Paper Title\* (use style: paper title)

\*Note: Sub-titles are not captured in Xplore and should not be used

line 1: 1st Given Name Surname   
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

line 1: 4th Given Name Surname  
line 2: *dept. name of organization*  
*(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCIDline 1: 2nd Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

line 1: 5th Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCIDline 1: 3rd Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

line 1: 6th Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

*Abstract*—This electronic document is a “live” template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. *\*CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract*. (*Abstract*)

Keywords—component, formatting, style, styling, insert (key words)

# **Introduction**

Software testing is a process designed to evaluate the functionality of a software application to ensure it meets specified requirements. This involves executing the software under controlled conditions and analyzing the results to identify defects or bugs. Testing is essential for ensuring software quality, verifying that the software meets the intended requirements, is reliable, secure, and performs as expected. It is a critical part of the software development lifecycle and encompasses various types, such as unit testing, integration testing, system testing, and acceptance testing. Acceptance testing determines whether the software meets business requirements and is satisfactory to end users or stakeholders. Often, acceptance tests are written in natural language, particularly when using Behavior Driven Development (BDD).

BDD is a software development methodology that enhances collaboration between developers, testers, product owners, and other non-technical stakeholders. It does so by defining software behavior through natural language examples, known as scenarios. These scenarios are specified as a series of steps starting with keywords like Given, When, or Then, describing how a feature may be used from the client's perspective. Tools like Cucumber, SpecFlow, and Behave can interpret these natural language scenarios and execute the corresponding tests, facilitating non-technical stakeholders' understanding and participation in the testing process. BDD encourages stakeholders to write scenarios before implementing features, ensuring a clear understanding of requirements among different teams. These scenarios are then automated to easily verify whether the software meets user expectations. By aligning development with user expectations and promoting continuous collaboration, BDD helps teams deliver high-quality software that meets stakeholders' needs.

While BDD offers many benefits, it also presents challenges. Maintaining and continually writing natural language test steps can be time-consuming and complex, especially for sophisticated software. As the software evolves with new features and dependency updates, BDD scenarios may require significant modifications. A common pitfall in BDD is incorporating too many technical test steps, which offer little value to non-technical team members. Often, these natural language test steps are not even read by those outside development and testing, diminishing BDD's effectiveness. Another issue is that team turnover can lead to redundant test steps and code, increasing the cost of executing test steps and introducing "test smells" into the test suite, making the codebase harder to maintain. When dependencies are updated or new features are added, instead of updating a single test or piece of code, multiple updates might be necessary. This reduces testing effectiveness and slows feedback to developers. Slower feedback can result in bugs leaking into production, lowering software quality and worsening the user experience. Therefore, identifying reusable features in test cases is imperative to reduce the cost of test execution.

Manually identifying redundant test steps and test cases would require extensive domain knowledge over the entire test suite. If the application is large, this can prove to be a difficult task. Simple text matching is not sufficient due to differences in vocabulary as similar test steps may be written using different words but semantically mean the same thing. Therefore, we explore using similarity and clustering based approaches to automate this task. There are several problems in software engineering that could benefit from this task apart from BDD, such as general document similarity, duplicate requirements detection, and plagiarism detection.

Previous studies in this field have manually mined private codebases to identify redundant test cases and steps. This approach demands significant effort to thoroughly examine test cases and steps and determine their redundancy. Besides the considerable time and effort required, this method presumes the codebase contains numerous redundant test cases. However, in smaller codebases, redundancy may be minimal. This study differs, in that, instead of manually identifying similar test steps and test cases, we use similarity algorithms to see whether these algorithms can capture enough semantic meaning from the test cases to cluster them together. Since similar test cases are already grouped within the same feature file, our goal is to see if similarity algorithms can effectively cluster test cases into the same feature.

Rather than focusing on finding redundant test cases, we investigate whether similarity algorithms can cluster similar test cases. This approach can be applied to large software applications with thousands of test cases and steps to group similar tests together. When multiple testers are involved, they might not realize that a new test case they are working on fits into an existing feature. Therefore, a method to notify a tester of redundant test code could help reduce maintenance and execution costs. This study explores how word embeddings, similarity algorithms, and clustering can be used to group similar tests, reduce redundancy, and improve maintainability.

# **Background**

Before calculating the similarity between test cases, the test case (including both the step name and underlying step definition) need to be embedded into a numerical vector. There are a variety of embedding techniques such as Word2Vec, BERT, and TF-IDF to name a few. In this study, we use TF-IDF to embed the test cases before calculating similarity. We explore four different similarity algorithms: Normalized Compression Distance (NCD), Cosine Similarity, Manhattan distance, and Euclidean distance. We use Mean Average Precision and Mean Reciprocal Rank to quantify the results and K-means clustering for clustering the similarity calculations.

# **2.1 Text Embedding**

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure of how important a word in a document is relative to a collection of documents (corpus). It is often used in natural language processing tasks to convert textual information into numerical features used by machine learning algorithms. The Term Frequency (TF) is the number of times a word appears in a document divided by the total number of terms in the document.

A higher term frequency indicates higher importance within that document. The Inverse Document Frequency (IDF) is the logarithm of the total number of documents in the corpus divided by the number of documents containing the word.

A higher IDF value indicates that the word appears infrequently across the corpus. The TF-IDF value for a word/term t in a document d is the product of its TF and IDK scores:

The combined TF-IDF score can be calculated for each term to build a matrix that emphasizes important terms rather than simply frequent terms, which can help cluster similar documents and classify them into categories.

# **2.2 Text Similarity**

Once our text is represented with a vector of numbers, we can apply machine learning and similarity algorithms to potentially find features that can be reused in the test cases. Four similarity algorithms are applied to the embedded test cases: Normalized Compression Distance (NCD), cosine similarity, Manhattan distance, and Euclidean distance.

NCD is an equation that measures the similarity between objects (text, files, compressed data) by measuring how much they can be compressed together versus separately. The more similar two objects are, the more they share and thus can be compressed together. NCD is defined as follows:

Where is the compressed size of object , is the compressed size of object , and is the compressed size of the concatenation of and . An NCD value close to 0 indicates high similarity while close to 1 indicates low similarity. The NCD is calculated pairwise between each test case in a repository. NCD can be calculated directly on string and does not need a numerical vector.

Cosine similarity is a measure that calculates the cosine of the angle between two vectors. The cosine similarity value ranges from -1 to 1 where 1 indicates that the two vectors are identical, 0 means that the two vectors are orthogonal (not similar), and -1 indicates that the two vectors are opposite. The cosine similarity between two vectors is defined as:

where is the dot product of vectors A and B, is the Euclidean norm of the vector. The cosine similarity is also calculated pairwise for each test case.

Manhattan distance, also known as L1 norm, is the sum of the absolute difference of each respective component in a vector and is represented by the following expression:

Where and are vectors, which in this case are the test cases embedded as vectors.

The Euclidean distance, also known as L2 norm, is similar to the Manhattan distance except the differences are squared rather than taking the absolute value.

# **2.3 Text Clustering**

Text clustering is a technique used to group a set of documents into clusters, where documents within each cluster are more similar to each other than to those in other clusters. This process helps in organizing and summarizing large volumes of data. K-means clustering is one particular clustering technique where we segment our data into K clusters. The following steps are performed in K-Means clustering:

1. *Initialization* - K initial centroids are chosen randomly
2. *Assignment* - Each document is assigned to the nearest centroid based on distance metric such as Euclidean distance
3. *Update* - New centroids are calculated by averaging the vectors of the documents in each cluster
4. *Iterate* - Repeat the assignment and update steps until the centroids do not change significantly or a maximum number of iterations has been reached

K-means clustering is efficient and scales well to large datasets but requires the number of clusters to be specified in advance and assumes clusters to be roughly spherical and of similar size. This clustering method is unsupervised since there is no labeled data being used and therefore arbitrary labels are created for each cluster. However, in this case we want to evaluate the predicted cluster with the true test clusters from our dataset. Therefore, we use the Hungarian (Kuhn-Munkres) algorithm to match predicted clusters with the true clusters, thus allowing us to evaluate the performance of the similarity algorithms. The Hungarian algorithm is an optimization algorithm used to solve the assignment problem. It is useful in cluster identification and matching problems where there is a need to assign items to clusters in an optimal way.

# **2.4 Evaluation Metrics**

To evaluate the predicted clusters against the true clusters, the Mean Average Precision and Mean Reciprocal Rank were calculated for step names, step definitions, and scenario titles. The Mean Average Precision is formally defined with the following formula.

Where is the number of classes and is the average precision of class . In this case, we calculate the precision of each predicted cluster and then average them all together.

The Mean Reciprocal Rank is often used in information retrieval to evaluate the effectiveness of a system that returns a ranked list of possible responses to a query. It is defined as:

Where is the number of queries and is the rank position of the first relevant result for the i-th query. In the context of clusters, we take each test case in each test cluster and find which cluster it was predicted to be in by the clustering algorithm. The index of the cluster is its rank, and the Mean Reciprocal Rank is interpreted as a measure of how spread out a test cluster is among the predicted results. The more spread out the clusters are, the smaller the MRR value and the worse the performance of the algorithm. The less spread out the better since if each test case inside a test cluster all belong to a single predicted cluster then it has perfectly predicted the test cluster.

# **Related Work**

There are a few related works that also used similarity algorithms, natural language processing techniques, and clustering in relation to natural language-based software testing.

Li et al [\*] propose an approach to cluster similar natural language test steps together such that the test steps in each cluster can be mapped to the same test API method. The approach uses a domain-specific word embedding technique along with using the Relaxed Word Mover’s Distance metric to analyze the similarity of test steps. The approach also combines hierarchical agglomerative clustering and K-means clustering for manually adjustable clustering results. The baseline approach was simply based on keyword extraction and removal of duplicates. The methods are evaluated on a private, industrial mobile app codebase which shows that compared to the baseline approach, this approach achieves 79.8% improvement on cluster quality and reduces 65.9% of clusters.

Viggiato et al [\*] analyzed both similar test steps and test clusters using text embedding, text similarity, and clustering techniques to help identify redundant test cases. They compared five different embedding techniques such as Word2Vec, BERT, Sentence-BERT, Universal Sentence Encoder, and TF-IDF. Two similarity metrics were used: Word Mover’s Distance and Cosine Similarity. For clustering, they used Hierarchical Agglomerative Clustering and K-Means Clustering. To identify redundant test cases, they parsed through a private codebase and manually identified redundant test steps and test cases. They further processed the test steps by manually inspecting misspelled words, removing stopwords, applying lemmatization, and removing words that only appear once. The results showed that an ensemble technique achieved the best performance with an F-Score of 87.39% while a similar performance could be achieved solely with the Word2Vec embedding technique which resulted in an F-Score of 86.99%.

Chang et al [\*] approach the redundant test case problem in a different angle where test cases are categorized into entities and relations. In their new detection approach, Tscope, they dissect test cases into atomic test tuples with five entities restricted by their associated relations. Each test case is split into entity categories which include Component, Behavior, Prerequisite, Manner, and Constraint. There are also relation categories which include Act, Require, Use, and Satisfy. The dataset, consisting of 10 different private codebases, is parsed and manually judged to decide whether a test case is redundant or not. Each test case is then manually labeled with the entities and relations. BERT embedding is used in combination with a deep learning architecture to create Tscope, which automatically extracts test-oriented entities and associated relations and uses them to conduct similarity comparison between test pairs. Tscope resulted in 91.8% precision, 74.8% recall, and 82.4% F1.

Pan et al [\*] use test code similarity to minimize black box test cases and they differ from the previously mentioned studies as they use evolutionary search algorithms on the underlying code's Abstract Syntax Tree. The AST-based Test case Minimizer relies on four tree-based similarity measures to apply genetic algorithms to minimize test cases. The approach is evaluated on 16 Java projects and was shown to achieve better results compared to FAST-R, another minimization method. Differently from the previously mentioned studies, this study does not focus on natural language based test cases and is simply for general black box testing.

Aranda et al [\*] developed a catalog of transformations to remove seven smells from natural language based test suites. Natural language test smells include behaviors such as unverified actions, eager actions, conditional tests, and more. They use the spaCy library to develop a tool to automatically detect smells based on action verbs and sentence connectors. The tool was evaluated on real-practice tests from the Ubuntu OS and resulted in an F-score of 83.7% for identifying smells.

# **Dataset**

Many of the previous studies on this topic use private codebases that were only accessible by the authors. In this study, we only use publicly available tests from open source repositories on GitHub. This study also differs in that we look at multiple codebases with step definitions written in different languages and different styles. The repositories also span many types of software such as command-line tools, software development kits, web development frameworks, and desktop software. The repositories also range in size from 1,000 - 4,000,000 lines of code and between 2 and 1,000 contributors. The repositories are summarized in the table below. The repositories were found by searching for “.feature” files (Gherkin language) indicating which repositories are using a Behaviour Driven Development (BDD) framework such as Behat or Cucumber to run End-2-End tests.

**Table 1: GitHub repositories and details regarding repo and test cases.**

|  |  |  |  |
| --- | --- | --- | --- |
| Repository Name | # of Test Cases | LoC | Contributors |
| Jekyll | 286 | 24,384 | 1,032 |
| Aws-sdk-js | 189 | 3,451,423 | 207 |
| Hub | 552 | 14,837 | 244 |
| Keygen-api | 5,767 | 93,444 | 2 |
| Trema | 114 | 1313 | 16 |
| Aws-sdk-ruby | 156 | 4,021,900 | 215 |

The number of test cases is the total number of scenarios extracted from the feature files. A group of similar scenarios will often be in the same feature file and run consecutively after one another. Each repository consists of feature files which consist of test(s) written in natural language (Gherkin), as well as the underlying step definitions for each step in the test. The feature files can be easily found by simply searching and collecting all of the files ending with the .feature extension. Step definitions are usually specified using keywords such as “Then”, “When”, and “Given” followed by a regular expression used to match the step definition with the step in the test case. To find all of the files containing step definitions, a Python script was used along with a regular expression to find files containing the aforementioned keywords. After identifying each file containing step definitions, the file was parsed into an Abstract Syntax Tree (AST) to extract the function inside each step definition. For each step in a test case, the corresponding step’s underlying function was matched. This was done for all of the steps in a test case, and for all test cases in each repository. There are 7,064 total test cases collected across 6 GitHub repositories. Here is a sample of what the “raw” dataset looks like:

# **Methodology**