Further, for the production corpus,

$$P(\text{bar\_mode} = \text{"new"} \mid \text{foo\_endpoints} = \text{"e4"}) = 1.$$

In the test environment, foo_endpoints is observed taking the value “e4”, but only together with bar_mode = “legacy”. Then it may be concluded that {foo_endpoints = “e4” AND bar_mode = “new”} in production is a pattern rather than a mere coincidence; this pattern is flagged as a test gap.

Likewise, a high NPMI indicates above-chance co-occurrence, and can also be used for bigram coverage criterion, although the critical value for conditional probability and for NPMI can be different.

<u>N-gram coverage analysis</u>: An n-gram is an unordered collection of $n \geq 2$ unigrams. N-gram analysis is based on topic modeling. For each topic generated using the production corpus, the list of dominant unigrams is found using the weight of the unigrams in the topic and optionally such parameters as the maximum and minimum allowed list length. The pipeline uses the topics and the associated dominant unigram lists to match each document in the production and the testbed corpus. If there is a match, the pipeline emits an n-gram which is made of the unigrams in the list. The pipeline then counts the occurrence of each n-gram in both production and testbed. If an n-gram thus obtained is seen in production, it is likely a usage pattern of a group of unigrams that occur together as a non-coincidence; an n-gram, on the other hand, missing in the testbed is indicative of a test gap.

Metrics such as topic coherence can be used to evaluate the quality of each topic. False positives can be prevented by discarding topics with low coherence scores before being used to compute the n-gram coverage.

In this manner, the described techniques leverage connections between software usage pattern discovery and natural language processing (NLP) to apply computational linguistics to software testing. Besides test coverage, computational linguistics analysis, as described herein, can be used to understand any form of structured data with an evolving schema. The techniques of tokenizing structured log entries followed by computational linguistics analysis represent a union between the fields of test coverage analysis and NLP. The techniques apply to structured logs of any format, e.g., protobuf, JSON, etc. Software or service vendors can use the techniques to improve their software quality via improved test coverage. Cloud vendors can use the techniques to build a library that enterprise customers can leverage without having to implement their own.

Further to the descriptions above, and with regard to production data analyzed, a user may be provided with controls allowing the user to make an election as to both if and when systems, programs, or features described herein may enable the collection of user information (e.g., information about a user's code, data, or preferences), and if the user is sent content or communications from a server. In addition, certain data may be treated in one or more ways before it is stored or used so that personally identifiable information is removed. For example, a user's identity may be treated so that no personally identifiable information can be determined for the user, or a user's geographic location may be generalized where location information is obtained (such as to a city, ZIP code, or state level) so that a particular location of a user cannot be determined. Thus, the user may have control over what information is collected about the user, how that information is used, and what information is provided to the user.

CONCLUSION

This disclosure describes computational linguistics techniques for software usage patterns and test coverage. Structured input data (which can have arbitrary and evolving schema), obtained from production software and from testbeds, are tokenized using tree traversal to generate unigrams and unigram statistics. Documents are transformed into bags of words (BoW). BoWs are subjected to statistical analysis to programmatically and intelligently discover software usage patterns in production, to identify test coverage, and to flag gaps in testing.

REFERENCES

[1] Gao, Yifan; Rafferty, Ben; Sampson, Tylor; and Kubik, Joseph, “Test Coverage Analysis by Diffing Production Logs Against Integration Test Logs”, Technical Disclosure Commons, (March 01, 2021) https://www.tdcommons.org/dpubs_series/4113

[2] Chambers, Craig, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. “FlumeJava: easy, efficient data-parallel pipelines.” *ACM Sigplan Notices* 45, no. 6 (2010): 363-375.

[3] Blei, David M., Andrew Y. Ng, and Michael I. Jordan. “Latent dirichlet allocation,” *Journal of Machine Learning Research*, 3 (2003): 993–1022.

[4] Ahmed, Amr, James Long, Daniel Silva, and Yuan Wang. "A practical algorithm for solving the incoherence problem of topic models in industrial applications." In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1713-1721. 2017.