Clustering Test Steps in Natural Language toward Automating Test Automation

Linyi Li
University of Illinois at Urbana-Champaign
Urbana, USA
linyi2@illinois.edu

Zhenwen Li
Key Laboratory of High Confidence Software Technologies, MoE (Peking University)
Beijing, China
lizhenwen@pku.edu.cn

Weijie Zhang
Jun Zhou
Pengcheng Wang
Tencent Inc.
Guangzhou, China

Jing Wu
Guanghua He
Xia Zeng
Tencent Inc.
Guangzhou, China

Yuetang Deng
Tencent Inc.
Guangzhou, China
yuetangdeng@tencent.com

Tao Xie*
Key Laboratory of High Confidence Software Technologies, MoE (Peking University)
Beijing, China
taoxie@pku.edu.cn

ABSTRACT

For large industrial applications, system test cases are still often described in natural language (NL), and their number can reach thousands. Test automation is to automatically execute the test cases. Achieving test automation typically requires substantial manual effort for creating executable test scripts from these NL test cases. In particular, given that each NL test case consists of a sequence of NL test steps, testers first implement a test API method for each test step and then write a test script for invoking these test API methods sequentially for test automation. Across different test cases, multiple test steps can share semantic similarities, supposedly mapped to the same API method. However, due to numerous test steps in various NL forms under manual inspection, testers may not realize those semantically similar test steps and thus waste effort to implement duplicate test API methods for them. To address this issue, in this paper, we propose a new approach based on natural language processing to cluster similar NL test steps together such that the test steps in each cluster can be mapped to the same test API method. Our approach includes domain-specific word embedding training along with measurement based on Relaxed Word Mover's Distance to analyze the similarity of test steps. Our approach also includes a technique to combine hierarchical agglomerative clustering and K-means clustering post-refinement to derive high-quality and manually-adjustable clustering results. The evaluation results of our approach on a large industrial mobile app, WeChat, show that our approach can cluster the test steps with high accuracy, substantially reducing the number of clusters and thus reducing the downstream manual effort. In particular, compared with the baseline approach, our approach achieves **79.8**% improvement on cluster quality, reducing **65.9**% number of clusters, i.e., the number of test API methods to be implemented.

CCS CONCEPTS

• **Software and its engineering** → **Documentation**; **Use cases**;
• **Computing methodologies** → *Information extraction.*

KEYWORDS

Clustering, software testing, natural language processing

ACM Reference Format:
Linyi Li, Zhenwen Li, Weijie Zhang, Jun Zhou, Pengcheng Wang, Jing Wu, Guanghua He, Xia Zeng, Yuetang Deng, and Tao Xie. 2020. Clustering Test Steps in Natural Language toward Automating Test Automation. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368089.3417067

1 INTRODUCTION

To test a large industrial application, testers analyze the application's requirements, design user scenarios, and then instantiate them in *manually written test cases*, typically described in natural language (NL). The main content of an NL test case is a sequence of *test steps* described in NL. Despite the advances in automated testing, manually written test cases are still prevalent in industrial practice, because they are easily adjustable and interpretable, provide clear targets of user scenarios, and are sometimes inherited from legacy systems.

Given that NL test cases are not automatically executable, it becomes inevitable to automate the execution of these test cases, referred to as test automation [32], especially for the purpose of continuous integration and regression testing. Without test automation, to execute a test case, human testers must read through its test steps and carry them out by hand by interacting with the application under test. When the application becomes large, the number of these test cases can reach thousands, making test automation a necessity.

*Corresponding Author

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417067

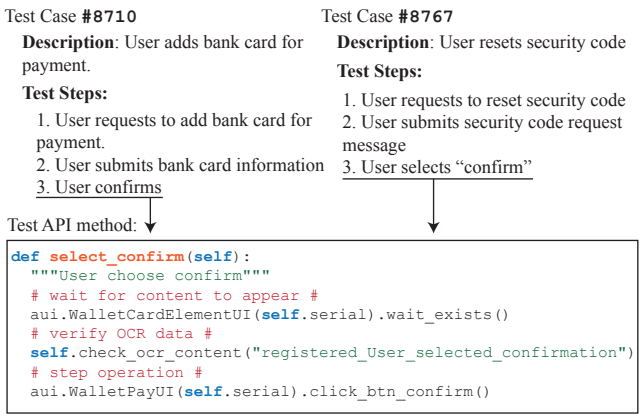

Figure 1: Examples of NL test cases for testing WeChat. The two test cases, namely case #8710 and case #8767, are designed for simulating different user scenarios. Each test case contains three test steps. Each test step should be implemented by a test API method for test automation. The third step of both test cases, despite having different descriptions, can be clustered and mapped to the same test API method illustrated in the bottom (`select_confirm()`).

In the existing industry practice, test automation of a test case consists of two phases, with the first phase requiring substantial manual effort: (1) manually implementing test-step API methods (in short as test API methods) and (2) automatically composing these test API methods. In particular, in Phase 1, testers manually translate each *test step* in the test case to a *test API method* that implements the described action in the test step. This phase is done manually due to the high complexity of user interface elements and pre/post-validations. For a large application under test, the number of test steps can reach thousands. Hence, manually implementing test API methods for these test steps requires substantial manual effort. In Phase 2, the corresponding test API methods for the test steps in the test case are sequentially composed to form an executable test script. This phase of composition is automated.

Even worse, quite some manual effort in the first phase is often wasted because in practice testers often may not realize the semantic similarity of multiple test steps and waste development effort for implementing duplicate test API methods (instead of just one) for them. According to our empirical investigation on existing test API methods implemented by WeChat testers, it is common that multiple test steps written differently share semantic similarity, and are supposed to be implemented by the same single test API method. The main reason is that test steps are written by multiple people, and it is hard for them to know whether and how other testers express the same test step. Figure 1 illustrates two example test cases from the industrial practice of testing WeChat, an industrial mobile app with over a billion active users. Although these two test cases' third test steps have different descriptions, i.e., "user confirms" and "user selects 'confirm' ", the two steps are supposed to be mapped to the same test API method (as illustrated in the bottom of Figure 1).

To address this issue, in this paper, we propose a new approach, with a supporting tool named *Clustep*, to cluster test steps based on their NL similarities, overcoming the limitations of existing clustering approaches. Some existing clustering approaches [3, 15, 36, 37] simply compare the literal equivalence or word-sharing ratio between test steps, but these approaches tend to miss similar test steps that should be merged together. Some other existing clustering approaches [1, 26] compare code similarity of their corresponding test API method implementations, but these approaches require substantial duplicate effort on manual implementation of test steps beforehand.

We propose our approach based on our four domain-specific insights for our application setting. First, the NL test steps use a very limited number of words, many of which are synonyms but not identical. Second, the NL test steps have very simple and similar grammatical structure. Third, besides improving clustering accuracy, it is highly important to reduce the number of clusters, because it is more efficient to implement related (somewhat but not highly similar) test steps (sharing some lines of code) by a single parameterized test API method, than writing multiple test API methods with some lines of code duplicated across them. Fourth, the clustering results should be manually adjustable, so that testers can further refine the results.

To produce high-quality clustering results while reducing the number of clusters, our approach includes multiple novel techniques for similarity analysis and clustering, proposed based on our domain insights. Inspired by Insight 1, our approach includes word embedding retraining [21] for word similarity analysis. Based on Insight 2, our approach measures test step distance based on Relaxed Word Mover's Distance [11]. From Insights 3 and 4, our approach includes an effective combination of hierarchical agglomerative clustering [29] with K-means clustering [17] post-refinement to produce high-quality and manually adjustable clustering results. The combined-clustering technique focuses on reducing the number of clusters while maintaining high clustering accuracy, and in the meantime producing manually adjustable clustering results.

Our approach is purely unsupervised, i.e., it does not rely on the existing labeled data set of mapping a test step to a test API method. Nor does our approach require manually derived features, thus assuring its generalizability and universality to various application settings.

To assess our approach, we conduct an evaluation on a large-scale test case dataset of WeChat, an industrial mobile app with over a billion active users. We measure the clustering accuracy by F-score, which penalizes both false positives and false negatives. Our evaluation results show that our approach achieves **79.8**% improvement on cluster accuracy, while reducing **65.9**% manual effort in terms of the number of test API methods, compared with a baseline approach based on keyword extraction and duplicate removal. These results show that our approach can substantially improve productivity and reduce downstream manual effort on test automation in large industry setting. The implementation for our approach has been integrated into the app testing system deployed for WeChat testing practice.

This paper makes the following main contributions:

- *Natural Language Processing Techniques for Test Step Analysis.* We develop domain-specific word embedding training, and

measurement based on Relaxed Word Mover's Distance to analyze the similarity of test steps.
- *High-Quality and Adjustable Test Step Clustering.* We develop a technique to combine hierarchical agglomerative clustering and K-means clustering post-refinement to derive high-quality and manually-adjustable clustering results.
- *Evaluation and Discussion.* We implement our approach with a supporting tool named *Clustep*, and conduct an evaluation, whose results demonstrate high effectiveness of our approach. We also discuss the failure cases (i.e., cases where our approach fails) and discuss multiple issues and limitations in the existing practice of writing NL test steps.

The remainder of this paper is organized as follows. §2 discusses related work. §3 states the preliminaries. §4 presents our approach in detail. §5 discusses the implementation. §6 presents the evaluation and discussion. §7 discusses failure cases and threats of validity. §8 concludes the paper.

2 RELATED WORK

Our work aims for automating test automation and is closely related to clustering and natural language processing.

Automating Test Automation. The concept of automating test automation is proposed by Thummalapenta et al. [32]. Their work proposes an approach to infer a sequence of action-target-data tuples from test cases manually written in NL. Their work uses POS Tagger [34] to annotate words and then calculates dependencies and grammar relationships between the words. The differences between their work and ours are two folds. First, their work uses traditional POS analysis for extracting a certain type of relation (i.e., "action" done on "target" using "data") by sentence structure analysis and field filling using literal entities. Their work cannot detect similarities between test steps, while our work makes use of similarities. Second, their work specifically generates DOM UI test cases for web applications, whereas our work is more general—it can be used whenever test steps are available. Little and Miller [15] translate keyword commands to executable code by keyword matching. Recent progress comes from Wang et al. [37], whose work generates system tests based on NL requirements using keyword/transformation rule matching and word relation extraction. All the preceding work uses traditional NLP techniques and relies on human-encoded rules to extract certain types of relations.

On manual analysis of use cases, Fantechi et al. [7] and Sinha et al. [28] use linguistic techniques to analyze use cases. The automatic test generation approach proposed by Sinha et al. [28], though preliminary, can be seen as a step toward automating test automation. Another direction for test automation (e.g., Text2Test [27], CoTester [18], and Cucumber [4]) is to write use cases and test cases using scripting languages, which are both human friendly and machine friendly.

In contrast to the related work, our work can effectively detect the semantic similarities between general NL test steps without any human supervision or tagging. Moreover, our work does not rely on use cases or test cases written in scripting languages.

Natural Language Processing (NLP) and Clustering. Neural networks and deep learning have boosted NLP in recent year, including text classification [12], machine translation [35], and reading comprehension [5]. Word2vec [21] can be viewed as one-layer fully-connected autoencoder neural network that generates word embeddings. Word embeddings are high-dimension number vectors that represent the words. The embeddings can capture the semantic and syntactic meanings of words and logical relations. Various approaches have been proposed to represent sentences or passages in embeddings [13] and measure sentence or document distance by embeddings of their contained words [11]. Our proposed approach leverages word embeddings, but includes new techniques tailored for test steps. Clustering techniques [9] can classify patterns or data items into groups (i.e., clusters). Popular clustering techniques include K-means [17], DBSCAN [6], GMM [24], and Hierarchical Agglomerative Clustering [29]. Our proposed approach combines existing clustering techniques to address domain-specific requirements in our application setting.

3 PRELIMINARIES

Objective. Our approach is a clustering approach—the clustering result is a partition of the given set of test steps. The objective of our approach is to produce *manually adjustable* clusters for achieving *high clustering accuracy*, while *minimizing the number of clusters*. First, the high accuracy is defined as minimizing inconsistency between the produced clustering results and the ground-truth results. We use the widely-used F-score [19, 30] as the evaluation metric for the clustering. Higher accuracy indicates less manual post-adjustment needed. Second, minimizing the number of clusters is also highly important in our objective. Fewer clusters indicate fewer test API methods needed to be implemented. To produce fewer clusters while still achieving high clustering accuracy, we can leverage the parameters in test API methods to distinguish different but sufficiently similar test steps. Third, the clustering results shall be amenable to manual adjustment. Since there exists no guarantee for the clustering results to be 100% correct, manual adjustment is needed in practice of applying the approach. It is expected that the approach supports real-time manual adjustment including cluster splitting, cluster merging, and changing cluster attribution of individual samples.

Dataset Setting and Characteristics. We study the manually constructed test cases for testing WeChat, a large industrial mobile app with over a billion active users. Table 1 shows some basic statistics of the dataset.

This dataset can serve as a reasonable representative of system test cases for a large industrial app for three main reasons. First, the dataset structure is standardized. All test cases are manually written in a typical structured format. Second, the dataset is clean. There are no unfinished nor invalid test cases. Third, the dataset size is relatively large. There are several thousands of test steps.

In particular, the dataset consists of a set of test cases. Each test case contains five fields: Target System, Business Type, Main Executor, Case Description, and Test Description. The "Test Description" field is a sequence of test steps. We consider the clustering of all test steps *across all test cases*. From manual inspection of existing test API methods, besides test steps themselves, considering only the *Main Executor* field of the belonging test case is sufficient to know the mappings from test steps to test API methods. Thus, we

Table 1: Dataset statistics. "#" stands for "number". "Avg." stands for "Average".

# Test Cases	745	# Test Steps	3,664
# Distinct Target Systems	10	# Distinct Test Steps	1,379
# Distinct Business Types	127	# Distinct Words in Test Description	718
# Distinct Main Executors	125	Avg. # Words per Test Description	4.043
# Distinct Case Descriptions	512	Avg. Word Frequency	20.63

consider only these two fields in our approach. In a sense, *each test step can be viewed as a* ⟨Main Executor, Test Description⟩ *tuple, where both fields are described in NL.*

Figure 1 shows the examples of NL test cases including test steps and a mapped test API method. As we can see, the Test Description fields specify test actions. The Main Executor fields (not shown in the figure) typically specify the identity of action initiator, such as "Authenticated User", "Manager", and "Merchant". The test API methods have no parameter, because the data generated during testing is either handled by the implementation of test API methods or handled by our testing infrastructure.

From Table 1 and Figure 1, we have four main observations. First, these test steps use limited words: only 718 distinct words occur in 3,664 test steps. Second, these test steps are highly duplicated: there are only 1,379 distinct test steps. Third, the grammatical structure of these test steps is simple and monotonic, mainly following the "user+*verb.*+*object*" form. Fourth, synonyms are prevalent: from manual inspection, we find that synonyms such as "confirm"/"accept", "click"/"choose" frequently occur. Related work [32] also observes similar NL characteristics from manually written use cases and test cases. The design of our approach makes use of these characteristics.

4 APPROACH

The overview of our approach is shown in Figure 2. First, we do preprocessing for all test steps. The preprocessing parses the Test Description sentences and Main Executor phrases to the lists of notional words. Then, we feed the lists to train domain-specific word embeddings using word2vec [21]. The word embeddings encode the words as numerical vectors, where the mutual distances capture the word semantic similarities. After that, given the word embeddings, we compute the pairwise test step distance using RMWD-based distance measurement [11]. Finally, the test steps are clustered using our effective combination of hierarchical agglomerative clustering [29] and K-means clustering [17].

Running Example. The bottom of Figure 2 shows a running example. First, the input dataset has two test steps whose Test Description fields are "User chooses return." and "User chooses cancel.". After preprocessing, the sentences are parsed to a list of words in their stem form, where "User chooses return." becomes `["user", "choose", "return"]`. Then, we collect all occurring words in the dataset and train domain-specific word embeddings. As a result, each test step can be viewed as a list of word vectors. After that, we compute the pairwise test step distance based on the word vectors using RMWD-based distance measurement. For example, the distance between the preceding two test steps can be as small as 0.1 due to the semantic similarity between "return" and "cancel". Finally, we use our effective clustering technique to group these two steps into one cluster, which is then assigned a single test API method.

We next describe each technique in detail, and also discuss multiple variants of our approach.

4.1 Preprocessing

Dataset preprocessing is the first stage of our approach. We split each test case field's text to an ordered list of words. To avoid bias from scarce words and irregular expressions, we remove all words that occur only once over the whole corpus. Furthermore, we also remove stop words, i.e., words that frequently occur but have no actual meaning. As a result, we remove 21 words and keep 697 words. After this procedure, all fields of test steps are sequences of notional words.

Then, we label domain-specific phrases as a single word. From manual inspection, we find that domain-specific phrases frequently occur, but similar phrases can result in totally different meanings. For example, "complete registered payer" and "simplified registered payer" have two words in common, but they are treated as distinct entities, and they correspond to different test API methods. Previously, we have maintained a terminology list for human test step writers to regularize the wording. We parse this terminology list and label each of these phrases in the list as a single word.

4.2 Training of Word Embeddings

Word embeddings, i.e., high-dimension number vectors, are trained to represent words. Fine-trained word embeddings preserve semantic and syntactic meanings of the words in the vector space [21].

Motivation. The dataset statistics (Table 1) show that the domain of test cases is dense in terms of words, i.e., the whole corpus uses limited words (718) and an individual word occurs with high frequency (20.63) on average. Therefore, although the size of the dataset is relatively small, the high density still supplies sufficient samples to learn the word embeddings. This characteristic differentiates test steps from data in other domains with similar size.

Model. We use word2vec [21], concretely, the skip-gram model, for word embedding training. To apply skip-gram, we use all pairs of co-occurring words as the input. "Co-occurring words" explicitly denote the words located closely to each other in a sentence or phrase. The motivation of the training scheme is that the neighboring words represent the context and imply the meaning of this word. This motivation also holds with test steps. For example, "return" and "cancel" have similar meanings in test steps, and they both co-occur with "choose", "previous", etc.

Improvements. Although the preceding observation indicates that only the Test Description and Main Executor fields are useful for clustering test steps, to better learn the context, we consider all test case fields including the Case Description field to construct the training dataset.

The training dataset is the set of adjacent word pairs. However, the fields of test steps are relatively short (on average only 4.043 words, see Table 1), so adjacent word pairs are limited. To mitigate this issue, we observe that test steps inside a test case are ordered by sequence, i.e., except the head and tail, each test step has its adjacent steps. Thus, we take adjacent steps into consideration by

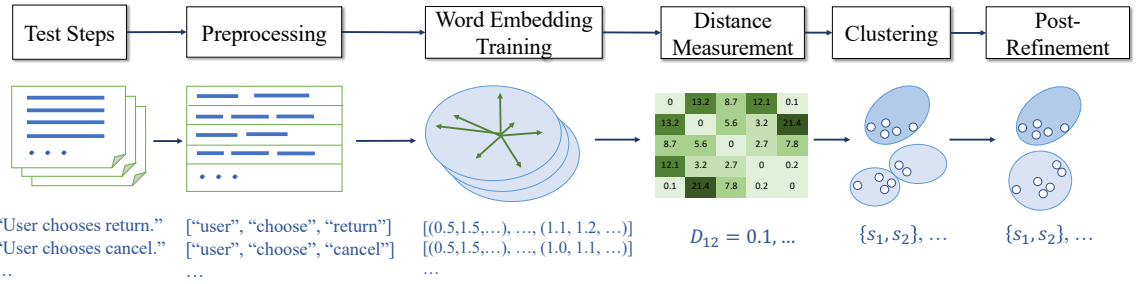

Figure 2: An overview of our approach. The running example is shown in the bottom. The word embedding training is based on word2vec. The distance measurement is relaxed word mover's distance (RWMD). The clustering is based on hierarchical agglomerative clustering, and is further refined by K-means.

concatenating the whole sequence of test steps in each test case to form a single long sentence to generate word pairs.

Use of Pre-trained Model Weights. We initialize the word embeddings using pre-trained word2vec model weights for common words on a large mixed corpus [22]. The word embeddings have 300 dimensions. The pre-trained model weights initialize 87.80% (612) of the total 697 words. We compute the mean and standard derivation of these 612 words and initialize the other words by sampling over the normalized distribution parameterized by this mean and standard derivation. Although the pre-trained model contains word embeddings for over 1 million common words, there are still only 87.80% of test-step words that can be initialized. This result indicates that the language domain of test cases is relatively specific, justifying the inevitability of retraining over this special corpus rather than directly using the pre-trained model.

Training Results. We calculate the pairwise likelihood between co-occurring words as the indicator of word embedding quality (the higher likelihood the better).

After initialization from the pre-trained model, the initial likelihood is just around 70%, indicating the large domain discrepancy between the general language context and the test step language domain. The discrepancy is shrunk quickly—after 6 epochs, the likelihood exceeds 90%. The overall process takes less than 5 minutes on a typical laptop CPU.

Variant: RNN (**RNNEmbed**). A variant of our technique is to use the recurrent neural network (RNN) [25], which is widely used in NLP. We adopt long short-term memory based RNN [8] for our application setting. We build an RNN language model similar to the RNN for widely used Penn Treebank dataset [20]. In each iteration, for test steps, we feed the Main Executor field concatenated with the Test Description field as sentences. The model is trained to convergence in 30 epochs and 5 min. Then, we retrieve the weights of word embedding layer and directly use them as the word embeddings. We denote this variant **RNNEmbed**.

4.3 Measurement of Test-Step Similarity

Kusner et al. [11] propose a fast and efficient sentence distance measurement—Relaxed Word Mover's Distance (RWMD). RWMD is a tight lower bound of Word Mover's Distance (WMD). WMD uses Euclidean distance of word embeddings as the word transformation cost and measures the sentence distance by the minimum cost of transforming each word from one sentence to the other. The direct calculation of WMD is too costly, but the relaxed lower bound RWMD can be efficiently computed. The tightness of the bound and the efficiency are verified in multiple classical NLP tasks [11].

Formally, for word w_i, we use f_i to denote its number of occurrences in the current sentence, and then we represent the sentence by normalized word frequency vector $\mathbf{x}$, where each component $x_i = f_i / \sum_j f_j$ encodes the normalized frequency of word w_i. Let the vector $\mathbf{v}_i$ denote the word embedding of word w_i that we have obtained in §4.2. On two sentences $\mathbf{x}$ and $\mathbf{x}'$, the RWMD distance is defined as below:

$$\mathrm{RWMD}(\mathbf{x}, \mathbf{x}') = \max\left(\ell_1(\mathbf{x}, \mathbf{x}'), \ell_2(\mathbf{x}, \mathbf{x}')\right), \text{ where}$$

$$\ell_1(\mathbf{x}, \mathbf{x}') = \sum_{i,j=1}^{n} t_{i,j} \|\mathbf{v}_i - \mathbf{v}_j\|_2 \quad \text{s.t. } t_{i,j} = \begin{cases} x_i, & j = \operatorname{argmin}_j \|\mathbf{v}_i - \mathbf{v}_j\|_2 \\ 0; & \text{otherwise} \end{cases}$$

$$\ell_2(\mathbf{x}, \mathbf{x}') = \sum_{i,j=1}^{n} t'_{i,j} \|\mathbf{v}_i - \mathbf{v}_j\|_2 \quad \text{s.t. } t'_{i,j} = \begin{cases} x'_j, & i = \operatorname{argmin}_i \|\mathbf{v}_i - \mathbf{v}_j\|_2 \\ 0. & \text{otherwise} \end{cases}$$

Another lower bound of WMD is Word Centroid Distance:

$$\mathrm{WCD}(\mathbf{x}, \mathbf{x}') = \left\| \sum_{i=1}^{n} x_i \mathbf{v_i} - \sum_{j=1}^{n} x_j \mathbf{v_j} \right\|_2.$$

Combining these two lower bounds, we develop the final distance metric $d(\mathbf{x}, \mathbf{x}') = \max(\mathrm{WCD}(\mathbf{x}, \mathbf{x}'), \mathrm{RWMD}(\mathbf{x}, \mathbf{x}'))$.

We calculate the distance between the Main Executor and the Test Description fields of test steps separately. All pairs of Main Executor distances and Test Description distances are normalized to the same mean. Then, the final distance between two test steps, denoted as d, is the weighted sum of the distance of these two fields, namely d_1 and d_2. Let w_{ME} be the relative weight of the Main Executor field with respect to the Test Description field, we have $d = w_{\mathrm{ME}} d_1 + d_2$. From a search on the parameters, we find that $w_{\mathrm{ME}} = 0.075$ yields the best performance.

Variants: Vectorization Based Measurements. Another category of measurements is based on the vector representation of test steps. We next introduce multiple variants that generate numerical vectors for test steps. Then we use the Euclidean distance of the corresponding vectors as the metric.

The TF-IDF (Term Frequency times Inverse Document Frequency) is a classical numerical statistic intended to reflect the word importance to a document in the corpus [23]. Formally, let m denote the number of words, and all words comprise the set $\{w_i : 1 \le i \le m\}$. TF ($\mathrm{tf}_{t,i}$) is the number of occurrences of the word w_i in field t. It

measures the local word frequency. Let N denote the number of test steps, and n_i denote the number of test steps where word w_i occurs. IDF (idf_i) measures the global word importance: $\text{idf}_i = \log_2(N/n_i)$.

TF-IDF Based Vectorization (**TFIDF**). We can represent test steps by TF-IDF field vectors. For each field $f \in \{\text{ME}, \text{TS}\}$, define $\text{tfidf}_{t,i,f} = \text{tf}_{t,i,f} \times \text{idf}_{i,f}$ to measure the weight of each word w_i. Then the field is represented by the vector $[\text{tfidf}_{1,f}, \text{tfidf}_{2,f}, \ldots, \text{tfidf}_{m,f}]^\top$. Let w_{ME} be an adjustable weight parameter similar to before, and $\mathbf{z}_1, \mathbf{z}_2$ are such vectors of the two fields, respectively. The weighted normalized concatenation, $(w_{\text{ME}}\mathbf{z}_1) \oplus \mathbf{z}_2$, is the TF-IDF based vectorization of the test step. Given that the vectorization is relatively sparse as each field has only a few words, we apply PCA [10] to reduce the vector dimension to 200. We denote this variant by **TFIDF**.

Word Embedding + IDF Vectorization (**IDFEmbed**). Because IDF measures the global word importance, another vectorization technique is the IDF weighted sum of word embeddings. Compared to TF-IDF based vectorization, this vectorization technique benefits from word embeddings. We denote this variant by **IDFEmbed**.

RNN (**RNNHidden**). The RNN language model that we have trained (§4.2) can provide not only word embeddings but also sentence embeddings. Concretely, after the RNN model receives the test step sentence as the input, on each word, the LSTM neurons of the model have a "hidden memory" of the current state, which is a numerical vector. We compute the average of these vectors from each word as the test step vector. We denote this variant by **RNNHidden**.

4.4 Hierarchical Agglomerative Clustering with K-means Post-Refinement

To this point, we attain the pairwise distance between test steps. We can then apply the clustering algorithm.

Hierarchical Agglomerative Clustering. In Hierarchical Agglomerative Clustering [29], at first, each sample is a cluster, and then the algorithm iteratively merges two nearest clusters together until there is only one cluster remaining. The merging criterion is choosing the pair of clusters that have the minimum average distance between all pairs of their elements. Thus, a binary tree indicating the cluster hierarchy can be constructed. Using this feature, the clustering results support manual adjustment: the split operation is to replace a cluster by two children clusters that are merged to form the cluster; the merge operation is to replace two children clusters by their parent cluster; the individual attribution adjustment is to change the placement of a subtree.

K-means Post-Refinement. The downside of hierarchical agglomerative clustering is that the merging criterion cares only about the entire optimality, but not single-sample optimality, i.e., samples may not be assigned to their closest cluster. Moreover, hierarchical agglomerative clustering considers only the cluster average distance. A small cluster might be close to some instances of a large one, but with large average distance, the small cluster cannot be merged.

To mitigate these limitations while reducing the number of clusters, we conduct K-means clustering [17] for post-refinement. In K-means, first, we generate k initial "means" from random sampling from the dataset. Then, in each iteration, we execute the following two steps: (1) assign each sample to the closest cluster by the Euclidean distance; (2) recalculate the new cluster means from the new sample assignments. The algorithm terminates when reaching convergence, i.e., no cluster attribution changes.

Our approach uses hierarchical agglomerative clustering as the initialization of K-means clustering, and then executes the iteration routine of K-means until convergence. We also consistently remove empty clusters at the end of each iteration routine. Compared to pure hierarchical agglomerative clustering, K-means post-refinement strictly guarantees that every sample is assigned to its nearest cluster, i.e., the assignment brings sample-wise optimum. Additionally, the removal of empty clusters reduces the number of clusters. Compared to pure K-means clustering, our combined clustering algorithm is *deterministic*. Note that K-means post-refinement converges fast. For our 3,664 test steps, K-means post-refinement on average runs 6.369 iterations. Each iteration costs only 5 seconds.

Preserving Manual Adjustment Availability. Given that K-means post-refinement destroys the tree structure of hierarchical agglomerative clustering results, we recover, or more precisely, reconstruct the tree structure using the following three steps: (1) run "local" hierarchical agglomerative clustering inside each cluster, yielding a cluster hierarchical binary tree for each cluster; (2) run "global" hierarchical agglomerative clustering where each cluster is viewed as a single data point; the clustering finally merges all clusters to a single one; (3) concatenate the "local" hierarchical tree of each cluster to the "global" hierarchical tree. Note that the leaf nodes of the "global" tree and the root nodes of the "local" trees represent the same set of clusters, and thus the concatenation is straightforward. With the tree structure, we support manual adjustments including cluster splitting, merging, and individual attribution adjustment on the K-means post-refined results.

4.5 Summary

Combining these preceding techniques forms our approach **Clustep** and **Clustep+**. **Clustep** uses retrained word2vec word embeddings with our proposed RWMD-based metric and then applies hierarchical agglomerative clustering. **Clustep+** further applies K-means post-refinement to reduce the number of clusters. At the expense, **Clustep+** may sacrifice some degree of global optimum, i.e., yielding worse cluster accuracy.

Some variants of the approach are also available and evaluated in §6. **RNNEmbed** uses word embeddings from an RNN language model and measures the test step distance with RWMD. **RNNHidden** uses the sentence embeddings from the RNN language model, and measures the Euclidean distance as the test step distance. **TFIDF** uses TF-IDF based vectorization with hierarchical agglomerative clustering. **IDFEmbed** uses word embeddings from **Clustep**, additionally weighted by IDF. We also evaluate the corresponding K-means post-refinement version for **RNNEmbed** and **IDFEmbed**, and they are denoted as **RNNEmbed+** and **IDFEmbed+**, respectively. Finally, **Dedup** is the baseline approach in §6, where two test steps are clustered together if and only if both fields of these test steps are exactly the same after preprocessing.

Table 2 summarizes all these approaches and variants.

Table 2: Overview of approaches and their techniques. Clustep and Clustep+ are our main approach.

Approach		Pre-processing	Embedding		Measurement				Clustering	
			Word2vec	RNN	TF	IDF	Euclidean	RWMD	Agglo.	K-means
Baseline	**Dedup**	✓								
Approach Variants	**TFIDF**	✓			✓	✓	✓		✓	
	IDFEmbed	✓	✓			✓		✓	✓	
	IDFEmbed+	✓	✓			✓		✓	✓	✓
	RNNEmbed	✓		✓				✓	✓	
	RNNEmbed+	✓		✓				✓	✓	✓
	RNNHidden	✓		✓			✓		✓	
Main Approach	**Clustep**	✓	✓					✓	✓	
	Clustep+	✓	✓					✓	✓	✓

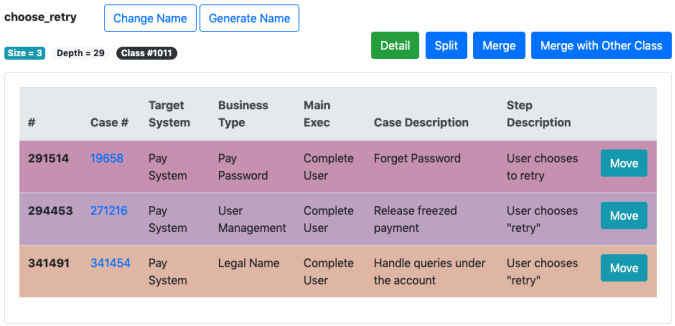

Figure 3: The UI screenshot of Clustep.

5 IMPLEMENTATION

We implement our approach with a supporting tool named Clustep mainly in Python. Critical modules such as clustering and distance computation are written in C++ for acceleration. To enable manual result adjustment, our tool also includes web-based UI using Django. The tool has been integrated into the testing system deployed for the WeChat testing practice.

Given an entire set of NL test cases, the tool produces the clustering results of the test steps included in the test set. Based on the clustering results, for each NL test case, the tool produces an *executable test case* by the following two steps: (1) replace each test step in the test case with the call of the test API method whose corresponding cluster includes the test step; and (2) compose sequentially these test API method calls.

Runtime Cost. Except word embedding training, the pipeline within the tool runs within 40 s on Intel Xeon E5-2650 CPU using a single core, including preprocessing, distance calculation, and hierarchical agglomerative clustering. If we additionally use K-means post-refinement, then the pipeline within the tool runs within 300 s. The word embedding training takes 15 min to reach over 90% likelihood but it needs to be executed only once.

User Interface. The user interface screenshot is shown in Figure 3. The UI supports all aforementioned manual adjustments. Besides, the UI recommends cluster candidates for adjustment, recommends cluster names, and shows the inner taxonomy of each cluster. Further details are omitted due to the space limit.

6 EVALUATION

To assess our approach, we conduct an evaluation on a large-scale test case dataset of WeChat, an industrial mobile app with over a billion active users. In particular, we intend to answer the following two main research questions:

- **RQ1:** How effectively can our approach improve over related approaches?
- **RQ2:** How much does each of our techniques contribute to the overall effectiveness achieved by our approach?

We first describe how we generate the clustering ground truth from test API method implementations and then discuss the results for addressing the preceding research questions.

6.1 Ground Truth Generation

We have the access to the test API method implementations for the test steps used in our evaluation. The bottom part of Figure 1 shows an example. Before the work in this paper, to reduce the number of test API methods and save maintenance cost, we merged test API methods with similar implementations. As a close approximation, we extracted the function call sequences using Python's `ast` package, compared the sequence equivalence, and merged test API methods with identical sequences. After that, we produced the ground truth for clustering accuracy evaluation by putting all test steps implemented by the same test API method to the same cluster.

We remark that the generated ground truth is more based on implementation similarity of test steps than semantic similarity. We generate ground truth from implementation similarity because in this way, the test steps in the same ground truth cluster can surely be implemented by a single test API method. Thus, the criterion directly aligns with our goal of reducing test API methods to implement.

6.2 Evaluation Metric

We evaluate the clustering accuracy using the F score on the test steps with ground truth clusters. The F score (also called F_1 score) has wide applications in statistics, machine learning, and NLP [19, 30, 33]. The F score considers each pair of different test steps: if the pair of test steps belongs to the same ground-truth cluster, and is assigned to the same cluster, the pair is a *true positive* instance; if the pair belongs to different ground-truth clusters, but is assigned to the same cluster, the pair is a *false positive* instance; if the pair belongs to the same ground-truth cluster, but is assigned to different

Table 3: Comparison of clustering accuracy measured by F score for both strict and loose settings. The highest numbers for both settings are bolded. The baseline approach is Dedup. In the strict setting, the number of clusters is required to be fewer or equal to 600. In the loose setting, there is no limit on the number of clusters. Note that Dedup does not support adjustment on the number of clusters.

Approach		Strict Setting		Loose Setting	
		Best F Score	# Cls. of Best	Best F Score	# Cls. of Best
Baseline	**Dedup**	/	/	45.35%	1,719
Approach Variants	**TFIDF**	14.44%	590	68.28%	1,320
	IDFEmbed	65.50%	570	79.43%	1,310
	IDFEmbed+	67.20%	567	83.22%	1,241
	RNNEmbed	50.57%	600	80.61%	1,160
	RNNEmbed+	65.08%	440	82.40%	863
	RNNHidden	35.27%	600	51.64%	790
Main Approach	**Clustep**	80.87%	600	**83.43%**	1,090
	Clustep+	**81.55%**	586	82.86%	1,345

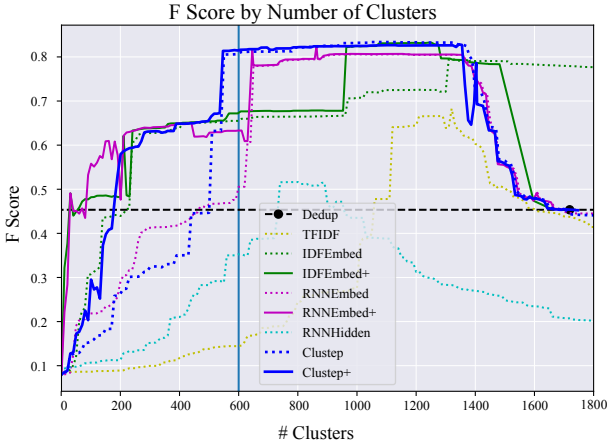

Figure 4: Clustering accuracy (measured by F score) with respect to the number of clusters. Our main approach, Clustep and Clustep+, is shown in the bolded blue dashed line and bolded blue solid line, respectively. The black point represents the baseline approach (Dedup). All other lines represent the approach variants. In the middle, the blue vertical line labels the constraint 600 on the number of clusters separating the strict and loose settings.

clusters, the pair is a *false negative* instance; if the pair belongs to different ground-truth clusters, and is assigned to different clusters, the pair is a *true negative* instance. With TP, FP, FN, TN used to denote the number of these instances, respectively, from *precision* and *recall*, the F score is defined as

$$precision := TP/(TP + FP), \quad recall := TP/(TP + FN),$$
$$F := 2 * precision * recall/(precision + recall).$$

Easily seen, the F score lies in range $[0, 1]$. A larger F score indicates better clustering quality.

6.3 Evaluation Setup

We conduct our evaluation on the dataset from our large industrial mobile app WeChat, as mentioned earlier (§3). All approaches are evaluated using the F score. Except for the baseline, all evaluated approaches have two adjustable parameters: the number of clusters and adjustable weights for the Main Executor field (w_{ME}). Following a suggested practice [2], we do a random search on these parameters and select the best parameters for each approach.

Note that these approaches are all *deterministic*. These approaches, even including the ones with K-means post-refinement, have no randomness because of deterministic initialization from hierarchical agglomerative clustering. Thus, for each setting, the approach needs to run only *once*.

We consider two evaluation settings: *strict* setting and *loose* setting. In the strict setting, we require the number of clusters to be smaller or equal to 600, and consider only clustering results satisfying this constraint. In the loose setting, we discard this constraint. A smaller number of clusters mean fewer test API methods to be implemented, while a larger number of clusters may have better cluster accuracy. The threshold 600 is determined empirically from the number of clusters (1,719) achieved by the baseline **Dedup**, and the total number of test steps (3,664). Since the clustering quality is dependent on the adjustable number of clusters, in the strict setting, for each approach, we let the number of clusters be 10, 20, ..., 590, 600 after agglomerative hierarchical clustering. For some approaches, we then run K-means post-refinement for each of these numbers of clusters. After that, we measure the best F score of each approach for all these numbers of clusters. In the loose setting, we remove the constraint on the number of clusters, and measure the best F score similarly. Table 3 shows the detailed results for both settings.

RQ1: How effectively can our approach improve over related approaches?

Strict Setting. The left part of Table 3 shows the results. The F score, as mentioned earlier, is the measurement of clustering accuracy. Our **Clustep+** approach is designed for this setting. The table shows that our approach (achieving 81.55%) is substantially better than other variants (at best 67.20%), as expected. *Especially, when compared with the baseline, our approach's F score is* 79.8% *better (*81.55% *vs* 45.35%*) and the number of clusters is* 65.9% *fewer (*586 *vs* 1,719*)*. The main reason for the improvement is the effective reduction on the number of clusters from K-means post-refinement and the strength of RWMD-based measurement combined with word embeddings.

Loose Setting. The best clustering accuracy is shown in the right part of Table 3. Our **Clustep** approach is designed for this setting. The table shows that our approach achieves the best clustering accuracy (83.43%), being much better than the baseline (45.35%). Other variants also achieve good clustering accuracy, such as 83.22% achieved by **IDFEmbed+** and 82.40% achieved by **RNNEmbed+**. We further discuss the implication of this result in RQ2.

In both settings, our approach, including **Clustep** and **Clustep+**, achieves the best clustering accuracy, and is substantially better than the baseline approach **Dedup**.

Time Savings. We estimate the actual time savings using our approach. From actual practice, we find that it takes an experienced human tester 5 min to write a test API method, and it takes a familiar user of our tool 1 min to adjust an incorrect cluster. Therefore, without our tool, transforming test steps to test API methods takes about

18 000 min (3, 664 test steps ×5 min); while with our tool, transforming test steps to test API methods takes about 3000 min (586 test steps ×5 min + (586 × 81.55%) incorrect clusters ×1 min). With 10 human testers, the whole transforming process is estimated to last only 5 hr instead of 30 hr—our tool saves enormous time.

Besides, after clustering, we need to implement much fewer test API methods. There are 9.105 lines of code on average for each test API method. Now we need to implement only 586 test API methods instead of 3, 664, and thus the estimated saved lines of code are 28, 000. Also, fewer lines of code substantially save the maintenance cost.

RQ2: How much does each of our techniques contribute to the overall effectiveness achieved by our approach?

Clustep and **Clustep+** use four major techniques: domain-specific preprocessing, word2vec word embedding training, RWMD-based step distance measurement, and hierarchical agglomerative clustering with K-means post-refinement. In Figure 4, to analyze the effect of each technique, we plot the clustering accuracy of different approaches and the variants with respect to the number of clusters.

Domain-Specific Preprocessing. All the evaluated approaches use domain-specific preprocessing, including the baseline. Before preprocessing, all 3, 000+ test steps are different in at least one field, and cannot be clustered together. With the preprocessing including low-frequency word removal, domain phrase concatenation, and stop word removal, even simple baseline **Dedup** reaches 45.35% F score and removes over half of clusters.

Word Embedding Training and RWMD-Based Distance. Except **Dedup**, **TFIDF**, and **RNNHidden**, other approach variants and our main approach all use word embeddings. Among them, **IDFEmbed**, **IDFEmbed+**, **Clustep**, and **Clustep+** use word2vec trained word embeddings combined with RWMD-based distance measurement. **RNNEmbed** and **RNNEmbed+** use RNN trained word embeddings with RWMD-based distance measurement.

From Table 3 and Figure 4, we find that word embeddings and RWMD-based distance jointly improve the clustering accuracy and reduce the number of clusters substantially. Under the loose setting, the approaches with word embeddings and RWMD-based distance measurement achieve clustering accuracy of 79.43% − 83.43%. However, without these techniques, **Dedup** and **TFIDF** have only 45.35% and 68.28% clustering accuracy. **RNNHidden** achieves only 51.64%. The gap between the variants with and without these techniques is larger than 10%. Similar trends can be observed under the strict setting.

The main reason is that word embeddings map semantically similar words to similar embeddings [11, 13, 21] and the RWMD-based distance measurement effectively measures the similarity of test steps from the similarity of their word embeddings [11]. Therefore, semantically similar test steps are correctly clustered together, unlike the baseline **Dedup**, the variant **TFIDF**, or existing approaches in the literature [31, 32]. These baseline and variants treat distinct words as equally different ones. **RNNHidden** uses sentence embeddings directly extracted from RNN neurons. This approach is shown to be powerful in a large and diversified NLP corpus [14, 16]. However, our test step corpus has small size and short document length. These characteristics pose difficulties in learning sentence embeddings from RNN.

Table 4: Ablation study for K-means post-refinement under the strict setting. Column "+" shows the F score with K-means post-refinement, and Column "-" shows the F score without it.

Approach Variants	-	+	Improvement
IDFEmbed/IDFEmbed+	65.60%	67.20%	+1.70%
RNNEmbed/RNNEmbed+	50.57%	65.08%	+14.51%
Clustep/Clustep+	80.87%	81.55%	+0.68%

K-Means Post-Refinement. K-means post-refinement is particularly proposed for the strict setting. Thus, we do ablation study under this setting, and the results are shown in the left part of Figure 4 and Table 4. In Figure 4, the left side of the vertical line corresponds to the strict setting. We can observe that **Clustep+** in the blue solid line is substantially better than **Clustep** in the blue dashed line under the same number of clusters. If we further reduce the number of clusters from 600 to 400 or 200, the advantage of K-means post-refinement becomes much more pronounced. For example, with 400 clusters, **Clustep** has roughly only 35% F score while **Clustep+** achieves over 64%. In Table 4, each variant with suffix "+" is the K-means post-refinement version of that without "+", so the pairs can be directly compared. K-means post-refinement, as expected, improves the clustering accuracy for different variants ranging from 0.68% to 14.51%. We note that sometimes the improvements are small. The reason could be reaching the performance limit of the current approach pipeline as increasing the number of clusters cannot improve much (see Figure 4). Further analysis is left to future work.

All these techniques contribute to the overall effectiveness. From the preceding discussion, we rank the importance from high to low as domain-specific preprocessing, word embeddings combined with RWMD-based distance, and K-means post-refinement. On the other hand, as indicated in Figure 4, the performance is in some degree independent of how the word embeddings are trained (e.g., by word2vec in **Clustep** or by RNN in **RNNEmbed**), or how the different keywords are weighted (e.g., equally-weighted in **Clustep** or IDF-weighted in **IDFEmbed**), because all these variants have very similar performance.

7 DISCUSSION

In this section, we first present a study of failing causes for suggesting future directions. We then discuss threats to validity.

7.1 Study of Failing Causes

Despite the satisfactory clustering results, there are still incorrect clusters. We manually inspect the best clustering results under the strict setting ($F = 81.55\%$ from **Clustep+**), and analyze the failing causes of our approach. The incorrect cluster examples are shown in Figure 5. We next summarize the four main causes of incorrect clusters, and Figure 6 shows the frequency statistics of each cause. The statistics of Figure 6 are calibrated from independent studies of two researchers to assure soundness.

No.	Main Executor	Test Description	Pred. Label
1	User	user chooses to agree	1
3	Complete User	user chooses to agree	1
4	User	user chooses not to agree	1
6	Complete User	user chooses not to agree	1

(a)

No.	Main Executor	Test Description	Pred. Label
1	User	user chooses to confirm	2
2	Complete User	user chooses to confirm	2
3	Auth. User	user chooses to confirm	2
6	Simple User	user chooses to confirm	2
7	Complete User	feedback info. failed, go to 9	3

(b)

No.	Main Executor	Test Description	Pred. Label
1	Auth. User	user chooses "upload personal ID"	5
2	User	user chooses to upload personal ID	5
3	User	user chooses "upload personal ID picture"	5

(c)

Figure 5: Incorrect cluster examples for failing cause analysis. Each table corresponds to a ground truth cluster. Gray rows show wrong samples. The "Pred. Label" shows the cluster label predicted by our approach.

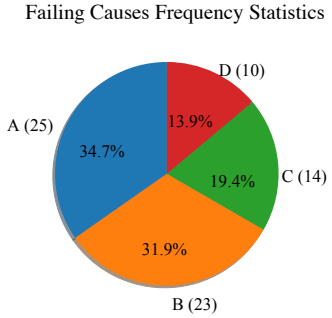

Figure 6: Failing cause statistics out of 72 incorrect clusters from the 586 clusters in total. A, B, C, and D correspond to those in §7. Each of the numbers in the parenthesis denotes the number of clusters falling into each category.

A *Independence between test API method implementation similarity and test step semantic similarity.* In Figure 5(b), the 7th row is a test step that should be in the same ground truth cluster with preceding rows but is assigned to another cluster. The row shares little semantic similarity with preceding rows but their test API method implementation is the same.

B *Inappropriate handling of Main Executor difference.* In Figure 5(b), the gray row is different from others in the Main Executor field. Those whose Main Executor is "Simple User" correspond to a special test API method, while all others correspond to the same but another test API method.

C *Bad word embeddings for scarce words.* When the words are rare in the context, their word embeddings cannot be well trained, and thus are close to their initialization values and are similar to common words. Thus, test steps containing these words are likely to be clustered to other common clusters.

D *Ignorance of difference between few important words.* In Figure 5(a), the rows containing "not" are wrongly clustered. In Figure 5(c), the erroneous last row differs from others in the last word "picture". The reason is that the difference in a single word may be diluted when all other words are similar.

The most common causes, A and B, reveal that professional knowledge should be provided, e.g., when the Main Executor matters and when the implementation can be the same. Without this knowledge, even human cannot decide correctly. These two causes account for roughly 66% of incorrect clusters. This large portion reflects that the ambiguity and language inconsistency of NL test steps in industrial apps is the major cause of failure. As a solution, we propose to require the human test step writers to explicitly tag the specific objects that have context-specific meanings. We have not collected large enough data to study the effectiveness of this new principle, and we leave the study as future work.

Cause C calls for better word embedding training algorithms, especially learning accurate word embeddings when the dataset size is limited. For cause D, we have tried some heuristics such as manually labeling some special words as keywords such as "not", "fail", "logout", and "wrong". However, we find that in many other cases, these words should be ignored instead. For example, "user chooses not to input password" has the same test API method as "user chooses to ignore password input". As a result, without considering the context, such heuristics bring even worse results, reaching only 54.5% F score. We believe that better context handling techniques, or a more regularized writing style of test steps, could be helpful to handle cause D.

7.2 Threats to Validity

The preceding study shows an internal threat: the implementation similarity of test API methods may be different from the semantic similarity of test steps. However, our high clustering accuracy indicates that these cases are relatively rare. Another threat comes from the incremental gains from K-means post-refinement in some cases. We plan to conduct a further study on K-means post-refinement in future work.

A major external threat is related to whether the proposed approach and the evaluation can be generalized well to other similar scenarios. The approach itself is general enough for any NL test steps sharing the structures and characteristics shown in §3, and existing studies on NL test cases [32] indicate that these characteristics are not rare. However, it is currently hard to evaluate the approach for these similar scenarios since few test case datasets are publicly available.

8 CONCLUSION

In this paper, we have proposed an approach with multiple novel techniques to cluster similar NL test steps together. The approach can cluster the test steps with high accuracy and reduce the number of clusters to substantially reduce the downstream manual effort. We have evaluated the effectiveness of the approach on test cases of WeChat, a large industrial app, and have integrated the approach's implementation into the testing system for the app. In future work, we plan to achieve higher accuracy and extend our approach to handle more-unstructured test cases.

REFERENCES

[1] Nadim Asif, Faisal Shahzad, Najia Saher, and Waseem Nazar. 2009. Clustering the source code. *WSEAS Transactions on Computers* 8 (12 2009), 1835–1844.

[2] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, 10 (2012), 281–305. http://jmlr.org/papers/v13/bergstra12a.html

[3] Ravishankar Boddu, Lan Guo, Supratik Mukhopadhyay, and Bojan Cukic. 2004. RETNA: From requirements to testing in a natural way. In *Proceedings of the 2004 12th IEEE International Requirements Engineering Conference (RE '04)* (Kyoto, Japan). IEEE, 262–271.

[4] Cucumber. 2019. Cucumber. https://cucumber.io. Accessed: 05-03-2020.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (NAACL-HLT '19)* (Minneapolis, Minnesota). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[6] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 1996 2nd International Conference on Knowledge Discovery and Data Mining (KDD '96)* (Portland, Oregon). AAAI Press, 226–231.

[7] Alessandro Fantechi, Stefania Gnesi, Giuseppe Lami, and Alessandro Maccari. 2003. Applications of linguistic techniques for use case analysis. *Requirements Engineering* 8, 3 (2003), 161–170.

[8] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[9] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. 1999. Data clustering: A review. *ACM Computing Surveys (CSUR)* 31, 3 (1999), 264–323.

[10] Ian T Jolliffe. 1993. Principal component analysis: A beginner's guide—II. Pitfalls, myths and extensions. *Weather* 48, 8 (1993), 246–253. https://doi.org/10.1002/j.1477-8696.1993.tb05899.x

[11] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From word embeddings to document distances. In *Proceedings of Machine Learning Research* (Lille, France), Francis Bach and David Blei (Eds.), Vol. 37. PMLR, 957–966. http://proceedings.mlr.press/v37/kusnerb15.html

[12] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In *Proceedings of the 2015 29th AAAI Conference on Artificial Intelligence (AAAI '15)* (Austin, Texas). AAAI Press, 2267–2273.

[13] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of Machine Learning Research* (Bejing, China), Eric P. Xing and Tony Jebara (Eds.), Vol. 32. PMLR, 1188–1196. http://proceedings.mlr.press/v32/le14.html

[14] Zachary C Lipton, John Berkowitz, and Charles Elkan. 2015. A critical review of recurrent neural networks for sequence learning. *CoRR* abs/1506.00019 (2015). arXiv:1506.00019 http://arxiv.org/abs/1506.00019

[15] Greg Little and Robert C. Miller. 2006. Translating keyword commands into executable code. In *Proceedings of the 2006 19th Annual ACM Symposium on User Interface Software and Technology* (Montreux, Switzerland) *(UIST '06)*. Association for Computing Machinery, 135–144. https://doi.org/10.1145/1166253.1166275

[16] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. 2016. Recurrent neural network for text classification with multi-task learning. In *Proceedings of the 2016 25th International Joint Conference on Artificial Intelligence* (New York, New York) *(IJCAI '16)*. AAAI Press, 2873–2879.

[17] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[18] Jalal Mahmud and Tessa Lau. 2010. Lowering the barriers to website testing with CoTester. In *Proceedings of the 2010 15th International Conference on Intelligent User Interfaces* (Hong Kong, China) *(IUI '10)*. Association for Computing Machinery, 169–178. https://doi.org/10.1145/1719970.1719994

[19] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press, New York, New York.

[20] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The penn treebank. *Computational Linguistics* 19, 2 (1993), 313–330. https://www.aclweb.org/anthology/J93-2004

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 2013 26th International Conference on Neural Information Processing Systems (NIPS '13)* (Lake Tahoe, Nevada), Vol. 2. Curran Associates Inc., 3111–3119.

[22] Yuanyuan Qiu, Hongzheng Li, Shen Li, Yingdi Jiang, Renfen Hu, and Lijiao Yang. 2018. Revisiting correlations between intrinsic and extrinsic evaluations of word embeddings. In *Proceedings of the 2018 17th Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data (NLP-NABD '18)* (Changsha, China), Maosong Sun, Ting Liu, Xiaojie Wang, Zhiyuan Liu, and Yang Liu (Eds.). Springer International Publishing, 209–221.

[23] Anand Rajaraman and Jeffrey David Ullman. 2011. *Mining of massive datasets.* Cambridge University Press, Cambridge, UK.

[24] Carl Edward Rasmussen. 1999. The infinite Gaussian mixture model. In *Proceedings of the 1999 12th Advances in Neural Information Processing Systems (NIPS '99)* (Denver, Colorado). MIT Press, 554–560.

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.

[26] Mark Shtern and Vassilios Tzerpos. 2012. Clustering methodologies for software engineering. *Advances in Software Engineering* 2012, Article 1 (Jan. 2012). https://doi.org/10.1155/2012/792024

[27] A. Sinha, S. M. S. Jr., and A. Paradkar. 2010. Text2Test: Automated inspection of natural language use cases. In *Proceedings of the 2010 3rd International Conference on Software Testing, Verification and Validation (ICST '10)* (Paris, France). IEEE, 155–164. https://doi.org/10.1109/ICST.2010.19

[28] A. Sinha, A. Paradkar, P. Kumanan, and B. Boguraev. 2009. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09)* (Lisbon, Portugal). IEEE, 327–336. https://doi.org/10.1109/DSN.2009.5270320

[29] Peter HA Sneath and Robert R Sokal. 1973. *Numerical taxonomy: The principles and practice of numerical classification.* W. H. Freeman, San Francisco, California.

[30] Alaa Tharwat. 2018. Classification assessment methods. *Applied Computing and Informatics* (2018). https://doi.org/10.1016/j.aci.2018.08.003

[31] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. Kumar, and S. Kumar. 2013. Efficient and change-resilient test automation: An industrial case study. In *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE '13)* (San Francisco, California). IEEE, 1002–1011. https://doi.org/10.1109/ICSE.2013.6606650

[32] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra. 2012. Automating test automation. In *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE '12)* (Zurich, Switzerland). IEEE, 881–891. https://doi.org/10.1109/ICSE.2012.6227131

[33] Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the 2003 7th Conference on Natural Language Learning at HLT-NAACL 2003 (CoNLL '03)* (Edmonton, Canada), Vol. 4. Association for Computational Linguistics, USA, 142–147. https://doi.org/10.3115/1119176.1119195

[34] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL '03)* (Edmonton, Canada), Vol. 1. Association for Computational Linguistics, USA, 173–180. https://doi.org/10.3115/1073445.1073478

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 2017 31st International Conference on Neural Information Processing Systems (NIPS '17)* (Long Beach, California). Curran Associates Inc., 6000–6010.

[36] Carlos Videira, David Ferreira, and A Silva. 2006. Patterns and parsing techniques for requirements specification. In *Proceedings of the 2006 1st Iberian Conference on Information Systems and Technologies (CISTI '06)* (Ofir, Portugal), Vol. 2. 375–390.

[37] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2019. Automatic generation of system test cases from use case specifications: An NLP-based approach. *CoRR* abs/1907.08490 (2019). arXiv:1907.08490 http://arxiv.org/abs/1907.08490