Using Natural Language Processing Techniques to Improve Manual Test Case Descriptions

Markos Viggiato, Dale Paas, Chris Buzon, Cor-Paul Bezemer*

ABSTRACT

Despite the recent advancements in test automation, software testing often remains a manual, and costly, activity in many industries. Manual test cases, often described only in natural language, consist of one or more test steps, which are instructions that must be performed to achieve the testing objective. Having different employees specifying test cases might result in redundant, unclear, or incomplete test cases. Manually reviewing and validating newly-specified test cases is time-consuming and becomes impractical in a scenario with a large test suite. Therefore, in this paper, we propose an automated framework to automatically analyze test cases that are specified in natural language and provide actionable recommendations on how to improve the test cases. Our framework consists of configurable components and modules for analysis, which are capable of recommending improvements to the following: (1) the terminology of a new test case through language modeling, (2) potentially missing test steps for a new test case through frequent itemset and association rule mining, and (3) recommendation of similar test cases that already exist in the test suite through text embedding and clustering. We thoroughly evaluated the three modules on data from our industry partner. Our framework can provide actionable recommendations, which is an important challenge given the widespread occurrence of test cases that are described only in natural language in the software industry (in particular, the game industry).

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Game testing, Natural language processing, Language modeling, Association rules, Clustering

1 INTRODUCTION

Software testing is a fundamental and widely-performed, yet costly, activity for quality assurance of a software system [12, 13, 16]. Despite the recent advancements in test automation, software testing often remains a manual activity in industry, such as in the gaming industry where developers face challenges to automate tests [30, 32]. In a manual testing scenario, the testing and test case design activities require even more effort and time from the development team and the Quality Assurance (QA) engineers (testers), which makes testing even more costly for the company.

Manual test cases are often described in natural language and consist of a sequence of one or more steps, which are instructions that need to be executed by the tester to achieve the test case objective (e.g., to test a new feature of the system). Those test cases are often defined by employees from different departments, such as QA engineers or developers, which may result in redundant (i.e., test cases with the same testing objective), unclear/ambiguous, or even incomplete (e.g., when necessary steps are missing from a test case description) test cases as the system evolves and the test suite grows [34]. Problematic test case descriptions can hinder the manual testing activity efficiency and effectiveness, and can also affect the performance of techniques such as Natural Language Processing (NLP) techniques and text clustering [22].

Having several employees manually review new test cases (e.g., to check if they are clear, unambiguous and complete) and identify redundant test cases is time-consuming and becomes impractical in a scenario with a large test suite in which test cases are constantly added to it. Also, prior work has indicated the need for automated approaches that can be integrated into the testing process of video games [32]. Therefore, an automated approach to analyze the test cases specified in natural language and provide actionable insights to improve their descriptions is needed to support QA and developers and help make testing more efficient and effective.

In this paper, we propose an automated framework for providing feedback on how to improve a new test case that is specified in natural language. In particular, we discuss three modules for analysis that we implemented so far for our framework. These analysis modules provide the following recommendations:

- Recommendations to **improve the terminology** of a new test case description based on existing test case descriptions through language modeling.
- Recommendations of **potentially missing test steps** for a new test case through frequent itemset and association rule mining.
- Recommendations of **similar test cases** that already exist in the test suite through a similarity detection technique that we proposed in a prior work [22, 35]

All three analysis modules were thoroughly evaluated and optimized for the data from our industry partner (Prodigy Education),[1] and we provide access to the source code of the experiments that we performed.[2] The framework's analysis modules are unsupervised (i.e., they do not require manually labelled data or human intervention). In this work we use the term "existing test cases" to refer to all test cases that are already part of the test suite and "new test case" to refer to a newly-specified test case that is not yet part of the test suite. Our framework should be used right after a new test case is specified to analyze it and provide feedback to improve

*Markos Viggiato and Cor-Paul Bezemer are with the Analytics of Software, Games and Repository Data (ASGAARD) lab at the University of Alberta, Canada. E-mail:{viggiato, bezemer}@ualberta.ca
Dale Paas and Chris Buzon are with Prodigy Education, Toronto, Canada. E-mail: {dale.paas, christopher.buzon}@prodigygame.com.

[1] https://www.prodigygame.com/main-en/
[2] https://github.com/asgaardlab/21-markos-test_case_improvement_framework-code

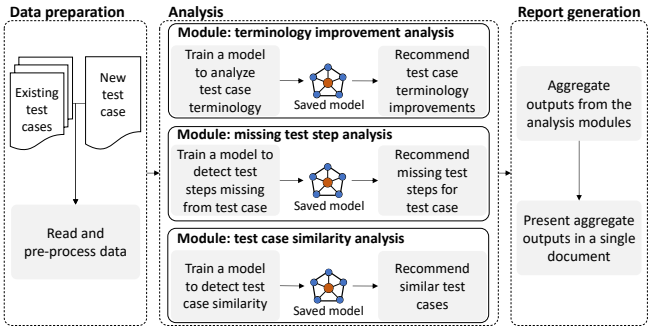


Figure 1: Our automated framework for analysis and feedback of test cases in natural language.

the test case description. Then, the improved test case can be added to the test suite and manually executed.

The goal of our framework is to help QA engineers and developers to reduce the time and effort needed for manual testing by improving the overall quality of test cases that are described in natural language. The framework also supports the creation and maintenance of a high-quality, more consistent and more standardized test suite. In particular, our framework can be useful and benefit new employees who do not yet have much knowledge about the existing test suite. Furthermore, a more consistent test suite can reduce the challenges when automating tests in the future [22] as the overall quality and consistency of the test suite will be higher.

The remainder of the paper is structured as follows. In Section 2, we explain our framework. In Sections 3, 4, and 5, we detail the approaches that were used to implement the framework's analysis modules, with the performed experiments and the results. We then present related work and threats to validity in Sections 6 and 7. Finally, we conclude the paper in Section 8.

2 OUR AUTOMATED FRAMEWORK FOR ANALYSIS AND FEEDBACK

Our automated framework provides feedback to improve the description of the test cases designed to test the *Prodigy Math game*, which is a proprietary, online, web-based educational math game with more than 100 million users around the world. The game has a curriculum-aligned educational content and features over 50,000 math questions spanning Grade 1-8. It is an RPG-style game, which means that players play the role of a character (a wizard) in the Prodigy world and can go to the several different world zones available in the game. As the players answer math questions, their wizards can evolve, learn new spells, and acquire new equipment and in-game items. While the game is available to every student, there is an optional membership subscription, that allows members to have an increased level of character customization, level up faster, among other benefits not available to non-members. The membership is not required to access the in-game curriculum-aligned content.

Our framework consists of three main components, which correspond to the steps that are performed: data preparation, analysis, and report generation. The framework's analysis component consists of individual configurable modules. Each module implements an approach that provides a different capability regarding automated analysis and feedback for test cases that are described in natural language. New modules with new capabilities can be easily added to the framework. Figure 1 presents an overview of our automated framework, which currently consists of the following components and modules:

- **Data preparation** component
- **Analysis** component, which currently contains modules for the following: (1) terminology improvement, and analyzing (2) missing test steps and (3) similar test cases.
- **Report generation** component

The three implemented modules within the analysis component were driven based on our experience at Prodigy and reports from experienced QA engineers and developers that indicated the need to support these types of test case improvements. Furthermore, prior work highlighted the need for more consistent and standard test case descriptions in a manual testing scenario and automated approaches to better support the testing process of games [22, 32].

Our framework first reads and pre-processes the data from existing and new test cases (data preparation component). Then, the pre-processed data is fed into one or more modules (analysis component) and the framework generates a report with the modules' outputs (report generation component). Each analysis module takes the data through a *training* and an *inference* phase. In the *training phase*, users can train new models using the pre-processed existing test cases. In the *inference phase*, users can use the trained models to analyze a new test case. The modules are independent from each other and can be enabled or disabled depending on the desired analysis that the users wish to perform. Next, we demonstrate each framework's component using the test case examples in Table 1.

2.1 Data preparation component

This component loads and pre-processes the data. The input to our approach consists of unprocessed test cases that are written in natural language: there is no source code available for these test cases. Each test case contains one or more test steps, which each give an instruction that must be manually performed by a human tester. Table 1 presents examples of two test cases TC1 and TC2 from the Prodigy game. TC1 is already in the test suite and TC2 is about to be added to it (and hence is not used to train the models in the analysis modules). Each test case has a name, an objective, and one or more test steps.

We applied several pre-processing steps to each test case's name, objective and test step(s). We used tokenization to transform the sentences into lists of words. We then removed stop words (e.g., "of" and "the") as they do not add meaning to the text. Finally, we converted all words to their root form (lemmatization), such as "playing" to "play", to have more consistent terminology. The data preparation component can be adapted if users wish to apply other pre-processing steps for an analysis module.

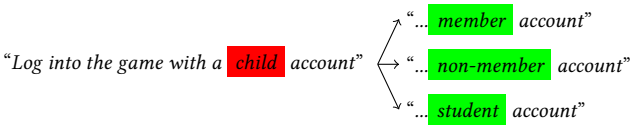
2.2 Analysis component

2.2.1 Module: terminology improvement analysis. This module uses the pre-processed test cases to train models to analyze the terminology of test cases. The models are then used to recommend improvements by identifying words in the description that could be

Table 1: Examples of test case descriptions from the Prodigy game.

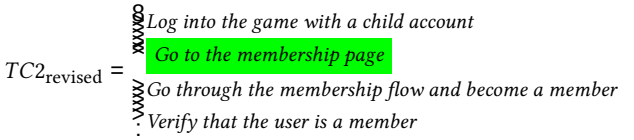
Test case name	Test case objective	Test step identifier	Test step
Membership purchase (TC1)	Verify the membership flow through the HUD (Heads Up Display)	TS1	Log into the game with a non-member account
		TS2	Go to the membership page
		TS3	Click on the member icon in the HUD
		TS4	Click on "Continue to buy a membership"
		TS5	Go through the membership flow and become a member
		TS6	Verify that the user is a member
Membership flow (TC2)	User successfully purchases membership	TS7	Log into the game with a child account
		TS8	Go through the membership flow and become a member
		TS9	Verify that the user is a member

replaced by more likely alternatives, based on their usage in existing test cases. For our example test case (TC2) in Table 1, the top-3 recommendations of this module are to change the word *child* to *member*, *non-member*, or *student* in test step TS7:



Using the original word *child* makes the test step unclear (as we are not sure which type of child account should be used as there are different ones) and would require further clarification with other QA engineers or developers. For example, replacing *child* with *non-member*, would be more appropriate as the tester would be aware that an account of the *non-member* type must be used to verify if a non-member can purchase the membership.

2.2.2 Module: missing test step analysis. This module analyzes how test steps of the existing test cases appear together to assess a new test case's completeness. The module builds a model that recommends potentially missing test steps for the new test case based on test steps that frequently appear together across the existing test cases. For test case TC2, this module recommends to add the test step TS2 ("*Go to the membership page*"):



TS2 ("*Go to the membership page*") appears in the test case TC1 but is missing from the new test case. Adding TS2 to TC2 can help the tester to execute the test more efficiently as a clearer direction is given (instructing the tester to go to the membership page).

2.2.3 Module: test case similarity analysis. This module trains a model with the pre-processed descriptions of existing test cases to identify and retrieve test cases that have a testing objective or test steps which are similar to the ones of a new test case. For test case TC2, this module retrieves TC1 as a similar test case. TC1 has a similar testing objective as TC2 (which is to go through the membership flow and assure that a user can purchase the membership) and similar, but more detailed test steps, which can be help to improve the new test case. Identifying similar test cases that are already in the test suite also helps avoid adding redundant test cases.

2.3 Report generation component

The purpose of this component is to aggregate the outputs of each used analysis module and present the results in a single report to QA engineers and developers.

2.4 Using the framework in practice

All the functionalities of our framework are provided through a web API, which can be used, for instance, to build other applications that rely on our framework. We are currently working to integrate our framework with Prodigy's data warehouse and cloud infrastructure through a web application that can be easily used by Prodigy's QA engineers and developers. The application allows users to perform the automated analysis and visualize the generated report with the results in a usable way. Users can also choose which module they want to execute and provide additional configurations to the techniques used for the analysis (e.g., if our recommendations of similar test cases are too broad, users can increase the similarity threshold and less recommendations will be provided). Furthermore, the web application allows users to automatically apply the recommended changes to the new test case, making the use of our framework more efficient. We discuss the experiments to train and evaluate the models for each module in Sections 3, 4, and 5.

2.5 A description of our dataset

To build the models and perform the experiments for each module that we previously discussed, we collected all the 3,323 test case descriptions written in natural language from the Prodigy test suite. The test cases under study were manually designed to test the *Prodigy Math game*. In total, the test cases in our data set contain 15,644 steps, with an average of 4.82 test steps per test case and a vocabulary size of 2,701 unique words across all the test cases.

3 THE TERMINOLOGY IMPROVEMENT ANALYSIS MODULE

Our approach for recommending terminology improvements consists of using statistical and neural language models to analyze the description of a test case and identify words that should be replaced by more likely words. We train unidirectional and bidirectional

n-grams, BERT-based models, and a combination of both types. We then use the characteristics of the sentences in the test case description to choose the most suitable model to identify words in the description that can be replaced by more likely words. Figure 2 presents an overview of our approach to recommend terminology improvements to test cases, which consists of a training phase, evaluation of the models, and inference phase as we explain below.

3.1 Training phase

The test case descriptions in our dataset have sentences with very different lengths, ranging from 3 words to more than 30 words. Furthermore, even though many test cases have a similar terminology, as new features are included in the game, new test cases with a terminology that is different from the existing ones are added to the test suite. Based on the characteristics of our data, we chose two different types of language models to be evaluated: statistical models (n-grams) and neural models (BERT-based models).

Statistical models such as the n-gram capture regularities in the corpus used to build the model and perform well with highly predictable corpora that have repetitive patterns [18], which often appear in our data. N-gram is a popular generative statistical language model that estimates the probabilities based on the frequency with which words appears in the training corpus (a.k.a. the *maximum likelihood estimate*) [7, 18, 19]. For any sequence s of words: $w_1 w_2 w_3 w_4 \cdots w_n$, a common way of estimating the probability of a word is to use a fixed-size window of (n-1) context words. For example, using a bigram, the probability of w_i depends only on w_{i-1} and uses the number of times $w_{i-1}w_i$ appeared in the training corpus relative to the number of times that w_{i-1} appeared:

$$p(w_i|w_1 \cdots w_{i-1}) = p(w_i|w_{i-1}) = \frac{count(w_{i-1}w_i)}{count(w_{i-1})}$$

N-gram models have been traditionally used for the *next word prediction* task, in which only the leftward context words are used to predict the next word (unidirectional n-gram) [19, 23]. However, in our work, we also experiment with n-gram models to perform the *fill-in-the-blank* task as both leftward and rightward context words are available (bidirectional n-gram) [9].

Neural models present several benefits over n-grams. For example, neural models can handle longer dependencies in a sentence, which can be an advantage for the longer sentences in our data. Also, neural models generalize better than statistical models [4, 28], which can be advantageous for cases with an unseen context (e.g., when a new test case with new terminology is added to the test suite). In particular, transformer-based neural language models have shown a higher performance than other types of neural models (e.g., RNN/LSTM) [21] and have achieved the state-of-the-art performance in many NLP tasks [23, 31]. In our work, we use transformer-based neural language models because they outperform other neural architectures and there are several large pre-trained models available [39].

3.1.1 Training n-gram language models. We trained unidirectional and bidirectional unigram, bigram, trigram, 4-gram, and 5-gram models. For n-grams with an order above 1 (bigram, trigram, and so on), a word might appear in a context in the test set that has not appeared in the training set. To avoid having a zero probability prediction and to have a usable prediction, we adopt a simple and effective smoothing technique called *stupid backoff* [6, 18, 19]. In this case, if the model has not seen a certain 5-gram, for example, it can back off from the 5-gram and use the probabilities of the 4-gram, and so on. To handle the cases in which an unknown (out-of-vocabulary) word appears in the test set, we introduce a new token <*unk*> in our vocabulary, which replaces rare words (a random sample of words that occur only once in our corpus). We then estimate the probabilities for the <*unk*> token from its counts just like another regular word [8, 19]. Also, an n-gram model automatically backs off to a lower-order if the length of the context word sequence is smaller than n. For example, when using a unidirectional 4-gram and analyzing the third word of a sentence, there are only two words on the left, so the model uses a trigram (two context words plus the target word). Finally, for the bidirectional n-gram, we estimate the probability of a word w_i by averaging the probability using the leftward words and the probability using the rightward words, as shown below for a bigram:

$$p(w_i|w_{i-1}w_{i+1}) = \frac{p_{\text{left}}(w_i|w_{i-1}) + p_{\text{right}}(w_i|w_{i+1})}{2}$$

3.1.2 Training BERT-based language models. To train our BERT-based language model, we used the *BertForMaskedLM* class from Huggingface [39] with a pre-trained model. To tokenize the data and format it as required by BERT, we used BERT's own tokenizer provided by Huggingface. Finally, similarly to what was originally performed to train BERT from scratch [10], we fine-tuned the pre-trained BERT-based models with the masked language modeling objective by randomly masking 15% of the words in the training data. We evaluated three pre-trained models: *BERT base uncased* (trained on lower-cased English text), *DistilBERT base uncased* (a light transformer model based on *BERT base uncased*), and *BERT large uncased whole word masking* (which was trained using whole word masking). For each of the three pre-trained models, we also evaluated their fine-tuned versions. For simplicity, we will use these names to refer to the used BERT models: *BERT* for *BERT base uncased*, *DistilBERT* for *DistilBERT base uncased*, and *BERT whole word* for *BERT large uncased whole word masking*.

3.2 Evaluation

3.2.1 Evaluation setup. To train and evaluate the language models, we used all the data that we collected (test case name, objective and steps). We shuffled the data and split it into training (80%) and testing (20%) sets. For this approach, a preliminary analysis showed that keeping the stop words and words in their original format (i.e., not performing lemmatization) increases the language models' performance as more context information is available. We used the intrinsic evaluation metric called perplexity [8, 18, 19]. A good language model can capture the patterns and regularities of the training corpus and should be able to predict the words in a new sequence W that comes from the same population as the training one with a high probability. That is, the model should not be "perplexed" by that new sequence. Perplexity (*PP*) can be defined as follows:

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1} \frac{1}{P(w_i \mid w_1 \ldots w_{i-1})}} \quad (1)$$

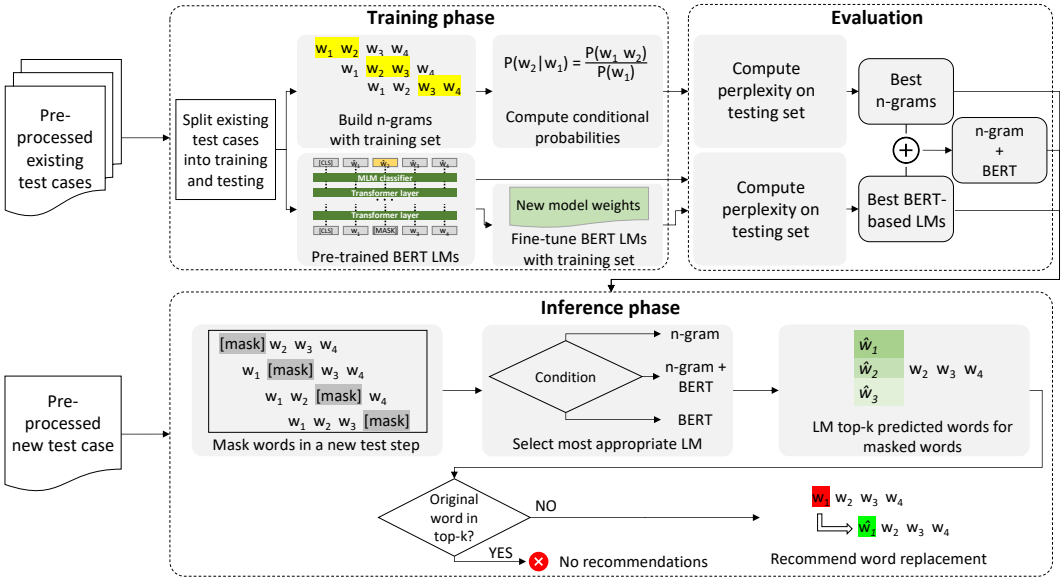
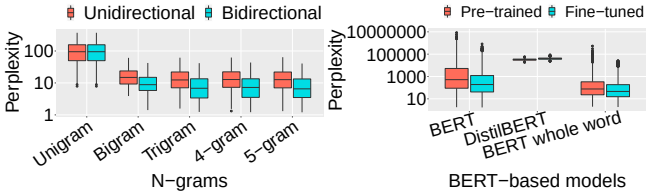


Figure 2: Our approach for recommending terminology improvements with n-grams and BERT-based language models (LMs).

Where W is a sequence of words and P is the conditional probability of w_i given the context words. Since a good model should assign a high probability to a new sequence of words and the perplexity is inversely related to the probability, the better the model, the higher the probabilities, and the lower the perplexity, which leads to a better generalization of the model [5]. We compare the distributions of perplexity for the different models with the non-parametric Wilcoxon rank-sum test [38] and compute the magnitude of the distribution difference with Cliff's delta effect size [24, 33].

3.2.2 Evaluation results. Figure 3 presents the distributions of the perplexity metric for all the evaluated models across the testing set (each data point corresponds to the perplexity of a sentence in the testing set). Figure 3a shows that for unidirectional n-grams, the unigram is the worst n-gram as it presents the highest perplexity median (94.95) and the higher the order of the n-gram, the lower the median perplexity (i.e., the better the model), with the trigram, 4-gram, and 5-gram presenting very similar median perplexities (12.41, 12.79, and 12.68, respectively). The bidirectional n-grams present a similar behavior across different n-gram orders, but with even lower perplexities. When comparing the unidirectional and bidirectional distributions for each n-gram order, the Wilcoxon rank-sum test shows that for all of them, except the unigram, the distributions are significantly different, with a medium Cliff's delta effect size. This shows that the bidirectional n-grams indeed achieve better performance. When we compare the distributions among the bidirectional n-gram models only, the bigram distribution is significantly different from higher-order n-grams, but with a small Cliff's delta effect size. However, there is no statistically significant difference between the distributions of the trigram, 4-gram, and 5-gram models (all with a negligible effect size). In practice, a trigram seems enough in our case, but given the low n-gram computational cost, we use the best-performing model (bidirectional 5-gram) in our approach for test case terminology improvement.



(a) Perplexity of n-gram models. (b) Perplexity of BERT models.

Figure 3: Distributions of the perplexity* metric of the evaluated language models. *Log-transformed perplexity for better visualization.

Figure 3b presents the perplexity distribution for the BERT-based models. For the pre-trained models, *DistilBERT* presents the highest median perplexity (32k), followed by *BERT* (520.64) and *BERT whole word* which has the lowest median perplexity (76.16). Fine-tuned models present a similar behavior, but with lower perplexities, with *BERT whole word* also having the lowest median perplexity (45.97). Except for *DistilBERT*, fine-tuning improves the model's performance by reducing the perplexity of the model as it sees new sequences. When comparing the distributions between the pre-trained and fine-tuned models, the Wilcoxon rank-sum test shows that for all the three types of models there is a statistically significant difference between the distributions, with a large effect size for *DistilBERT*, a small effect size for *BERT*, and a negligible effect size for *BERT whole word*.

3.2.3 Comparing N-gram and BERT-based language models. Since we cannot use perplexity to compare models built with different vocabularies [8, 19], we used a recommendation system-like metric (*accuracy@k*) to compare the best-performing n-gram (bidirectional 5-gram) to the best-performing BERT-based model (fine-tuned *BERT*

Table 2: Median *accuracy@k* (acc@k) for combinations of different types of language models. **BERT whole word* refers to the *BERT large uncased whole word masking* model.

Language model	Entire testing set			Short test step sentence			Long test step sentence		
	acc@3	acc@5	acc@10	acc@3	acc@5	acc@10	acc@3	acc@5	acc@10
Bidirectional 5-gram	**0.67**	**0.71**	**0.75**	**0.34**	**0.5**	**0.5**	0.81	0.84	0.86
Fine-tuned BERT whole word*	0.17	0.22	0.25	0	0	0.17	0.21	0.27	0.30
N-gram$_{\text{unseen-context}}$ + BERT	**0.67**	0.70	**0.75**	**0.34**	**0.5**	**0.5**	**0.84**	**0.85**	**0.88**

whole word). We compared their performance for our task (word recommendation). To compute the *accuracy@k*, we first translate the problem of word recommendation to a binary problem. Suppose we are analyzing a test step composed of a sequence of words $w_1 w_2 \cdots w_n$. We mask one word at a time (i.e., replace the word by the *[mask]* token, as shown in Figure 2) and get the top-k most likely words predicted by the language model for each masked word. For the top-k words predicted by a model for a single masked word, if the original word is among the k predictions, we consider it a correct recommendation (1), otherwise, we consider it a wrong recommendation (0). Then, we have a correct (1) or wrong (0) recommendation for each masked word in a test step sentence, and compute the *accuracy@k* for the whole test step sentence as: $\frac{\text{count(correct suggestions)}}{\text{count(all suggestions)}}$. We evaluated the bidirectional 5-gram and the fine-tuned *BERT whole word* models on the entire testing set using top-3, top-5, and top-10 suggestions. Table 2 shows the median *accuracy@k* for both models across the entire testing set in the *Entire testing set* column (we computed the *accuracy@k* for each test step in the testing set and calculated the median), for which the bidirectional 5-gram performed better than *BERT whole word* for $k \in \{3, 5, 10\}$.

To further understand the scenarios in which the bidirectional 5-gram and fine-tuned *BERT whole word* models fail, we manually inspected a sample of test steps for which either the n-gram has an *accuracy@10* of zero and BERT has an *accuracy@10* of one, or vice-versa. We focused on the cases where one model is totally unable to provide a correct recommendation (even recommending the top-10) while the other provides all the recommendations correctly to be able to identify the characteristics that might cause one model to fail but not the other. This allows us to better understand in which scenarios we can combine both models. We made two observations: (1) the n-gram model performs very well when context words were seen during training but the performance degrades when the model needs to back off until reaching the unigram (because of unseen context) and the n-gram's prediction probability is low (even for the first-ranked predicted word) and (2) BERT struggles to make correct predictions when the test step has very domain-specific terms and is short (in terms of number of words).

Using those two observations with the fact that BERT usually outperforms other language models for long sentences, we also evaluated a combination of the bidirectional 5-gram with *BERT whole word* for different lengths of test steps. Using the distribution of number of words per test step in our data, we split the testing set into two groups: short test step sentences (less than 5 words, which correspond to the bottom 20% of the testing set) and long test step sentences (more than 12 words, which corresponds to the top 20% of the testing set). To combine the bidirectional 5-gram with *BERT whole word*, we adopt the following procedure: for each masked word in a test step sentence, we verify if the n-gram backed-off until the unigram to make the prediction (i.e., if the n-gram found an unseen context) and if the n-gram probability for the first-ranked word is lower than 0.5 (empirically defined). If those conditions occur, we assume that the BERT predictions are more reliable (since in an unseen context, more generalizable models, e.g. BERT, perform better) and use them. Otherwise, we keep the n-gram predictions. Our goal is to evaluate if switching to the predictions made by *BERT whole word* boosts the overall performance of word recommendation for different test step sentence lengths.

Table 2 shows the performance of the combined models (N-gram$_{\text{unseen-context}}$ + BERT) and how it compares to each individual model's performance for all the sentence length scenarios. Using both the entire testing set or only short sentences, the performance of the bidirectional 5-gram is superior than *BERT whole word*'s performance for the top-3, top-5, and top-10 predictions. That is, using the combined N-gram$_{\text{unseen-context}}$ + BERT does not increase the performance. However, for longer test step sentences, the combined the models increases the performance compared to each model's individual performances. For the top-3 predictions, the *accuracy@3* increased from 0.81 to 0.84 (almost 4%), while the *accuracy@5* increased from 0.84 to 0.85 (around 1.2%) and the *accuracy@10* increased from 0.86 to 0.88 (around 2.3%).

3.3 Inference phase

Finally, in the inference phase, we apply the best-performing n-gram and BERT-based models (bidirectional 5-gram and *BERT whole word*) to analyze the test steps of a new test case and recommend improvements if necessary. We follow a similar process as we did to compare the n-gram to the BERT model: we mask each word at a time in the test step sentence and get the top-k predictions from the n-gram. Then, we verify if (1) the n-gram backed off to the unigram, (2) the n-gram has a prediction probability less than 0.5 for the first-ranked word, and (3) the sentence length is above 12 words. If all the three conditions occur, we use the bidirectional 5-gram combined with the *BERT whole word* model, otherwise we use only the bidirectional 5-gram. Then, if the original word is among the top-k predicted words, the most appropriated word is already being used, so we do not recommend any changes. Otherwise, we present the recommendations to the user. Note that we filter out stop word-related recommendations as they do not meaningfully improve the test case descriptions.

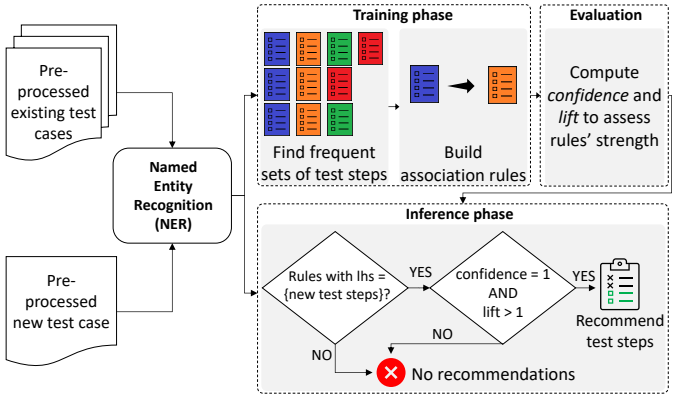


Figure 4: Our approach for recommending missing test steps using association rules.

4 THE MISSING TEST STEP ANALYSIS MODULE

Our approach for recommending test steps that are missing from test cases first trains a model to identify sets of test steps that frequently appear together in existing test cases and then builds association rules based on those sets. The high-confidence rules are then used to recommend test steps that are missing from a new test case. Figure 4 presents an overview of our approach for recommending missing test steps, which consists of a training phase, evaluation of the model, and inference phase as we explain below.

4.1 Training phase

4.1.1 Named Entity Recognition (NER). Our approach first trains a Named Entity Recognition (NER) model to identify proper names of game assets (e.g., in-game items and game zones) in the test steps and replace the assets' names by the related entity. In our test cases, similar test steps are executed to test different assets in the game. For example, suppose the following steps are used to test if a user can purchase item A: ["*Log into the game*", "*Purchase item A*", "*Verify item A is part of the student's asset list*", "*Log out of the game*"]. Now, suppose that a new item B is added to the game and a new test case is added to the test suite to test if a user can purchase item B: ["*Log into the game*", "*Purchase item B*", "*Verify item B is part of the student's asset list*", "*Log out of the game*"]. The second and third steps of test cases for A and B are textually different but have the exact same meaning (the tester performs the same action just with different items). Those steps are considered different items when we use a frequent itemset technique. By replacing the item names (*A* and *B*) with a placeholder that represents the entity (e.g., *asset_item*), the second and third steps become the same item for the mining technique and we can successfully identify the frequent test steps and association rules. By using the NER model, our approach for recommending missing steps becomes agnostic to such named entities and flexible to support the evolution of the game since different assets (e.g., items) are frequently added to the game. Note that a pure keyword-based search is infeasible since asset names might appear written differently in test cases (e.g., a mix of lowercase and uppercase, different spacing, etc). Furthermore, the list of keywords would need to be constantly updated. In contrast, a trained NER model is capable of identifying asset names with a high accuracy (including newly-added entities) based on the learned textual patterns and sentence structure. Our trained NER model was obtained by customizing the NER model provided by Spacy.[3]

4.1.2 Finding frequent sets of test steps and building test step association rules. Frequent itemset mining is the process of finding sets of items that occur together frequently across different transactions [1, 2]. Using frequent itemsets, we can build association rules which have the form $\{X\} \rightarrow \{Y\}$, where X (the antecedent) and Y (the consequent) represent sets of one or more items that occur together. Our goal is to discover sets of (one or more) test steps that appear together across different test cases. Therefore, we mapped the transactions to test cases and the items to test steps. To obtain the frequent sets, and as the majority of the test steps does not occur very frequently across different test cases, we empirically set the minimum support (minimum frequency with which the sets must occur in the test cases to be considered frequent) to 0.005. This means that every test step set that occurs in at least 0.5% of the test cases is considered a frequent set. Using the sets of test steps that appear together, we build association rules to recommend missing test steps from a new test case. In this work, we train a model that uses the popular and efficient *FP-Growth* (Frequent Pattern Growth) algorithm [14, 15] to mine frequent itemsets and association rules. *FP-Growth* is suitable for our work since we use a low minimum support threshold, for which *FP-Growth* is very efficient [14].

4.2 Evaluation

4.2.1 Evaluation metric. To evaluate the strength of the built rules, we focus on the confidence and lift metrics. The confidence of a rule corresponds to the conditional probability of the consequent occurring (the right-hand side of the rule) given that the antecedent occurred (the left-hand side of the rule). Even though the confidence metric already indicates the rules' strength, it might be misleading in scenarios of a highly frequent consequent, in which the confidence would be misleadingly high. Therefore, we also use the lift metric to asses the rules' strength and interestingness [3, 26]. The lift of a rule $\{X\} \rightarrow \{Y\}$ represents how much the probability of Y occurring with the knowledge that X occurred (i.e., the conditional probability of Y given X) changes related to the occurrence frequency of Y alone. In practice, a lift above 1 indicates that the occurrence of X has a positive effect on the occurrence of Y.

4.2.2 Evaluation setup. Our evaluation setup consists of simulating the process of designing a new test case and adding it to the test suite. We assume that a certain number of test cases are already in the test suite and use those test cases to build the association rules. In our case, we selected the first 2500 test cases in our data (about 75% of the data) to build the rules. We then applied the built rules to the 2501^{th} test case, which we suppose is a new one. In the next iteration, we added the 2501^{th} test case to the test suite, built the rules with those 2501 test cases, and applied the rules to the 2502^{th} test case. This process continued until we reached the last test case (the 3323^{th} one).

[3] https://spacy.io/

For each iteration, we computed the accuracy of the rules' recommendations for a new test case to verify how often the recommended test steps are correct. To do this, for each new test case, we removed one of its test steps at a time, applied the rules to the remaining test steps, and checked if the removed test step was among the test steps recommended by the rules. If it was, the rule made a correct suggestion (1), otherwise it was a wrong suggestion (0). Then, we computed the accuracy (proportion of correct suggestions) using all the selected rules. We followed this process for every test step in a new test case and computed the average and median accuracy for the whole test case. Note that we only selected the rules that have a minimum confidence (*min_confidence*) and a minimum lift (*min_lift*). For our experiments, we used a *min_confidence* of 0.5 and a *min_lift* above 1, and a stricter criteria with a *min_confidence* of 1 (the highest possible) and a *min_lift* above 1 as well.

4.2.3 Evaluation results. Using a *min_confidence* of 0.5 with a *min_lift* above 1, we obtained 1,060 association rules to recommend missing test steps for a new test case. Those rules have an average accuracy of 0.72 (and a median of 1) across all the new test cases as we explained in Section 4.2.2. This means that, on average, the recommendations by the rules are correct 72% of the time per test case. For a *min_confidence* of 1 with a *min_lift* above 1, we obtained 475 association rules, which is less than before as we applied a stricter *min_confidence*. Those rules have an average accuracy of 0.98 (and a median of 1) across all the new test cases, which means that, on average, the recommendations by the stricter rules were correct 98% of the time per test case.

4.3 Inference phase

Finally, we use the built association rules with high confidence and lift metrics with the test steps of a new test case to recommend test steps that are potentially missing from the new test case. To be consistent, we also apply the trained NER model to the test steps of the new test case to identify and replace game assets' names. We then use two criteria to select strong rules to be used. First, we only select the rules for which the antecedent (left-hand side) matches exactly to the set of test steps of the new test case since we want to suggest other test steps that occurred together with the newly-specified ones. Second, we select the best-performing rules, i.e., only rules with a confidence of 1 (the highest confidence possible) and a lift metric above 1. These criteria help us to ensure we are using strong rules to provide suggestions and reduce false positives.

5 THE TEST CASE SIMILARITY ANALYSIS MODULE

Our approach for recommending similar test cases was proposed in our prior work [35]. The approach consists of two stages: (1) clustering similar test steps using text embedding, text similarity, and clustering techniques (test step-level stage), which is based on the work by Li et al. [22] and (2) identifying similar test cases using the clusters of test steps (test case-level stage). Figure 5 gives an overview of how our approach for identifying similar test cases was integrated as an analysis module which consists of a training phase, evaluation of the models, and an inference phase as we explain below. In this section, we give a concise overview of the approach that was presented in detail in our prior work [35].

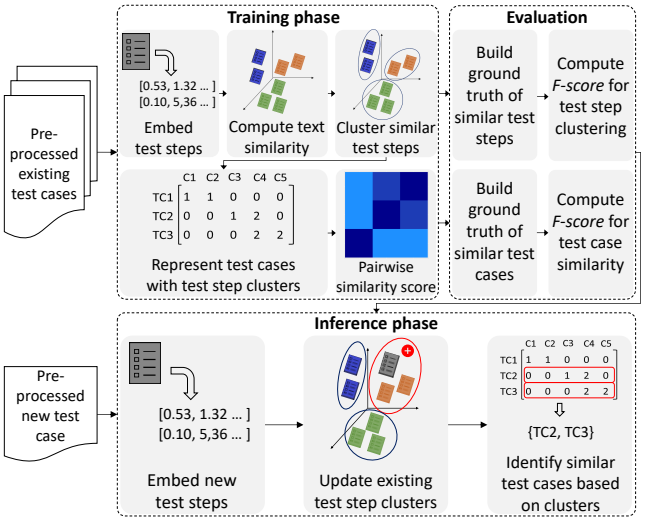


Figure 5: Our approach for recommending similar test cases using text embedding and clustering techniques.

5.1 Training phase

Our approach starts by transforming the test step sentences into one or more numeric vectors (text embedding), which is necessary to apply a machine learning algorithm [37]. The pairwise distance between test step embeddings is then computed, which we use to capture the similarity between the test steps. In particular, embeddings that are close in the embedding space should represent similar test steps. Finally, our approach leveraged the computed distance to identify clusters of similar test steps (i.e., test steps that have a small distance between them should belong to the same cluster).

In the second stage, our approach leverages the obtained clusters of similar test steps together with the test case name to identify similar test cases. The approach first obtains a numeric representation (i.e., a vector) for each test case based on the clusters to which the test steps of that test case belong, as shown in Figure 5. Then, the pairwise similarity between test cases is computed (which we call the *test step cluster-based similarity score* since it is computed using the test step clusters). Next, to incorporate knowledge from the test case name, the approach embeds the test case names and computes their pairwise similarity (which we call the *test case name-based similarity score* since it is compute using the test case names). Finally, the approach computes a final similarity score which is a weighted average between the *test step cluster-based similarity score* and the *test case name-based similarity score*. In our prior work [35], we thoroughly evaluated the described approaches with several different techniques using the data from our industry partner.

5.2 Evaluation

Based on our prior work [35], we selected the best performing approach for clustering similar test steps, which uses Word2Vec [27]

to embed the test steps, the Word Mover's Distance (WMD) metric [20] to compute the similarity between test step embeddings, and K-means [11] to cluster the test steps. We also selected the best-performing approach for identifying similar test cases, which uses cosine to compute the similarity between test cases' numeric representations with an optimal threshold of 0.75. This means that if an existing test case has a final cosine similarity score of more than 0.75 with the new test case, the existing test case is considered a similar test case. Furthermore, our prior work indicated that the optimal balance between the *test step cluster-based similarity score* and the *test case name-based similarity score* is 50%.

5.3 Inference phase

Finally, in the inference phase, we use the best-performing models to cluster similar test steps (Word2Vec + WMD + k-means) and identify similar test cases to retrieve the existing test cases that are similar to the new test case. Our approach starts by embedding the test steps of the new test case using Word2Vec. Then, the existing test step clusters, obtained with the test step clustering approach, are updated with the new test steps (using the distance between their embeddings). Finally, the approach to identify similar test cases is used to retrieve all the existing test cases that have a cosine similarity score higher than 0.75 compared to the new test case.

6 RELATED WORK

In this work, we apply several NLP techniques to automatically analyze and provide feedback to improve the description of test cases specified in natural language. Prior work used those techniques to assist software testing and other software engineering tasks in many different ways, as we discuss below.

Wang et al. [36] proposed an approach to automate the generation executable, system-level test cases for acceptance testing from natural language use case specifications. The approach relies on a domain model (i.e., a class diagram) and uses several NLP techniques (e.g., Named Entity Recognition and part-of-speech tagging). Two industrial case studies were used to evaluate the approach, which correctly generated test cases that exercise different scenarios manually implemented by experts. Mai et al. [25] proposed an approach to automatically generate executable test cases from use case specifications that capture malicious behavior of users. The evaluation through a case study in the medical domain indicated that the proposed approach can automatically generate test cases that can detect vulnerabilities. Hemmati and Sharifi [17] proposed an approach to predict the failure of system-level test cases specified in natural language. The approach relies only on the test case description in natural language and seeks to enhance the performance of history-based prediction models (i.e., models that use test execution failure history) by including natural language features (e.g., obtained through Part-of-Speech tagger) weighted with TF-IDF. The approach evaluation showed that using natural language features improve the performance of the failure prediction model. Finally, Hemmati et al. [16] investigated approaches to prioritize test cases described only in natural language. The authors used three types of heuristics for test case prioritization, including topic coverage-based and risk-driven heuristics (using the test case risk of detecting a fault based in its fault detection history).

The aforementioned works used NLP for different tasks, such as to analyze use cases described in natural language and automatically generate executable test cases. In contrast, we use several NLP techniques such as text embedding and Named Entity Recognition as part of an automated framework for automatically analyzing newly-specified manual test cases and providing feedback to improve the test case descriptions.

Language modeling is another NLP technique that has been used in software engineering, mainly for code completion tasks [23, 29]. For instance, while Nguyen et al. [29] used program analysis and a statistical language model (n-gram) to develop a technique to complete code, Liu et al. [23] used a transformer-based neural architecture to develop multi-task learning based pre-trained language model for code understanding and code generation. Differently from those works, we are the first, to the best of our knowledge, to use language modeling to model test case specifications in natural language and recommend improvements by identifying words in the description that could be replaced by more likely words, based on word usage in previous test cases.

7 THREATS TO VALIDITY

A threat to the **external** validity concerns to the generalizability of our automated framework and its module evaluations. Our findings rely on the test cases from an educational math game and using test cases of a system from a different domain might yield different results. Another threat is that the achieved results might differ if other text embedding, clustering, or frequent itemset mining techniques are used. Future studies should investigate whether our analysis modules can be improved using other techniques.

A threat to the **internal** validity is related to the selection of the association rules used to recommend missing test steps for a new test case. First, we only use rules with either a confidence above 0.5 or exactly 1 and a lift above 1. Second, our rules only recommend one test step (i.e., there is only one set in the consequent of a rule). Future work should investigate a wider range of the confidence and lift metrics and whether having more than one consequent in a rule improves our missing test step analysis module. Another threat concerns the choice of only one architecture (transformers) for the neural language models. Other model architectures (e.g., RNN/LSTM), should also be investigated. Finally, the evaluations performed for the analysis modules used the existing test cases, which are unoptimized. Even though the test cases were written by experienced QA engineers and developers, at this moment, we are focusing on ensuring that new test cases are improved as much as possible before they are entered into the test suite. In the future, we will also work on improving the existing test cases.

8 CONCLUSION

In this paper, we propose an automated framework for automatically analyzing and providing feedback on how to improve the description of manual test cases. We discuss three analysis modules that were implemented for our framework so far. These modules are capable of recommending improvements to the following: (1) the terminology of a new test case, (2) potentially missing test steps for a new test case, and (3) recommendations of similar test cases

that already exist in the test suite. The three modules were thoroughly evaluated on the data from our industry partner with the test cases designed to test the *Prodigy Math game*. Our evaluation results show that we can achieve a high accuracy (up to 88%) to recommend terminology improvements with statistical and neural language models. Also, on average, our association rules can correctly recommend missing test steps 98% of the time per test case. Finally, we can identify similar test cases with a high performance (an F-score of approximately 83%) using text embedding, text similarity, and clustering techniques. Our proposed framework uses an innovative and efficient way of combining traditional and state-of-the-art techniques to automatically analyze test cases in natural language. The framework is capable of providing actionable recommendations, which is an important challenge given the widespread occurrence of test cases that are written in natural language in the software industry (in particular, the game industry).

ACKNOWLEDGMENTS

The research reported in this article has been supported by Prodigy Education and the Natural Sciences and Engineering Research Council of Canada under the Alliance Grant project ALLRP 550309.

REFERENCES

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *ACM sigmod record*, Vol. 22. ACM, 207–216.

[2] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. 1996. Fast discovery of association rules. *Advances in knowledge discovery and data mining* 12, 1 (1996), 307–328.

[3] D Magdalene Delighta Angeline et al. 2013. Association rule generation for student performance analysis using apriori algorithm. *The SIJ Transactions on Computer Science Engineering & its Applications (CSEA)* 1, 1 (2013), 12–16.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A neural probabilistic language model. *The Journal of Machine Learning Research* 3 (2003), 1137–1155.

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

[6] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. 2007. Large language models in machine translation. (2007).

[7] Peter F Brown, Vincent J Della Pietra, Peter V Desouza, Jennifer C Lai, and Robert L Mercer. 1992. Class-based n-gram models of natural language. *Computational linguistics* 18, 4 (1992), 467–480.

[8] Christian Buck, Kenneth Heafield, and Bas Van Ooyen. 2014. N-gram Counts and Language Models from the Common Crawl.. In *LREC*, Vol. 2. 4.

[9] Tze Yuang Chong, Rafael E Banchs, and Eng Siong Chng. 2012. An empirical evaluation of stop word removal in statistical machine translation. In *Proceedings of the Joint Workshop on Exploiting Synergies between Information Retrieval and Machine Translation (ESIRMT) and Hybrid Approaches to Machine Translation (HyTra)*. 30–37.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Richard O Duda, Peter E Hart, and David G Stork. 1973. *Pattern classification and scene analysis*. Vol. 3. Wiley New York.

[12] Vahid Garousi, Sara Bauer, and Michael Felderer. 2020. NLP-assisted software testing: A systematic mapping of the literature. *Information and Software Technology* 126 (2020), 106321.

[13] Vahid Garousi and Junji Zhi. 2013. A survey of software testing practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376.

[14] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. In *ACM sigmod record*, Vol. 29. ACM, 1–12.

[15] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery* 8, 1 (2004), 53–87.

[16] Hadi Hemmati, Zhihan Fang, and Mika V Mantyla. 2015. Prioritizing manual test cases in traditional and rapid release environments. In *Proceedings of the 8th Int'l Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

[17] Hadi Hemmati and Fatemeh Sharifi. 2018. Investigating nlp-based approaches for predicting manual test case failure. In *Proceedings of the 11th Int'l Conference on Software Testing, Verification and Validation (ICST)*. 309–319.

[18] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[19] Daniel Jurafsky and James H Martin. 2009. Speech and Language Processing. (2009).

[20] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From word embeddings to document distances. In *Proceedings of the 32nd Int'l Conference on Machine Learning (ICML)*. 957–966.

[21] Ke Li, Zhe Liu, Tianxing He, Hongzhao Huang, Fuchun Peng, Daniel Povey, and Sanjeev Khudanpur. 2020. An empirical study of transformer-based neural language model adaptation. In *Int'l Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 7934–7938.

[22] 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 Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1285–1295.

[23] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th Int'l Conference on Automated Software Engineering (ASE)*. 473–485.

[24] Jeffrey D Long, Du Feng, and Norman Cliff. 2003. Ordinal analysis of behavioral data. (2003).

[25] Xuan Phu Mai, Fabrizio Pastore, Arda Göknil, and Lionel Briand. 2018. A natural language programming approach for requirements-based security testing. In *Proceedings of the 29th Int'l Symposium on Software Reliability Engineering (ISSRE)*.

[26] Paul David McNicholas, Thomas Brendan Murphy, and M O'Regan. 2008. Standardising the lift of an association rule. *Computational Statistics & Data Analysis* 52, 10 (2008), 4712–4721.

[27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. *arXiv preprint arXiv:1310.4546* (2013).

[28] Frederic Morin and Yoshua Bengio. 2005. Hierarchical probabilistic neural network language model. In *Int'l Workshop on Artificial Intelligence and Statistics*. 246–252.

[29] Son Nguyen, Tien Nguyen, Yi Li, and Shaohua Wang. 2019. Combining program analysis and statistical language model for code statement completion. In *Proceedings of the 34th Int'l Conference on Automated Software Engineering (ASE)*. 710–721.

[30] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. How is video game development different from software development in open source?. In *Proceedings of the 15th Int'l Conference on Mining Software Repositories (MSR)*. 392–402.

[31] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365* (2018).

[32] Cristiano Politowski, Fabio Petrillo, and Yann-Gäel Guéhéneuc. 2021. A Survey of Video Game Testing. *arXiv preprint arXiv:2103.06431* (2021).

[33] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices. In *Annual meeting of the Southern Association for Institutional Research*. 1–51.

[34] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability* 12, 4 (2002), 219–249.

[35] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. 2021. Identifying Similar Test Cases That Are Specified in Natural Language. http://asgaard.ece.ualberta.ca/papers/preprint/markos_preprint_test_similarity.pdf. [This work was submitted to the IEEE Transactions on Software Engineering journal and is undergoing a major revision].

[36] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel Briand. 2020. Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering* (2020).

[37] Sholom M Weiss, Nitin Indurkhya, Tong Zhang, and Fred Damerau. 2010. *Text mining: predictive methods for analyzing unstructured information*. Springer Science & Business Media.

[38] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[39] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. , 38–45 pages. https://www.aclweb.org/anthology/2020.emnlp-demos.6