# Leveraging Natural Language Processing Techniques to Improve Manual Game Testing

by

Markos Viggiato de Almeida

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

# Abstract

The gaming industry has experienced a sharp growth in recent years, surpassing other popular entertainment segments, such as the film industry. With the ever-increasing scale of the gaming industry and the fact that players are extremely difficult to satisfy, it has become extremely challenging to develop a successful game. In this context, the quality of games has become a critical issue. Game testing is a widely-performed activity to ensure that games meet the desired quality criteria. However, despite recent advancements in test automation, manual game testing is still prevalent in the gaming industry, with test cases often described in natural language only and consisting of one or more test steps that must be manually performed by the Quality Assurance (QA) engineer (i.e., the tester). This makes game testing challenging and costly. Issues such as redundancy (i.e., when different test cases have the same testing objective) and incompleteness (i.e., when test cases miss one or more steps) become a bigger concern in a manual game testing scenario. In addition, as games become bigger and the number of required test cases increases, it becomes impractical to execute all test cases in a scenario with short game release cycles, for example.

Prior work proposed several approaches to analyze and improve test cases with associated source code. However, there is little research on improving manual game testing. Having higher-quality test cases and optimizing test execution help to reduce wasted developer time and allow testers to use testing resources more effectively, which makes game testing more efficient and effective. In addition, even though players are extremely difficult to satisfy, their priorities are not considered during game testing. In this thesis, we investigate how to improve manual game testing from different

perspectives.

In the first part of the thesis, we investigated how we can reduce redundancy in the test suite by identifying similar natural language test cases. We evaluated several unsupervised approaches using text embedding, text similarity, and clustering techniques and showed that we can successfully identify similar test cases with a high performance. We also investigated how we can improve test case descriptions to reduce the number of unclear, ambiguous, and incomplete test cases. We proposed and evaluated an automated framework that leverages statistical and neural language models and (1) provides recommendations to improve test case descriptions, (2) recommends potentially missing steps, and (3) suggests existing similar test cases.

In the second part of the thesis, we investigated how player priorities can be included in the game testing process. We first proposed an approach to prioritize test cases that cover the game features that players use the most, which helps to avoid bugs that could affect a very large number of players. Our approach (1) identifies the game features covered by test cases using an ensemble of zero-shot techniques with a high performance and (2) optimizes the test execution based on highly-used game features covered by test cases. Finally, we investigated how sentiment classifiers perform on game reviews and what issues affect those classifiers. High-performing classifiers can be used to obtain players' sentiments about games and guide testing based on the game features that players like or dislike. We show that, while traditional sentiment classifiers do not perform well, a modern classifier (the OPT-175B Large Language Model) presents a (far) better performance.

The research work presented in this thesis provides deep insights, actionable recommendations, and effective and thoroughly evaluated approaches to support QA engineers and developers to improve manual game testing.

# Preface

The research work presented in this thesis has been conducted in the Analytics of Software, GAmes, And Repository Data (ASGAARD) lab led by Dr. Cor-Paul Bezemer. This thesis is an original work by Markos Viggiato de Almeida.

Chapter 2 has been published as: M. Viggiato, D. Paas, C. Buzon and C.-P. Bezemer, "Identifying Similar Test Cases That Are Specified in Natural Language", in IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2022.3170272. I was responsible for developing the ideas, collecting and processing the game testing data, creating and evaluating the unsupervised approaches for similarity detection, analyzing the data, and manuscript composition. D. Paas and C. Buzon provided access to the data and assisted with the data collection. Dr. Bezemer was the supervisory author and was involved in concept formation and manuscript composition.

Chapter 3 has been published as: M. Viggiato, D. Paas, C. Buzon and C.-P. Bezemer, "Using Natural Language Processing Techniques to Improve Manual Test Case Descriptions", 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2022, pp. 311-320, doi: 10.1145/3510457.3513045. I was responsible for developing the ideas, collecting and processing the game testing data, creating and evaluating the automated framework, analyzing the data, and manuscript composition. D. Paas and C. Buzon provided access to the data and assisted with the data collection. Dr. Bezemer was the supervisory author and was involved in concept formation and manuscript composition.

Chapter 4 has been submitted for review as: M. Viggiato, D. Paas and C.-P. Bezemer, "Prioritizing Natural Language Test Cases Based on Highly-Used Game

Features", 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2023. I was responsible for developing the ideas, collecting and processing the game testing and the game execution data, creating and evaluating the optimization approaches, analyzing the data, and manuscript composition. D. Paas provided access to the game execution log data and assisted with the data collection. Dr. Bezemer was the supervisory author and was involved in concept formation and manuscript composition.

Chapter 5 has been published as: M. Viggiato, D. Lin, A. Hindle and C.-P. Bezemer, "What Causes Wrong Sentiment Classifications of Game Reviews?", in IEEE Transactions on Games, vol. 14, no. 3, pp. 350-363, Sept. 2022, doi: 10.1109/TG.2021.3072545. I was responsible for developing the ideas, collecting part of the game review data, processing and analyzing the game review data, and manuscript composition. D. Lin was responsible for collecting the data and contributed to manuscript edits. Dr. Hindle and Dr. Bezemer were the supervisory authors and were involved in concept formation and manuscript composition.

Chapter 6 has been submitted for review as: M. Viggiato, D. Lin and C.-P. Bezemer, "Leveraging the OPT Large Language Model for Sentiment Analysis of Game Reviews", in IEEE Transactions on Games. I was responsible for developing the ideas, collecting part of the game review data, processing and analyzing the game review data, and manuscript composition. D. Lin was responsible for collecting the data. Dr. Bezemer was the supervisory author and was involved in concept formation and manuscript composition.

# Acknowledgements

I would like to thank everyone who contributed to the works presented in this thesis. First, I would like to express my sincere gratitude to my supervisor, Dr. Cor-Paul Bezemer, who has guided me and taught me so much during my studies. Dr. Bezemer has always been patient and motivated and has given me invaluable advice over all these years. This research would not have been possible without his support.

I would like to express my gratitude to the members of my examination committee: Dr. Fabio Petrillo, Dr. Denilson Barbosa, Dr. Marek Reformat and Dr. James Miller. I would also like to thank the Alberta Innovates research and innovation agency and Prodigy Education, which supported the research presented in this thesis.

I am also thankful for all the friends that I made during my studies, in particular my friends at the ASGAARD lab, who were always there when I needed support and made my journey more joyful.

Finally, I am grateful to my parents, my sister, and my partner for all their unconditional love, care and support during my studies. I could not have come this far without them.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The gaming industry is a market segment that has rapidly grown in recent years and has become a multi-billion dollar industry bigger than the global movie and North American sports industries combined [198]. The global gaming market is projected to reach approximately $546 billion in 2028 [56] and it has seen a surge in the number of players, with an expected number of 3.32 billion players worldwide by 2024 [78]. With the scale of the gaming industry and the fact that players are extremely difficult to satisfy [32, 97, 105, 106], developing a successful game has become challenging and the quality of games has become a critical issue [97, 105, 106].

## 1.1 Motivation

Game testing is a widely-used and essential quality assurance activity during the game development to ensure that the game meets the desired quality criteria [15, 58, 69, 73]. Despite the recent advancements in the test automation field [111, 140, 173], prior work showed that manual testing is still prevalent in the gaming industry [130, 140, 145, 180]. For instance, Politowski *et al.* [145] showed, through a survey, that manual game testing is the primary testing technique used by game developers and, as a result, the gaming industry relies almost solely on manual labour to test games. In addition, Murphy-Hill *et al.* [130] interviewed 14 developers with experience in game and non-game development and surveyed 364 practitioners and showed that game

developers face several challenges to write automated tests, which was also shown in other prior works [140, 145]. One example of such a challenge is the non-determinism that is present in games due to multithreading, artificial intelligence, and purposely injected randomness. In addition, the large state space that needs to be explored and the difficulty of asserting what the expected behavior is make it challenging to write automated tests [130].

In a manual game testing scenario, test cases are often described in natural language only and consist of one or more test steps that must be manually performed by the Quality Assurance (QA) engineer (i.e., the tester) to test the target game. Manually executing test steps is a tedious activity and requires a large amount of human effort (e.g., testers may need to play through several levels of a game to verify that the game logic is working as expected).

An additional challenge for manual testing is to organize and maintain test cases, mainly in a situation with a large test suite, and ensure the high quality of test case descriptions. For instance, having redundant test cases in the test suite (i.e., when different test cases have the same testing objective) or incomplete test cases (i.e., when test cases miss one or more steps) are bigger concerns for manual testing (than for automated testing). Those kinds of problems make the testing activity even more costly, might result in wasted QA and developer time and effort and, ultimately, lead to less effective and efficient testing. Furthermore, as games grow and the number of test cases increases, it becomes impractical to execute all manual test cases, mainly in a scenario with short release cycles.

Test cases with associated source code have been widely studied [11, 27, 36, 81, 112, 116, 141]. However, research targeting how natural language test cases in a manual game testing scenario can be improved is scarce [111]. Having well-maintained, organized, and high-quality test cases (e.g., test cases with a clear and objective description) help to reduce wasted developer time and make testing more efficient and effective [101]. In addition, optimizing the execution of test cases allows developers

and testers to use the available resources (e.g., time) more effectively when there is not time to execute all the test cases (e.g., during regression testing). For instance, the execution of tests can be optimized based on player priorities such as the game features that players use the most. This helps to guide testing based on features that are more relevant to players and avoid bugs that could affect a large number of players.

## 1.2 Thesis objectives

This thesis has the following objectives:

1. Objective 1: Investigate how natural language test cases for manual game testing can be improved.
    (a) Investigate how we can reduce redundancy in the test suite by identifying similar natural language test cases.
    (b) Investigate how we can reduce the number of unclear, ambiguous and incomplete test cases by improving the descriptions of newly-designed natural language test cases.
2. Objective 2: Investigate how we can take player priorities into account during the manual game testing process.
    (a) Investigate how we can prioritize test cases that cover game features which are relevant to players by optimizing the test execution using the game features that players use the most.
    (b) Investigate how we can focus testing on the game features that players like or dislike by guiding testing based on the sentiment that players express through game reviews.

To achieve our objectives, we performed five research studies. In the first study (which targets Objective 1.a), we evaluated several unsupervised approaches to automatically identify similar test cases that are specified in natural language only. In the second study (which targets Objective 1.b), we proposed an automated framework that analyzes the test cases specified in natural language and provides actionable insights to improve their descriptions. In the third study (which targets Objective 2.a), we investigated how to optimize test case execution to prioritize test cases that cover highly-used game features, which are more relevant to players. Finally, to verify if we can leverage sentiment analysis techniques to obtain players' sentiment about games, in the fourth and fifth studies (which target Objective 2.b) we investigated how sentiment classifiers perform on game reviews and the reasons why they might fail.

We performed industrial case studies to thoroughly evaluate the approaches proposed in studies 1, 2, and 3 with the data of our industry partner, *Prodigy Education*.[1] We summarize the motivation and findings of our research studies below:

**Research Study 1: Identifying Similar Test Cases That Are Specified in Natural Language (Chapter 2)**

**Motivation:** Despite prior work having proposed approaches for test case similarity, these approaches have important limitations, such as the large manual effort needed to use the approach (e.g., to specify formal descriptions of test cases based on textual descriptions) or the need for the test case source code. Therefore, in Research Study 1, we explore unsupervised approaches to automatically identify similar test cases described only in natural language. QA engineers and developers can use our approach to identify groups of similar test cases, which can help to identify and remove redundant test cases from the test suite.

**Findings:** We found that we can leverage text embedding, text similarity and

[1]https://www.prodigygame.com/main-en/

clustering techniques to identify groups of similar test cases with a high performance. Our evaluations showed that using a two-stage approach achieves the best results: (1) our approach first clusters similar test steps and then (2) uses the test step clusters to identify groups of similar test cases.

**Research Study 2: Using Natural Language Processing Techniques to Improve Manual Test Case Descriptions (Chapter 3)**

**Motivation:** Manual test cases are often specified by employees from different departments, such as QA engineers or developers. This may result in problematic test cases, such as unclear, ambiguous or incomplete test case descriptions, which can hinder the efficiency and effectiveness of the manual testing activity. Having an automated approach to analyze and suggest improvements to test cases helps to reduce the manual testing effort of QA engineers and developers by improving the quality of test case descriptions. It also helps the creation and maintenance of a high-quality, more consistent and more standardized test suite, which can be useful and benefit new employees who do not yet have much knowledge about the existing test suite. Therefore, in Research Study 2, we propose an automated framework that provides the following actionable recommendations to improve test case descriptions to QA engineers: recommendations to improve the terminology of a new test case, recommendations of potentially missing test steps in a new test case, and recommendations of existing similar test cases.

**Findings:** We found that we can combine traditional (statistical) and state-of-the-art (neural) language models to effectively recommend terminology improvements. We also found that association rules are an effective approach to identify potentially missing test steps in a newly-specified test case. Text embedding, text similarity, and clustering techniques can be used to identify and recommend existing test cases which are similar to a newly-designed test case.

**Research Study 3: Prioritizing Natural Language Test Cases Based on**

**Highly-Used Game Features (Chapter 4)**

**Motivation:** To include player priorities regarding the game features that they use the most in the testing process, we need to be able to automatically prioritize test cases that test the most used game features. However, most existing techniques for prioritizing test cases do not work for manual test cases. For instance, they might depend on test case source code, which does not exist for manual test cases, or the execution history of test cases, which could be difficult to be accessed or is generally not meaningful in a manual testing scenario. Therefore, in Research Study 3, we investigate how we can prioritize test cases specified in natural language without source code. In particular, we prioritize test cases that cover highly-used game features. Focusing the test execution on highly-used game features helps to avoid bugs that could affect a very large number of players.

**Findings:** Our results show that our approach can successfully identify the game features covered by test cases and prioritize test cases that cover highly-used game features. This means that our approach can find test case orderings that cover highly-used game features early in the test execution while keeping the test execution time as short as possible.

**Research Study 4: What Causes Wrong Sentiment Classifications of Game Reviews? (Chapter 5)**

**Motivation:** To include player priorities with respect to players' sentiment about the game features in the testing process, we need to be able to automatically identify the players' sentiment about games and their features. The first step is to investigate how well sentiment classifiers perform on game reviews provided by players and what are the problems with those classifiers (if any). Therefore, in Research Study 4, we investigate how widely-used traditional sentiment classifiers perform on game reviews from the Steam platform[2] and which factors cause wrong classifications. This investi-

[2]https://steamcommunity.com/

gation provides insights for game developers and researchers about whether existing classifiers can be used to identify players' sentiment about the game features and how to improve the performance of sentiment classification techniques.

**Findings:** At the time that we conducted this study, we found that traditional sentiment classification techniques performed poorly on game reviews. Furthermore, we identified four main causes for wrong classifications, such as reviews that point out advantages and disadvantages of the game, reviews with game comparisons, and reviews that contain sarcastic text, which might confuse the classifier.

**Research Study 5: Leveraging the OPT Large Language Model for Sentiment Analysis of Game Reviews (Chapter 6)**

**Motivation:** The findings from Research Study 4 showed that traditional sentiment classifiers, which were available at the time that study was performed, do not perform well on game reviews. However, the Natural Language Processing (NLP) field has seen major improvements in several different tasks, such as sentiment classification, in the last few years. Therefore, to better understand if we can now use sentiment analysis techniques to obtain players' sentiment about games and include that information in the testing process, we performed a follow-up study. In Research Study 5, we investigate how a pre-trained Large Language Model performs on the sentiment classification of game reviews and what issues affect the performance of such a model. This study provides insights about whether a modern sentiment classifier can be used to effectively capture players' sentiment about games.

**Findings:** We found that the pre-trained Large Language Model that we used performs (far) better on the sentiment classification of game reviews and that most issues that affect traditional classifiers have been solved. Therefore, a Large Language Model can be used to capture players' sentiment about game features and we can now include this information in the testing process.

Together, the research studies discussed above provide deep insights and action-

able recommendations to improve manual game testing. They also present novel approaches to support QA engineers and developers to improve test case descriptions and optimize the test execution taking into account player priorities, which makes the manual testing more effective and efficient and improve the experience of players with the game. In the next section, we discuss an example of a typical natural language test case that is used in a manual game testing scenario.

## 1.3 Natural language test cases

In a manual game testing scenario, test cases are written in natural language only and do not have source code. A natural language test case contains the following fields:

- a meaningful test case name.
- an objective with the main goal of the test case.
- the duration of the test case execution, as provided by developers and QA engineers.
- one or more steps with instructions that must be manually performed by a human tester.

Table 1.1 presents two examples of typical test cases in natural language that were used in the research studies that we performed. The first example is a simple test case that verifies if the login functionality is working as expected. The tester needs to manually log in to the game with an existing account designed for testing and verify the success of the operation in the system. The second test case aims at verifying if a membership can be purchased. The tester needs to log in to the game using a non-member account designed for testing, perform the steps to purchase a membership, and verify in the system that the membership was successfully purchased.

Table 1.1: Examples of natural language test case descriptions.

| **Name** | **Objective** | **Duration** | **Steps** |
|---|---|---|---|
| Login - Existing account | Verify if players can log in to the game with an existing account | 1 minute | 1. Log in to the game using an existing player account<br>2. Verify if the player is successfully logged in |
| Membership purchase | Verify if players without a membership can purchase it | 3 minutes | 1. Log in to the game using a non-member player account<br>2. Go to the membership page<br>3. Click on the membership icon<br>4. Go through the membership flow<br>5. Verify that the player successfully purchased the membership |

For the research studies presented in Chapters 2, 3, and 4, we collaborated with an industry partner, *Prodigy Education*, and used the test cases designed to test the *Prodigy Math game*.[3] The *Prodigy Math game* is a proprietary, online, web-based serious math game with a curriculum-aligned educational content. The game features over 50,000 math questions spanning Grade 1-8. In the game, players play the role of a character (a wizard) in the Prodigy world and can go to the several different world zones. As the players answer math questions, their wizards can evolve, learn new spells, and acquire new equipment and in-game items. Furthermore, differently from entertainment-only games, the *Prodigy Math game* has been designed with a primary focus on supporting the learning of math, and aspects such as providing entertainment and fun to players are used to keep players engaged in the learning process.

[3]https://www.prodigygame.com/main-en/

## 1.4 Thesis outline

The remainder of this thesis is organized as follows: Chapter 2 presents an investigation of several unsupervised approaches to identify similar test cases specified in natural language. Chapter 3 presents a study in which we investigated how NLP techniques can be used to improve the description of manual test cases. Chapter 4 presents an approach to prioritize test cases that cover highly-used game features. Chapter 5 presents a study with an evaluation of traditional sentiment classifiers on game reviews and the causes of wrong sentiment classifications. Chapter 6 presents an evaluation of a Large Language Model on the sentiment classification of game reviews and the challenges of using such a model for game review sentiment classification. Finally, Chapter 7 concludes the thesis by highlighting the findings and contributions of our research studies and discussing directions for future research.

# Chapter 2

# Identifying Similar Test Cases That Are Specified in Natural Language

## 2.1 Abstract

Software testing is still a manual process in many industries, despite the recent improvements in automated testing techniques. As a result, test cases (which consist of one or more test steps that need to be executed manually by the tester) are often specified in natural language by different employees and many redundant test cases might exist in the test suite. This increases the (already high) cost of test execution. Manually identifying similar test cases is a time-consuming and error-prone task. Therefore, in this chapter, we propose an unsupervised approach to identify similar test cases. Our approach uses a combination of text embedding, text similarity and clustering techniques to identify similar test cases. We evaluate five different text embedding techniques, two text similarity metrics, and two clustering techniques to cluster similar test steps and three techniques to identify similar test cases from the test step clusters. Through an evaluation in an industrial setting, we showed that our approach achieves a high performance to cluster test steps (an F-score of 87.39%) and identify similar test cases (an F-score of 86.13%). Furthermore, a validation with developers indicates several different practical usages of our approach (such as identifying redundant test cases), which help to reduce the testing manual effort and time.

## 2.2 Introduction

Despite the many recent improvements in automated software testing, testing is still a manual process in many industries. For example, in the gaming industry, game developers face several challenges and difficulties with writing automated tests [130, 140, 145]. As a result, test cases are often described in natural language and consist of a sequence of one or more test steps, which have instructions that must be manually performed to test the target game. Furthermore, those test cases are usually defined by employees from different departments, such as Quality Assurance (QA) engineers or developers, which may result in redundant test cases (i.e., test cases that are semantically similar or even duplicates) as the system evolves and the test suite grows [157]. Having redundant test cases is problematic in particular in a manual testing scenario, due to the tediousness and cost of executing such manual tests.

Manually identifying similar or duplicate test cases to reduce test redundancy is an expensive and time-consuming task. In addition, naive approaches (e.g., searching for exactly matching test cases) are not sufficient to capture all similarity, as different test case writers may use different terminology to specify a test case, even for similar test objectives. Approaches proposed by prior work [36, 101, 186] have limitations in terms of scope (e.g., the work by Li *et al.* [101] can only cluster test steps but not entire test cases), the large manual effort necessary to specify formal descriptions of test cases [186], or the need for the test case source code [36]. Therefore, an automated and unsupervised technique to identify similar test cases (which can be applied directly to the natural language description of entire test cases) is necessary as it can prevent the QA and development teams from wastefully executing test cases that perform the same task. Throughout this chapter, for brevity we adopt the term "similar test cases" to refer to semantically similar and duplicate test cases.

In this chapter, we propose an approach to identify similar test cases that are specified in natural language. More specifically, (1) we use text embedding, text

similarity, and clustering techniques to cluster similar test steps that compose test cases and (2) we compare test cases based on their similarity in terms of steps that belong to the same cluster.

In the first part of the chapter, we study how text embeddings obtained from different techniques, text similarity metrics, and different clustering algorithms can be leveraged to identify semantically similar test steps. We compare embeddings from five different techniques (Word2Vec, BERT, Sentence-BERT, Universal Sentence Encoder, and TF-IDF), two similarity metrics (Word Mover's Distance and cosine similarity), and evaluate two different clustering techniques (Hierarchical Agglomerative Clustering and K-Means). In particular, we address the following research question for this part of the chapter:

**RQ1: How effectively can we identify similar test steps that are written in natural language?**

*Understanding if we can effectively identify similar test steps automatically allows to know if we can rely on test step clusters to identify similarity between entire test cases. We found that we can achieve the highest performance (an F-score of 87.39%) using an ensemble approach that consists of different embedding and clustering techniques. In addition, we show that using Sentence-BERT instead of Word2Vec (which was identified as the best-performing model by prior work [101]) yields a slightly lower performance but reduces the execution time from 150 minutes to about 2 minutes.*

In the second part of the chapter, we leverage the previously detected clusters of test steps to identify similar test cases. We compared three different techniques and related variations to compute a similarity score (using the simple overlap, Jaccard, and cosine metrics) to measure the similarity of test cases based on the test step clusters that they have in common. In particular, we address the following research question for this part of the chapter:

**RQ2: How can we leverage clusters of test steps to identify similar test cases?**

*Given the difficulty of identifying similar test cases, which are usually composed of several steps, we use clusters of similar test steps to identify similar test cases. We found that test step clusters can be used to identify test case similarity with a high performance (an F-score of 86.13%).*

Our work presents an approach to identify similar test cases based only on their natural language descriptions. We highlight that our approach is unsupervised as it does not require labelled data nor requires human supervision. In addition, no test source code or system model is necessary. QA engineers and developers can use our approach to obtain groups of similar test cases, which can be used, for example, to identify and remove redundant test cases from the test suite. Furthermore, existing groups of similar test cases can be leveraged to support the design of new test cases and help to maintain a more consistent and homogeneous terminology across the test suite. Finally, we provide access to the source code of our approach and the experiments that we performed.[1]

The remainder of the chapter is organized as follows. In Section 2.3, we present background information about text embedding, clustering techniques and game testing. We discuss related work in Section 2.4 and our proposed approach in Section 2.5. Section 2.6 presents the dataset that we used to evaluate our approach. Sections 2.7 and 2.8 discuss the experiments that we performed to evaluate the two main stages of our approach. In Section 2.9, we discuss our results and the approach validation. Finally, Sections 2.10 and 2.11 present the threats to validity and conclude the chapter, respectively.

## 2.3 Background

In this section, we present an overview of the terminology and concepts that we use throughout this chapter. In this work, we use "test cases" to refer to manual test cases that are described in natural language as a sequence of steps, i.e., test cases for

[1] https://github.com/asgaardlab/test-case-similarity-technique

which there is no source code associated.

### 2.3.1 Text representation

In order to use text data as input for a machine learning algorithm, we first need to convert the text into a numeric vector through a process called *text embedding* [193, 194]. Different methods can be used to obtain a text embedding, and the embedding can be done at different granularity levels, such as at word and sentence-level. Below, we explain the different techniques that we use in this work to obtain the numeric representation of words and sentences.

#### Word Embedding

A word embedding is the representation of a single word through a real-valued (and usually high-dimensional) numeric vector. In this study, we use two natural language processing techniques to obtain word-level embeddings: Word2Vec [124] and BERT [46]. Figure 2.1a presents two examples of pre-processed test steps along with part of their word embeddings obtained by the Word2Vec and BERT models. Next, we explain how each word embedding technique works and how the example embeddings presented in Figure 2.1a are computed.

**Word2Vec** transforms words into high-dimensional numeric vectors that are able to maintain the syntactic and semantic relationships between words in the vector space [123, 124]. This means that embeddings of similar words will (most of the time) be close in the vector space (i.e., the distance between the embedding vectors is small). Furthermore, with Word2Vec, each word is assigned a single numeric vector regardless of the context in which it is used, as we can see for the words "verify" and "item" in the two steps in Figure 2.1a. In this work, we used the continuous bag-of-words (CBOW) model architecture of Word2Vec, which is faster than the other possible architecture, called skip-gram [123].

Differently from Word2Vec, **BERT (Bidirectional Encoder Representations**

| Test step | Word2Vec | BERT |
|---|---|---|
| [verify item name] | [(-0.93, -0.16, ...), (0.57, 0.21, ...), (0.12, 0.85, ...)] | [(-0.12, -0.11, ...), (-0.59, -0.13, ...), (-0.24, -0.58, ...)] |
| [verify item description] | [(-0.93, -0.16, ...), (0.57, 0.21, ...), (-0.03, -0.27, ...)] | [(-0.12, 0.07, ...), (-0.61, -0.08, ...), (-0.24, -0.50, ...)] |

(a) Examples of word embeddings for test steps.

| Test step | SBERT | USE | TF-IDF |
|---|---|---|---|
| [verify item name] | [(0.32, 0.02, ...)] | [(0.46, 0.52, ...)] | [(0.0 ... 0.63, 0.67 ... 0.0)] |
| [verify item description] | [(0.31, -0.09, ...)] | [(-0.15, 0.81, ...)] | [(0.0 ... 0.76, 0.55 ... 0.0)] |

(b) Examples of sentence embeddings for test steps.

Figure 2.1: Examples of test step embeddings. Note that we provide only the first two elements of the embedding vector due to space constraints as the actual vectors have a high dimension.

**from Transformers)** is a transformer-based model that can be used to extract contextual word embeddings, i.e., embeddings that change depending on the context in which a word is present [46]. The context of a target word refers to the words that surround it, i.e., the words that appear before and after the target word. This means that the same word may have different embedding vectors, as we can see in Figure 2.1a, where the BERT embeddings for the words "verify" and "item" are different in the two test steps because those words are in different contexts.

BERT is available as a model that was pre-trained on lower-cased English text (uncased BERT). This pre-trained model can further be trained with a domain-specific training set (known as domain-adaptive pre-training [65]). The BERT model uses WordPiece tokenization [202], in which a word may be split into sub-words. For example, the word "validate" is composed of the sub-words "valid" and "ate", each one with its own embedding vector. Therefore, when extracting embeddings of words that are split into sub-words, we need to aggregate the embeddings of the sub-words (e.g., by averaging the embedding vectors).

### Sentence Embedding

Differently from word embedding, sentence embedding is the representation of a whole sentence with a real-valued (and usually high-dimensional) numeric vector. In this work, we use three different techniques to extract sentence embeddings (SBERT, USE, and TF-IDF). Figure 2.1b presents two examples of pre-processed test steps along with part of their sentence embeddings obtained by the SBERT, USE, and TF-IDF techniques. Next, we explain how each sentence embedding technique works.

**Sentence-BERT (SBERT)** is a BERT-based framework that allows us to directly extract numeric representations of full sentences [152]. The embeddings of sentences that are semantically similar are close in the embedding space. We can use this information for different purposes, such as identifying paraphrases and clustering similar sentences. For instance, the SBERT embeddings of the two test steps presented in Figure 2.1b are close in the embedding space (i.e., have a small distance between them). Among several generic and task-specific SBERT pre-trained models that are available[2], three models are suitable for our task (identifying similar test steps): *paraphrase-distilroberta-base-v1*, *stsb-roberta-base*, and *stsb-roberta-large*. While the first model is optimized to identify paraphrases and was trained on large scale paraphrase data, the second and third ones are the base and large versions of a model that was optimized for semantic textual similarity.

**Universal Sentence Encoder (USE)** is an embedding technique that can be used to directly extract embeddings from sentences, phrases, or short paragraphs to be used in another task, such as textual similarity and clustering tasks [31]. With a similar behavior to SBERT, the two examples presented in Figure 2.1b have close embedding vectors.

Finally, we also used the **TF-IDF (Term Frequency–Inverse Document Frequency)** method to represent sentences. TF-IDF computes the importance of a word

[2]https://www.sbert.net/docs/pretrained_models.html

to a document by combining the word frequency in the document and the word frequency across all the other documents [86, 87, 161]. In our case, the test steps (i.e., sentences) are considered documents. We built a numeric vector for each test step using the word importance values. Words that are not present in the step are assigned a value of zero. We can observe a typical vector obtained with TF-IDF in the examples presented in Figure 2.1b, in which the values different from zero correspond to the importance of the words presented in the "verify item name" and "verify item description" steps.

### 2.3.2 Clustering techniques

**Hierarchical Agglomerative Clustering (HAC)** [155] is a clustering algorithm that works in a bottom-up manner. Initially, each data point corresponds to a single cluster itself, and as the algorithm iterates, different clusters are merged with the aim of minimizing a specific linkage criterion. The result of the iterative merging process is a tree structure that can represent the data points (and their clusters), known as a dendrogram. Although the dendrogram can be used to identify the number of clusters, in our work we determined that parameter empirically and used the number that maximizes our evaluation metric (as explained in Section 2.7.5). Different linkage criteria can be used, such as single-linkage (the algorithm uses the minimum of the distances between all data points of two sets) and average-linkage (the algorithm uses the average of the distances between all data points of two sets).

The **K-means clustering** [50] algorithm splits the data points into $k$ different clusters. Different from HAC, no hierarchical cluster structure is generated with K-means. The goal of K-means is to group data points in order to minimize the distance between points belonging to the same cluster compared to the distance of points from different clusters. Using the Expectation-Maximization algorithm [127], K-means starts with $k$ centroids. Then, the algorithm (1) assigns each data point to the closest cluster (in terms of the distance between the point and the centroids)

and (2) computes the new centroids using the updated data point assignments. The execution finishes when there is no change to the allocation of data points.

### 2.3.3 Game testing

Video game testing is substantially different from traditional software (e.g., desktop or mobile) testing. While there have been advances in test automation for traditional software, games still rely mainly on high-level, black-box, manual testing, in which human testers play through the game to assert its expected behavior (which is known as gameplay testing) [130, 140, 145]. Furthermore, the focus of game testing is more related to the overall user experience than to the accuracy of the test [145]. The test cases in a test suite of a game must also verify different types of requirements compared to traditional software, such as fun, entertainment, gameplay and other user experience aspects that traditional testing cannot satisfy [145]. Test automation is significantly more difficult in games for a number of reasons, such as (1) the difficulty of separating the user interface from the rest of the game, (2) the difficulty to explore the often large state space in games, (3) the challenge in asserting what the expected behavior is, and (4) the non-determinism that games have (e.g., because of multithreading, distributed computing, and AI agents) [130]. Finally, the common scenario of manual testing and the difficulties to automate tests in games show the need for new methodologies that can support QA engineers and developers during the game testing and enable game test automation in the future [140].

## 2.4 Related Work

In this section, we discuss prior work that applied clustering techniques [10, 34, 101, 186, 214] and natural language processing (NLP) [101, 113, 119, 120, 175, 189, 190] to software testing.

### 2.4.1 Clustering techniques for software testing

Our work is based on the study of Li *et al.* [101], which proposed an approach to cluster test steps written in natural language based on the steps' similarities. The study used text embeddings (including embeddings obtained with the Word2Vec technique) together with the Relaxed Word Mover's Distance (RWMD) metric [93] to measure similarity between embeddings. The test steps were then clustered with the hierarchical agglomerative and K-means clustering techniques. The approach was evaluated on a large-scale dataset of a mobile app and achieved an F-score of 81.55% in the best case. The proposed approach also reduced the manual effort for implementing test-step API methods by 65.90%. Differently from Li *et al.*'s work [101], we evaluated more recent NLP techniques to obtain word and sentence embeddings (BERT, SBERT, and Universal Sentence Encoder). Furthermore, we extended Li *et al.*'s work [101] for the purpose of identifying similar test cases using the identified clusters of test steps.

Walter *et al.* [186] proposed an approach to improve the efficiency of test execution. The approach removes redundant test steps and uses clustering techniques to rearrange the remaining steps. To use the approach, the textual descriptions of test cases must be converted into a representation form of parameters concatenated by first-order logic operators (AND, OR, NOT). The approach was evaluated in a case study with a system from an automotive industry company. The results indicated a test load reduction of 18% due to the removal of redundant test steps and rearranging of the remaining steps. Chetouane *et al.* [36] proposed an approach to reduce a test suite by clustering similar test cases (based on their source code) with the K-means algorithm. 13 Java programs were used to evaluate if the approach could efficiently reduce the test suite and assess the impact on coverage metrics. The evaluation showed that the approach can reduce the test suite by 82.2% while maintaining the same coverage metric as the original test suite. Even though the work of Walter *et al.* [186]

addressed the problem of test step redundancy, their approach requires all test steps to contain a formal description of their precondition, action and postcondition. Creating these formal descriptions requires a large amount of manual effort which causes scalability issues and reduces the applicability of the approach in practice. Our approach does not require such manual effort. The approach proposed by Chetouane *et al.* [36] requires test cases that have source code associated with them. The test cases on which our approach focuses consist of only natural language descriptions and do not have any source code associated with them.

Pei *et al.* [141] proposed distance-based Dynamic Random Testing (DRT) approaches with the goal of improving the fault detection effectiveness of DRT. The work clustered similar test cases based on their source code with three clustering methods: K-means, K-medoids, and hierarchical clustering. The information of distance between the test case groups was used to identify test cases that are closer to failure-causing groups. 12 versions of 4 open-source programs were used to evaluate the approaches. The evaluation showed that the proposed strategies achieve a larger fault detection effectiveness with a low computational cost compared to other DRT approaches. Arafeen and Do [6] investigated whether clustering of test cases based on similarities in their requirements could improve traditional test case prioritization techniques. The paper used TF-IDF and the K-means clustering algorithm to group test cases that have similar requirements. Two Java programs were used to evaluate the approach. The evaluation showed that the use of requirements similarity can improve the effectiveness of test case prioritization techniques but the improvements vary with the cluster size.

Differently from the works above, our study aims at finding similar test cases that are written in natural language and for which there is no associated source code. We experimented with different NLP and clustering techniques to find clusters of similar test steps, which are used with test case names to obtain similar test cases. Furthermore, differently from the work of Walter *et al.* [186], which converts natural language

descriptions of test cases into a representation form of parameters concatenated by logic operators to be used with their approach, our proposed approach works in an unsupervised manner with the original test cases written in natural language.

### 2.4.2 Natural Language Processing techniques for software testing

Wang *et al.* [190] proposed an approach to automate the generation of executable system test cases. The approach applies NLP techniques (such as tokenization and part-of-speech tagging) to textual data obtained from use case specifications. Furthermore, a domain model of the system under analysis is necessary to generate test data and oracles. Wang *et al.* [190] performed an industrial case study with automative software to demonstrate the feasibility of the proposed approach. Wang *et al.* [189] extended their previous work [190] by further providing empirical evidence about the scalability of the approach to generate executable, system-level test cases for acceptance testing from natural language requirements. In addition, Wang *et al.* [189] focused on embedded systems and demonstrated the effectiveness of the proposed approach using two industrial case studies, in which the approach correctly generated test cases that exercise different scenarios manually implemented by experts, including critical scenarios not previously considered.

Yue *et al.* [210] proposed a Test Case Specification (TCS) language, called Restricted Test Case Modeling (RTCM), and an automated test case generation tool, called *aToucan4Test*, to transform textual test cases into executable test cases. RTCM provides a template that combines natural language with restriction rules and keywords for writing TCS. Two case studies were performed to assess the applicability of RTCM and a commercial video conferencing system was used to evaluate the *aToucan4Test* tool. *aToucan4Test* could correctly generate 246 executable test cases from 9 test case specifications of subsystems of the video conferencing system. The study also evaluated the effort to use RTCM and *aToucan4Test* using the average time for

deriving the executable test cases, which is 0.5 minutes. Mai *et al.* [113] addressed the problem of automatically generating executable test cases from security requirements in natural language. Mai *et al.* proposed an approach to generate security vulnerability test cases from use case specifications that capture malicious behavior of users. Similarly to previous work, Mai *et al.* evaluated the approach with an industrial case study in the medical domain. The evaluation indicated that the proposed approach can automatically generate test cases detecting vulnerabilities.

Prior work also used NLP techniques for test case prioritization and fault localization. Peng *et al.* [142] investigated program change-based test case prioritization using Information Retrieval (IR) techniques, in which the textual similarity between the program changes and the tests is used to rank tests for execution. Four techniques were used to compare and rank the tests, such as BM25, LDA, LSI, and TF-IDF, which transforms the text data into a numeric vector using bag-of-words. The proposed techniques were evaluated using cost-aware and cost-unaware metrics related to the Average Percentage of Faults Detected (APFD). Lachmann *et al.* [94] investigated test case prioritization of system-level, black-box test cases written in natural language. Test case textual descriptions were pre-processed (with techniques such as tokenization and stemming) and converted to numeric vectors using the frequency of terms occurring in the test cases. These vectors were combined with other test case meta-data (e.g., failures revealed by the test cases) to rank test cases based on their importance. Hemmati *et al.* [74] also studied test prioritization using natural language, black-box test cases. Three techniques were proposed (text diversification, topic modeling, and history-based test prioritization) and evaluated on Mozilla Firefox projects. The evaluation showed that, in rapid release environments, test case failure history can be used to effectively prioritize test cases for execution. DiGiuseppe and Jones [48] proposed a Semantic Fault Diagnosis (SFD), which automatically provides natural language descriptions of software faults. Using information extracted from the source code text (e..g, class names, comments, and other keywords), SFD can present

developers not only with the pass and fail outcome of a test execution, but also a list of words that describe the topics related to the fault. Finally, Fry and Weimer [55] presented an approach that relies on textual features (e.g., term frequency vectors) from source code and defect report descriptions to localize defects in the source code. Using a similarity score to compare the representations of a defect report and the source code files, the approach ranks the source code files such that files at the top are more likely to contain the defect.

The aforementioned works used different NLP techniques to perform several tasks related to testing, such as to automatically generate different types of test cases, test case prioritization, and fault localization. In contrast, we propose an approach that leverages different NLP techniques to extract text embeddings and can automatically identify similar test cases. The approach can be used to identify and remove redundant test cases written in natural language.

## 2.5 Proposed approach

In this section, we demonstrate our proposed approach through a running example. Our approach starts by clustering similar test steps, which are then used to identify similar test cases. We adopt a test step-based approach since test steps have a simpler grammatical structure compared to whole test cases. Also, a whole test case, which consists of all of its test steps aggregated, is not a coherent document as the test steps in a test case might be very different from each other. For example, in the same test case, one test step might be related to the 'login' functionality and another test step might be related to 'purchasing a membership'. Finally, using a test step-level approach provides more flexibility for recommending improvements not only related to whole test cases but also to individual test steps in the future. Figure 2.2 presents an overview of our approach, which consists of three stages: (1) pre-processing of test cases, (2) clustering of similar test steps and (3) identification of similar test cases. Next, we explain the stages of our approach, and we present an example that

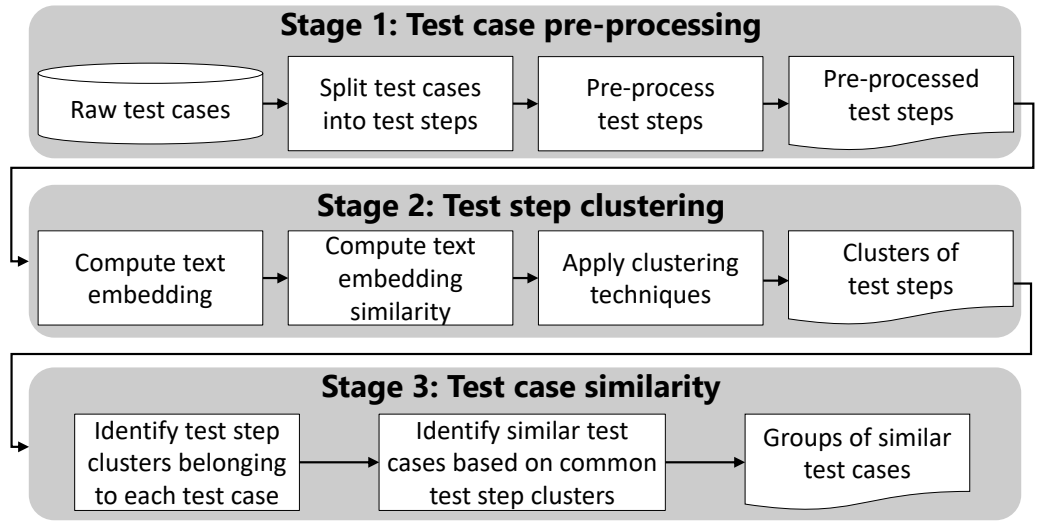



Figure 2.2: Overview of our proposed approach.

demonstrates the necessity of our approach.

### 2.5.1 Stage 1: Test case pre-processing

Our approach relies only on test cases that are written in natural language, which means that there is no source code available for our test cases. The input to our approach consists of unprocessed (raw) test cases. Table 2.1 presents three test cases (TC1, TC2, and TC3) that we use as a running example to describe how our approach identifies similar test cases. As we can observe, each test case contains an identifier, a name and a type. In addition, a test case has one or more test steps, which are instructions that the tester must perform in order to achieve the overall objective of the test case. Note that this objective is generally not explicitly specified. Test steps might be related to one or more game assets, which are the content of the game (e.g., in-game items, characters, and maps). The test steps that we collect to perform our experiments are explicitly identified (i.e., each test step has its own field within a test case). Therefore, we can directly collect the test steps and identify to which test case they belong. Each test step is assigned a unique identifier and is pre-processed. Initially, we used tokenization to transform the step sentences into a list of words. To ensure that we have high-quality data, we obtained a list of the unique words in our

Table 2.1: Running example that shows the test case fields: test case identifier (TC ID), test case name (TC name), test case type (TC type), test step identifier (TS ID), test step (TS) before pre-processing, and test step (TS) after pre-processing.

| TC ID | TC name | TC type | TS ID | TS (before pre-processing) | TS (after pre-processing) |
|---|---|---|---|---|---|
| TC1 | Log in to an existing account | Login | TS1.1 | Login to the game using an existing account that has completed the tutorial | [login, game, using, existing, account, completed, tutorial] |
| | | | TS1.2 | Select the Playing from School portal | [select, playing, school, portal] |
| TC2 | Assignment with many students | Education | TS2.1 | Update the assignment adding students | [update, assignment, adding, student] |
| | | | TS2.2 | Request the next skill and question from the algorithm gateway for the 1st student on the assignment | [request, next, skill, question, algorithm, gateway, student, assignment] |
| | | | TS2.3 | Request the next skill and question from the algorithm gateway for the middle student on the assignment | [request, next, skill, question, algorithm, gateway, middle, student, assignment] |
| TC3 | Student has multiple assignments | Education | TS3.1 | Request the next skill and question from the algorithm gateway for one of the students that was on the assignment | [request, next, skill, question, algorithm, gateway, one, student, assignment] |
| | | | TS3.2 | Remove student from the first assignment | [remove, student, first, assignment] |
| | | | TS3.3 | Request the next skill and question from the algorithm gateway for one of the students that was on the assignment | [request, next, skill, question, algorithm, gateway, one, student, assignment] |
| | | | TS3.4 | Remove the student from the second assignment | [remove, student, second, assignment] |

data and manually inspected the list to identify misspelled words, which were used to build a list of [misspelled_word, fixed_word] tuples. The manually built tuple list was used to programmatically replace misspelled words with the corresponding fixed words across the entire dataset. We then removed stopwords (such as "a", "of", and "the") as they do not add meaning to the sentences. Also, we applied lemmatization to the words to have a consistent terminology across the data. Finally, similar to prior work [101], we removed words that occur only once in the whole dataset (507 out of 2,599 unique words) as they may result in incorrect embeddings due to the small amount of data for these words. Overall, a test case instance can be represented

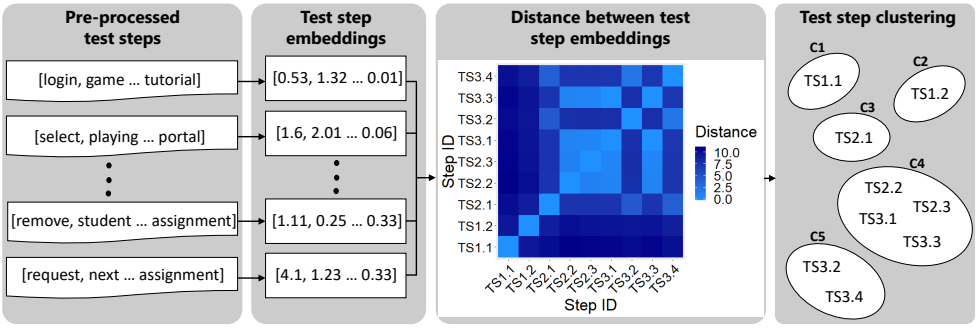



Figure 2.3: Overview of stage 2 of our approach with the running example.

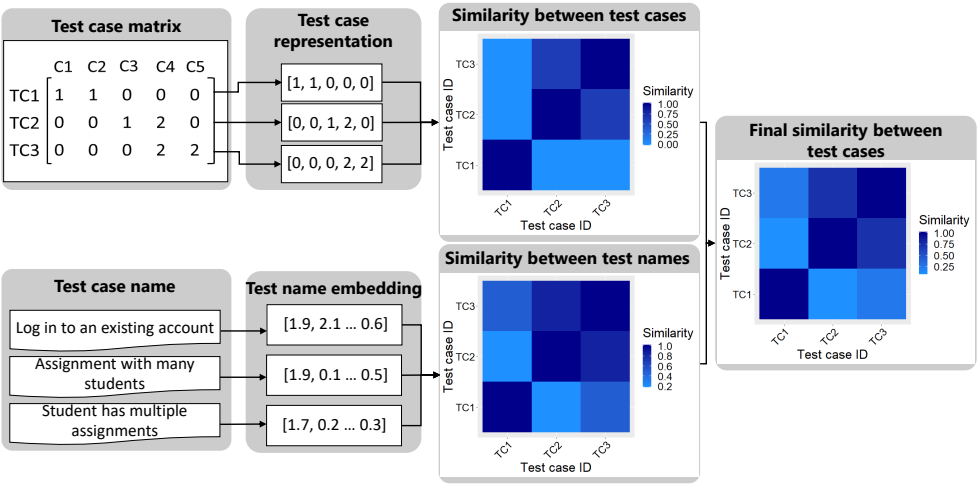



Figure 2.4: Overview of stage 3 of our approach with the running example.

by the triple:

$$<test\_case\_name, test\_case\_type, test\_steps>$$

### 2.5.2 Stage 2: Test step clustering

In the second stage, our approach clusters similar test steps. Figure 2.3 shows how the steps of the three test cases are processed in this stage. Before applying a machine learning algorithm to text data, we need to transform the text into a numeric representation [193, 194]. Our approach starts by transforming each test step into one or more numeric vectors (text embedding). The pairwise similarity between steps (in terms of embedding distance) is then computed. The computed distances between the text embeddings of the test steps can be used to capture their similarity. In par-

ticular, embeddings that are close in the embedding space should represent similar steps.

Finally, our approach leverages the computed distances to identify clusters of similar test steps. While steps that have a small distance between them should belong to the same cluster, steps with larger distances should be in different clusters.

### 2.5.3 Stage 3: Test case similarity

In the last stage, our approach leverages the clusters of test steps identified in stage 2 together with the test case name to find similar test cases. Figure 2.4 shows how the TC1, TC2, and TC3 test cases are processed in this stage. The relationship between test cases and test step clusters is represented through a matrix in which each row is a test case (TC1, TC2, and TC3) and each column is a step cluster (C1, C2, C3, C4, and C5). Initially, for each test case (matrix row), the approach identifies the test step clusters (matrix column(s)) that contain one or more steps of the test case. Our approach supports the use of binary (which yields a matrix consisting of 0's and 1's) or numeric flags. Note that a numeric flag represents the number of test steps present in the identified cluster. After filling in the matrix, each test case is represented by the corresponding binary or numeric vector (a row in the matrix) with a length corresponding to the total number of test step clusters. Test cases are then compared to each other in terms of the similarity between their representation vectors. Finally, to incorporate knowledge from the test case name, the approach computes the pairwise similarity between test case name embeddings and combines this similarity metric with the one obtained from the test step clusters. The final test case similarity score is a weighted sum between the test step cluster and the test case name metrics. For the running example, our approach identifies the TC2 and TC3 test cases as similar but both are different from the TC1 test case. A QA engineer can then investigate those test cases to decide, for example, whether they are redundant or should be improved.

Table 2.2: Motivational example of two similar test cases.

| **Test case name 1** | **Test steps 1** |
|---|---|
| Boots - Ruin Dweller Boots (Got Item) | 1. Verify item name<br>2. Verify item icon |
| **Test case name 2** | **Test steps 2** |
| Boots - Got Item | 1. Verify item name<br>2. Verify item icon<br>3. Sanity check on wearable item - check "Got item dialogue" for Boots |

### 2.5.4 Motivational Example

Li et al. [101] proposed an approach to cluster similar test steps in natural language. Even though their work is supposed to be used for test steps only (and not entire test cases composed of one or more test steps), we evaluated an adaptation of their approach on our dataset using their best-performing techniques. The adaptation approach consists of using Word2Vec for text embedding, Word Mover's Distance (WMD) for text similarity, and hierarchical agglomerative clustering together with K-means for clustering. To be able to apply their approach, we considered a test case to be represented by either (1) the test case name concatenated with all the test steps or (2) all the test steps together. In both scenarios, the approach failed to cluster the two similar test cases presented in Table 2.2 (and there are many more examples in our dataset which could not be identified as similar by our adaptation of Li et al.'s approach).

The intended purpose of Li et al.'s approach is to cluster test steps (and not test cases). When considering a whole test case as a single test step, the granularity of Li et al.'s approach becomes too coarse and it considers the differences between the two test cases too large to cluster them together. However, when comparing the test

cases step by step, instead of as a single blob of text, our approach detects that many of the steps overlap, hence clustering the two test cases in the same cluster.

## 2.6 Dataset and ground truth

We collected 3,323 test case descriptions written in natural language. The test cases under study were manually designed to test the *Prodigy Game*[3], a proprietary, educational math game with more than 100 million users around the world. Each test case is composed of one or more test steps and, in total, there are 15,644 steps. There is an average of 4.71 (and a median of 2) test steps per test case. We also collected the predefined type of the test case regarding the part of the game that is being tested. The test case type is available for 2,053 test cases (62% of the total number of test cases). All the test steps are pre-processed according to the pre-processing steps as explained in Section 2.5.1. Manual testing using test cases that are described only in natural language is still a common practice across several industries [73, 74, 101, 130, 140, 145, 186]. The test cases of the Prodigy game are similar in structure to natural language test cases from other projects. Hemmati *et al.* [74] studied Mozilla Firefox projects, with manual test cases described only in natural language, with an objective and one or more test steps, similarly to our test cases. Li *et al.* [101] studied natural language test cases of a large industrial app (WeChat) with similar characteristics, such as 4.04 words per test step description on average (our average is 3.92) and test steps with simple grammatical structure. Walter *et al.* [186] studied automotive test cases in natural language, also with similar characteristics, such as an average of 3.57 test steps per test case for one of the studied systems (our average is 4.71).

To evaluate the performance of our approach (stage 2, for test step clustering, and stage 3, for test case similarity), we used our dataset to manually build a ground truth of similar test steps (stage 2) and similar test cases (stage 3), as we explain below.

[3]https://www.prodigygame.com/main-en/

**Ground truth of similar test steps (stage 2 of our approach).** We randomly selected a representative sample from all 15,644 test steps with a confidence level of 95% and a confidence interval of 5%, which corresponds to 394 steps. The test step samples were manually analyzed in an incremental manner: when analyzing test step *n*, we looked at all the (*n-1*) previously clustered steps to verify if step *n* should be assigned to an existing cluster or to a new cluster. To determine if two test steps are similar, we looked for two main characteristics of the data: (1) if the steps are textually similar or (2) if the steps give the same or similar instructions for testing, even if the textual descriptions are not similar. If any of those two characteristics are observed, we cluster the two samples together. Below, we show examples of pairs of test steps to demonstrate both characteristics:

(1) Textually similar test steps:

- "Play before 4pm and attempt to play video."
- "Play before 8am and attempt to play video."

(2) Test steps with similar instructions for testing:

- "Verify the game zones that can be selected by the student."
- "Check which game zones are available to the student."

The ground truth of similar test steps ended up with a total of 213 clusters and an average of 1.9 (standard deviation of 2.0) test steps per cluster. We also found that the largest cluster has 15 test steps. The fourth author independently validated the ground truth on a sample of 80 randomly selected pairs of test steps, which corresponds to a representative sample with a confidence level of 95% and a confidence interval of 10%. For each pair of test steps, the fourth author indicated if the two test steps should be in the same cluster (i.e., if they are similar) or not. We reached an agreement of 96.25% (which corresponds to a kappa coefficient [40, 95] of 0.89 or

almost perfect agreement). The reached agreement demonstrates that the manual clustering process is straightforward (though time-consuming).

**Ground truth of similar test cases (stage 3 of our approach).** We randomly selected a representative sample of test cases with a confidence level of 95% and a confidence interval of 5%, which corresponds to 381 test cases. Similarly to the way that we built the ground truth of similar test steps, the test case samples were manually analyzed in an incremental manner: when analyzing test case $n$, we looked at all the ($n$-1) previously clustered test cases to verify if test case $n$ should be assigned to an existing cluster or to a new cluster. To determine if two test cases are similar, we looked for the same characteristics (1) and (2) as for the test steps. If any of those two characteristics are observed, we cluster the two samples together. Note that, to analyze test cases, we consider the test case name, test case type, and all the steps that compose the test case. The ground truth of similar test cases ended up with a total of 242 clusters and an average of 1.6 (standard deviation of 1.9) test cases per cluster. For this ground truth, we found that the largest cluster has 21 test cases.

## 2.7 Evaluating our approach for clustering similar test steps

In this section, we discuss the experiments that we performed to evaluate our approach for clustering similar test steps in an industrial setting.

### 2.7.1 Evaluated techniques

Our approach consists of several steps that can be implemented through different techniques and models. To evaluate our approach, we performed experiments with combinations of five different text embedding techniques, two similarity metrics, and two clustering techniques. Figure 2.5 presents an overview of the experiments that we performed to address RQ1. Different NLP techniques can be used for text embedding

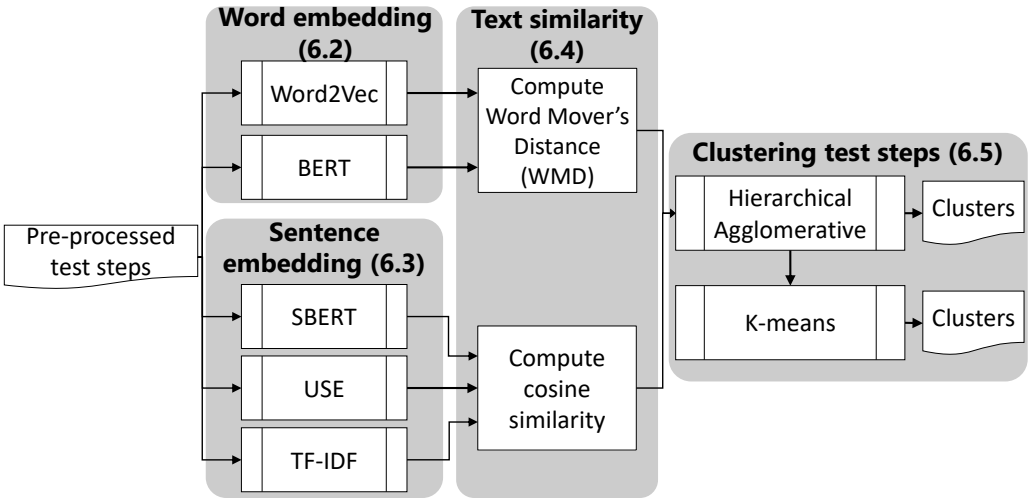



Figure 2.5: Overview of the experiments to identify clusters of similar test steps.

at different granularities, such as words, sentences, and short paragraphs [31, 46, 96, 123, 124, 152]. As our test steps usually consist of a single sentence and the test steps are transformed into a list of words after pre-processing, we adopt word-level and sentence-level text embedding. We used two techniques to obtain text embeddings at the word-level (Word2Vec [123, 124] and BERT [46]) for the test steps and computed the embedding similarity using the Word Mover's Distance (WMD) metric [93]. For text embeddings at the sentence-level, we used three techniques (SBERT [152], Universal Sentence Encoder [31], and TF-IDF [86, 161]) and used the cosine similarity to compare the embeddings. For both types of embeddings, we applied the hierarchical agglomerative [169] and K-means [50] clustering techniques to obtain clusters of similar steps.

### 2.7.2 Configuration of the word embedding techniques

*Word2Vec.* We trained a Word2Vec model using all 15,644 test steps that we collected. Furthermore, to provide more context to the embedding model during training, we concatenated the test case type (available for 2,053 test cases) and test case name to each step. We used an embedding vector of length 300 (as in the original study that proposed the Word2Vec model [124]). We used the continuous bag-of-words (CBOW)

model architecture of Word2Vec with two context words as this configuration provides the highest test step clustering performance, which was determined through an experiment in which we varied the number of context words from one to ten. We initialized the word embeddings with the weights from the large-scale pre-trained model released by Google.[4] This model contains 3 million word embeddings with dimension 300 and was trained on a Google News corpus with approximately 100 billion words. For words that are present in our dataset but not in the pre-trained model (and, therefore, cannot be initialized with pre-trained weights), we followed a process proposed by Li *et al.* [101] to initialize the word embeddings. We computed the mean and standard deviation of the initialized words and initialized the remaining words with samples of a normal distribution parameterized by the computed mean and standard deviation. Finally, the outcome of the training process is the word embeddings learned with our data.

*BERT.* In this work, we used the pre-trained model released by Google[5] (*pre-trained BERT*) to obtain contextual embeddings of the test steps. Furthermore, we used a model with additional pre-training using our own corpus of test steps (*domain-adaptive pre-trained BERT*) to obtain the contextual embeddings. We explain the configurations of both models below.

Pre-trained BERT. For the pre-trained model, we used the uncased (case-insensitive) version of the base model [46, 179]. We transformed the test step text into the BERT format by adding the *[CLS]* and *[SEP]* tokens respectively to the start and end of each test step text. The test step was then tokenized with BERT's own tokenizer. Finally, we used the tokenized steps to extract the contextual embeddings. As explained in Section 2.3.1, we can adopt different pooling strategies to obtain the embedding vector for a word. We performed experiments with four different pooling strategies to combine the layers (as suggested by the original paper's authors [46]): using only

[4]https://code.google.com/archive/p/word2vec/
[5]https://github.com/google-research/bert

the second-to-last layer, summing the last four layers, averaging the last four layers, and concatenating the last four layers. We found that summing the last four layers achieves the best performance with our data. Finally, we used the average of sub-word embeddings (see Section 2.3.1) to obtain the original out-of-vocabulary word embedding.

Domain-adaptive pre-trained BERT. We also performed additional pre-training of BERT with our corpus. For the additional pre-training, after experimenting with the base and large models, we decided to use the uncased version of the BERT large model as the initial checkpoint (i.e., we performed the additional pre-training on top of the pre-trained large model). We followed the same process to configure the test step text to a BERT-friendly format. However, differently from the pre-trained model, using the second-to-last layer (instead of summing the last four layers) achieves the best results for the domain-adaptive pre-trained BERT model.

### 2.7.3 Configuration of the sentence embedding techniques

*Sentence-BERT (SBERT).* We performed experiments with three available pre-trained SBERT models suitable for our task (see Section 2.3.1): *paraphrase-distilroberta-base-v1*, *stsb-roberta-base*, and *stsb-roberta-large*. We decided to use the *paraphrase-distilroberta-base-v1* model since it achieves the best results with our data. To obtain the embeddings for the test steps, we just provided the test steps directly as parameters to the SBERT model.

*Universal Sentence Encoder (USE).* To obtain the test step embeddings with the USE model, we provided the steps directly as parameters to the USE model.

*TF-IDF.* Finally, we also used TF-IDF to extract the numeric vector representations of the test steps. For each word, we computed its importance in a single test step relative to all the other test steps. We used the *TfidfVectorizer* class provided by sklearn[6] with default parameters, which includes a smoothing parameter of 1 so that

[6]https://scikit-learn.org/stable/

out-of-vocabulary words can be properly handled.

### 2.7.4 Computing the test step similarity

*Word Mover's Distance (WMD).* We used the Word Mover's Distance (WMD) [93] metric to measure the similarity between test step word-level embeddings. The WMD metric is suitable to be used together with the Word2Vec and BERT models because of the property that distances between embedded words in the embedding space are semantically meaningful, which is a property that WMD relies on [93]. Therefore, for word-level embeddings, we used the WMD metric instead of other metrics, such as the cosine similarity. We computed the pairwise WMD metric between any two test steps and built a distance matrix of dimension *[15,644 x 15,644]*. The more similar two steps are, the lower is the WMD metric, with the lowest bound being zero for exactly matching steps.

*Cosine similarity.* Since cosine is a widely used metric to measure similarity between text vectors [60, 79, 99, 162], we used the cosine to measure the similarity between test step sentence-level embeddings. Note that we cannot use the WMD metric for sentence-level embeddings since WMD requires the embeddings of each word individually instead of a whole sentence embedding. Similarly to the way we computed the WMD metric, we computed the pairwise cosine similarity between any two test steps and built a distance matrix of dimension *[15,644 x 15,644]*. As the cosine similarity score measures the cosine of the angle between the numeric vectors of two steps, the smaller the angle, the larger its cosine and the more similar the two test steps are.

### 2.7.5 Clustering test steps

*Hierarchical Agglomerative Clustering.* We applied the hierarchical agglomerative clustering technique to the distance matrix that we built in the previous step (Section 2.7.4). We used the average linkage criterion (with Euclidean distance), which means that the clustering algorithm merges pairs of test step clusters that minimize

the average distance between each observation of the pairs.

*K-means.* To apply the K-means clustering technique, we used the test step embeddings obtained with the word/sentence embedding techniques (Sections 2.7.2 and 2.7.3). Note that, for word-level embeddings, we transformed the embedding vectors of the words of a test step into a single vector to represent the whole test step by computing the word embeddings' average [117, 216]. Furthermore, to speed up the execution of K-means, we used the centroids of the clusters obtained by the hierarchical approach as the initialization centroids, similarly to prior work [101, 110].

Regarding the number of clusters for both clustering techniques, we chose the number of clusters that maximized the F-score (which is our evaluation metric, as explained in Section 2.7.6). We performed a search by varying the number of clusters from 50 up to 15,000 with a step of 50, and for each value we executed both clustering approaches and computed the F-score. Finally, we selected the (optimal) number of clusters for which each clustering technique achieved the highest F-score. Note that the optimal number of clusters might be different for the hierarchical clustering and K-means.

**Ensemble approach.** Each text embedding technique that we used has different characteristics and properties to extract word or sentence embeddings, which leads to different clusters of test steps. Therefore, attempting to mitigate each model's specific weaknesses and based on prior work [47, 187] which showed that ensemble approaches might perform well for certain tasks (e.g., classification and clustering), we built an ensemble approach that uses majority voting. The approach uses the clusters generated by each previous single approach and starts by getting the set of all the test steps in the data. Then, it iterates through each test step and performs pairwise comparisons with all the other test steps. Suppose the approach (1) starts with test step $TS_n$. Then, (2) for each pair ($TS_n$-$TS_{n+1}$, $TS_n$-$TS_{n+2}$, etc.), the approach verifies if the majority of the single approaches (i.e., at least three out of five) assigned that

pair to the same cluster or not and does the same assignment (i.e., puts the pair together if the majority did so or just skips the test step being compared to $TS_n$). After this first pass, we have all the test steps that are similar to $TS_n$. (3) The test steps that are clustered with $TS_n$ are removed from the main set of test steps (i.e., they will not be analyzed anymore). (4) We then repeat procedures (1), (2), and (3) for the next test step that is not part of the $TS_n$ cluster. When there is no test step left in the main set of test steps, the approach finalizes and we have a set of clusters of similar test steps.

**Baseline.** We used two baselines to evaluate the performance of our proposed approaches for test step (TS) clustering. The first baseline (*TS-Baseline 1*) assigns test steps to the same cluster only if those steps are exactly the same after pre-processing, similarly to Li *et al.* [101]. The second baseline (*TS-Baseline 2*) uses the Word2Vec technique together with the WMD similarity metric and only assigns two test steps to the same cluster if the WMD similarity of those steps is zero (i.e., their embeddings are the same).

### 2.7.6 Evaluation metric

We are interested in penalizing both the false positives (to avoid excessive suggestions of similar test steps when they are not similar) and false negatives (to avoid missing out many similar test steps). Therefore, we used the F-score metric (as shown in Equation 2.1) to evaluate the test step clustering approaches as this metric captures the trade-off between precision (related to false positives) and recall (related to false negatives). Even though we focus on the F-score, we also report the precision and recall of the proposed techniques along with the time necessary to execute the techniques. The executed time consists of the median time (in minutes) of five executions. Using the test steps present in the manually built ground truth of similar test steps, we analyzed all the pairs of test steps, similarly to prior work [101]:

- *True positive (TP)*: when a pair of steps is included in the same cluster by our

Table 2.3: Precision, recall, and F-score of the test step clustering approaches along with the execution time (in minutes) and the optimal number of clusters obtained using HAC and K-means. In the last column, we show the F-score distribution for a number of clusters between 2,150 and 3,000.

| Text embedding technique | Clustering | Precision | Recall | F-score | Exec. time (min) | Num. clusters | F-score for num. of clusters between 2,150 and 3,000 |
|---|---|---|---|---|---|---|---|
| TS-Baseline 1 | Identical text | 100.00 | 54.32 | 70.40 | 1 | 4,407 | - |
| TS-Baseline 2 | Identical embeddings | 100.00 | 54.32 | 70.40 | 151 | 4,393 | - |
| Word2Vec | HAC | 93.74 | 79.19 | 85.85 | 149 | 2,650 | 80  87 |
| Word2Vec | K-means | 94.24 | 80.77 | 86.99 | 150 | 2,650 | 80  87 |
| BERT | HAC | 89.57 | 80.25 | 84.65 | 157 | 3,050 | 80  87 |
| BERT | K-means | 91.14 | 79.89 | 85.15 | 160 | 3,050 | 80  87 |
| Domain-adaptive BERT | HAC | 93.89 | 78.66 | 85.60 | 159 | 3,300 | 80  87 |
| Domain-adaptive BERT | K-means | 94.29 | 78.66 | 85.77 | 162 | 3,300 | 80  87 |
| SBERT | HAC | 94.67 | 78.30 | 85.71 | 2 | 3,350 | 80  87 |
| SBERT | K-means | 95.09 | 78.66 | 86.10 | 2 | 3,350 | 80  87 |
| USE | HAC | 90.26 | 78.48 | 83.96 | 1 | 3,050 | 80  87 |
| USE | K-means | 86.91 | 82.01 | 84.39 | 1 | 2,900 | 80  87 |
| TF-IDF | HAC | 91.90 | 82.01 | 86.67 | 2 | 2,500 | 80  87 |
| TF-IDF | K-means | 91.80 | 80.95 | 86.03 | 2 | 2,500 | 80  87 |
| Ensemble | - | 94.47 | 81.30 | 87.39 | 317 | 3,158 | - |

approach and the steps indeed belong to the same cluster in the ground truth.

- *False positive (FP)*: when a pair of steps is included in the same cluster by our approach but the steps do not belong to the same cluster in the ground truth.
- *True negative (TN)*: when a pair of steps is not included in the same cluster by our approach and the steps do not belong to the same cluster in the ground truth.
- *False negative (FN)*: when a pair of steps is not included in the same cluster by our approach but the steps belong to the same cluster in the ground truth.

We then computed the F-score metric as follows:

$$\text{F-score} = 2 \times \frac{precision \times recall}{precision + recall} \tag{2.1}$$

Where the precision corresponds to the proportion of true positives regarding all the pairs identified as positive ($\frac{TP}{TP+FP}$) and the recall corresponds to the proportion of true positives regarding all the existing positive instance ($\frac{TP}{TP+FN}$).

### 2.7.7 Findings

**Similar test steps that are written in natural language can be identified with an F-score of 87.39% by applying the ensemble approach. Word2Vec (K-means), TF-IDF (HAC) and SBERT (K-means) also have high F-scores (86.99%, 86.67%, and 86.10%, respectively), but TF-IDF and SBERT considerably reduce the execution time (from 150 minutes to 2 minutes compared to Word2Vec).** Table 2.3 presents the precision, recall, and F-score of all the approaches along with the execution time and the optimal number of clusters.

All the proposed approaches achieve a similar and high performance, with an F-score between 83.96% and 87.39%, except for both baselines, which have the same F-score of 70.40%. More specifically, the ensemble approach achieves the highest performance, with an F-score of 87.39%. If we look at the performance of the single models, Word2Vec with K-means has the highest F-score (86.99%), which is very close to the ensemble approach performance. TF-IDF with HAC achieves the second highest F-score (86.67%) among the single models, followed closely by SBERT with K-means (86.10%) and Domain-BERT with K-means (85.77%). By analyzing the F-score obtained by all the approaches for all the searched number of clusters (from 50 up to 15,000), we observed that the F-score plateaus when we use a number of clusters of 6,000 or higher. This means that, in practice, we do not need to search for the optimal number of clusters with values above 6,000. We also noticed that the F-score is always above 80% when the number of clusters is between 2,150 and 3,000. We can therefore use a number of clusters in that range to avoid searching for the

optimal number of clusters frequently.

Regarding the two versions of the BERT model, we observe that the domain-adaptive pre-trained BERT is a little better, with F-scores of 85.60% (using HAC) and 85.77% (using K-means), in comparison to the generic pre-trained BERT, with F-scores of 84.65% (using HAC) and 85.15% (K-means). One possible reason for the small gain is that we do not have large amounts of data for the domain-adaptive pre-training. However, our findings indicate that the additional pre-training is capable of improving the model performance and might be more helpful with larger datasets.

We can observe that for all the approaches except for TF-IDF, running K-means on top of HAC is beneficial as this increases the F-score. Note, however, that the gain in performance is minimal, such as 1.14% and 0.50% in absolute percentage point for Word2Vec and BERT, respectively. On average, applying K-means on top of hierarchical clustering increases the performance by 0.33% in absolute percentage point. The number of clusters obtained by the approaches with HAC does not necessarily need to be the same as the number of clusters obtained by the approaches with K-means. HAC and K-means are two different clustering techniques that we evaluate and, since they follow different procedures to cluster the data, they might achieve a different number of clusters. Note that, since we use the centers of the clusters obtained by HAC to initialize the K-means' centroids, K-means will converge fast as those initial cluster centers are often close to optimal or are in fact optimal in terms of F-score (which is the case when using HAC and K-means achieves the same number of clusters).

Table 2.3 also presents the precision and recall for all the approaches. Except for the baselines, we can observe that the precision varies from 86.91% (USE with K-means) up to 95.09% (SBERT with K-means) and the recall varies from 78.30% (SBERT with HAC) up to 82.01% (USE with K-means and TF-IDF with HAC). For the best performing models (ensemble approach and Word2Vec with K-means), both the precision and recall metrics are similar. Regarding the baselines, both of them

present a very high precision (100%) but with a low recall (54.32%).

Finally, the presented execution time is the median time (in minutes) of five executions of the techniques. Even though the ensemble approach has the highest performance for clustering test steps, this approach is computationally expensive as it requires the implementation and execution of all the other approaches, which takes around 317 minutes (about 5 hours) in total (using six cores on an Intel i7-8700 CPU to compute the WMD metric and a single core for all the other computations). However, we can achieve a very close performance with a single technique, such as Word2Vec with K-means (which takes around 150 minutes to execute using six cores to compute the WMD metric). TF-IDF (HAC) and SBERT (K-means) also achieve similar high F-scores (86.67% and 86.10%, respectively) but present much shorter execution times using a single core (2 minutes for both). Our experiments showed that both Word2Vec and BERT present a large execution time due to the large computational cost of computing the Word Mover's Distance, which makes SBERT a great alternative since it is considerably faster despite the slightly lower performance, and does not require further configurations or training as it uses a pre-trained model. The reported execution times are for the full test step clustering pipeline (test step pre-processing, word embedding training, test step similarity, and clustering) using the optimal number of clusters.

## 2.8 Evaluating our approach for identifying similar test cases

In this section, we discuss the experiments that we performed to evaluate our approach for identifying similar test cases that are specified in natural language. Below, we discuss four different techniques to identify similar test cases using the test step clusters obtained by the best-performing approach in Section 2.7 (ensemble approach).

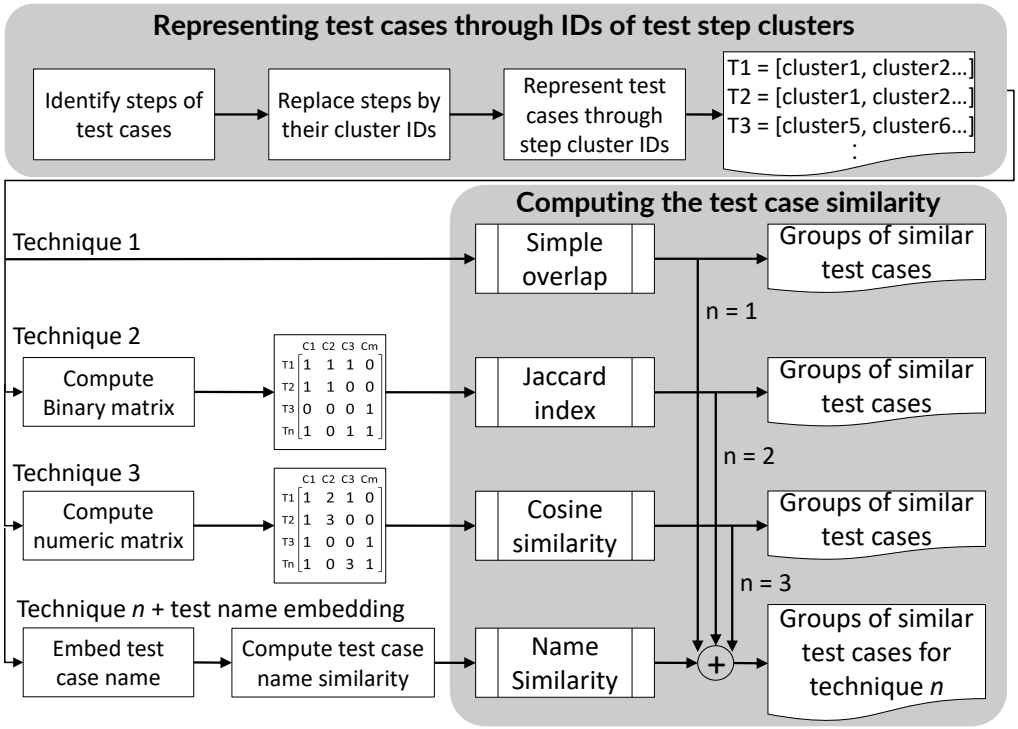



Figure 2.6: Overview of the experiments to identify similar test cases.

Table 2.4: Examples of test case representations (through vectors) obtained with the experimented three techniques and their versions with test case name embedding (*Technique n + name embedding*).

| **Test case** | **Test step** | **Test step cluster** | **Technique 1** | **Technique 2** | **Technique 3** | **Technique *n* + name embed.** |
|---|---|---|---|---|---|---|
| TC1 | TS1, TS2, TS3, TS4 | C1, C2, C3, C1 | [C1, C2, C3] | [1, 1, 1, 0, 0] | [2, 1, 1, 0, 0] | Technique $n$ + [TC1 Name embedding] |
| TC2 | TS1, TS5, TS6, TS7, TS8 | C1, C4, C5, C2, C5 | [C1, C2, C4, C5] | [1, 1, 0, 1, 1] | [1, 1, 0, 1, 2] | Technique $n$ + [TC2 Name embedding] |

### 2.8.1 Evaluated techniques

We performed experiments with three different techniques and variations of those techniques to identify similar test cases using the previously identified clusters of test step (with the ensemble approach). Figure 2.6 gives an overview of the experiments. To explain how each technique works, we use the two example test cases presented in Table 2.4.

In the example, there are two test cases (TC1 and TC2). TC1 contains four steps

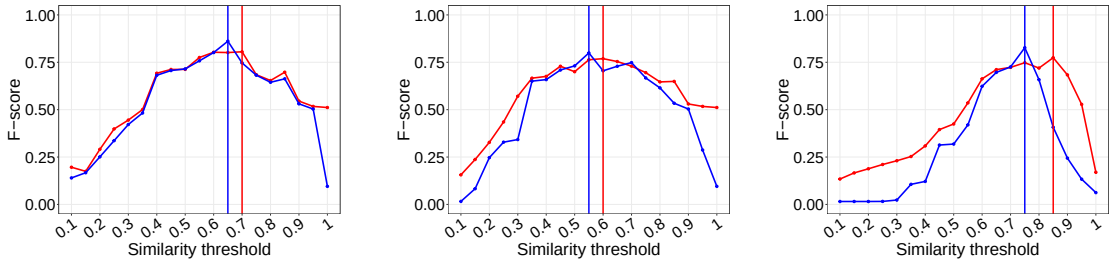
(TS1, TS2, TS3, TS4) and TC2 contains five steps (TS1, TS5, TS6, TS7, TS8). As we can see, only the TS1 step is shared between the test cases. In the test step cluster column, we can see the cluster ID to which each step belongs (TS1 belongs to the C1 cluster, TS2 belongs to the C2 cluster, and so on), where *Cn* is the ID of the cluster *n*. Note that different steps (such as TS2 and TS7) might belong to the same cluster (C2). Next, we explain each proposed technique using this example.

**Technique 1: Test step cluster overlap.** For this technique, we used only the identifiers of the test step clusters to represent test cases. For each test case, we gathered the unique list of cluster IDs that contain the test steps. For our running example, the TC1 test case is represented through the *[C1, C2, C3]* vector, while TC2 is represented through the *[C1, C2, C4, C5]* vector. Finally, we computed the pairwise similarity of any two test cases using a simple overlap metric, which indicates the proportion of overlap that test cases have in terms of test step cluster IDs, as shown below:

$$\text{Overlap} = \frac{length((TCn) \cap (TCm))}{max(length(TCn), length(TCm))} \tag{2.2}$$

Where $TCn$ and $TCm$ correspond to the representations of the test cases $n$ and $m$ through the unique cluster IDs, respectively. Intuitively, test cases that have a large overlap of test step clusters (even if the test steps themselves are different) should be similar since test steps in the same cluster are (most of the time) similar. For our example, the length of TC1 is three (C1, C2, C3), the length of TC2 is four (C1, C2, C4, C5), and the length of the intersection between TC1 and TC2 is two (C1, C2). Therefore, the overlap between the TC1 and TC2 test cases is: $\frac{2}{max(3,4)} = \frac{2}{4} = 0.5$ (50%).

We used the computed overlap as the similarity metric to compare the test cases. Furthermore, in order to determine the optimal similarity threshold (i.e., with the




(a) F-score for different similarity thresholds for *Technique 1* and *Technique 1 + test name*.

(b) F-score for different similarity thresholds for *Technique 2* and *Technique 2 + test name*.

(c) F-score for different similarity thresholds for *Technique 3* and *Technique 3 + test name*.

Figure 2.7: F-score for different similarity thresholds for our proposed techniques. The vertical line indicates the threshold that maximizes the F-score (red for Techniques 1, 2, and 3 and blue for their versions with the test name).

optimal tradeoff between false positives and false negatives) to be used to identify similar test cases, we performed a search by varying the threshold from 0.1 (10% of overlap) up to 1.0 (100% of overlap). Figure 2.7a shows how the F-score changes with the similarity threshold (the optimal threshold is indicated with the vertical red line). As we can see, our search showed that the threshold that provides the maximum F-score is 0.70, which means that two test cases should be considered similar if their overlap metric is at least 70%.

**Technique 2: Binary representation of test cases.** Similarly to Technique 1, for Technique 2 we used the test step clusters to represent test cases. However, instead of using the cluster IDs directly, we used a binary vector for each test case, in which we flagged the clusters that contain at least one test step of that case with a "1". Otherwise, we used "0". For our example, both test cases TC1 and TC2 are represented through a vector of length five because there are five different test step clusters in total (C1, C2, C3, C4, C5). TC1 is represented through the *[1,1,1,0,0]* vector (because TC1 has steps that belong to the clusters C1, C2, and C3, but no step belongs to the clusters C4 and C5), while TC2 is represented through the *[1,1,0,1,1]* vector. We built a matrix of dimension *[#test_cases x #test_step_clusters]*, where each row corresponds to a test case and each column corresponds to a test step cluster.

Finally, we computed the pairwise similarity of any two test cases using the Jaccard index, as used in prior work [1, 2] to calculate the similarity between binary vectors. Our search (see Figure 2.7b) shows that the optimal lower threshold for the Jaccard index is 0.60 (vertical red line), which means that two test cases are similar if their Jaccard index is equal to or larger than 0.60.

**Technique 3: Numeric representation of test cases.** Using a binary vector to represent test cases might not be sufficient for situations where test cases have more than one step in a cluster. Therefore, we modified the previous technique so that, instead of representing test cases as a binary vector, we represent test cases as a numeric vector. This numeric vector corresponds to the number of test steps that the test case has in each cluster. For our example, TC1 is represented through the *[2,1,1,0,0]* vector (because TC1 has two steps in the C1 cluster, one step in each of the C2 and C3 clusters, and no step in the C4 and C5 clusters). TC2 is represented through the *[1,1,0,1,2]* vector. We found that using a threshold of 0.85 (see Figure 2.7c, vertical red line) achieved the best performance in terms of F-score. This means that all the pairs of test cases that have a cosine similarity equal to or larger than 0.85 are considered similar by Technique 3.

**Including the test case name embedding.** For each technique mentioned above (Techniques 1, 2, and 3), we evaluated their versions with the test case name embedding as well: *Technique 1 + test name*, *Technique 2 + test name*, and *Technique 3 + test name*. For our example, both test cases TC1 and TC2 are represented through the same vectors discussed above for Techniques 1, 2, and 3. For the versions with the test case name, we combined the test step clusters with the test case name embedding.

To obtain the embeddings for the name, we used the best-performing text embedding technique from the experiments for test step clustering (which is Word2Vec). Following a similar process as we did for the test step clustering, we computed the pairwise similarity for any two test case name embeddings. To compute the final similarity score for test cases, we used the weighted sum between the similarity score

obtained with the test step clusters and the similarity score obtained with the test case name embeddings, as shown in Equation 2.3. Two test cases are considered similar if the final score is above a certain threshold.

$$\begin{aligned}\text{Final score} = \ & (\text{weight}) * (\text{test step cluster similarity}) \\ & + (1 - \text{weight}) * (\text{test case name similarity})\end{aligned} \tag{2.3}$$

To determine the best weight and threshold for the final score, we performed a search similarly to the threshold search that we did for Techniques 1, 2, and 3. We varied the weight from 0.1 up to 1.0 with a step of 0.1. For each weight, we varied the threshold from 0.10 up to 1.0 with a step of 0.05. For each combination of weight and threshold, we obtained the clusters of similar test cases and computed the evaluation metrics.

For *Technique 1 + test name*, we found that the optimal weight is 0.9, i.e., the similarity score from the test step clusters contributes with 90%, while the similarity score from the test case name contributes with 10% to the final score. For *Technique 2 + test name*, the optimal weight is 0.8 and for *Technique 3 + test name*, the optimal weight is 0.5. Furthermore, as Figures 2.7a, 2.7b, and 2.7c show (vertical blue line), the optimal similarity thresholds for *Technique 1 + test name*, *Technique 2 + test name*, and *Technique 3 + test name* are 0.65, 0.55, and 0.75, respectively. Note that, due to space constraints and for a better visualization, Figures 2.7a, 2.7b, and 2.7c only display how the F-score changes with the threshold already using the optimal weights for each technique.

**Baseline.** We compared the performance of our proposed approaches with several baselines for test case (TC) similarity identification. *TC-Baseline 1* considers two test cases to be similar if they have the exact same steps (regarding the text of the step). *TC-Relaxed baseline 1* considers two test cases similar if they differ only in N test steps (with N=1), with the remaining test steps being exactly the same. Note that *TC-Baseline 1* has N=0. *TC-Baseline 2* considers two test cases to be

similar if they have the same name. *TC-Relaxed baseline 2* considers two test cases similar if their test case name embedding vectors are close, for which we embedded test case names with Word2Vec (as in Section 2.7.2) and searched for the optimal similarity threshold. *TC-Baseline 1*, *TC-Relaxed baseline 1*, *TC-Baseline 2*, and *TC-Relaxed baseline 2* are simple, intuitive and computationally cheap methods. *TC-Baseline 3* and *TC-Baseline 3 + test name* are based on existing document clustering techniques [79, 131]. We transformed the textual description of the test cases into a numeric vector (vectorization) using a traditional vectorization method (TF-IDF). We then computed the pairwise cosine similarity between the test cases and applied the HAC and K-means clustering algorithms. For both baselines, we consider the whole test case as a single document. For *TC-Baseline 3*, we represent a test case with all its test steps aggregated, while for *TC-Baseline 3 + test name* we consider the test case name concatenated to the aggregated test steps. For both cases, we followed a similar procedure as for the test step clustering (Section 2.7.5), in which we searched for the optimal number of clusters.

### 2.8.2 Evaluation metric.

To evaluate our approaches for finding similar test cases, we used the manually built ground truth of similar test cases to compute the precision, recall, and F-score. We followed the exact same process as we did previously for the test step clustering (Section 2.7.6).

### 2.8.3 Findings

**Clusters of similar test steps and test case name embeddings together can be used to identify similar test cases with an F-score of 86.13%**. Table 2.5 presents the F-score of all the techniques that we evaluated along with precision, recall, execution time and the optimal similarity threshold.

We observe that *Technique 1 + test name* achieves the highest performance in

Table 2.5: Precision, recall and F-score of the test case similarity techniques along with the execution time (in seconds) and the optimal similarity threshold.

| Technique | Technique name | Prec. | Recall | F-score | Exec. time (sec) | Thresh. |
|---|---|---|---|---|---|---|
| TC-Baseline 1 | Identical steps | 99.42 | 31.07 | 47.35 | 18.28 | - |
| TC-Relaxed baseline 1 | Close number of identical steps | 93.91 | 38.57 | 54.68 | 23.36 | - |
| TC-Baseline 2 | Identical name | 50.00 | 0.18 | 0.35 | 6.53 | - |
| TC-Relaxed baseline 2 | Close test name embedding | 59.77 | 27.86 | 38.00 | 7.98 | - |
| TC-Baseline 3 | Aggregated test steps | 68.87 | 72.68 | 70.72 | 325.82 | - |
| TC-Baseline 3 + test name | Test case name + aggregated test steps | 27.25 | 39.46 | 32.24 | 343.29 | - |
| Technique 1 | Test step cluster overlap | 82.43 | 78.75 | 80.54 | 16.33 | 0.70 |
| **Technique 1 + test name** | **Test step cluster overlap + test name embedding** | **83.19** | **89.29** | **86.13** | **33.63** | **0.65** |
| Technique 2 | Binary repres. of test cases | 78.34 | 75.53 | 76.90 | 144.09 | 0.60 |
| Technique 2 + test name | Binary repres. of test cases + test name embedding | 77.80 | 81.96 | 79.82 | 154.16 | 0.60 |
| Technique 3 | Numeric repres. of test cases | 90.45 | 67.67 | 77.42 | 6.52 | 0.85 |
| Technique 3 + test name | Numeric repres. of test cases + test name embedding | 94.37 | 74.82 | 83.47 | 23.84 | 0.75 |

terms of F-score (86.13%), followed by *Technique 3 + test name*, which achieves an F-score of 83.47%. We also observe that, even though *Technique 3* achieves a higher performance than *Technique 2*, the improvement is very small (0.52 in absolute percentage point). This indicates that using the number of test steps in each cluster (instead of just flagging whether the cluster contains a test step) slightly improves the performance of the test case similarity technique. Further incorporating the test case name information considerably improves the performance for all the three techniques. *Technique 1 + test name* improved the F-score in 5.59 absolute percentage point from *Technique 1*. The absolute percentage point improvements for Techniques 2 and 3

were 2.92 and 6.05, respectively.

Regarding the baselines, all the experimented techniques perform considerably better than all the baseline methods. We observe that *TC-Baseline 2* achieves an extremely low F-score (0.35%) and that *TC-Baseline 3* presents the best F-score among the baseline methods (70.72%). Note that for *TC-Baseline 3*, HAC performed better than K-means, while for *TC-Baseline 3 + test name*, K-means performed better. We only report the results using the best-performing clustering algorithms. We also see that the precision of the *TC-Baseline 1* is very high (99.42%), but the recall is very low (31.07%). Two main reasons explain why all our proposed approaches perform better than the baseline methods. First, *TC-Baseline 1 and 2* and *TC-Relaxed baseline 1 and 2* are too simple to capture all the different types of similar test cases (e.g., test cases that have a different name and number of test steps, which, despite describing similar testing activities, are written differently). Second, *TC-Baseline 3* and *TC-Baseline 3 + test name* consider a whole test case as a single document. However, a whole test case is not a coherent document as test steps might be very different from one another. For example, in the same test case, one test step might be related to the 'login' functionality and another test step might be related to 'purchasing a membership'. Therefore, using whole test cases as documents for the similarity detection is a much more difficult task.

The presented execution time is the median time (in seconds) of five executions of the techniques and reflect the time necessary to run the test case similarity techniques considering that the clusters of test steps (obtained in the previous stage, as described in Section 2.7) are available. Apart from the *TC-Baseline 3* and *TC-Baseline 3 + test name* and *Technique 2* and *Technique 2 + test name*, all the other techniques present a similar execution time, which ranges from 6.52 seconds (*Technique 3*) up to 33.63 seconds (for the best-performing *Technique 1 + test name*). Using the optimal similarity threshold, the best technique (*Technique 1 + test name*) found 427 groups of similar test cases with two or more test cases in each group. The 427 groups contain

a total of 2,193 test cases (65.9%), i.e., there are 2,193 test cases in the test suite that have at least one similar test case, according to this technique. This leaves 1,130 test cases for which there is no other similar test case. On average, each group has two similar test cases, with a standard deviation of four.

To understand the output produced by our best technique, we manually inspected a representative sample of 100 of the obtained groups of similar test cases. The sample was randomly selected with a confidence level of 95% and a confidence interval of 10% from the 427 groups of similar test cases obtained by the best technique (*Technique 1 + test name*). We identified four main types of similar test cases (shown in Table 2.6). While Type 1 corresponds to test cases with the same steps for different game assets, Type 2 regards test cases that have slightly different steps to indicate the asset being tested (e.g., **backpack hat** and **backpack wand**). Type 3 refers to test cases with a large overlap of steps but one of them has more/fewer steps, which might indicate unnecessary or missing steps. Finally, Type 4 regards redundant test cases, which are written differently and may have a different number of steps, but the testing objective is the same. The last type of similarity helps to identify and remove redundant test cases from the test suite.

## 2.9 Discussion

In this section, we revisit the research questions and discuss the validation of our approach.

**RQ1: How effectively can we identify similar test steps that are written in natural language?**

Our experiments demonstrate that we can identify similar test steps with a high performance in terms of F-score. We showed that an ensemble approach using a combination (majority voting) of different techniques (five text embedding techniques with two similarity metrics and two clustering algorithms) achieves the highest performance. Such ensemble approach has a large computational cost as it requires the

Table 2.6: Examples of the four types of test case similarity. Differences between test cases' steps are highlighted in bold.

| **Similarity type** | **Test case name** | **Test steps** |
|---|---|---|
| (1) Same steps for different game assets | Check Hat - In Backpack | 1. Verify item name<br>2. Verify item icon |
| | Check Wand - In Backpack | 1. Verify item name<br>2. Verify item icon |
| (2) Slightly different steps for different game assets | Equip Hat | 1. Trigger equip functionality via **backpack hat** item slot<br>2. Trigger unequip functionality via **backpack hat** item slot |
| | Equip Wand | 1. Trigger equip functionality via **wand backpack** item slot<br>2. Trigger unequip functionality via **wand backpack** item slot |
| (3) Test cases with additional/missing steps | Check Consumables (Water Resist) | 1. Use in battle<br>2. Check battle bonus<br>3. **Check item card name**<br>4. **Check item card stats** |
| | Check Food (Popcorn) | 1. Use in battle<br>2. Check battle bonus |
| (4) Redundant test cases | Catch Firefly in Forest | 1. **Catch firefly in forest** |
| | Firefly Forest - Catch Firefly | 1. **Catch a firefly** |

execution of several different techniques. However, we showed that using a single technique (such as Word2Vec or TF-IDF) can also provide a high performance while being less computationally expensive.

**RQ2: How can we leverage clusters of test steps to identify similar test cases?**

Our experiments demonstrate that we can use the clusters of similar test steps identified in the first part of the study to represent test cases and identify the similar ones. More specifically, representing test cases through a vector that captures the number of test steps in each cluster boosts the similarity technique performance. Furthermore, we showed that combining the clusters of similar test steps with the embedding of the test case name achieves an even higher performance. Our experiments showed that the optimal weight (for our data) for the test step clusters and the test case names is 90% (i.e., the similarity score from the test step clusters contributes with 90%, while the similarity score from the test case name contributes with 10% to the final similarity score). In addition, we can use a threshold of 0.65 for the final similarity score to decide whether two test cases are similar (i.e., two test cases are similar if their final similarity score is equal to or larger than 0.65).

**Validation with developers.** To validate the results of our approach, we did an informal interview with a QA expert at Prodigy Education to discuss whether our results are valid and how they can be used in practice and improve the testing process. We selected a purposive sample [53, 63, 178] to explicitly select test cases that cover the different types of similar test cases that we identified.

Overall, the expert validated the different types of test case similarity that we identified and mentioned that our approach can help the QA to improve the quality of the test cases. More specifically, the QA expert pointed out four practical usages of our approach, as we explain next.

- **Identification of redundant test cases**, which are test cases that are described differently (e.g., because they were written by different professionals) but test exactly the same aspect/asset of the game.

- **Reuse of existing test cases** when creating new ones for new features of the

game. In this case, existing descriptions of test cases can either be fully or partially (e.g., a few test steps) reused. The reuse can be full (e.g., when a new test case instructs the tester to perform the exact same steps of an existing test case but for a new game asset, such as a new consumable item in the game) or partial (e.g., when a new test case performs a similar test as an existing test case but with a few differences, such that only part of the test steps of the existing similar test case can be reused). By reusing test cases, the overall quality of the test suite improves with more consistent and homogeneous descriptions in terms of terminology. Furthermore, reusing test cases reduces the manual effort and time required for designing and creating new test cases.

- **Identification of test cases with missing steps**. A few test case samples that we discussed with the expert were indeed groups of similar test cases which perform the same task, but some of the cases had fewer steps than what is actually performed by a tester. We further investigated those cases with the QA expert and found out that the missing steps were scattered across the test suite (in different cases) and should be merged with the steps of the main test case.

- **Identification of test cases which are redundant but one of the cases has additional steps.** This occurs when a new test case is created based on existing ones, but some steps are added for clarification purposes and the older test case is not removed from the test suite.

## 2.10 Threats to Validity

**External validity** relates to the generalizability of our findings. One threat is that our findings rely on the test case descriptions of an educational game company. Test cases of organizations from different domains might be different (e.g., in terms of the used terminology, grammar complexity and structure, and characteristics of the

data, such as the distribution of test steps across the test cases) and might affect the results. However, as we explained in Section 2.6, our test cases are similar in structure to natural language test cases from other domains studied by prior work, such as WeChat [101], automotive systems [186], and Mozilla Firefox projects [74]. Our approach can be applied to natural language test cases from other industrial projects with similar characteristics as the test cases from our industry partner, such as the test cases from the projects and companies discussed above. Furthermore, our approach can be applied to well-maintained open-source projects which have test cases described in natural language that are composed of one or more individual test steps. Note that our approach consists of using clusters of similar test steps together with the similarity of test case name embeddings, which might be computationally expensive for large datasets. In addition, specifying a ground truth to be used with our approach can be challenging and time-consuming. Finally, further investigation is necessary to apply our approach to projects with different test case characteristics, such as different distributions of test steps across test cases or different test case structures (e.g., test cases which are not composed of individual test steps). Another threat is that our thresholds for optimal values (e.g., the number of clusters and the similarity score) likely do not apply to other systems. However, our method for searching for these values is generalizable.

**Internal validity** concerns the bias and errors due to the experimental design. One threat concerns the methods that we used for text embedding, text similarity and clustering. Even though we mitigated this threat by studying several different types of techniques (five different text embedding techniques, three similarity metrics and two clustering algorithms), different results might be achieved with other techniques. Future studies should further investigate additional methods and algorithms for text embedding, text similarity and clustering. Another threat is related to the manual analysis of the samples of test steps and test cases performed by one author to build the ground truth. The manual analysis is subject to error and bias because of hu-

man factors. To mitigate this threat, another author independently cross-validated a subset of the 20 randomly selected test steps, which achieved an agreement of 100%. Furthermore, a QA engineer with more than 5 years of experience in the company further validated the output produced by our technique.

**Construct validity** concerns the choices made during the construction of our experiments. One threat is related to the chosen parameters for the embedding techniques. We mitigated this threat by using well-studied or recommended values for such parameters. For example, our chosen length for the Word2Vec embedding vector is 300. This is a popular choice and was used in the original study that proposed the Word2Vec model [124], but models with different lengths might achieve different performances for the clustering task. Another threat concerns the pooling strategies that we used to combine the layers of the BERT models and extract the embedding vectors. Different strategies provide different word embedding vectors. Even though several strategies can be used, we compared the four strategies recommended by the original paper on BERT [46]. The use of the clusters obtained by the HAC algorithm to initialize K-means' centroids is another threat to validity. Even though we adopted this process to speed up the execution of K-means, using different initialization methods might produce different clusters and achieve a different performance. Future studies should investigate how the initialization of the K-means' centroids affects the performance of the test step clustering. Finally, another threat is regarding the evaluation of the test step clustering and test case similarity identification. We evaluated our proposed techniques using a random sample of test steps/cases, which may not reflect the characteristics of the entire population. To mitigate this threat, we randomly sampled the data with a confidence level of 95% and a confidence interval of 5%, which yielded a statistically representative sample.

## 2.11 Conclusion

Test cases written in natural language are often defined by different people who may use different terminology to refer to the same concept. As a result, many similar or redundant test cases may exist in the test suite, which increases the manual testing effort and the usage of development resources. Since manually identifying similar test cases is a time-consuming task, an automated technique is necessary.

In this chapter, we propose an approach to identify similar test cases specified in natural language. First, we evaluated different text embedding techniques, similarity metrics, and clustering algorithms to identify clusters of similar test steps (which compose test cases). We then leveraged the identified test step clusters together with the test case name to identify similar test cases. To evaluate the approach, we used test cases from an educational game company. We manually built a ground truth of similar test steps and test cases and computed the F-score metric. The approach evaluation shows that similar test steps can be identified with a high performance (an F-score of 87.39%) using an ensemble approach which consists of different NLP techniques. We can also achieve a similar performance (an F-score of 86.99%) using a single technique (Word2Vec). Furthermore, we identified similar test cases with a high performance (an F-score of 86.13%) using clusters of similar test steps combined with the similarity between test case names.

We show how we can identify similar test cases based only on their description in natural language with an unsupervised approach, which requires no labelled data nor human supervision. As indicated in an informal interview with a QA engineer, our approach has several usages in practice, such as supporting QA and developers to identify and remove redundant test cases from the test suite. Furthermore, existing groups of similar test cases can be leveraged to create new test cases and help to maintain a more consistent and homogeneous terminology across the test suite.

# Chapter 3

# Using Natural Language Processing Techniques to Improve Manual Test Case Descriptions

## 3.1 Abstract

Despite the recent advancements in test automation, 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 chapter, 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).

## 3.2 Introduction

Software testing is a fundamental and widely-performed, yet costly, activity for quality assurance of a software system [58, 73]. 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 [140, 145]. 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 [157]. 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 [101].

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 [145]. There-

fore, 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 chapter, 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 [101, 183].

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

[1] https://www.prodigygame.com/main-en/

[2] https://github.com/asgaardlab/21-markos-test_case_improvement_framework-code

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 [101] as the overall quality and consistency of the test suite will be higher.

The remainder of the chapter is structured as follows. In Section 3.3, we explain our framework. In Sections 3.4, 3.5, and 3.6, 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 3.7 and 3.8. Finally, we conclude the chapter in Section 3.9.

## 3.3 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.

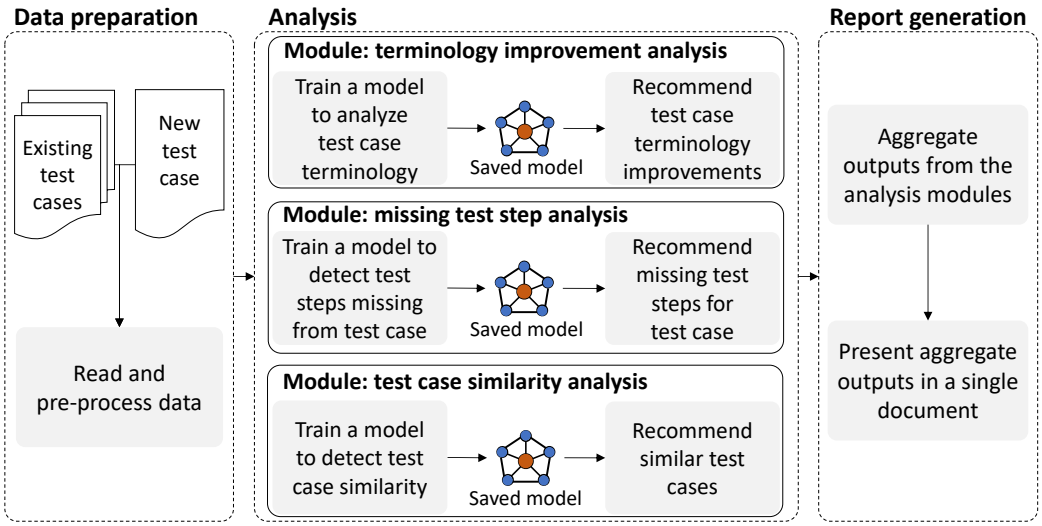



Figure 3.1: Our automated framework for analysis and feedback of test cases in natural language.

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 3.1 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 [101, 145].

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 3.1.

### 3.3.1 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 gives an instruction that must be manually performed by a human tester. Table 3.1 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

Table 3.1: Examples of test case descriptions from the Prodigy Math game.

| Test case name | Test case objective | Test step ID | 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 |

be adapted if users wish to apply other pre-processing steps for an analysis module.

### 3.3.2 Analysis component

**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 replaced by more likely alternatives, based on their usage in existing test cases. For our example test case (TC2) in Table 3.1, 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.

**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*"):

$$TC2_{\text{revised}} = \begin{cases} \textit{Log into the game with a child account} \\ \textit{Go to the membership page} \\ \textit{Go through the membership flow and become a member} \\ \textit{Verify that the user is a member} \end{cases}$$

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).

**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.

### 3.3.3 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.

### 3.3.4 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, 3.5, and 3.6.

### 3.3.5 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.

## 3.4 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 3.2 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.

### 3.4.1 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

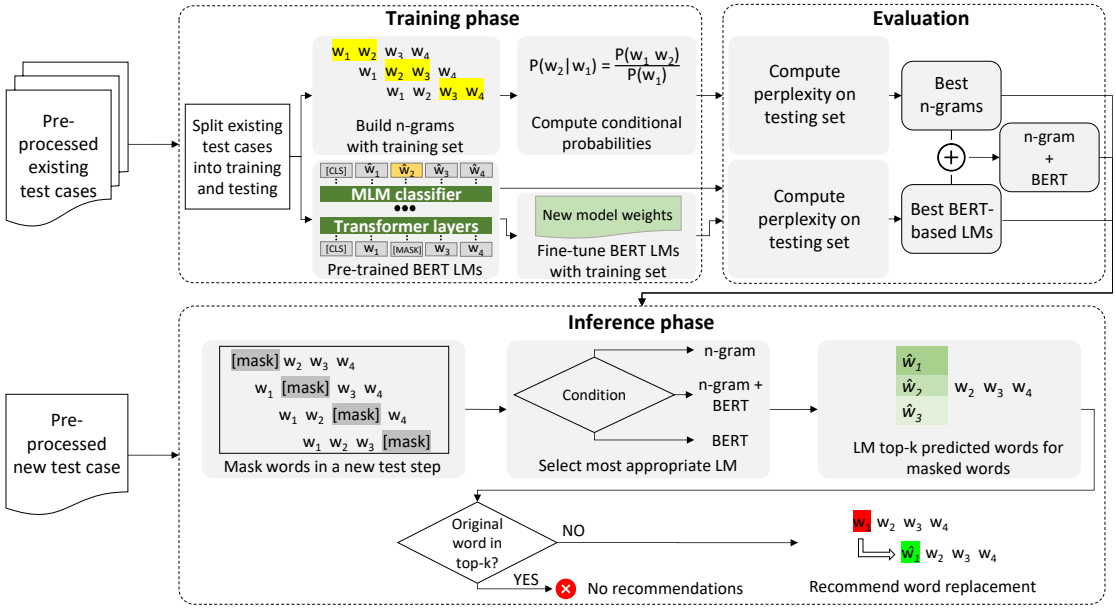



Figure 3.2: Our approach for recommending terminology improvements with n-grams and BERT-based language models (LMs).

(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 [77], 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*) [24, 77, 90]. 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) [90, 107]. 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) [38].

Neural models present several benefits over n-grams, such as the ability to 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 [14, 128], 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) [100] and have achieved the state-of-the-art performance in many NLP tasks [107, 143]. 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 [200].

**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* [23, 77, 90]. 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 $\langle unk \rangle$ 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 $\langle unk \rangle$ token from its counts just like another regular word [26, 90]. 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:

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

**Training BERT-based language models.**

To train our BERT-based language model, we used the *BertForMaskedLM* class from Huggingface [200] 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 [46], 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*.

### 3.4.2 Evaluation

#### Evaluation setup

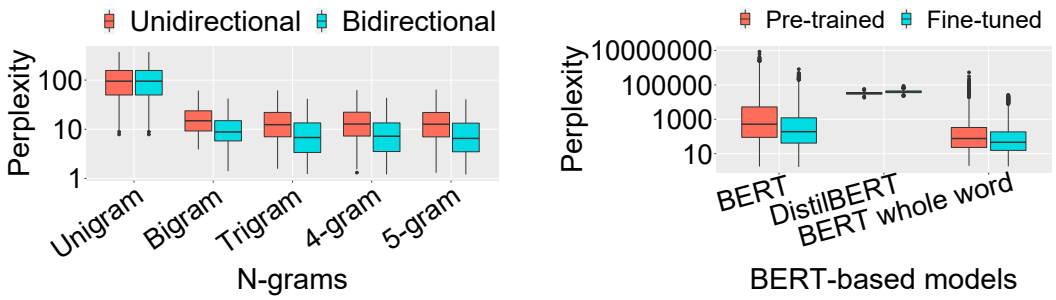
To train and evaluate the language models, we used all the data that we collected (test case name, objective and steps). We shuffled the data and split it into training (80%) and testing (20%) sets. For this approach, a preliminary analysis showed that keeping the stop words and words in their original 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 [26, 77, 90]. 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:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P\left(w_i \mid w_1 \ldots w_{i-1}\right)}} \tag{3.1}$$

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 [21]. We compare the distributions of perplexity for the different models with the non-parametric Wilcoxon rank-sum test [197] and compute the magnitude of the distribution difference with Cliff's delta effect size [109, 156].

#### Evaluation results

Figure 3.3 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




(a) Perplexity of n-gram models.

(b) Perplexity of BERT models.

Figure 3.3: Distributions of the perplexity* metric of the evaluated language models. *Log-transformed perplexity for better visualization.

sentence in the testing set). Figure 3.3a 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.

Figure 3.3b 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*.

**Comparing N-gram and BERT-based language models**

Since we cannot use perplexity to compare models built with different vocabularies [26, 90], 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 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 3.2) 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 3.2 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

Table 3.2: 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 |
| $\text{N-gram}_{\text{unseen-context}}$ + BERT | **0.67** | 0.70 | **0.75** | **0.34** | **0.5** | **0.5** | **0.84** | **0.85** | **0.88** |

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 3.2 shows the performance of the combined models ($\text{N-gram}_{\text{unseen-context}}$ + 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 $\text{N-gram}_{\text{unseen-context}}$ + 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%).

### 3.4.3 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.

## 3.5 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 3.4 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.

### 3.5.1 Training phase

#### 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*”].

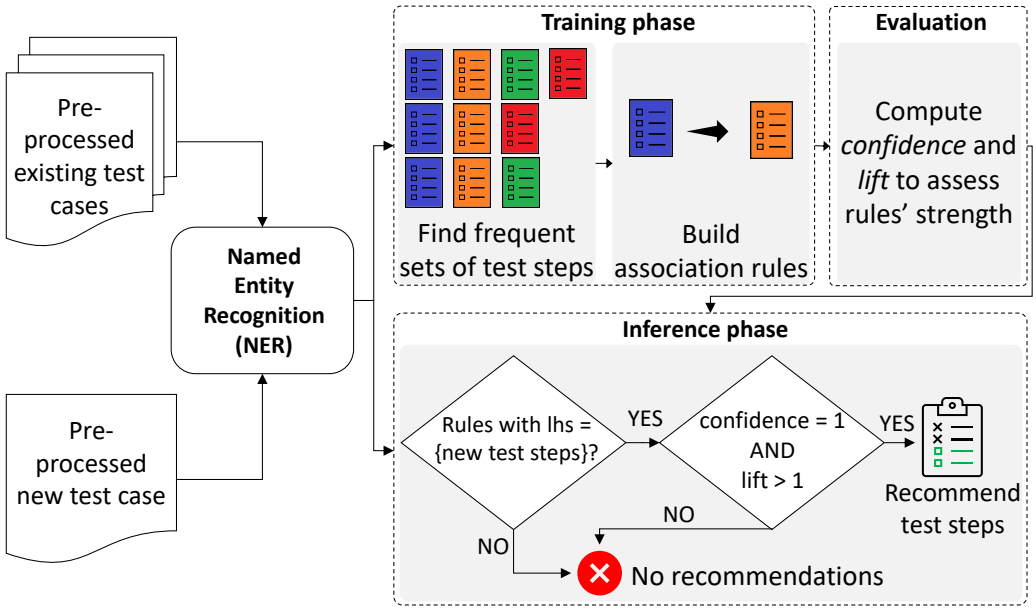



Figure 3.4: Our approach for recommending missing test steps using association rules.

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]

**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 [4, 5]. 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 [70, 71] 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 [70].

### 3.5.2 Evaluation

#### Evaluation metric

To evaluate the rules' strength, 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

[3]https://spacy.io/

side of the rule). The confidence metric 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 [121]. 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.

**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 $2501^{\text{th}}$ test case, which we suppose is a new one. In the next iteration, we added the $2501^{\text{th}}$ test case to the test suite, built the rules with those 2501 test cases, and applied the rules to the $2502^{\text{th}}$ test case. This process continued until we reached the last test case.

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.

**Evaluation results**

Using a *min_confidence* of 0.5 together 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 3.5.2. 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.

### 3.5.3 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., 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.

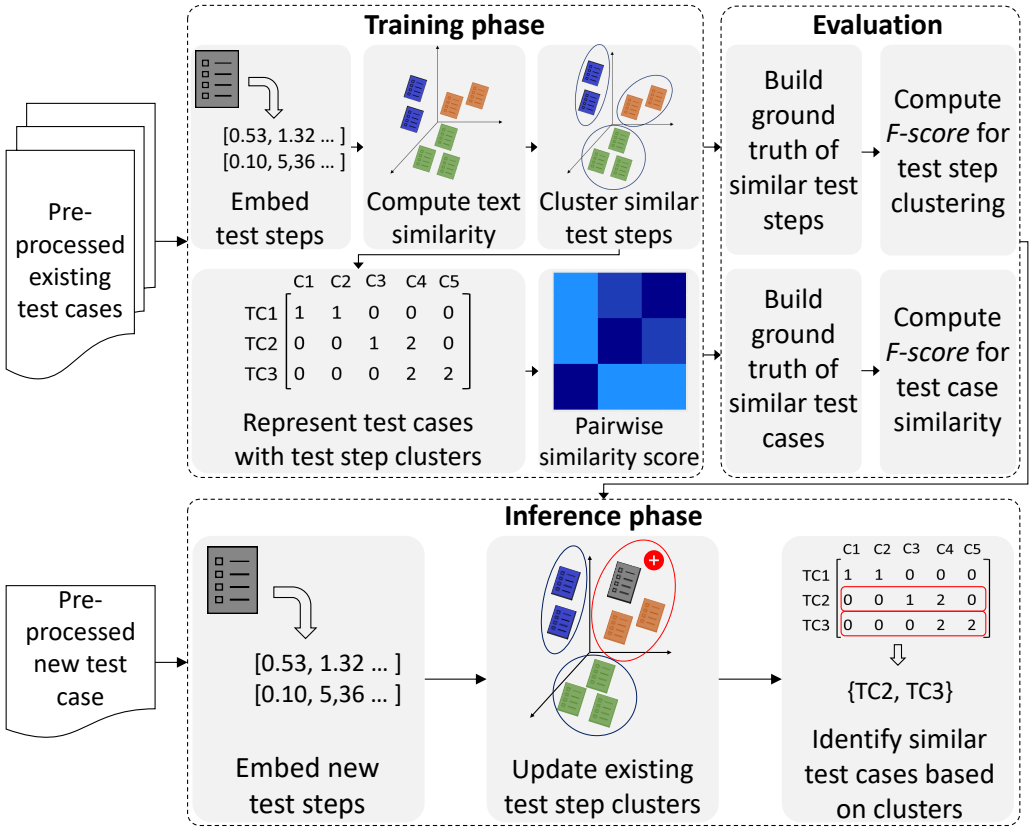



Figure 3.5: Our approach for recommending similar test cases using text embedding and clustering techniques.

## 3.6 The test case similarity analysis module

Our approach for recommending similar test cases was proposed in our prior work [183]. 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. [101] and (2) identifying similar test cases using the clusters of test steps (test case-level stage). Figure 3.5 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 [183].

### 3.6.1 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 [194]. 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 3.5. 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 [183], we thoroughly evaluated the described approaches with several different techniques using the data from our industry partner.

### 3.6.2 Evaluation

Based on our prior work [183], we selected the best performing approach for clustering test steps, which uses Word2Vec [124] to embed the test steps, Word Mover's Distance (WMD) [93] to compute the similarity between test step embeddings, and K-means [50] 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 0.65. This means that if an existing test case has a final cosine similarity score of more than 0.65 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 90% (i.e., the test step clusters contribute with 90% and the test case name contributes with 10% to the final similarity score).

### 3.6.3 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 0.65 compared to the new test case.

## 3.7 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 in many different ways, as we discuss below.

Wang *et al.* [189] proposed an approach to automate the generation of executable, system-level test cases 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.* [113] 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 [75] proposed an approach to predict the failure of system-level test cases 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 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.* [73] 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 [107, 133]. For instance, while Nguyen *et al.* [133] used program analysis and a statistical language model (n-gram) to develop a technique to complete code, Liu *et al.* [107] 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 specifi-

cations 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.

## 3.8 Threats to Validity

A threat to the **external** validity concerns to the generalizability of our framework and its evaluations. Our findings rely on the test cases from an educational math game and using test cases from a different domain might yield different results. Another threat is that the results might differ if other text embedding or clustering techniques are used. Future studies should investigate if our 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. 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. Future work should investigate a wider range of those metrics and whether having more than one consequent in a rule is better. 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 were performed with the existing test cases, which are unoptimized. Even though the test cases were written by experienced QA engineers, 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.

## 3.9 Conclusion

In this chapter, 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 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).

# Chapter 4

# Prioritizing Natural Language Test Cases Based on Highly-Used Game Features

## 4.1 Abstract

Software testing is still a manual activity in many industries, such as the gaming industry. But manually executing tests becomes impractical as the system grows and resources are restricted, mainly in a scenario with short release cycles. Test case prioritization is a commonly used technique to optimize the test execution. However, most prioritization approaches do not work for manual test cases as they require source code information or test execution history, which is often not available in a manual testing scenario. In this chapter, we propose a prioritization approach for manual test cases written in natural language based on the tested application features (in particular, highly-used application features). Our approach consists of (1) identifying the tested features from natural language test cases (with zero-shot classification techniques) and (2) prioritizing test cases based on the features that they test. We leveraged the NSGA-II genetic algorithm for the multi-objective optimization of the test case ordering to maximize the coverage of highly-used features while minimizing the cumulative execution time. Our findings show that we can successfully identify the application features covered by test cases using an ensemble of pre-trained models

with strong zero-shot capabilities (an F-score of 76.1%). Also, our prioritization approaches can find test case orderings that cover highly-used application features early in the test execution while keeping the time required to execute test cases short. QA engineers can use our approach to focus the test execution on test cases that cover features that are relevant to users.

## 4.2 Introduction

Software testing is an essential, yet costly, quality assurance activity during the software development life cycle [15, 59, 69, 73]. Despite the recent advances in test automation techniques [111, 140, 145, 173], manual tests are still widely performed across different industries [69, 140, 145, 180, 196]. In the gaming industry, for example, game developers face several challenges to automate tests and, consequently, manual testing is a predominant practice [140, 145, 180, 183, 184]

Manually executing tests is a tedious activity and requires a large amount of human effort as testers need to perform several steps to achieve the testing goal [183, 184]. As systems grow and the number of test cases increases, it becomes impractical to execute all manual test cases, mainly in a scenario with short release cycle [69, 73, 75, 94].

Prior work proposed several approaches to optimize the execution of test cases when resources are restricted, such as prioritizing test cases during regression testing [52, 73, 116, 134, 136, 146, 147, 166, 191, 217]. However, most proposed approaches do not work for manual test cases as (1) they depend on test case source code, which does not exist for manual test cases and (2) the execution history of test cases, which could be out-of-date or difficult to be accessed [206] or is generally not meaningful for manual test cases as they tend to be specified at a higher-level. Because of these two limitations of manual test cases, it is difficult to automatically prioritize their execution based on a meaningful metric.

In this chapter, we propose an approach for prioritizing manual test cases that

are written in natural language based on the application feature(s) that they test. In particular, we prioritize test cases that test highly-used application features, to ensure that the limited testing resources are used to test features in which bugs will affect the largest group of users. Our approach performs a multi-objective optimization with the widely-used NSGA-II genetic algorithm [44] to find optimal orderings of test cases according to two objectives: (a) highly-used feature coverage and (b) test case execution time. For objective (a), we need to identify the link between the test cases and the application features to identify which features are covered by test cases. We then collect the feature usage data for each feature. To identify this link, we leverage the strong zero-shot capabilities of several pre-trained language models.

We evaluated and optimized our approach for the data of a game from our industry partner (*Prodigy Education*).[1] Our experiments were performed with the test cases in the test suite of *Prodigy Education*.

The main contributions of our work are as follows:

- We build an automated mapping between natural language test cases and the game feature(s) that they cover.
- We propose a novel prioritization technique for natural language test cases based on the game feature(s) that they cover, in particular, the highly-used game features.

The source code of our experiments is available online.[2] The remainder of this chapter is structured as follows. Section 4.3 describes our industrial case subject and Section 4.4 gives an overview of our approach for test case prioritization. Section 4.5 presents the experiments and results to build our zero-shot ensemble model. Sections 4.6 and 4.7 present and discuss the prioritization experiments and results. We discuss practical aspects of our approach in Section 4.8, related work in Section 4.9

[1] https://www.prodigygame.com/main-en/
[2] https://github.com/asgaardlab/natural-language-test-prioritization

and the threats to validity in Section 4.10. Finally, Section 4.11 concludes the chapter.

## 4.3 Industrial case study subject

In this chapter, we applied our approach to the *Prodigy Math game* (from Prodigy Education[3]), which is a proprietary, online, RPG-style educational math game with more than 100 million users around the world. The game contains over 50,000 math questions spanning Grade 1-8. The players play the role of a character (a wizard) in the Prodigy world and can go to several 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. We use the test cases designed by Prodigy Education developers, the usage data generated by the players and the features of the *Prodigy Math game* as input for our approach.

**Dataset characteristics.** Our case study subject has 1,146 test cases that are written in natural language. Each test case contains the following fields:

- a test case name.
- an objective with the main goal of the test case.
- the time required to execute the test case, as provided by developers and QA engineers.
- one or more steps that the tester must perform.

The total combined execution time of the test cases is 133 hours. In total, the test cases cover 110 features of the *Prodigy Math game*. Every test case covers at least one feature, and a feature may be covered by more than one test case. For example, the "login" feature is covered by 27 test cases. In our data, a test case covers a median of 2 game features.

[3]https://www.prodigygame.com/main-en/

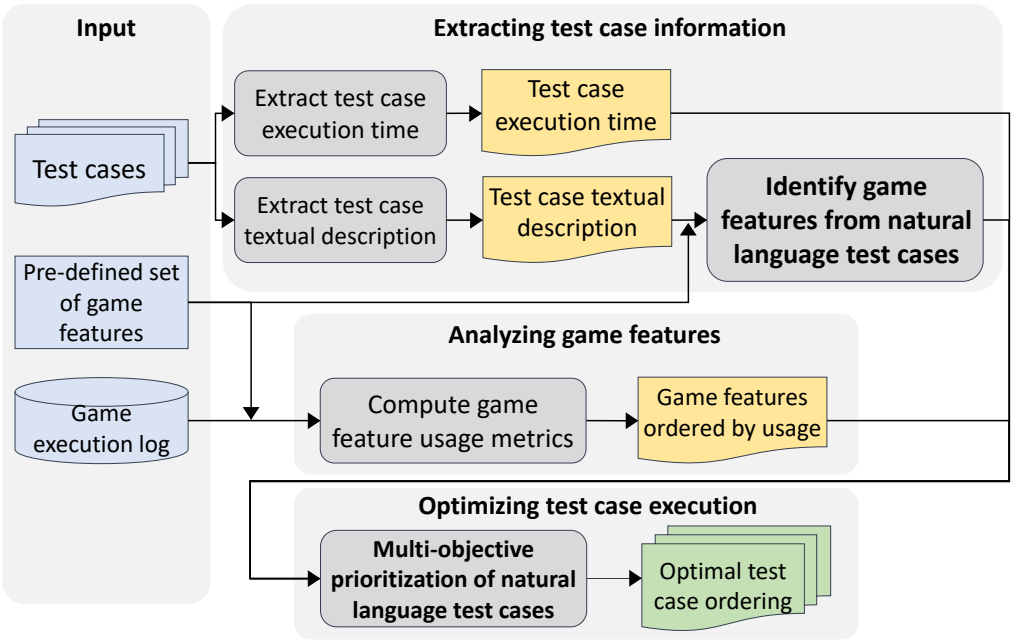



Figure 4.1: Overview of our approach for prioritizing natural language test cases.

## 4.4 Overview of our approach for test case prioritization

Our approach for prioritizing natural language test cases consists of two steps: (1) automatically identifying the tested game feature(s) from natural language test case descriptions and (2) prioritizing test cases based on the highly-used game features covered by test cases. Our approach finds the orderings of test cases that maximize the number of highly-used features covered early in the test execution and minimize the execution time. Figure 4.1 presents an overview of our approach.

### 4.4.1 Input

Our approach takes as inputs (1) manual test cases specified in natural language, (2) a pre-defined set of features of the application under test, and (3) the data generated from the interaction of users with the system (e.g., an execution log).

### 4.4.2 Extracting test case information

Our approach starts by extracting the execution time and textual descriptions from test cases. We use the concatenation of the test case name and objective as the textual description of the test case. We then use several techniques with strong zero-shot capabilities to identify the game features tested by test cases as we do not have a mapping between test cases and the game feature(s) that they cover. Since a test case might cover more than one feature, our approach performs a multi-label classification of test cases with those techniques. We chose the zero-shot approach because we do not have labeled data to train a classifier from scratch or even to fine-tune pre-trained models, as they require large amounts of data. Manually labeling data to train a classifier is not feasible because we have more than a hundred labels. Also, a manual classification of all the data is error-prone and infeasible due to the large number of test cases.

Table 4.1 shows an example test case with the corresponding covered features as identified by our zero-shot classification techniques. The test case named "Co-op: Joining a team" verifies whether players can join another player's team during a cooperative battle, and therefore, covers the cooperative battles feature (named "co-op") and, more specifically, the feature that allows players to join another player's team in a cooperative battle (named "co-op join"). We store the features identified by the zero-shot techniques in a *feature coverage vector*, which is *[co-op, co-op join]* in our example. And after this stage, every test case has a corresponding *feature coverage vector*.

### 4.4.3 Analyzing game features

Our approach uses game feature usage data to prioritize test cases that test highly-used features. We collect the *total number of uses* for each feature of the game for a specific period of time from the execution logs (in our case, the events that are stored in Prodigy's data warehouse). As the feature usage metric in our experiments, we

Table 4.1: Test case example with the covered features.

| | |
|---|---|
| **Name** | Co-op: Joining a team |
| **Objective** | Verify the functionality of joining another player's team |
| **Test steps** | 1. Log into the game<br>2. Try to join the team of another player<br>3. Verify that the student joined the other player's team |
| **Execution time** | 1 (minute) |
| **Covered features** | co-op, co-op join |

used the average number of feature uses per week for an entire school year (September 2021 to June 2022).

### 4.4.4 Optimizing test case execution

Finally, our approach performs a multi-objective optimization with the test case descriptions (with the corresponding *feature coverage vector*), the feature usage metric (*total number of uses*), and the test case execution time. Our approach optimizes the test case order based on a maximization of the number of highly-used game features covered by test cases and a minimization of the cumulative test execution time.

## 4.5 Identifying game features from natural language test cases

In our work, we leverage techniques with strong zero-shot capabilities to identify the link between the manual test cases and the features that they cover. Recently proposed pre-trained language models (such as BART [98]) have strong zero-shot capabilities, which means that their knowledge (obtained from very large amounts of data used during pre-training) can be transferred to a new domain which has no

labeled data [25, 46, 98]. As a result, we can apply these pre-trained models to new data and classes. Prior work has demonstrated the success of zero-shot learning in different fields, such as computer vision, speech, and natural language processing [35, 42, 45, 148, 173, 205].

### 4.5.1 Experiment setup

We did experiments to evaluate the performance of each individual zero-shot classification technique in our dataset. In addition, to have a more robust zero-shot classification, we experimented with different ensembles of the individual zero-shot techniques, as we explain below. For all the experiments, we used the 1,146 test cases of the *Prodigy Math game* and a list of 110 features that was defined by the game developers.

**Techniques for zero-shot classification.** We used three techniques that have strong zero-shot capabilities as demonstrated by prior work [45, 203, 205, 209]:

`BartLargeMNLI`

facebook/bart-large-mnli [98] is a model trained on the Multi-Genre Natural Language Inference (MNLI) dataset which has been shown to have strong zero-shot capabilities for text classification [209].

`CrossEncoderNLI`

cross-encoder/nli-distilroberta-base[4] is a model trained with a cross-encoder architecture to learn sentence embeddings [153] using the MNLI and the Stanford Natural Language Inference (SNLI) datasets, which also has zero-shot capabilities for text classification. For both `BartLargeMNLI` and `CrossEncoderNLI` models, we provide the textual description of a test case and a list of all the game features of the *Prodigy Math game*. The models output the game features sorted by their probability of being related to the test case.

[4]https://huggingface.co/cross-encoder/nli-distilroberta-base

`LatentEmb`

This is an unsupervised, similarity-based technique that uses text embedding methods to embed sentences (to be classified) and the candidate labels and uses a similarity metric (e.g., cosine) to find the labels that are similar to the sentence [45, 203, 205]. The sentence is then classified into the most similar labels (i.e., labels that are close to the sentence in the embedding space). We need to use a sentence embedding model to embed sentences and a word embedding models to embed labels (which are usually single words). In our work, we use the popular Sentence-BERT (SBERT) model [153] to embed test case textual descriptions (i.e., sentences), with the *sentence-t5-large* pre-trained checkpoint, and the Word2Vec embedding model [124] to embed the game features (i.e., labels). However, we cannot compute the cosine similarity directly between the embedding vectors from SBERT and Word2Vec since they have different scales (SBERT vectors are 768-dimensional, while Word2Vec vectors are 300-dimensional). To compare the embeddings, we need to have the embeddings from test case description and game features in the same space. Therefore, we performed a least-squares linear regression to learn a mapping between the SBERT and the Word2Vec spaces.[5] In practice, the mapping is a "transfer" matrix that can be used to transfer embeddings from one space to the other. We can then embed test case descriptions and game features with SBERT, use the matrix to transfer all embeddings to the Word2Vec space, and compute the consine similarity in the Word2Vec space. Figure 4.2 shows how we used `LatentEmb` to identify the game features covered by a test case example (test case 1).

To build the mapping, we need to embed the same set of words with both SBERT and Word2Vec and then perform a linear regression with those embedding vectors. We used the top-20k most frequent words from Word2Vec for the linear regression. With the computed matrix, we can embed the description of a test case and the game

[5]https://joeddav.github.io/blog/2020/05/29/ZSL.html

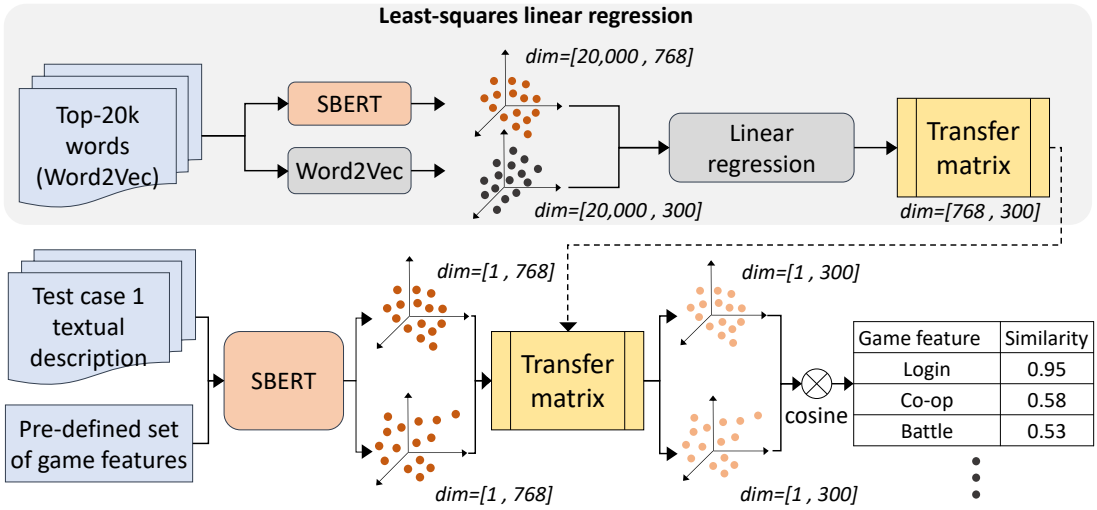



Figure 4.2: Overview of our `LatentEmb` technique for test case 1.

features with the SBERT model and used the matrix to transfer the embeddings to the 300-dimensional Word2Vec embedding space, where we can compute the cosine metric between the test case embedding and the game feature embeddings. We performed a preliminary analysis to evaluate other word embedding models (*Glove* and *Fasttext*), but using Word2Vec with the top-20k words achieved the best performance. We also used the preliminary analysis to determine the optimal classification threshold to be used for the cosine similarity in the `LatentEmb` approach and for the two pre-trained models (as their outputs contain the game features along with their probabilities). We used the best thresholds found in our analysis: 0.9 for `BartLargeMNLI`, 0.6 `CrossEncoderNLI`, and 0.2 for `LatentEmb` (we give more details on how we could evaluate the models in Section 4.5.2 below).

**Ensembles of techniques for zero-shot classification.** We also experimented with different ways of aggregating the classifications from each individual zero-shot technique to build an ensemble. Below, we explain the different aggregation methods that we explored.

**Ensemble with majority voting (`EnsMajorVoting`)**

Our initial idea is to use a majority voting approach to obtain the final classifications. This ensemble uses the sets of labels obtained from each individual zero-shot model and selects the labels provided by at least two models.

**Ensemble with full intersection (`EnsFullInters`)**

Aiming at having more robust and high-confidence classifications, this ensemble uses only the labels that were provided by all the three models.

**Ensemble with back-off using top-2 models (`EnsBackOffTwo`)**

The ensemble above (`EnsFullInters`) might be too strict sometimes, so we evaluated an ensemble that first obtained the labels that were provided by all the three models and, if that results in an empty set, this ensemble backs-off to the intersection of the two best individual models. Note that this is different from majority voting, which uses the labels provided by a minimum of any two models (not only the top-2 best models).

**Ensemble with back-off using all models (`EnsBackOffComplete`)**

If the intersection of all three models is empty, our approach backs-off and inspects the intersection of the two best individual models. Then, if the top-2 intersection is still an empty set, the approach backs-off again and inspect the intersection of the best and third best model. At last, if that also results in an empty set, we use the intersection between the second and third best models. Note that for all the ensembles, if the final result set is empty, we do not assign any labels to the test case.

**Baseline.** To have a "sanity check", we use a keyword search approach as baseline. We search the feature name in the test description to find if that feature is covered by that test case.

Table 4.2: Example of multi-label classification of test cases. Binary vectors for the "battle" feature are highlighted in green (true) and orange (predicted).

| Test name | True feature coverage vector | Predicted feature coverage vector | True binary vector [battle, login, co-op] | Pred. binary vector [battle, login, co-op] |
|---|---|---|---|---|
| Log in on mobile device | [login] | [co-op] | [0, 1, 0] | [0, 0, 1] |
| Start a co-op battle | [battle, co-op ] | [login, co-op ] | [1, 0, 1] | [0, 1, 1] |
| Check animations in battle | [battle ] | [battle ] | [1, 0, 0] | [1, 0, 0] |

### 4.5.2 Evaluation

To evaluate our proposed approaches, we manually labeled a subset of the test cases in the test suite of Prodigy Education. Please note that we only labeled the data to be able to evaluate the zero-shot and ensemble models. To use our approach in practice, no manual data labeling is necessary. To label the test cases, the first author, who has an extensive knowledge of the *Prodigy Math game*, randomly selected test cases until there was at least one labeled example for each label. In total, we labeled 211 test cases and there are, on average, 3 examples for each label. Using the labeled data, we computed the precision, recall, and F-score for all the evaluated approaches. As the F-score metric penalizes both the false positives and false negatives, we focus the discussions on that metric. To compute the evaluation metrics for our multi-label classification task, we used the `scikit-learn` package, which computes the metrics for each individual label and obtains the average. We used a weighted average because our labels are imbalanced (i.e., the number of labeled examples for each label is different). To clarify how we computed the F-score for our multi-label classification, Table 4.2 shows examples of three test cases, their true feature coverage vector, and the feature coverage vector predicted by a model. We used the *MultiLabelBinarizer* from `scikit-learn` to obtain the binary vectors from the feature coverage vectors. The binary vectors follow a fixed order of the features, such as [battle, login, co-op]

in our example, and contain '1' in case that feature is present and '0' otherwise. We then compute the evaluation metrics (e.g., F-score) for each label individually (i.e., for each game feature) and average the per-label metrics. For example, for the "battle" feature, we use the binary elements that correspond to the position of "battle" in the ground truth binary vectors (i.e., the first elements, which are highlighted in green), which gives [0, 1, 1]. We do the same for the predicted binary vectors (elements highlighted in orange) and obtain [0,0,1]. We then compute the F-score between those two vectors, which gives an F-score of 0.67. We do the same procedure for all game features and compute their average weighted by the number of times each game features appears in the ground truth (e.g., "battle" appears two times, in the second and third test cases, while "login" appears once, in the first test case).

### 4.5.3 Results

Table 4.3 presents the results of our experiments with the zero-shot models. All the ensemble approaches perform better than the individual models and the baseline. The *EnsBackOffComplete* approach has the best performance, with an F-score of 76.1, followed closely by the *EnsBackOffTwo* approach, with an F-score of 76.0. The best individual model is the *LatentEmb*, with an F-score of 72.3, while the *BartLargeMNLI* and the *CrossEncoderNLI* achieved F-scores of 70.5 and 69.9, respectively. Based on these results, we used the *EnsBackOffComplete* approach to classify all the test cases in our dataset.

## 4.6 Multi-objective prioritization of natural language test cases

To optimize the execution of manual test cases, our approach performs a multi-objective optimization using a genetic algorithm. Below, we explain how we applied the non-dominated sorted genetic algorithm (NSGA-II) genetic algorithm [44] to test case prioritization, the performed experiments and the obtained results.

Table 4.3: Results of experiments with the zero-shot models.

| Zero-shot approach | F-score | Precision | Recall |
|---|---|---|---|
| Baseline (keyword search) | 59.8 | 65.5 | 60.0 |
| BartLargeMNLI | 70.5 | 69.9 | 79.9 |
| CrossEncoderNLI | 69.9 | 73.5 | 75.3 |
| LatentEmb | 72.3 | 71.5 | 80.9 |
| EnsMajorVoting | 74.1 | 71.5 | 84.4 |
| EnsFullInters | 74.7 | 74.2 | 83.1 |
| EnsBackOffTwo | 76.0 | 78.3 | 78.9 |
| **EnsBackOffComplete** | **76.1** | 78.0 | 79.2 |

### 4.6.1 Multi-Objective Genetic Algorithms

A genetic algorithm is a search-based heuristic that uses the concept of natural evolution to find the best solutions from a large number of possible solutions [43]. In our case, a possible solution is a specific test case ordering that is searched among all possible test case orderings (i.e., all the permutations of orderings). We apply the NSGA-II algorithm [44] because it has been widely used for multi-objective optimization for different purposes in the software engineering field [102, 164, 167, 191]. At each iteration, the algorithm uses an objective function (i.e., fitness function) to evaluate the candidate solutions that we generated. Differently from a single-objective optimization, in which the candidates are evaluated using a single objective, in a multi-objective scenario, there is a trade-off between the multiple objectives. NSGA-II uses the concept of dominance [43, 91] to determine the best solutions. A solution s1 dominates solution s2 (s1 $\preceq$ s2) if s1 is no worse than s2 for all the objectives and s1 is strictly better than s2 for at least one objective. In the end, the algorithm outputs a set of non-dominated solutions, which are called the *Pareto front* [91].

### 4.6.2 Test Case Prioritization Using NSGA-II

For our work, a *Pareto front* consists of a set of test case orderings with the optimal trade-off between the objectives. To use NSGA-II, we need to define the *solution encoding* (i.e., how a solution is represented). In our case, as a solution corresponds to a specific test case ordering, we assign a unique integer to identify each test case. Therefore, a solution is represented by an ordered sequence of integers $[1, 2, \cdots, n]$, where n is the total number of test cases in our test suite. We initialize NSGA-II by randomly sampling a subset of all the possible test case orderings. For the required genetic operators, we use the default operators provided by the Python package that we used (`pymoo` [20]) for permutation problems: binary tournament for the *selection* operator, order-based crossover for the *crossover* operator (order-based is a crossover operator proper for permutation encoded chromosomes, such as our case), and inversion mutation for the *mutation* operator.

### 4.6.3 Objective functions for NSGA-II

We defined three objective functions that are used during the optimization process in our experiments. Similarly to prior work [166, 167], we use normalized metrics as the objective functions to avoid bias of the model towards functions with larger values. In addition, normalized objective functions are inherently more interpretable.

As we mentioned in Section 4.4, the goal of our prioritization approach is to search for test case orderings such that (1) highly-used features are covered early in the test execution and (2) test cases with shorter durations are executed early in the testing. During our experiments, for comparison purposes, as we explain in Section 4.6.5, we also performed an optimization such that (3) a large number of features (not necessarily the highly-used features) are covered early in the test execution and (2) test cases with shorter durations are executed early in the testing. To capture these three criteria during the test case prioritization, we defined the following objective functions.

#### Feature ranking similarity (featRankSim)

This metric measures the similarity between two rankings: the feature usage ranking (in which the features are sorted by their total number of uses) and the feature testing ranking (in which the features are sorted according to the order in which they are covered when executing the ordered test cases). Ideally, the feature testing ranking is the same as the feature usage ranking. To measure the ranking similarity, we used the normalized discounted cumulative gain (NDCG) metric [85, 192]. NDCG is commonly used to compute ranking quality in Information Retrieval-based systems [122] and uses a graded-scale relevance for documents, where the usefulness of a document is measured based on its position in the ranking (highly-relevant documents should be at the top of the ranking). The cumulative gain score is computed as we move from top to bottom in the ranking. The lower the position of a document in the ranking, the lower the gain that it provides to the final score. Because of this, NDCG gives greater importance to documents in the top of the ranking. For example, differences in the top of the ranking have a larger impact on the score than differences in the bottom of the ranking. We used the `scikit-learn` implementation of NDCG, which lies in the range *[0,1]*, with 1 indicating a perfect match between the obtained ranking and the ideal ranking. In our case, a document is a game feature and we use the total number of uses of the feature as its relevance score. The ideal ranking is obtained by sorting the features by their usage (feature usage ranking). Finally, we want to maximize the *featRankSim* objective to have the feature testing ranking as similar to the feature usage ranking as possible (which means that highly-used features are tested early in the test case ordering).

#### Cumulative execution time (cumExecTime)

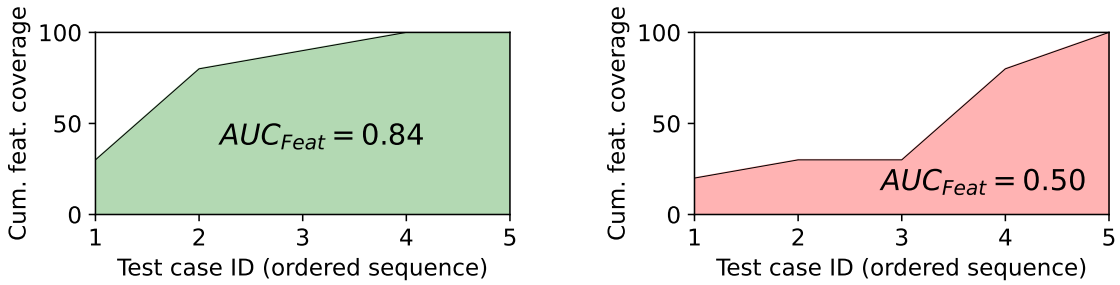
This metric captures how the cumulative execution time of test cases changes as test cases are executed in a specific order. For each executed test case, its execution time is added to the partial cumulative execution time. Since we want *cumExecTime* to

increase as slow as possible as we execute the ordered test cases, we use the AUC obtained as we move along the sequence of ordered test cases as the objective function ($AUC_{Time}$). We normalize $AUC_{Time}$ with regard to the maximum area (which is the test case ordering in which the first test case has an execution time that corresponds to the total execution time of the test suite).

**Cumulative feature coverage (cumFeatCov)**

This metric captures how the number of covered features increases as test cases are executed in a specific order. Since one feature might be tested in multiple test cases, we need to define a minimum number of test cases necessary to consider that a feature has been covered. We defined a threshold for the percentage of test cases that is sufficient to consider that a feature was indeed tested and can be counted as covered (which we call *per-feature coverage threshold*). For example, if the *per-feature coverage threshold* is 0.8, we only consider feature "A" covered after executing 4 out of 5 test cases that cover that feature. To obtain the *cumFeatCov* metric, we get the set of features associated with the test cases as they are executed one at a time and compute how many features are covered. A feature is considered covered if its *per-feature coverage threshold* is met. Since we want *cumFeatCov* to increase as quick as possible as we execute the ordered test cases, we use the area under the curve (AUC) obtained as we move along the sequence of ordered test cases as the objective function ($AUC_{Feat}$). We normalize $AUC_{Feat}$ similarly as we do for $AUC_{Time}$

Figure 4.3 presents examples of different test case orderings that achieve different $AUC_{Feat}$ and helps to clarify our goal of maximizing $AUC_{Feat}$. Figure 4.3a shows a test case ordering in which a large number of features is covered early in the sequence (which yields a large $AUC_{Feat}$ of 0.84), while Figure 4.3b shows that the number of covered features increases slower than in Figure 4.3a (which yields a smaller $AUC_{Feat}$ of 0.50). Since we want the number of covered features to increase as quick as possible, the ordering with the larger $AUC_{Feat}$ is preferable.




(a) Ordering w/ high $AUC_{Feat}$

(b) Ordering w/ low $AUC_{Feat}$

Figure 4.3: Examples to demonstrate our objective function.

### 4.6.4 Stopping Criteria for NSGA-II

Lastly, we need to define the stopping criteria for NSGA-II so that the algorithm can be stopped when no progress is made in the search for the optimal solutions. Similarly to prior work [167], and to have a systematic way of deciding when to stop the algorithm execution, we defined two stopping criteria that we used in our experiments.

**T-test**: for the non-dominated solutions $s_i$ of each new generation $g_i$ during NSGA-II execution, we run a t-test [185] for each objective to compare the non-dominated solutions of the new generation with the solutions $s_{i-1}$ of the previous generation $g_{i-1}$. For example, if generation $g_i$ has 10 solutions, there are 10 non-dominated test case orderings, i.e., 10 values for each objective: $AUC_{Feat}$, $AUC_{Time}$, and *featRankSim*. When the t-tests for all the objectives show that the difference between the two generations is insignificant (p-value $> 0.05$) for five consecutive generations, the algorithm execution is stopped.

**Mutual Dominance Rate (MDR)**: we also use the set of non-dominated solutions for two consecutive generations $g_{i-1}$ and $g_i$ to compute the mutual dominance rate (MDR) indicator [64, 118]. Consider a function $\Delta(g_{i-1}, g_i)$ that returns the set of solutions in $g_{i-1}$ that are dominated by at least one solution in $g_i$. We can then

formulate the MDR as:

$$MDR = \frac{|\Delta(g_{i-1}, g_i)|}{|g_{i-1}|} - \frac{|\Delta(g_i, g_{i-1})|}{|g_i|}$$

where $|g|$ is the number of elements in $g$.

The MDR indicator ranges from -1 to +1, in which an MDR of -1 indicates that the solutions of the current generation are not better than the solutions of the previous generation, while an MDR of +1 indicates that the current solutions are completely better than the previous solutions. An MDR of zero means that no significant progress has been made [118]. Since the MDR can have alternated signs due to the randomness of genetic algorithms, we consider that the algorithm can be stopped when MDR lies within a pre-defined range *[-a, a]* (as done in prior work [167]) for five consecutive generations (which is stricter than prior work [167]). We experimented with different MDR ranges, as we explain below.

### 4.6.5 Experiment setup

In this section, we describe the experiments that we performed to assess how our approach works in different scenarios and with different parameters. For all experiments, we used our dataset of 1,146 test cases, with a total execution time of 133 hours and 110 game features covered by test cases. Similarly to prior work [8, 72, 166, 167], we used random-based search approaches as the baselines with which we compare our approaches. We randomly selected 50 test case orderings, named $Random_{50}$, and 100 test case orderings, named $Random_{100}$. The random orderings were selected without replacement from the entire population of test case orderings. Also, following the literature guidelines [9, 166], we used a population of 100 in all our experiments. We executed NSGA-II 50 times during the experiments to mitigate the randomness involved in genetic algorithms and we report the results from all 50 runs.

**Experiment 1: *number of covered game features* versus *test execution time* (without feature usage).** In this experiment, we performed a bi-objective

optimization for different combinations of *per-feature coverage threshold* and stopping criteria. We performed the test case prioritization only with the $AUC_{Feat}$ and $AUC_{Time}$ objective functions. Our goal is to understand the trade-off between game feature coverage and execution time when no feature usage information is included. We evaluated four *per-feature coverage thresholds*: 50%, 75%, 90%, and 100%. We consider that 50% is the minimum acceptable threshold to consider that a feature is covered. For each *per-feature coverage threshold*, we evaluated three approaches with different intervals for MDR in the stopping criteria: $Stop_{0.25}$, $Stop_{0.10}$, and $Stop_{0.05}$, with the following ranges: *[-0.25, 0.25]*, *[-0.10, 0.10]*, and *[-0.05, 0.05]*. For the stopping criteria, both the t-test (p-value $> 0.05$) and the MDR criteria must be satisfied for five consecutive generations.

**Experiment 2: *number of covered highly-used game features* versus *test execution time* (with feature usage).** In this experiment, we used the game feature usage in the optimization through the *featRankSim* objective function instead of only the number of covered game features. Our goal is to find test case orderings that test highly-used features early in the test execution in the shortest amount of time. We evaluated the same stopping criteria as in experiment 1, i.e., the *[-0.25, 0.25]*, *[-0.10, 0.10]*, and *[-0.05, 0.05]* MDR ranges together with the t-test. For experiment 2, as we included feature usage, we named the approaches as follows: $Stop_{0.25\_usage}$, $Stop_{0.10\_usage}$, and $Stop_{0.05\_usage}$.

### 4.6.6 Evaluation of test case prioritization approaches

For each approach, we report the number of non-dominated solutions obtained, the number of fitness evaluations of NSGA-II, and the execution time until the algorithm was stopped. In all cases, we report the median obtained from the 50 runs. The number of fitness evaluations corresponds to the number of test case orderings that were inspected during the optimization and represents the speed with which our approaches converge and their practical applicability. We also report the median of the

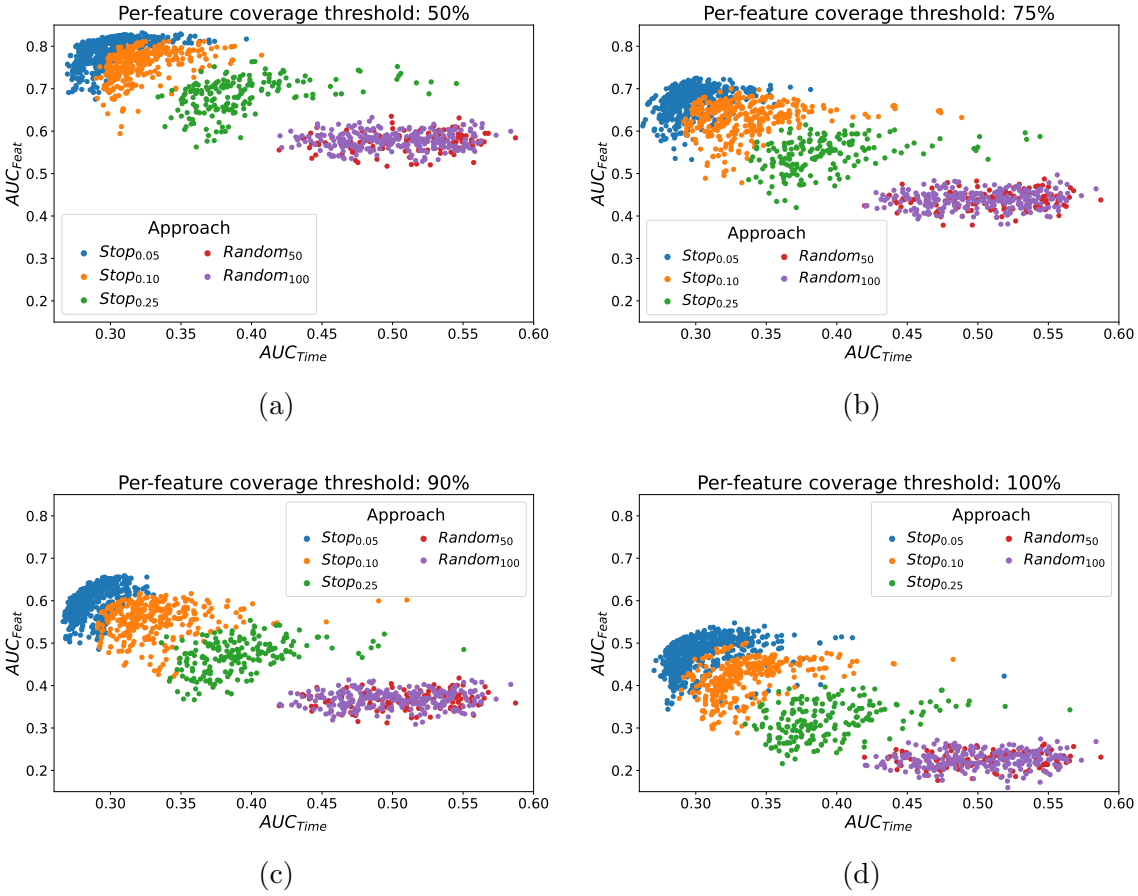



Figure 4.4: Experiment 1: Trade-off between $AUC_{Time}$ and $AUC_{Feat}$ for different *per-feature coverage thresholds* across our different approaches (**without feature usage**).

$AUC_{Feat}$ and $AUC_{Time}$ objective functions for experiment 1, and of the *featRankSim* and $AUC_{Time}$ objective functions for experiment 2. Following the literature recommendations [8], we used the Mann-Whitney U-test [114] and Cliff's delta $d$ effect size [109, 156] to statistically compare our approaches. We adopt the thresholds for $d$ as provided by Hess and Kromrey [76]:

$$\text{Effect size} = \begin{cases} negligible(N), & \text{if } |d| \leq 0.147 \\ small(S), & \text{if } 0.147 < |d| \leq 0.33 \\ medium(M), & \text{if } 0.33 < |d| \leq 0.474 \\ large(L), & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

### 4.6.7 Results

In this section we present the results of the two experiments that we performed. We report the results from all the 50 executions of the NSGA-II algorithm.

**Experiment 1: *number of covered game features* versus *test execution time* (without feature usage).** Figure 4.4 shows the non-dominated solutions found by 50 runs of NSGA-II for our approaches across different *per-feature coverage thresholds*. The $Stop_{0.05}$ approach found a median of 56.5 non-dominated solutions across all *per-feature coverage thresholds*, while the $Stop_{0.10}$ and $Stop_{0.25}$ approaches found a median of 34.0 and 18.0 non-dominated solutions. In terms of fitness evaluations (i.e., the number of test case orderings that were inspected until the algorithm was stopped), the $Stop_{0.05}$ approach had a median of 37,950 fitness evaluations across all *per-feature coverage thresholds*, while $Stop_{0.10}$ and $Stop_{0.25}$ had a median of 15,300 and 3,600 evaluations, respectively. As expected, the number of fitness evaluations for the $Stop_{0.05}$ approach was higher since the stopping criteria is stricter. The $Stop_{0.05}$ approach took a median of 85.98 seconds to execute, while $Stop_{0.10}$ and $Stop_{0.25}$ took a median of 32.11 seconds and 8.06 seconds, respectively.

Figure 4.4 shows that all our proposed approaches achieve better solutions than random search for all *per-feature coverage threshold* values. The $Stop_{0.25}$, $Stop_{0.10}$, and $Stop_{0.05}$ approaches found solutions with a better trade-off between the objective functions, i.e., with lower $AUC_{Time}$ and larger $AUC_{Feat}$. For a *per-feature coverage threshold* of 50%, presented in Figure 4.4a, the $Stop_{0.05}$ approach has a median $AUC_{Time}$ of 0.29, while the $Stop_{0.10}$ and $Stop_{0.25}$ approaches have larger median $AUC_{Time}$: 0.32 and 0.38, respectively. The $Random_{50}$ and $Random_{100}$ approaches have a median $AUC_{Time}$ of 0.49 and 0.50, respectively. Regarding the $AUC_{Feat}$, $Stop_{0.05}$ has a median of 0.80, while $Stop_{0.10}$ and $Stop_{0.25}$ have medians of 0.77 and 0.69. The $Random_{50}$ and $Random_{100}$ approaches have a median $AUC_{Feat}$ of 0.58 and 0.57, respectively. Among our proposed approaches, $Stop_{0.05}$ found the best solutions

since it has the best trade-off between the $AUC_{Time}$ and $AUC_{Feat}$. All the proposed approaches are significantly better than both random search approaches (p-value less than 0.05) and the Cliff's delta shows large effect sizes for both objective functions. Pairwise comparisons between the three approaches also show statistically significant differences with large effect sizes.

A similar behavior is observed for the *per-feature coverage thresholds* of 75%, 90%, and 100%, in Figures 4.4b, 4.4c, and 4.4d. In all these cases, the $Stop_{0.05}$ found the best solutions. However, the range in which the $AUC_{Feat}$ lies gets smaller as we increase the *per-feature coverage threshold*. This happens because a higher threshold means that we need to execute more test cases to consider a feature as covered, so the number of covered features increases more slowly as we execute the ordered test cases, which yields a smaller $AUC_{Feat}$ (as we explained in Section 4.6.3, in Figure 4.3).

**Experiment 2: *number of covered highly-used game features* versus *test execution time* (with feature usage).** Figure 4.5 shows the non-dominated solutions found by 50 runs of NSGA-II for our approaches. For this experiment, we did not use the *per-feature coverage threshold* since we did not use the $AUC_{Feat}$ objective function. The $Stop_{0.05\_usage}$ approach found a median of 42.5 non-dominated solutions for all 50 runs, while the $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ approaches found a median of 29.0 and 18.0 non-dominated solutions. In terms of fitness evaluations, the $Stop_{0.05\_usage}$ approach had a median of 47,850 fitness evaluations, while $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ had a median of 12,800 and 3,650 evaluations, respectively. As expected, the number of fitness evaluations for the $Stop_{0.05\_usage}$ approach was higher since the stopping criteria is stricter. Also as expected, the $Stop_{0.05\_usage}$ approach took longer to execute until the stopping criteria were satisfied, with a median of 100.16 seconds. The $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ approaches took a median of 26.36 seconds and 7.47 seconds, respectively. We can see that our best approach ($Stop_{0.05\_usage}$) is feasible to be used in practice as it can find the best solutions in less than 2 minutes.

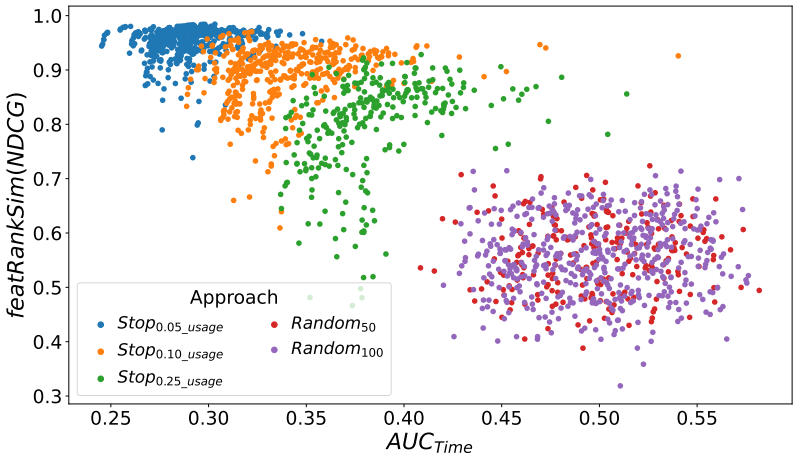



Figure 4.5: Experiment 2: Trade-off between $AUC_{Time}$ and *featRankSim* across our approaches (**with feature usage**).

Figure 4.5 shows that all our proposed approaches achieve better solutions than random search since our approaches present better trade-offs between the objective functions (i.e., lower $AUC_{Time}$ and larger *featRankSim*). The $Stop_{0.05\_usage}$ approach has a median $AUC_{Time}$ of 0.29, while the $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ approaches have larger median $AUC_{Time}$: 0.33 and 0.38, respectively. The $Random_{50}$ and $Random_{100}$ approaches have a median $AUC_{Time}$ of 0.49 and 0.50, respectively. Regarding *featRankSim*, $Stop_{0.05\_usage}$ has the largest median, with a value of 0.96, while $Stop_{0.10\_usage}$ and $Stop_{0.25\_usage}$ have medians of 0.90 and 0.82. The $Random_{50}$ and $Random_{100}$ approaches have a median *featRankSim* of 0.56 and 0.55, respectively. Among our proposed approaches, $Stop_{0.05\_usage}$ found the best solutions since it has the best trade-off between the $AUC_{Time}$ and *featRankSim*. This demonstrates that our $Stop_{0.05\_usage}$ approach can find test case orderings that cover highly-used game features early in the test execution (with a high *featRankSim* of 0.96) while keeping the cumulative test execution time small. All the proposed approaches are significantly better than the random search approaches (p-value less than 0.05) and the Cliff's delta shows large effect sizes for both objective functions. Pairwise comparisons between the three approaches also show statistically significant differences with large effect sizes.

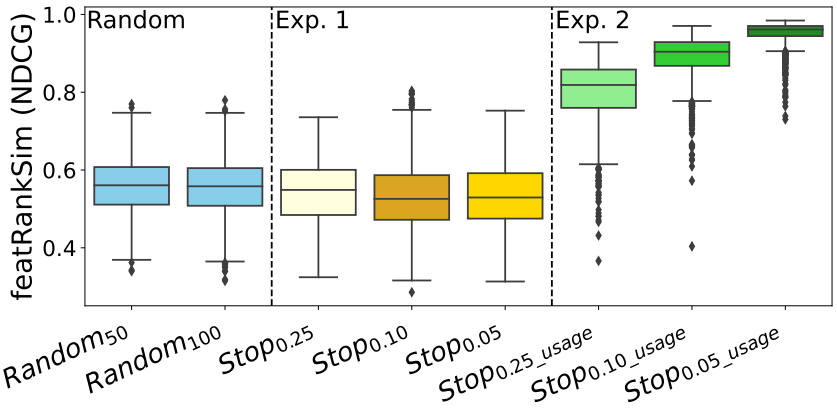



Figure 4.6: Distributions of *featRankSim* (NDCG) for our different approaches.

## 4.7 Discussion

In this section, we compare the results obtained with our experiments for prioritization with and without feature usage. Figure 4.6 shows the distribution of the *featRankSim* objective for our random approaches and for the non-dominated solutions obtained with our proposed approaches. In experiment 1, without feature usage, all the approaches (using a *per-feature coverage threshold* of 50%) achieve a similar median *featRankSim*: 0.55, 0.52, and 0.53 for $Stop_{0.25}$, $Stop_{0.10}$, and $Stop_{0.05}$, respectively. The random approaches achieve similar median values of *featRankSim*: 0.56 and 0.55 for $Random_{50}$ and $Random_{100}$, respectively. In contrast, the approaches in experiment 2 achieve larger median *featRankSim* values: 0.82, 0.90, and 0.96 for $Stop_{0.25\_usage}$, $Stop_{0.10\_usage}$, and $Stop_{0.05\_usage}$, respectively. The larger *featRankSim* obtained by our approaches in experiment 2 shows that those approaches, in particular $Stop_{0.05\_usage}$, can successfully obtain test case orderings that cover highly-used game features early in the test execution.

Often during regression testing, the main constraint is the time available to execute test cases. Therefore, we discuss below how (1) the percentage of covered game features and (2) the percentage of coverage of the top-k most used game features change for different test execution times. For this analysis, we used the solutions

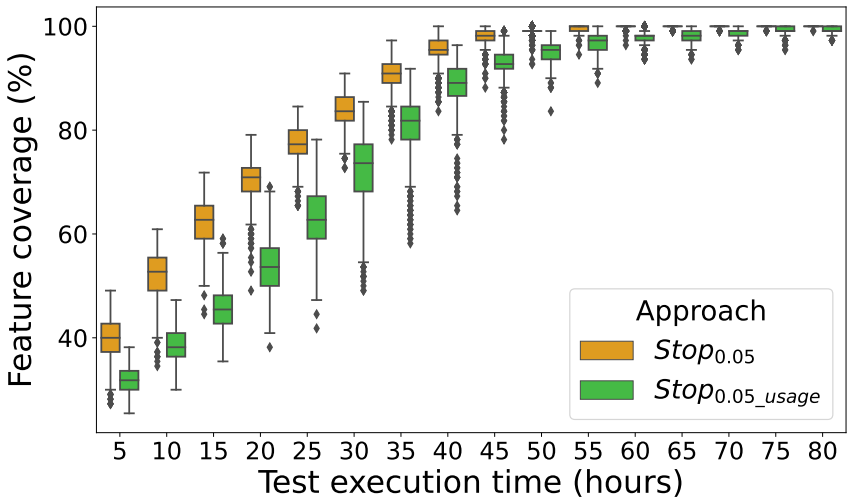



Figure 4.7: Comparison of game feature coverage for our best approaches in experiments 1 and 2.

found by our best approaches in experiments 1 and 2 ($Stop_{0.05}$, with a *per-feature coverage threshold* of 50%, and $Stop_{0.05\_usage}$).

Figure 4.7 shows how the percentage of covered game features changes for different execution times. A large amount of testing time is necessary to achieve large game feature coverage. For example, to achieve 100% coverage, a median of 55 hours for the $Stop_{0.05}$ approach and 70 hours for the $Stop_{0.05\_usage}$ approach are necessary. Even for lower coverage, a large amount of time is necessary. For example, to achieve 90% of coverage, $Stop_{0.05}$ requires 35 hours and $Stop_{0.05\_usage}$ requires 40 hours. We also observe that the last 10% of feat coverage requires an extremely large amount of time (approximately 30 hours additional testing time).

Achieving a high percentage of game feature coverage is not feasible in practice due to the large amount of time necessary, even for the $Stop_{0.05}$ approach that was optimized to achieve a large game feature coverage in the shortest time possible. If we analyze a more feasible scenario, with an available testing time of 5 hours, for example, $Stop_{0.05}$ covers a median of 40% of game features, while $Stop_{0.05\_usage}$ covers a median of 32% of game features. However, despite achieving a slightly smaller coverage for the same amount of time available, the test case orderings obtained

with $Stop_{0.05\_usage}$ cover highly-used features earlier in the test execution compared to the solutions obtained with $Stop_{0.05}$. For example, if we analyze the coverage of the top-20% of the most used features (which gives the top-16 most used features) in 5 hours, $Stop_{0.05}$ covers only 68% of those features, while $Stop_{0.05\_usage}$ covers 93%. With one additional hour, $Stop_{0.05\_usage}$ covers all the top-20% most used features, while $Stop_{0.05}$ covers 75%. Therefore, with a feature usage-based test prioritization, we can find test case orderings that cover most of the highly-used features early in the test execution, which helps to avoid bugs that would affect a large number of users. Finally, QA engineers can achieve a higher coverage of game features and highly-used game features by parallelizing the test execution. For example, two QA engineers can execute independent test cases in parallel to achieve higher coverages within 5 hours.

## 4.8 Using our prioritization approach in practice

We implemented an internal web application prototype of our approach to collect initial feedback on the outcome of our approach. We are now integrating the application into our industry partner's cloud infrastructure. Our approach can be used by QA engineers in an environment where resources (e.g., time) are restricted to obtain a set of test case orderings with the best trade-off between multiple objectives. Because of how we performed the optimization, in a situation of an early-stop of test execution, our approach ensures that the highest number of highly-used features are covered with the shortest time possible. The tester may also choose one particular test case ordering among the optimal orderings that maximizes one specific objective of interest to the detriment of the other. For example, an ordering that maximizes only the highly-used features may not have the shortest cumulative test execution time. Another practical aspect concerns a small subset of features that are critical to the *Prodigy Math game* of Prodigy Education (such as the "game membership purchase" feature). These features must be frequently tested during regression regardless of their usage. Therefore, we allow the users of our application to identify

the critical features and retrieve the associated test cases before executing the optimization. Those test cases are then removed from the set of test cases that we use in the optimization (as they will always be executed before the optimized ordering).

## 4.9 Related work

Optimizing the execution of test cases in a manual testing scenario is of extreme importance [69, 73, 75]. However, only a few works investigated prioritization techniques for manual test cases which are described only in natural language (i.e., no source code is associated with them) [73, 94]. Hemmati *et al.* [73] investigated approaches to prioritize manual test cases using test execution history and Latent Dirichlet Allocation (LDA) [21] to find the topics related to test cases. Our approach uses a pre-defined list of the application features to find the features being tested and leverages zero-shot models for that purpose, which does not require any manual analysis to further understand which features are covered by test cases (as LDA requires). Furthermore, our approach does not require the test execution history, which could be difficult to be accessed or not meaningful for manual test cases [206]. Lachmann *et al.* [94] investigated a supervised approach for prioritization of manual test cases using textual descriptions, test execution history, and the link between test cases and requirements. Their approach requires an expert to manually label test cases as (un)important to build the training set. In contrast, our approach does not require any manual data labeling nor the test execution history. Furthermore, none of the above mentioned works take into consideration the impact that bugs might have on users. We include the coverage of highly-used features in our approach, which helps to test those features more often and avoid impacting a large number of users.

Several works proposed prioritization techniques for test cases with associated source code [11, 27, 81, 112, 116, 125, 134, 136, 159, 171, 191, 207]. For instance, Marchetto *et al.* [116] performed test case prioritization with NSGA-II. However, differently from our work, they used code coverage and the link between re-

quirements and source code in their approach. Instead, we do not have source code test cases and we used the link between natural language test cases and the covered features. We also include the coverage of highly-used features in our approach. Wang *et al.* [191] proposed to use multi-objective search algorithms for a resource-aware test case prioritization using four objectives. Their goal was to achieve a test case ordering for a limited time budget while maximizing the usage of the available test resources. In contrast, we focus on prioritizing manual test cases to maximize the coverage of the game features and the coverage of highly-used features.

## 4.10 Threats to validity

A threat to the **external validity** concerns the generalizability of our zero-shot methods and prioritization techniques. Using applications from other domains might yield different results. Another threat regards the used techniques. Using different classification models and optimization algorithms might achieve different results. Future studies should investigate if our approaches can be improved with other techniques.

A threat to the **internal validity** concerns the percentage of test cases that we consider sufficient to count a feature as covered. To mitigate this threat, we experimented with different percentages (from 50% up to 100%), but using other values will achieve different results. Also, companies that already have the link between test cases and covered features might use a different percentage. Another threat is related to the feature usage metric that we use (*total number of uses*). Other metrics can also capture feature usage, such as using the *number of unique users* who used a feature, which might achieve different orderings of features based on usage. Finally, using different conditions to stop the optimization algorithm (e.g., other p-value thresholds or other MDR ranges) might result in different non-dominated test case orderings.

## 4.11 Conclusion

In this chapter, we propose a novel approach to prioritize natural language test cases. Our approach leverages zero-shot classification techniques to identify the features covered by the test cases of a game and uses this information to optimize the execution of test cases. In particular, we prioritize test cases that cover highly-used game features, in which bugs would affect a large group of players. Our findings show that we can successfully identify the game features covered by test cases with an ensemble of zero-shot models (an F-score of 76.1%). Also, our prioritization approaches can find test case orderings that cover highly-used game features early in the test execution while keeping the time required to execute test cases short. In practice, QA engineers and developers can use our approach to focus the test execution on test cases that cover game features that are relevant to players.

# Chapter 5

# What Causes Wrong Sentiment Classifications of Game Reviews?

## 5.1 Abstract

Sentiment analysis is a popular technique to identify the sentiment of a piece of text. Several different domains have been targeted by sentiment analysis research, such as Twitter, movie reviews, and mobile app reviews. Although several techniques have been proposed, the performance of current sentiment analysis techniques is still far from acceptable, mainly when applied in domains on which they were not trained. In addition, the causes of wrong classifications are not clear. In this chapter, we study how sentiment analysis performs on game reviews. We first report the results of a large scale empirical study on the performance of widely-used sentiment classifiers on game reviews. Then, we investigate the root causes for the wrong classifications and quantify the impact of each cause on the overall performance. We study three existing classifiers: `Stanford CoreNLP`, `NLTK`, and `SentiStrength`. Our results show that most classifiers do not perform well on game reviews, with the best one being `NLTK` (with an AUC of 0.70). We also identified four main causes for wrong classifications, such as reviews that point out advantages and disadvantages of the game, which might confuse the classifier. The identified causes are not trivial to be resolved and we call upon sentiment analysis and game researchers and developers to prioritize a research agenda that investigates how the performance of sentiment analysis of game

reviews can be improved, for instance by developing techniques that can automatically deal with specific game-related issues of reviews (e.g., reviews with advantages and disadvantages). Finally, we show that training sentiment classifiers on reviews that are stratified by the game genre is effective.

## 5.2 Introduction

Sentiment analysis is a widely adopted Natural Language Processing (NLP) technique to obtain the sentiment (expression of positive or negative feeling) from text data [103, 138]. This technique consists of identifying the sentiment that is present in a piece of text (words, sentences, or entire documents), which corresponds, in its most basic form, to finding whether the text has a positive, neutral, or negative sentiment [103]. Sentiment analysis is a research topic that has gained attention and has presented improvements [49, 170, 204], being developed and applied in several different domains, such as Twitter tweets [12, 16, 22], movie reviews [170], customer reviews of mobile applications [62, 139], video game reviews [172, 177], and various aspects of software development [83, 88, 89, 137, 154]. Sentiment analysis is valuable for game developers because it allows them to capture how players feel about the game and learn about previous games' success or failure factors [172]. This knowledge can help game developers improve their game development processes and guide them in future releases of their game (e.g., by focusing on features that users are more positive about).

Several studies have been published on sentiment analysis with the purpose of developing new techniques, improving current techniques, or applying current techniques and classifiers to existing datasets [30, 67, 83, 88, 89, 103, 172, 177]. However, the performance of such techniques is still far from acceptable, mainly when off-the-shelf sentiment analysis classifiers are applied out of domain, i.e., a classifier is trained in one domain and applied in a different domain without any configuration or adjustment. Normally, sentiment analysis techniques must be adapted to the target domain.

For instance, Thompson *et al.* [177] adapted a sentiment analysis technique that was initially designed for movie reviews to be used in video game chat messages. Despite the low performance of sentiment analysis, no study has investigated the reason(s) for the low performance.

In this study, we investigate how different sentiment classifiers perform on game review data. Game reviews from Steam differ from other types of data to which sentiment analysis is normally applied. Game reviews contain a more complex text structure and generally discuss several aspects of the game, such as the game's storyline, graphics, audio and controls [211]. Texts from micro-blogging and social media (e.g., Twitter) are usually very short [54, 132]. In addition, such texts are broader in scope since they are not necessarily reviewing a game. Prior work [106] also showed that game reviews are different from mobile app reviews in several aspects. For instance, game reviews contain game-specific terminology, which is a challenge for language processing tools.

Although the diversity of game reviews makes them a rich source of data, it also poses challenges to NLP techniques, such as sentiment analysis. For instance, players may mention the graphical aspects and the storyline of the game in the same review [172]. The two pieces of text corresponding to such aspects may have different sentiments, which could confuse the sentiment classifier when making a classification of the overall sentiment of the review.

By applying sentiment classifiers on game reviews, we are able to report the sentiment classification performance and identify cases where sentiment analysis fails. For instance, the following review is an example of a difficult classification task for current sentiment classifiers: "*Very nice programmed bugs*". The reviewer makes references to a positive word ("nice"), with a stronger intensity due to the use of an adverb ("very"), which might lead the classifiers to classify this instance as positive. However, the overall sentiment of this review should be negative as the reviewer is being sarcastic (the reviewer is pointing out that the game contains bugs). A deeper

investigation of wrong classifications (failing cases) allows us to find problematic text patterns for sentiment classifiers and provide insights for game developers about how to improve the performance of sentiment analysis.

In this chapter, we first report the results of a large-scale empirical study on the performance of sentiment analysis on 12 million game reviews. Our goals are (1) to investigate how existing sentiment classifiers perform on game reviews, (2) identify which factors impact the performance and (3) quantify the impact of such factors. Note that we do not aim to propose a new sentiment classification technique. Instead, we investigate reasons for wrong classifications of existing classifiers. We studied three widely-used and computationally accessible sentiment classifiers [83, 103]: `Stanford CoreNLP` [170], `NLTK` [19], and `SentiStrength` [176]. The selected classifiers adopt different approaches to classify the text, such as rule and machine learning-based approaches, which gives more confidence to our study and makes the results more generalizable.

We evaluated these classifiers on all the game reviews collected from the Steam platform up to 2016. We then selected the reviews of which all classifiers misclassified the sentiment. We manually analyzed a representative and statistically significant sample of 382 of these reviews to understand which factors might be causing wrong classifications. Finally, we performed a series of experiments to quantify the impact of each identified factor on the performance of sentiment analysis on game reviews. We address the following three research questions:

**RQ1: How do sentiment analysis classifiers perform on game reviews?**
Investigating the performance of sentiment analysis on game reviews is the first step to understand how current sentiment analysis classifiers work on game reviews and whether they are suitable for this task on such data. We found that sentiment classifiers do not perform well on game reviews, with AUC values ranging from 0.53 (`Stanford CoreNLP`), which is slightly better than random guess, up to 0.70 (`NLTK`).

**RQ2: What are the root causes for wrong classifications?**

Identifying the causes for wrong classifications contributes to obtain important insights about how to improve existing sentiment analysis for game reviews. We found several causes which mislead the classifiers, such as reviews that make comparisons to games other than the game under review, reviews with negative terminology (e.g., reviews that use the word "kill") which does not necessarily mean the content has a negative sentiment, and reviews with sarcasm.

**RQ3: To what extent do the identified root causes impact the performance of sentiment analysis?**

Quantifying the impact of each identified root cause to the performance of sentiment analysis is important to support game developers with the prioritization of causes to be resolved and a research agenda to address such issues. We found that reviews which point out advantages and disadvantages of the game have the highest negative impact on the performance of sentiment analysis, followed by reviews with game comparisons. In addition, we deepened our investigation and showed that training sentiment classifiers on reviews stratified by the game genre is effective.

Our study makes three major contributions:

- We evaluate the performance of widely-adopted sentiment analysis classifiers on game reviews from the Steam platform.
- We identify a set of root causes that can explain the wrong classifications of sentiment analysis classifiers on game reviews.
- We quantify the impact of each identified cause for wrong classifications on game reviews and provide a research agenda for addressing these causes.
- We provide access to the data[1] (URLs of game reviews from Steam with the sentiment classification provided by all three classifiers).

[1] https://github.com/asgaardlab/sentiment-analysis-Steam_reviews

The remainder of this chapter is organized as follows. Section 5.3 provides a background on sentiment analysis classification techniques. Section 5.4 discusses related work and Section 5.5 presents the proposed research methodology. Section 5.6 discusses the pre-study. In Sections 5.7, 5.8, and 5.9, we discuss the results, while in Section 5.10 we present our recommendations on how to perform sentiment analysis on game reviews. Finally, Section 5.11 concludes the chapter.

## 5.3 Sentiment Analysis

In this section, we present an overview of the main sentiment analysis techniques along with the most used classifiers that adopt these techniques. In this work, we use 'technique' to refer to the method adopted for the sentiment classification and 'classifier' (which can also be understood as 'tool' or 'framework') to refer to an implementation of a technique (i.e., an actual instance of the technique). Next, we discuss each technique and the representative classifier(s) we chose for our work. For this study, we focus on popular, open source and free-to-use sentiment analysis classifiers.

Sentiment analysis techniques are responsible for identifying the sentiment present in a piece of text, which can be either positive, neutral, or negative [103, 138]. Table 5.1 presents an overview of the main sentiment analysis techniques and classifiers which have been proposed in prior studies. This is not an exhaustive list of sentiment classifiers and it comprehends the most reported classifiers in prior studies. The grouping of classifiers under a specific technique category was done based on the method the classifier uses. Classifier names in bold refer to the ones studied in this work. The last column shows the type of data on which the classifier was originally trained. Next, we detail each technique and the corresponding classifier(s) we chose to use in our study.

Table 5.1: Sentiment analysis techniques, corresponding classifiers and default training dataset.

| Technique | Classifier | Default training set | Used by |
|---|---|---|---|
| Machine Learning | `NLTK`*[19] | Micro-blog texts | [88], [89], [103], [144], [126] |
| | `Stanford CoreNLP` [170] | Movie reviews | [88], [89], [150], [115], [199] |
| | `IBM Alchemy`** | — | [88], [89], [168], [18] |
| | `Senti4SD` [28] | Stack Overflow posts | [28], [82] |
| Rule-based | `SentiStrength` [176] | MySpace | [68] [67], [66], [88], [89] |
| | `SentiStrength-SE` [83] | JIRA | [83], [82] |
| | `EmoText` [29] | Stack Overflow, JIRA | [29], [82], [135] |

* Note that we use the machine learning version of `NLTK` instead of its VADER version (which uses a rule-based approach).
** IBM Alchemy is available as a service within IBM Watson at https://www.ibm.com/watson/services/tone-analyzer/.

### Machine Learning-based Techniques

Machine learning-based classifiers leverage machine learning algorithms, such as Support Vector Machines, Naïve Bayes, and Neural Networks. Examples of classifiers that adopt this technique are `NLTK` [19], `Stanford CoreNLP` [170], and `Senti4SD` [28]. For our study, we selected `NLTK` and `Stanford CoreNLP`, which are open source, free to use and very popular [83, 89].

`NLTK` is part of a larger NLP package that provides many other functions.[2] Regarding sentiment analysis, `NLTK` uses a bag of words model. In order to apply `NLTK`, we can adopt two different approaches: train a Naïve Bayes classifier on our data and apply the built model (as we did) or use the VADER (Valence Aware Dictionary and

[2]https://www.nltk.org/

| True sentiment | NLTK classification | Sentence |
|---|---|---|
| Negative | Positive ✗ | I am so happy the game keeps freezing |
| Positive | Positive ✓ | Was blown away by some of the developments in the story in this game, not gonna spoil but def a must try |

(a) Example of classifications made by `NLTK`.

| True sentiment | SentiStrength classification | Sentence | Positive strength | Negative strength |
|---|---|---|---|---|
| Negative | Positive ✗ | I am so happy the game keeps freezing | **2** | -1 |
| Neutral | Positive ✗ | The game was nothing special | **2** | -1 |
| Positive | Negative ✗ | Was blown away by some of the developments in the story in this game, not gonna spoil but def a must try | **1** | -2 |

(b) Example of classifications made by `SentiStrength`.

Figure 5.1: Examples of sentiment classifications.

sEntiment Reasoner) model, which was trained on social media texts, such as microblogs [103]. The latter approach provides four scores for each sentence: *compound* (varies from very negative to very positive as indicated by a score in the range [-1, +1]), *negative* (probability of being negative), *neutral* (probability of being neutral), and *positive* (probability of being positive). In the former approach, we train a Naïve Bayes model to classify each review (it provides the probability of being positive). Figure 5.1a presents examples of reviews classified by the machine learning version of `NLTK`. As we can see, the positive example is correctly classified. However, `NLTK` is not able to capture the negative sentiment of the sentence "*I am so happy the game keeps freezing*", which contains sarcasm.

`Stanford CoreNLP` was developed by the Stanford Natural Language Processing Group[3] at Stanford University. The authors propose a model called Recursive Neural Tensor Network, of which the implementation is based on a Recurrent Neural Network (RNN). The technique consists of parsing the text to be classified into a set of sentences and performing a grammatical analysis to capture the compositional semantics of each sentence [88, 103, 170]. Then, a score between '0' and '4' is assigned for each sentence, in which '0' means a *very negative* sentiment, '1' means *negative*, '2' refers

[3]https://nlp.stanford.edu/

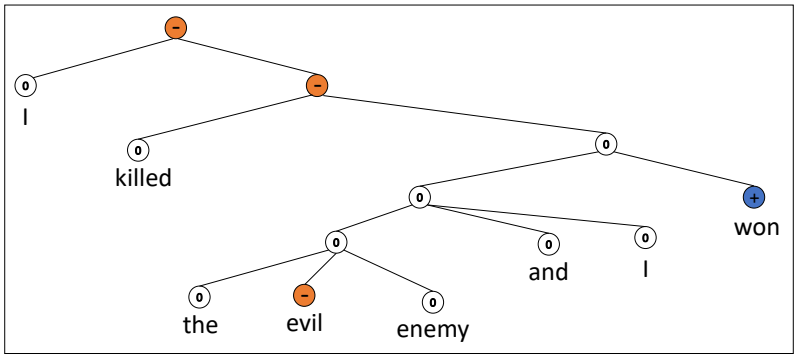



Figure 5.2: Example of the Recursive Neural Tensor Network predicting the sentiment in a sentence.

to a *neutral* sentiment, and '3' and '4' refer to *positive* and *very positive* sentiments, respectively. To classify a game review (composed of more than one sentence), we adopt the following approach [89]: -2*(#0) - 1*(#1) + 1*(#3) + 2*(#4), in which #0 refers to the number of sentences with score 0, and so on. If the resulting score is above zero, the review sentiment is positive; if it is below zero, the review sentiment is negative; otherwise, the review sentiment is neutral.

In Figure 5.2, we can see an example of how the sentence "*I killed the evil enemy and I won*", which is positive, is wrongly classified using `Stanford CoreNLP` (the root node indicates it is a negative sentence). As we can observe, each node in the parse tree is assigned a score (from *negative* to *neutral* to *positive*) and the final sentiment is obtained via the compositional structure of the tree. We can see that different nodes are assigned different sentiments (relative to the partial sentence composed up to that node) and the sentiment contained in the root is supposed to capture the overall sentiment of the full sentence, which is opposed to only inspecting the sentiment of each word individually and summating the scores. This example was obtained from the `Stanford CoreNLP` sentiment analysis website with the live demo tool.[4]

[4]http://nlp.stanford.edu:8080/sentiment/rntnDemo.html [Accessed online: March 11th, 2020]

### Rule-based Techniques

Rule-based classifiers are based on a predefined list of words along with their sentiment score. The piece of text is split into words, and the scores of each word are composed into a final score for the entire piece. Examples of rule-based classifiers are `SentiStrength` [176], `SentiStrength-SE` [83], and `EmoTxt` [29]. In this work, we apply `SentiStrength`, which is of one of the most used sentiment classifiers across different domains, such as social media (e.g., Twitter) [7], and movie reviews [129].

`SentiStrength` is a rule-based classifier to classify sentences into sentiments based on a word bank in which each word has a sentiment score associated with it (this is also called lexical analysis). This classifier is based on a model trained on the MySpace social media network [103]. The document under analysis must be tokenized into sentences, which are assigned two scores based on the summation of each word's score: a positive strength score (how positive is the text) that ranges from 1 (*not positive*) to 5 (*very positive*), and a negative strength score (how negative is the text) that ranges from -1 (*not negative*) to -5 (*very negative*).

Figure 5.1b presents some examples of classifications made by `SentiStrength`. We can see that the classifier's approach of getting the sentiment score of each word individually and summating the scores does not work for some cases. The classifier is not able to capture the negative sentiment in the sentence "*I am so happy the game keeps freezing*", which is sarcastic. This possibly happens due to the presence of the word "happy", which is a positive word and misleads the tool to classify the whole sentence as positive. In addition, `SentiStrength` is not able to capture the neutral sentiment in the sentence "*The game was nothing special*", possibly due to the presence of the positive word "special", which is positive.

## 5.4 Related Work

In this section, we describe prior work on the application of sentiment analysis on game data and on other types of data. We also discuss empirical studies on game reviews. Note that, in our work, we do not aim at proposing a new sentiment analysis technique. Instead, we investigate the performance of existing sentiment classifier on game reviews and reveal the root causes for wrong classifications. We focus on popular sentiment classifiers, which are not computationally expensive (e.g., deep learning-based classifiers).

**Sentiment Analysis on Game Data and Reviews**. Thompson *et al.* [177] studied how to extend a lexicon-based sentiment analysis technique for the purpose of analyzing StarCraft 2 player chat messages. The authors updated the entries to the word dictionary and tailored it to the gaming context. The approach was able to classify sentiment and identify toxicity of instant messages across 1,000 games. The best fitting model outperformed the baseline (which predicts that every message has a positive sentiment) for the sentiment classification. The authors also performed a niche analysis, which showed that the model performances remained relatively stable across regions, leagues, and different message lengths. Stråat and Verhagen [172] investigated user attitudes regarding previously released video games. The authors performed a manual aspect-based sentiment analysis on all user reviews from two game franchises: the PC-version of three games from the Dragon Age franchise and the three first games from the Mass Effect franchise. The data was collected from the Metacritic platform. The paper showed that the rating of a user review highly correlates with the sentiment of the aspect in question, in the case of a large enough data set. Zagal *et al.* [213] studied 397,759 game reviews to identify the sentiment of 723 adjectives used in the context of video games. The authors found that some words which are generally used with a negative (or positive) connotation have a positive (or negative) connotation in the game domain. Finally, Chiu *et al.* [37], Raison

*et al.* [151], and Wijayanto and Khodra [195], and Yauris and Khodra [208] analyzed the sentiment about specific aspects of the game (such as graphics and storyline) in reviews. For example, take the following sentence: "*An okay game overall, good story with very bad graphics*". The player has a positive feeling about the game's storyline, but a negative feeling about the game's graphics. An aspect-based sentiment analysis would compute a different sentiment score for each mentioned aspect.

The aforementioned leveraged different approaches (e.g., lexicon-based and aspect-based) to perform sentiment analysis on different types of data. In contrast, we evaluate existing sentiment analysis techniques on game reviews, identify the causes for misclassifications, and quantify the impact of those causes in the performance of the classifiers.

**Sentiment Analysis on Other Types of Data.** Agarwal *et al.* [3] built different types of models (a feature based model and a tree kernel based model) to perform two classification tasks using Twitter data: a binary task to classify tweets into positive and negative classes; and a 3-way task to classify tweets into positive, negative, and neutral classes. The authors showed that both models outperform the state-of-the-art approach by then, which consisted of a unigram model. The proposed models presented a gain of 4% in performance in comparison to the baseline. Saif *et al.* [160] also used Twitter-related data to build sentiment analysis models. The authors added semantic features into the three different training datasets: a general Stanford Twitter Sentiment (STS) dataset, a dataset on the Obama-McCain Debate (OMD), and one on Health Care Reform (HCR). The results showed that combining semantic features with word unigrams outperforms the baseline (only unigrams) for all datasets. On average, the authors increased the accuracy by 6.47%.

Lin and He [104] proposed a probabilistic modeling framework based on Latent Dirichlet Allocation (LDA) to detect sentiment and topic at the same time from a piece of text. The authors evaluated the model on a movie review dataset and they only consider two classes: positive and negative. The results showed that the pro-

posed approach obtained an accuracy of 84.60%, outperforming some state-of-the-art approaches. Guzman and Maalej [68] proposed an automated approach to analyze mobile app reviews. The authors used the `NLTK` classifier to identify fine-grained app features in the user reviews. They obtained the sentiment of these features and used topic analysis to group them into higher-level groups. The authors used 7 apps from the Apple App Store and Google Play Store and their approach presented a precision of 59% and a recall of 51%. Rigby and Hassan [154] used a psychometrically-based linguistic analysis tool called Linguistic Inquiry and Word Count (LIWC) to examine the Apache httpd server developer mailing list. The authors assessed the personality of four top developers, including positive and negative emotions present in the mailing list. Among the results, the authors found out that the two developers that were responsible for two major Apache releases had similar personalities, which were different from other developers on the traits of extroversion and openness. Bazelli *et al.* [13] analyzed StackOverflow posts to identify and compare developers' personality types. They also used the LIWC tool. The results show that, compared to medium and low reputed users, top reputed post's authors are more extroverted, indicating the presence of social and positive LIWC measures as well as the absence of tentative and negative emotional measures. In addition, authors of up voted posts present less negative emotions than authors of down voted posts.

The aforementioned works used data from three different sources: Twitter, movie reviews, and mobile app reviews. On the other hand, we focus on game reviews from a digital distribution platform (Steam).

**Studies on Game Reviews.** Zagal *et al.* [211] analyzed and characterized game reviews from different websites. The authors used open coding to come up with the topics present in the reviews. Their findings show that game reviews are rich and varied in terms of themes and topics covered. For instance, players might post descriptions of the game under review, their personal experience, advice to other players who read the review, and suggestions for game improvements. Zagal and Tomuro

[212] performed a study on a large body of user-provided game reviews aiming at comparing the characteristics of the reviews across two different cultures. The authors collected reviews from *Famitsu* and *Game World* (Japanese gaming websites) and from *Gamespot* and *Metacritic* (US gaming websites). Among the findings, the authors mention that American players value the replay of a game, while Japanese players are more strict towards bugs. The works mentioned above studied the characteristics of game reviews and what are the differences between reviews from different cultures. Differently, on our work, we use game reviews for the purpose of evaluating existing sentiment classifiers and come up with the causes for wrong classifications.

As we can see, all the aforementioned works proposed new sentiment analysis models and explored the characteristics of game reviews with regard to several different aspects. However, we still lack clarification regarding the performance of existing sentiment classifiers on game reviews, which game review text characteristics impact the performance of sentiment analysis and to what extent they impact it. In our study, we perform a large-scale study with more than 12 million reviews from Steam to evaluate existing sentiment classifiers and reveal text characteristics which are problematic for these classifiers.

## 5.5 Methodology

In this section, we detail the methodology that is used in our study to evaluate existing sentiment classifiers on game reviews from Steam and identify the root causes for wrong classifications. Figure 5.3 presents a complete overview of our methodology, which is detailed next.

### 5.5.1 Collecting Game Reviews

We collected the reviews of all 8,025 games that were available in the Steam Store on March 7th, 2016 using a customized crawler. We removed games that had less than 25 reviews from our initial dataset to reduce a possible bias in our results due to a

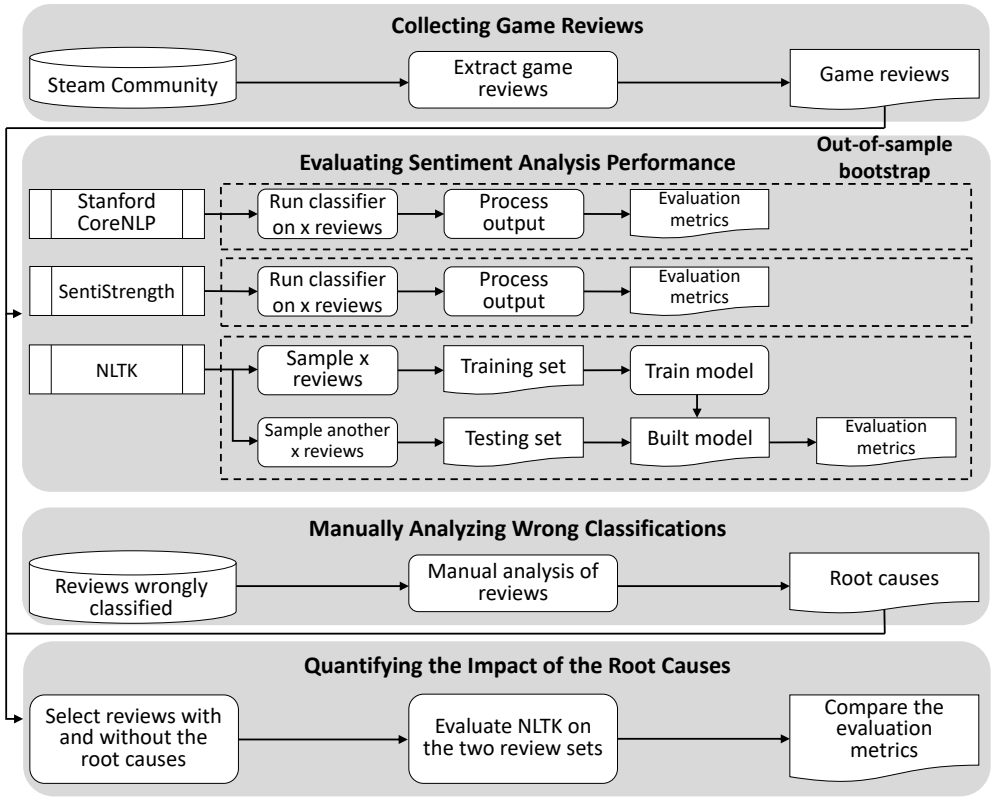



Figure 5.3: Study methodology overview.

small number of reviews (e.g., because a large portion of those reviews were posted by friends of the developers). In total, we collected reviews of 6,224 games. We extracted all the reviews for each game from the Steam Community and ended up with a total of 12,338,364 reviews across all supported natural languages. Steam provides a filter for the language of reviews for a game. We crawled the reviews in each language separately using this filter, to identify the language of each review. Most reviews are written in English (6,850,130), but there are also reviews in Russian (1,789,979), German (525,548), Spanish (469,582), Portuguese (441,145), and French (396,057), since the classifiers can handle several different languages. Besides the review itself, we also collected other available data: the recommendation flag (i.e., whether the reviewer recommended the game or not), early access status (i.e., whether a game is in the early access stage or not), the number of playing hours, the author ID, the date when the review was posted, helpful count, not helpful count, funny count, and

the URL of the review.

Note that our data consists of Steam game reviews, which is different from Metacritic reviews. The Metacritic website[5] aggregates game reviews from professionals and amateurs. While amateur reviews have similarities with Steam reviews (e.g., Metacritic amateur reviews contain gameplay and experience descriptions), professional reviews are much longer and more complex [163]. Therefore, further investigation is necessary to properly assess whether the sentiment classifiers adopted in our work can be applied to professional Metacritic reviews.

### 5.5.2 Evaluating Sentiment Analysis Performance

We evaluated the performance of three sentiment analysis classifiers on game reviews, namely `Stanford CoreNLP` (version 3.9.2) [170], `NLTK` (version 3.4) [19], and `SentiStrength` (Windows version) [176]. For the purpose of evaluation of the classifiers, we consider the game recommendation flag on Steam as the sentiment truth label in our data, that is, we make the assumption that a review that recommends a game has a positive sentiment, while a review that does not recommend a game has a negative sentiment. Our dataset contains 10,603,348 positive reviews (*recommendation* = 1) and 1,735,016 negative reviews (*recommendation* = 0).

Although our dataset is imbalanced, we do not fix the imbalance since our goal is to evaluate existing techniques and reveal the root causes for wrong classifications rather than proposing a new sentiment analysis technique that outperforms the state-of-the-art. Furthermore, in the real world, data distribution is often imbalanced [33, 204] and existing re-sampling techniques have serious defects for text data [204]. Finally, we adopt the Area Under the Receiver Operating Characteristic Curve (AUC) evaluation metric, which is a robust metric with imbalanced data [80].

Regarding `NLTK`, we have the option to train it on our own data using the Naïve Bayes algorithm. Since it is computationally expensive to train and test it on our

[5]https://www.metacritic.com/

entire data, we adopt the out-of-sample bootstrap technique [51] to perform the training and testing, since the use of this technique allows us to avoid possible bias in the training and testing sets as we would have with a simple one-time sampling. In our work, the out-of-sample bootstrap technique consists of randomly sampling 100K reviews (sample) with no replacement from the entire set of reviews (population) to train the classifier. Then, we randomly select another 100K reviews from the pool of remaining reviews to test the classifier. The sample size (100K) was appropriately determined in a pre-study (detailed in Section 5.6). The bootstrap process is repeated 1,000 times, which is enough to represent the entire population and reduce a possible bias in the training and testing sets. Note that for all executions of `NLTK`, before performing the classification itself, we have a preprocessing pipeline, which consists of tokenization, case normalization, and stop word removal. For the `SentiStrength` and `Stanford CoreNLP` classifiers, we also adopted the bootstrap technique and evaluated them on the same 1,000 samples used to test `NLTK`. Note also that we opted for not changing the configurations of the ready-to-use classifiers, such as `SentiStrength` and `Stanford CoreNLP`, as prior work has mostly used them without changes to their configuration [68, 88, 89, 103]. Keeping the configuration of classifiers similar to the configuration previously used allows us to evaluate and compare our results with existing literature more fairly.

For all the classifiers, we computed the Area Under the Receiver Operating Characteristic Curve (AUC). With the bootstrap process, we are able to obtain the AUC distribution for all the classifiers (1,000 AUC values corresponding to the 1,000 bootstrap iterations). The Receiver Operating Characteristic Curve plots the true positive rate against the false positive rate. The AUC measures the classifier's capability of distinguishing between positive and negative sentiments and ranges from 0.5 (random guessing) to 1 (best classification performance). For the cases in which the classification is neutral, we always consider it as a wrong classification since our data has only two labels: positive (the reviewer recommends the game) and negative (the reviewer

does not recommend the game).

We compared the AUC distributions using the Wilcoxon rank-sum test. The Wilcoxon rank-sum test is an unpaired, non-parametric statistical test, where the null hypothesis is that two distributions are identical [197]. If the p-value of the applied Wilcoxon test is less than 0.05, then we can refute the null hypothesis, which means that the two distributions are significantly different. In addition to checking whether the two distributions are different, we provide the magnitude of the difference between the two distributions using Cliff's delta $d$ [109] effect size. We adopt the following thresholds for $d$ [156]:

$$\text{Effect size} = \begin{cases} negligible(N), & \text{if } |d| \leq 0.147 \\ small(S), & \text{if } 0.147 < |d| \leq 0.33 \\ medium(M), & \text{if } 0.33 < |d| \leq 0.474 \\ large(L), & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

### 5.5.3 Manually Analyzing Wrong Classifications

To understand why classifiers are making wrong classifications and come up with the root causes which might be leading to the poor classification performance, we performed a manual analysis on the reviews that were wrongly classified by each of the three sentiment analysis classifiers we use. With this approach, we are more likely to identify characteristics from the review text itself that might confuse the classifier rather than wrong classifications due to bias in a classifier.

We adopt an inductive approach similar to the open-coding technique [41] to manually analyze the reviews. Initially, two authors independently read 100 reviews, being 50 wrongly classified as positive and 50 wrongly classified as negative. They then came up with causes that might have misled the classifiers. After discussing these causes and reaching an agreement on four causes (plus two categories in which the misclassification was unclear), we selected a representative sample with a confidence level of 95% and a confidence interval of 5%, which corresponds to 382 reviews. This

sample was then classified into the set of agreed upon causes by one author so we could obtain the percentage of reviews for each cause.

### 5.5.4 Quantifying the Impact of the Root Causes

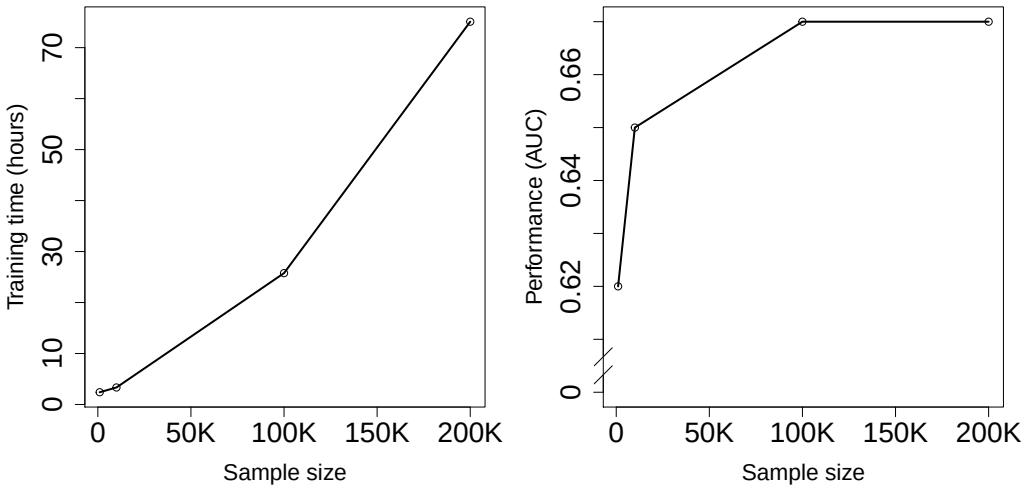
Based on the previous step, in which we extracted possible causes for wrongly classified reviews, we conducted a series of experiments to evaluate the impact of the identified causes, separately, on the performance of the sentiment analysis classifiers. For each cause, we selected the set of reviews that are affected by that cause (the affected set), the set of the remaining reviews (the unaffected set), computed the AUC distribution for both sets, and compared the AUC distributions using the Wilcoxon rank-sum test and the Cliff's delta effect size.

## 5.6 Pre-study

We need to find the best sample size to train the `NLTK` classifier. In this section, we present our pre-study to investigate the performance of `NLTK` with different sample sizes to train and test it.

Training `NLTK` on our entire dataset would be computationally expensive. We designed two experiments to determine the proper training and testing set sample sizes so we can apply the out-of-sample bootstrap technique, as we explained in Section 5.5.2. In Figure 5.4, we can see the plots regarding our experiments. For both cases, we used the following values for the sample size (number of reviews): 1K, 10K, 100K, and 200K. Figure 5.4a presents how the training time (in hours) varies with the sample size. As we can see, the time increases quickly with the increase in sample size (jumping from 26 hours, for 100K, to 75 hours of training time, for 200K). Therefore, a sample size larger than 200K would be infeasible.

Figure 5.4b presents how the performance of the `NLTK` classifier (by means of the median AUC) varies with the increase in the sample size. As we can observe, the plot plateaus when it reaches 100K (presenting an AUC of 0.67), which means using 100K




(a) Sample size versus training time. (b) Sample size versus AUC.

Figure 5.4: Plots of experiments to determine the sample size for `NLTK`.

reviews is sufficient for our purpose. Using the result of this experiment together with the result of the previous experiment, we decided to use a sample of 100K game reviews to train and test the `NLTK` classifier. We also used the same sample size to evaluate the `SentiStrength` and the `Stanford CoreNLP` classifiers.

These results provide evidence of the richness of game review data as we do not need the entire dataset to train our model, indicating that, although the sentiment classification is a tricky problem, we have a rich dataset for which the model does not need huge amounts of data to learn from.

## 5.7 RQ1: How do sentiment analysis classifiers perform on game reviews?

*Motivation:* It is important to verify the performance of widely-used sentiment analysis classifiers on game reviews as this is the first step to understand whether current sentiment analysis classifiers are suitable for classifying the sentiment of such data.

*Approach:* For this research question, we applied the out-of-sample bootstrap with

Table 5.2: Evaluation metrics (median) for unbalanced and balanced dataset.

| Classifier | Acc. | Precision | Recall | F-measure | AUC |
|---|---|---|---|---|---|
| NLTK | 0.61 | 0.60 | 0.70 | 0.54 | 0.70 |
| NLTK (balanced) | 0.67 | 0.73 | 0.67 | 0.65 | 0.67 |
| SentiStrength | 0.52 | 0.56 | 0.63 | 0.47 | 0.63 |
| SentiStrength (balanced) | 0.63 | 0.65 | 0.63 | 0.62 | 0.63 |
| Stanf. Core NLP | 0.37 | 0.52 | 0.53 | 0.35 | 0.53 |
| Stanf. Core NLP (balanced) | 0.53 | 0.54 | 0.53 | 0.51 | 0.53 |

1,000 iterations to evaluate the `NLTK`, `Stanford CoreNLP` and `SentiStrength` classifiers on the game review. To evaluate the classifiers, we computed five metrics: accuracy, precision, recall, F-measure, and AUC. We also performed an experiment to investigate how the length of the reviews affects the performance of the sentiment classification. The reviews were split into 51 groups according to their length: reviews with less than 20 characters, reviews with length between 20 and 40 characters (exclusive), reviews with length between 40 and 60 characters (exclusive), and so on up to the last group of reviews with more than 1,000 characters. We evaluated each classifier with a sample of 10K reviews from each length range. Finally, we compared the performance of the sentiment classification of game reviews with the sentiment classification of other three corpora (Stack Overflow posts, Jira issues, and mobile app reviews), as indicated by prior work [84, 103].

*Findings:* Table 5.2 presents the metrics for the imbalanced and balanced versions of the dataset. Note that we provide all these metrics for the purpose of comparisons with prior (and future) work, but for our discussions, we will focus on the AUC metric. `NLTK` **achieved the best performance of sentiment analysis (in the studied configuration) on game reviews while** `Stanford CoreNLP` **presented the worst performance.** Figure 5.5 presents the distribution of the AUC metric

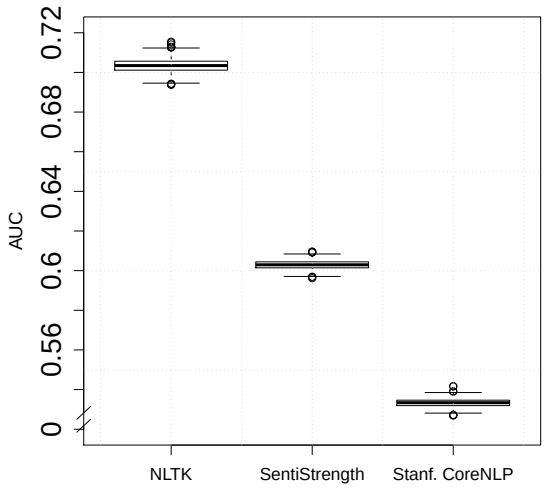



Figure 5.5: Distribution of the AUC.

for the classifiers (each value corresponds to an iteration of the bootstrap).

The AUC for `NLTK` varies from 0.69 up to 0.72, with a median value around 0.70. For `SentiStrength`, the AUC ranges from 0.60 to 0.61 with a median of 0.60, while for `Stanford CoreNLP`, the AUC ranges from 0.53 to 0.54 with a median of 0.53. For all classifier pairs ([`NLTK`, `SentiStrength`], [`NLTK`, `Stanford CoreNLP`], and [`SentiStrength`, `Stanford CoreNLP`]), the Wilcoxon rank-sum test shows that the two distributions are significantly different, with a large Cliff's delta effect size.

Figure 5.6 shows that the performance of the classifiers remains mostly stable across different review lengths, with the largest changes occurring for reviews with less than 20 characters (AUC of 0.65 for `NLTK`) and reviews with more than 1,000 characters (AUC of 0.61 for `NLTK`). We can also see that `NLTK`'s performance slightly reduces as the review length increases. Finally, we computed the distribution of different review lengths in our dataset. We found that 75% of the reviews are in the range 20-1000 characters (where `NLTK` performs best), while 20% of the reviews have less than 20 characters, and 5% of the reviews have more than 1,000 characters.

Finally, Table 5.3 presents the F-measure metric of the sentiment classification

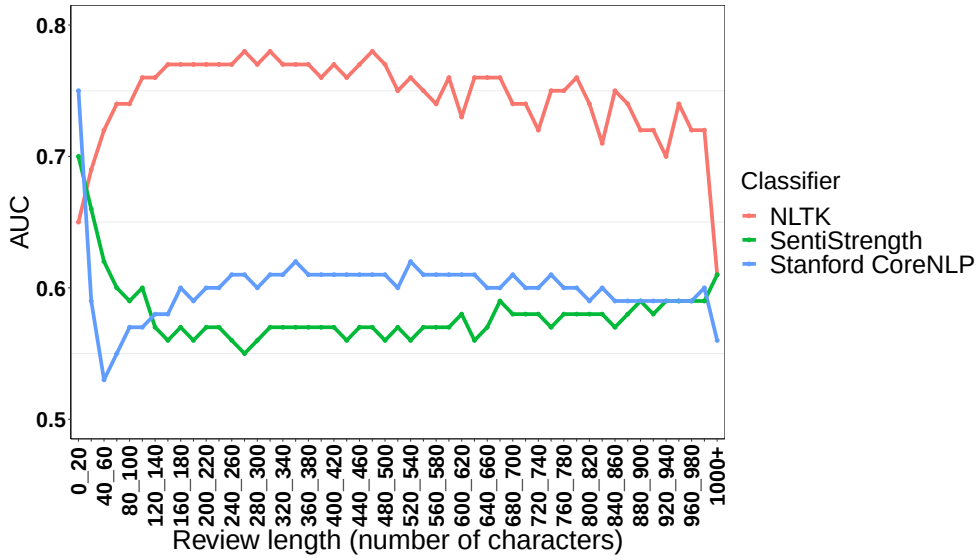



Figure 5.6: Performance of classifiers for different length ranges. Note that there is a data point for every range of 20 characters (0-20, 20-40, and so on). However, for the purpose of a better visualization, the figure only displays every other range in the $x$ axis (e.g., the label '20_40' is not shown in the plot, but the corresponding data point for that range is present in the plot).

Table 5.3: F-measure of sentiment classification across different corpora.

| Corpora | NLTK | SentiStrength | Stanford CoreNLP |
|---|---|---|---|
| Game reviews | 0.54 | 0.47 | 0.35 |
| Stack Overflow posts | 0.21 | 0.34 | 0.28 |
| Jira issues | **0.55** | **0.62** | **0.52** |
| Mobile app reviews | 0.53 | **0.64** | **0.74** |

across different corpora. The text in bold represents a classification performance better than for game reviews. As we can see, the classifiers usually perform better when using a corpus other than game reviews. After training `NLTK` on game reviews, it achieves a performance that is similar to the performance on the Jira issues and mobile app reviews corpora. However, `SentiStrength` and `Stanford CoreNLP` work much better on the Jira issues and mobile app reviews corpora compared to game reviews.

Table 5.4: Root causes for misclassifications in sentiment analysis (each review may be assigned to more than one root cause).

| **Root cause** | **Definition** | **Occurrence (%)** |
|---|---|---|
| Contrast conjunctions | The review points out both the advantages and disadvantages of the game, frequently using contrast conjunctions | 30 |
| Game comparison | The review contains a comparison with another game or with a previous version of the game itself | 25 |
| Negative terminology | The review contains words such as *kill* and *evil* which are not necessarily bad for specific game genres (e.g., action games) | 23 |
| Unclear | It is not clear what might have caused the wrong classification | 21 |
| Sarcasm | The review contains sarcastic text | 6 |
| Mismatched recommendation | The user might have entered a wrong recommendation: positive (negative) recommendation with a negative (positive) review content | 6 |

Overall, sentiment analysis classifiers do not achieve a high performance, performing worse on game reviews than on other domains. The median AUC ranged from 0.53 (`Stanford CoreNLP`) to 0.70 (`NLTK`).

## 5.8 RQ2: What are the root causes for wrong classifications?

*Motivation:* Understanding what is causing sentiment analysis classifier to make wrong classifications is essential to extract important insights about how to improve existing sentiment analysis for game reviews. Such knowledge can be used to fix problems in the classification pipeline and achieve a better performance.

*Approach:* We start by (1) selecting the reviews that were misclassified by all the three classifiers simultaneously (i.e., the intersection of misclassified reviews). We then (2)

use the pool of all misclassified reviews to select a statistically representative sample of 382 reviews for the manual analysis. We adopted an open coding-like approach to identify the root causes which could affect sentiment analysis classifiers' performance. Two authors independently analyzed a sample of 100 reviews (50% wrongly classified as positive and 50% wrongly classified as negative) to identify the root causes that may confuse the classifiers. Our manual analysis had an agreement of 83% between the two authors (we consider an agreement when both authors agreed that the root cause X is related to a review Y). After reaching the agreement, one author analyzed a statistically representative sample of 382 reviews (which yields a confidence level of 95% with a confidence interval of 5) to compute the frequency of occurrence of each cause. Note that each misclassification may be assigned to more than one root cause (if that is the case). The sample of 382 reviews for the manual analysis was obtained from the reviews that were misclassified by all three classifiers. We focused on reviews that were misclassified by all classifiers to better identify characteristics of the review text that affect the sentiment analysis classification, rather than a characteristic of only a single classifier.

*Findings:* **We revealed four types of possible causes for sentiment misclassifications: use of contrast conjunctions to indicate the advantages and disadvantages of a game in the same review, comparison to other games, reviews with negative terminology, and sarcasm.** Table 5.4 presents all the root causes we identified along with their definitions and percentage of occurrence. As we can see, the most common cause is contrast conjunctions (30%), followed by game comparison (25%), negative terminology (23%), and sarcasm (6%). Cases for which we are not able to clearly identify the cause for the wrong classification (unclear) occurred in 21% of the reviews. Cases in which the review content did not match the recommendation (mismatched recommendation) occurred in 6% of the reviews.

Next, we present each root cause in detail along with corresponding examples of

reviews.

## Root cause 1: Contrast conjunctions

**Description**: The review points out advantages and disadvantages of the game.

**Symptoms**: This type of review frequently makes use of contrast conjunctions (*but*, *although*, *though*, *even though*, and *even if*) when presenting positive and negative points about the game. As we can see in the example below, the review contains a positive view ("*I love this game...*") and a negative view ("*...it keeps flickering please help!*") about the game separated by the conjunction *but*.

**Example**: "*I love this game but it keeps flickering, please help!*".

## Root cause 2: Game comparison

**Description**: The review compares the game with another game or a previous version of the game itself. Such comparisons might make the sentiment classification more difficult since positive or negative points might refer to the other game or the game itself in a previous version instead of the current game version under review.

**Symptoms**: The review mentions one or more games [A, B...] in a review for another game [G], or mentions a version 1.x of the game [G] in a review for the version 2.x of the same game [G]. In the example below, the review for the *Terraria* game compares the reviewed version of the game with a previous version.

**Example**: "*Terraria was one of the best games I've ever played, but after they released 1.2, I stopped enjoying it!*".

## Root cause 3: Negative terminology

**Description**: The review uses (supposedly) negative terminology (i.e., words with a negative connotation), which might mislead the classifier towards a negative sentiment classification even though many times the review text has a positive sentiment (as indicated by the recommendation of the game).

**Symptoms**: The review contains words that are considered negative in many situations (e.g., *kill*, *evil*), which might not have a negative connotation for games of specific genres, such as first-person shooter games. The review in the example below contains supposedly negative words, such as *kill*, although it is just describing the role of the player in the game. In fact, the reviewer recommended the game and even made it explicitly by assigning a score of 10 out of 10 to the game.

**Example**: "*I've played like 15 games [...], zombies just go around you, you can't run, just keep trying to kill them.*".

## Root cause 4: Unclear

**Description**: We are not able to clearly identify a pattern or characteristic that might be confusing the classifier.

**Symptoms**: There is no symptom. We cannot identify a clear possible reason which might mislead the classifier. The review in the example below was classified as positive while it should be negative.

**Example**: "*Downloaded Game into steam, Played for 40 Hours total. Game disappeared from computer. Redownloaded, Played for a while, Game disappeared again. As someone with a download cap and 2 other gamers in the house, was. not. impressed*".

## Root cause 5: Sarcasm

**Description**: The review contains sarcastic text. Sarcasm occurs when an apparently positive text is actually used to convey a negative attitude (or vice-versa) [61]. Prior work has shown that sarcasm is difficult to automatically identify [138].

**Symptoms**: The review contains sarcasm, which is observed when the reviewer writes an (apparently) positive text intending to transmit a negative message (or vice-versa). The review in the example below contains sarcastic text as the reviewer makes use of positive words (e.g., *great*), when the person actually points out a

negative aspect about the game.

**Example**: "*Great for uninstalling 11/10 would uninstall again*".

### Root cause 6: Mismatched recommendation

**Description**: It means the reviewer might have entered a wrong recommendation, which does not match with the review content itself. Note that this root cause is different from sarcasm (as we cannot clearly identify a positive review intending to transmit a negative attitude or vice-versa), however both causes are hard to be automatically identified.

**Symptoms**: The reviewer is positive about the game, but they did not recommend the game (or vice-versa). The example below presents a review that was classified as positive (as expected since the text clearly expresses a positive sentiment). However, the reviewer did not recommend the game, which we assume was a mistake of the reviewer.

**Example**: "*I love this GAME!*".

We identified four root causes for wrong classifications of sentiment analysis classifiers: use of contrast conjunctions (30%), game comparisons (25%), negative terminology (23%), and sarcasm (6%).

## 5.9 RQ3: To what extent do the identified root causes impact the performance of sentiment analysis?

*Motivation:* It is important to quantify the impact of each identified root cause to the overall performance of sentiment analysis on game reviews. Such knowledge will support the prioritization of the causes to be addressed, the implementation of better sentiment analysis tools to be deployed in gaming contexts, and a research agenda to address such issues.

*Approach:* For this part of the study, we first identified the root causes which are

feasible to be automatically identified in reviews. Then, we implemented detection heuristics to identify reviews affected by each root cause. We focused on the following root causes for which we can automatically identify reviews: **contrast conjunctions**, **game comparison**, and **negative terminology**. After identifying such reviews, we re-ran the `NLTK` classifier on both groups: the set of identified reviews (affected set, which is supposedly harder for the classifier) and the set of remaining reviews (unaffected set, which is supposedly easier for the classifier since they do not contain the cause for the wrong classification).

Note that, in this last part, we focused only on the `NLTK` classifier as it presented the best performance (Section 5.7) and it can be trained on our data. Furthermore, we also applied the bootstrap technique with 1,000 iterations, as we previously did.

Next, we explain the implemented detection heuristics and the obtained results for each root cause.

### 5.9.1 Contrast Conjunctions

*Detection heuristic:* We noticed that reviews which point out the advantages and disadvantages of a game usually use contrast conjunctions to transmit the idea of contrast between advantages and disadvantages of the game. We defined a list with the contrast conjunctions we observed in our manual analysis and performed a keyword-based search in our dataset to identify reviews that contain one or more conjunctions of the list. Table 5.5 presents the selected conjunctions with examples.

Among the most frequent conjunctions found in the reviews, we have "but" (1,941,535), "although" (104,295), and "even if" (66,802). After the search, we ended up with 10,187,926 reviews in the remaining set (82% of the original dataset) and 2,150,438 reviews in the detected set (identified by the heuristic).

*Findings:* **Game reviews with contrast conjunctions are indeed more difficult to classify for `NLTK`, with a median AUC that is 11% lower than for reviews without contrast conjunctions.** Figure 5.7 presents the distributions

Table 5.5: Contrast conjunctions and corresponding examples.

| Contrast conjunction | Example |
|---|---|
| But | *Nice Matchmaking,* ***but*** *if you are not premium you have no chance...* |
| Although, though | ***Although*** *I really enjoy this game, I do think that PTTM still remains the best in the series...* |
| Even though, even if | ***Even though*** *it gets progressively difficult and you won't get the perfect items each run, you'll find yourself coming back for more...* |

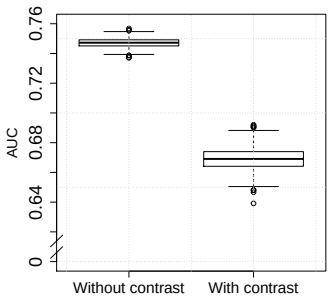



Figure 5.7: AUC distribution for reviews without and with contrast.

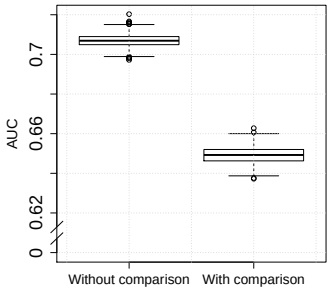



Figure 5.8: AUC distribution for reviews without and with comparison.

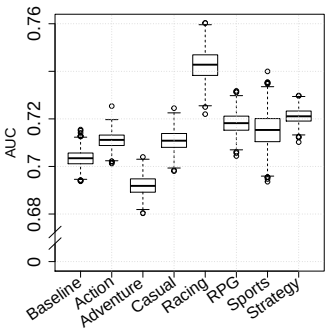



Figure 5.9: AUC distribution for reviews of all the game genres and the baseline.

of the AUC for the sets of reviews without and with contrast conjunctions. The Wilcoxon rank-sum test shows that the two distributions are significantly different, with a large (1.0) Cliff's delta effect size.

As we can observe, the AUC of reviews without contrast is much higher (large Cliff's delta effect size) than the AUC of reviews with the presence of contrast conjunctions. In fact, we found a median AUC of almost 0.75 for the group without contrast, while for the group with contrast the median AUC is around 0.67 (11% lower).

### 5.9.2 Game Comparison

*Detection heuristic:* We collected the top-500 most played games from Steam and, based on this list, we performed a keyword-based search in our dataset to identify reviews that mention other games.

We collected the most played games from the SteamDB platform.[6] This list was obtained based on the peak number of players who have played the game. For instance, the number one game in the list is *Playerunknown's Battlegrounds* (3,257,248 players), followed by *Dota 2* (1,295,114 players) and *Counter-Strike: Global Offensive* (854,801 players). This data was collected on January 10th, 2020. Note that, although the game reviews were collected in 2016, their age does not impact our analysis.

We performed a keyword-based search on the reviews in our dataset. We ensured that both the game name being searched and the review text were lower case during the search. Among the most mentioned games in the reviews, we found *Terraria* (31,860), *Dota 2* (30,385), and *Counter-Strike* (21,418). After the search, we ended up with 11,753,211 reviews in the remaining set (95% of the original dataset) and 585,153 reviews in the detected set (identified by the heuristic).

*Findings:* **Game reviews with comparisons are actually more difficult to classify for `NLTK`, with a median AUC that is 8% lower than for reviews without comparisons.** After training `NLTK` on both sets, we computed the AUC using the out-of-sample bootstrap with 1,000 iterations, as we did previously. Figure 5.8 presents the distributions of the AUC for the sets of reviews without and with comparison. The Wilcoxon rank-sum test shows that the two distributions are significantly different, with a large (1.0) Cliff's delta effect size.

As we can see, reviews without comparison present a higher AUC than reviews with comparison. In fact, we found a median AUC of almost 0.71 for the group

[6] https://steamdb.info/

without comparison, while for the group with comparison the median AUC is around 0.65 (8% lower), which indicates that, similarly to the case of reviews with contrast conjunctions, comparisons can also degrade the performance of sentiment analysis.

### 5.9.3 Negative Terminology

*Detection heuristic:* We noticed that some game reviews use words with a negative connotation, such as *kill*, *evil*, and *death*. Although such words might refer to negative aspects of something in a usual context, within the context of games they might be used without the negative connotation. For instance, when describing the role of a character in an RPG (Role-Playing Game) game, one might say they need to **defeat** and **kill** the **enemy**. Although the review uses (supposedly) negative words, its final content might be positive towards the game (i.e., the reviewer might recommend the game even when using negative words).

For this root cause, instead of adopting the approach as we did for the previous causes, we propose a stratified training process for the sentiment analysis classifier based on the game genre, which we call per-genre training. We used a customized crawler to collect the game genre from Steam for each review in our dataset and grouped reviews by genre so we could train the classifier separately by genre. We found a list of seven game genres (excluding generic genres reported by Steam, such as Early Access, Free to Play, and Indie): Action, Adventure, Strategy, RPG, Casual, Racing, and Sports.

We established a maximum period of one month to collect the game genres for a randomized version of our data, which resulted in genres for 4 million reviews. It would be infeasible to collect the genre for our entire dataset in a timely manner due to restrictions when using a crawler to collect online data (such as the limited number of requests allowed per a period of time). Furthermore, for some cases, the review or the profile itself was excluded by the user from the Steam platform. In the case of less popular genres for which we are not able to sample 100K reviews for the training and

testing sets (casual, racing, and sports genres), we adopted a 80/20 percentage split to train and test with the bootstrap technique. For instance, if we had 10K reviews for a specific genre, we would use 8K for training (80%) and 2K for testing (20%). Table 5.6 presents the number of reviews for each genre.

Table 5.6: Game genres and corresponding number of reviews.

| **Genre** | **Number of reviews** |
|---|---|
| Action | 741,569 |
| Adventure | 484,236 |
| Strategy | 395,595 |
| RPG | 372,033 |
| Casual | 128,590 |
| Racing | 43,899 |
| Sports | 33,890 |

*Findings:* **Per-genre training is effective when performing sentiment analysis on game reviews.** Figure 5.9 presents the distribution of the AUC for all the genres and also for the baseline, which is the evaluation of `NLTK` on the entire dataset (Section 5.7). We can see that, for all the genres except for adventure, the median AUC is higher than the median AUC for the baseline. In fact, we obtained the following median AUC values: 0.70 (baseline), 0.71 (action), 0.69 (adventure), 0.71 (casual), 0.74 (racing), 0.72 (RPG), 0.72 (sports), and 0.72 (strategy). The Wilcoxon rank-sum test shows that the AUC distribution for each genre is significantly different from the AUC distribution for the baseline with a large effect size.

Reviews that use contrast conjunctions to point out advantages and disadvantages of the game have the highest negative impact on the performance (11% lower AUC), followed by reviews with game comparisons (8% lower AUC). Furthermore, we show that per-genre training is effective for sentiment analysis on game reviews as it is mostly able to improve the performance of `NLTK`.

## 5.10 Recommendations and research directions for sentiment analysis on game reviews

In this section, we provide practical recommendations for performing sentiment analysis on game review data.

**No need for huge amounts of data.** Through our pre-study we showed that 100K reviews is a sufficient sample size to train and test sentiment analysis classifiers on game reviews. We showed that using more than 100K reviews does not improve the sentiment analysis performance as it plateaus after 100K reviews. Note that this is based on a Naïve Bayes classifier as we aim to provide recommendations for computationally accessible approaches rather than computationally intensive deep learning algorithms. Furthermore, this recommendation is based on the `NLTK` configurations adopted in the study (i.e., the machine learning version of `NLTK` with the same preprocessing steps).

**Prioritize on studying techniques that can deal with reviews with advantages and disadvantages of the game.** Based on the impact that each root cause has on the sentiment analysis performance, we suggest game developers and researchers to develop techniques that can analyze reviews which use contrast conjunctions to point out the advantages and disadvantages of the game under review as this might confuse the classifier. Secondly, we suggest to develop techniques that can deal with reviews which make comparison to games other than the game under review or to previous versions of the game itself. Finally, we suggest the development of techniques to analyze reviews that contain sarcasm.

**Stratify reviews by game genre.** Different game genres have different characteristics in terms of expressions used by reviewers. Therefore, we recommend to stratify the dataset by genre and train the classifier separately for each genre. This approach helps to avoid mixing different types of data when training the model. For instance, negative words (e.g., *evil*) are used for different purposes in reviews of different gen-

res, such as casual (where the reviewer probably uses it with a negative connotation) and first-person shooter (where the reviewer does not intentionally have a negative connotation).

## 5.11 Conclusion

In this chapter, we perform a large-scale study to understand how sentiment analysis works on game reviews. We collected 12 million reviews from the Steam platform. We investigate the performance of existing sentiment analysis classifiers on game reviews, identify which factors might impact such performance and to what extent.

Our study shows that sentiment analysis classifiers do not perform well on game reviews and we identified root causes for such performance, such as sarcasm and reviews with negative terminology. Reviews that point out advantages and disadvantages of a game (through the use of contrast conjunctions) have a high negative impact on the performance (reducing the median AUC by 11%), followed by reviews that contain comparisons to games other then the game under review (reducing the median AUC by 8%). Furthermore, we show that training classifiers on reviews stratified by the genre is effective and can improve the performance of sentiment analysis. For all genres except adventure, the median AUC was higher than the baseline, with significant different AUC distributions and large effect sizes.

Our study is the first important step towards identifying what are the root causes for wrong classifications in sentiment analysis on game reviews and the impact of each cause. Our study calls upon sentiment analysis and game researchers to further investigate how the performance of sentiment analysis on game reviews can be improved, for instance by developing techniques that can automatically deal with specific game-related issues of reviews (e.g., reviews with contrast conjunctions and reviews with game comparisons). Another future direction is to explore how user characteristics affect the performance of the sentiment classification of game reviews.

# Chapter 6

# Leveraging the OPT Large Language Model for Sentiment Analysis of Game Reviews

## 6.1 Abstract

Sentiment analysis is a popular technique to obtain the sentiment from a piece of text, such as a player-provided game review. Automatically extracting players' sentiments about games can help game developers to better understand the aspects of their games that players like or dislike. Prior work showed that traditionally widely-used sentiment analysis techniques do not perform well on game reviews. However, the NLP field has seen a steep progress in the latest years, with substantial performance improvements in many tasks, such as sentiment classification. Therefore, in this chapter, we investigate how a modern sentiment classifier performs on game reviews and compare it with the results of prior work using traditional classifiers. In particular, we use the OPT-175B Large Language Model, the largest model from the Open Pre-trained Transformer suite. Furthermore, we manually analyze the game reviews wrongly classified by OPT-175B to better understand the issues that affect the performance of that model and how those issues compare to the challenges faced by traditional classifiers. We found that OPT-175B achieves (far) better performance than traditional sentiment classifiers, with a 72%-increased F-measure and a 30%-

increased AUC compared to the best traditional classifier investigated in prior work. Also, we found that common challenges of traditional classifiers, such as reviews with game comparisons and negative terminology, have been mostly solved by the OPT-175B model. The players' sentiments about games and their features can be used for several different purposes, such as to guide game testing based on the game features that players like or dislike.

## 6.2 Introduction

As discussed in Chapter 5, widely-used traditional sentiment classifiers performed poorly on game reviews at the time our original study [182] was conducted. However, in the last 3 years, the NLP field has seen a steep improvement across several NLP tasks, such as sentiment classification [17, 25, 39, 46, 149, 158, 181, 188, 201]. Therefore, we perform a follow-up study in which we investigate how a modern sentiment classifier performs on Steam game reviews and if the issues with traditional sentiment classifiers are overcome by the modern model. In particular, we used the Open Pre-trained Transformer language model with 175 billion parameters (OPT-175B), which is a Large Language Model (LLM) with high performance on several NLP tasks [215]. We address the following research questions:

**RQ1: How does OPT-175B perform on the sentiment classification of game reviews?**

We investigate how the OPT-175B model performs on the sentiment classification of game reviews comparably to the traditional sentiment classifiers in the original study [182]. We found that OPT-175B achieves a (far) better performance than traditional classifiers, with an F-measure of 0.93 (72% higher than the best traditional classifier) and an AUC of 0.91 (30% higher than the best traditional classifier).

**RQ2: How do the root causes of wrong classifications made by OPT-175B compare to the root causes of wrong classifications made by traditional**

**sentiment classifiers?**

Just like in our original study, we analyze a representative sample of game reviews that were misclassified (only in this case, by the OPT-175B model) to reveal the causes for wrong classifications made by OPT-175B and compare these with the causes for wrong classifications made by traditional sentiment classifiers. We found that most of the issues with traditional sentiment classifiers (such as reviews with game comparisons and reviews with negative terminology) are solved by OPT-175B.

The remainder of this chapter is organized as follows. Section 6.3 presents a background on the OPT-175B Large Language Model. Section 6.4 presents the methodology that we used in this study. Sections 6.5 and 6.6 present and discuss the results of our study. Section 6.7 discusses the threats to the validity of the work. Finally, Section 6.8 concludes this chapter.

## 6.3 The OPT-175B Large Language Model

A language model is a probability distribution over a sequence of words, which, in summary, predicts the next word based on the previous words [90]. Language models have a wide variety of applications, such as text classification. In this study, we evaluate the OPT language model [215], which is a Large Language Model trained by Meta AI.[1] The model used in our evaluation (OPT-175B) is the largest model available in the OPT model family with 175 billion parameters. Differently from other models, such as GPT-3 [25] which is paid, OPT-175B is an open-source model.

OPT-175B is a prompt-based language model, which means that we need to define a proper prompt to get the sentiment classification of a game review with this model. Prompting consists of manually or automatically designing templates with natural language instructions which are used by a pre-trained model to solve a task [57, 92, 108, 165]. Figure 6.1 shows an example of the prompt structure that we used with a game review that contains sarcasm: "Great for uninstalling. 11/10 that I would

[1]https://github.com/facebookresearch/metaseq

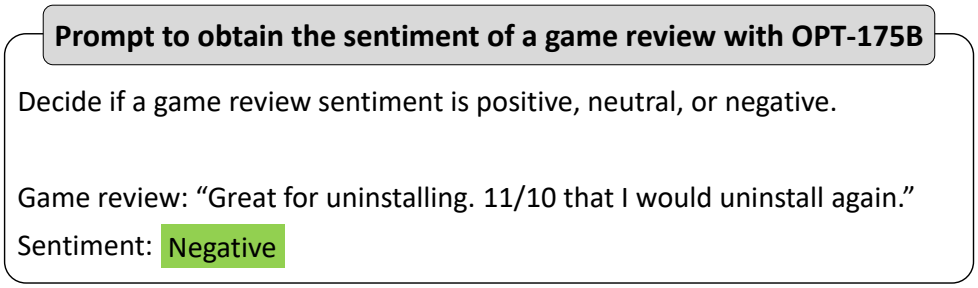



Figure 6.1: Example of a prompt to determine the sentiment of a game review. The text highlighted in green was generated by the OPT-175B model.

uninstall again.”. The text highlighted in green corresponds to the sentiment of the game review, which was generated by OPT-175B. In our prompt, we explicitly tell the model that the text of which it should identify the sentiment is a game review, which helps to capture any game knowledge the model has [174] and achieve a higher performance. We also clearly elicit the possible sentiment options in the prompt: positive, neutral, or negative.

To use Large Language Models such as OPT, we need to define a few parameters. We used the following parameters in our study:

- `temperature`: this parameter controls the randomness involved in the text generation. Lower temperature values result in less random, more repetitive texts. We set the `temperature` to zero to have a deterministic model, since in our task we expect the model to produce the same output given the same input.

- `top_p`: this parameter also controls the randomness and uniqueness in the text generated by the model. However, `top_p` corresponds to the inference time threshold used for sampling the model outputs. If `temperature` is set to zero, it is recommended to set `top_p` to one and vice-versa. Since we use a `temperature` of zero, we set `top_p` to one.

- `max_tokens`: this parameters corresponds to the maximum number of tokens generated by the model. We set `max_tokens` to six, which is sufficient to obtain

the positive, neutral, or negative sentiments from game reviews.

- `stop sequence`: this parameter determines when the model stops generating tokens. The model will stop generating further tokens when a `stop sequence` is generated. Due to the prompt structure that we use, the model generates the game review sentiment and returns to the beginning of the next line. Any generated text beyond that (within the max_tokens) is not important to us, therefore, we used "\n" as the `stop sequence`.

- `frequency penalty`: this parameter penalizes tokens that were already generated in a sentence by reducing their likelihood. Since we are not interested in avoiding repetitiveness, we set the `frequency penalty` to zero.

- `presence penalty`: similarly to the frequency penalty, this parameter reduces the likelihood of already-generated tokens. However, the `presence penalty` does not use the frequency with which the token appeared, but only if the token has been already generated or not. Similarly to the previous parameter, we set `presence penalty` to zero.

## 6.4 Methodology

In this section, we explain the methodology that we used in our study. We discuss the game reviews used to evaluate OPT-175B, the evaluation setup and how we performed the manual analysis. Figure 6.2 presents an overview of our methodology.

### 6.4.1 Selecting game reviews

For this study, we used the game reviews collected from the Steam platform in our prior work [182]. We executed OPT-175B for one week on randomly selected English-language reviews from the originally used game review dataset. After one week, we obtained the sentiment classifications for 614,403 game reviews, which we use for all our analyses and comparisons in this study.

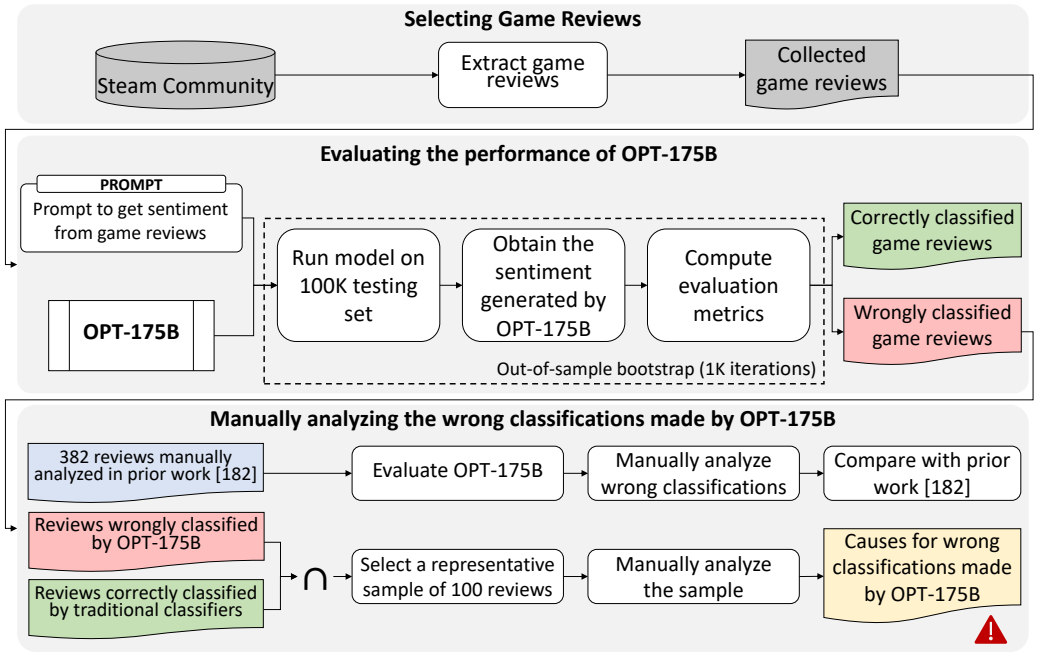



Figure 6.2: Overview of our methodology for evaluating OPT-175B.

### 6.4.2 Evaluating the performance of OPT-175B

To evaluate the performance of OPT-175B, similarly to our original research study [182], we consider the game recommendation flag on Steam as the truth label for the sentiment in our data. Therefore, we consider that if a review recommends a game, it has a positive sentiment, while a review that does not recommend a game has a negative sentiment. In the analyzed subset of 614,403 game reviews, there are 534,916 positive game reviews (recommendation = 1) and 79,487 negative game reviews (recommendation = 0).

We used the testing set samples generated with an out-of-sample bootstrap approach [51] in our previous work [182] to evaluate OPT-175B. In the original work, the bootstrap approach was used to obtain the training and testing sets to train and evaluate the sentiment classifiers. The use of out-of-sample bootstrap allows us to avoid possible bias samples as we would have with a simple one-time sampling. We randomly sampled 100K reviews with replacement from the entire set of reviews to train the sentiment classifiers. Then, we randomly selected another 100K reviews

from the pool of remaining reviews to test the sentiment classifiers. The bootstrap process was repeated 1,000 times, which is enough to represent the entire population and reduce a possible bias in the sampled data for evaluation. In the end, we had 1,000 training sets and 1,000 testing sets. In this study, we use the exact same samples obtained with the bootstrap in the previous work [182] for fair comparisons. However, since we do not need to train OPT-175B, we used only the testing set samples to directly evaluate the performance of OPT-175B with the sentiment classification task. In addition, as we mentioned earlier, we have the OPT-175B classifications for 614,403 game reviews. Therefore, we may not have the sentiment classification for all the 100K testing set reviews for all the 1,000 bootstrap samples. For example, for a particular testing set of 100K reviews, we may have the OPT-175B classification for only 50K reviews. In such cases, we use that subset of 50K reviews to evaluate OPT-175B.

We computed the distribution of the Area Under the Receiver Operating Characteristic Curve (AUC) obtained using all the bootstrap testing set samples (i.e., the 1,000 AUC values corresponding to the 1,000 bootstrap iterations). In addition, we report the following evaluation metrics: accuracy, precision, recall, and F-measure. Similarly to our previous study [182], we focus our discussions on the AUC metric, which measures the classifier's power of distinguishing between positive and negative sentiments and ranges from 0.5 (random guessing) to 1 (best classification performance). Also, we consider all the cases in which the OPT-175B classification is "neutral" as a wrong classification since our data has only two labels (positive and negative).

### 6.4.3 Manually analyzing the wrong classifications made by OPT-175B

To understand the causes of wrong classifications made by OPT-175B and how they compare to the causes for wrong classifications of traditional sentiment classifiers, we

performed two manual analyses. First, we ran OPT-175B with the 382 game reviews manually analyzed in the original study (for which all the traditional classifiers failed) and selected the reviews for which OPT-175B fails for manual inspection. Then, we followed an inductive approach similar to the open-coding technique [41] to analyze a representative sample of game reviews where OPT-175B fails but all the traditional sentiment classifiers (originally studied) get correct classifications, to verify whether OPT-175B is affected by new challenges for sentiment analysis.

## 6.5 RQ1: How does OPT-175B perform on the sentiment classification of game reviews?

*Approach:* We evaluated OPT-175B on the testing set samples obtained with the out-of-sample bootstrap approach in our prior work [182]. There are 1,000 samples and each sample contains 100K game reviews. We report the accuracy, precision, recall, F-measure, and AUC metrics. Similarly to our prior work, we focus our discussions on the AUC metric and report the AUC distribution for the 1,000 testing set samples.

*Findings:* Table 6.1 presents the evaluation metrics for OPT-175B. We also included the metrics for the traditional sentiment classifiers from the original study to facilitate the comparison. OPT-175B achieves (far) better performance for all the reported metrics. For instance, the median F-measure for OPT-175B (0.93) is 72% higher than the best median F-measure of the traditional sentiment classifiers (0.54 for NLTK). In addition, the median AUC for OPT-175B (0.91) is 30% higher than the best median AUC of the traditional sentiment classifiers (0.70 for NLTK).

Figure 6.3 presents the distribution of the AUC metric for the modern (OPT-175B) and traditional (NLTK, SentiStrength, and Stanford CoreNLP) sentiment classifiers (each value corresponds to an iteration of the bootstrap). OPT-175B presents the best performance, with an AUC varying from 0.89 up to 0.93 and a median value of 0.91. In contrast, traditional sentiment classifiers perform worse. The AUC for

Table 6.1: Evaluation metrics (median) for traditional and modern sentiment classifiers.

| Classifier type | Classifier | Acc. | Precision | Recall | F-measure | AUC |
|---|---|---|---|---|---|---|
| Traditional | NLTK | 0.61 | 0.60 | 0.70 | 0.54 | 0.70 |
| | SentiStrength | 0.52 | 0.56 | 0.63 | 0.47 | 0.63 |
| | Stanf. CoreNLP | 0.37 | 0.52 | 0.53 | 0.35 | 0.53 |
| Modern | OPT-175B | **0.92** | **0.94** | **0.92** | **0.93** | **0.91** |

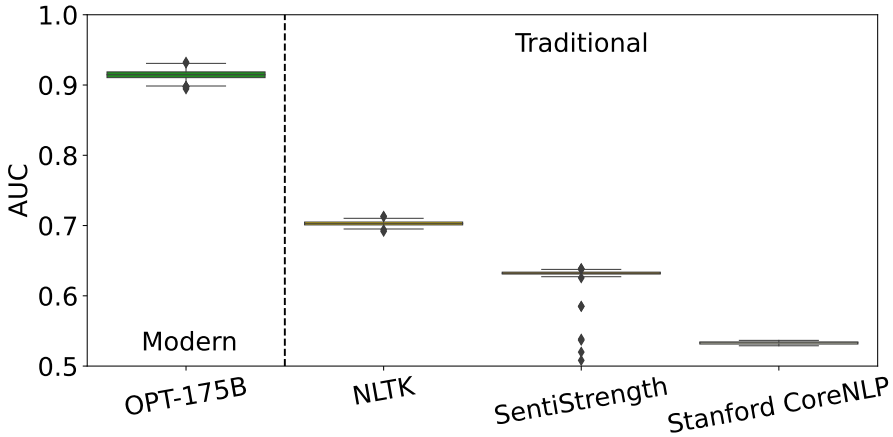



Figure 6.3: Comparison of the AUC distribution with bootstrap samples for modern (OPT-175B) and traditional (NLTK, SentiStrength, and Stanford CoreNLP) sentiment classifiers.

the NLTK classifier varies from 0.69 up to 0.71, with a median value of 0.70. For SentiStrength, the AUC varies from 0.51 up to 0.64, with a median value of 0.63. Finally, for Stanford CoreNLP, the AUC varies from 0.53 up to 0.54 only, with a median value of 0.53.

The OPT-175B model performs better than traditional sentiment classifiers for all the reported evaluation metrics, with a 72%-increased F-measure and a 30%-increased AUC compared to the best traditional classifier in the original study (NLTK).

## 6.6 RQ2: How do the root causes of wrong classifications made by OPT-175B compare to the root causes of wrong classifications made by traditional sentiment classifiers?

*Approach:* To compare the root causes of wrong classifications made by OPT-175B with the causes of wrong classifications made by traditional classifiers, we first executed OPT-175B on the representative set of 382 game reviews manually analyzed in the original study, computed its performance, and manually analyzed the misclassified game reviews. Then, to better understand if OPT-175B is affected by issues other than the issues previously identified for traditional sentiment classifiers, we obtained the set of game reviews correctly classified by all the traditional classifiers but for which OPT-175B fails. This resulted in 2,136 reviews. We selected a representative sample of 100 game reviews from that set (with a confidence level of 95% and a confidence interval of 10%) and manually inspected that sample to identify the causes of wrong classifications.

*Findings:* The OPT-175B model presents a high accuracy (66%) comparably to the traditional classifiers (0%) on the set of 382 game reviews for which all those traditional classifiers fail. Figure 6.4 shows the number of game reviews wrongly classified by the traditional sentiment classifiers (which sums up to 423 reviews) and by OPT-175B (which sums up to 129) for each root cause. Note that each review may be assigned to more than one root cause, which is the reason why the total number of reviews wrongly classified by traditional classifiers is larger than 382. By analyzing the 129 game reviews misclassified by OPT-175B, we can see that contrast conjunctions (which appear in game reviews that discuss several advantages and disadvantages of the game under review) is the main issue that affects OPT-175B, with 44 reviews being wrongly classified because of that (which is a much lower number than the 114 reviews misclassified by the traditional classifiers for that cause). In

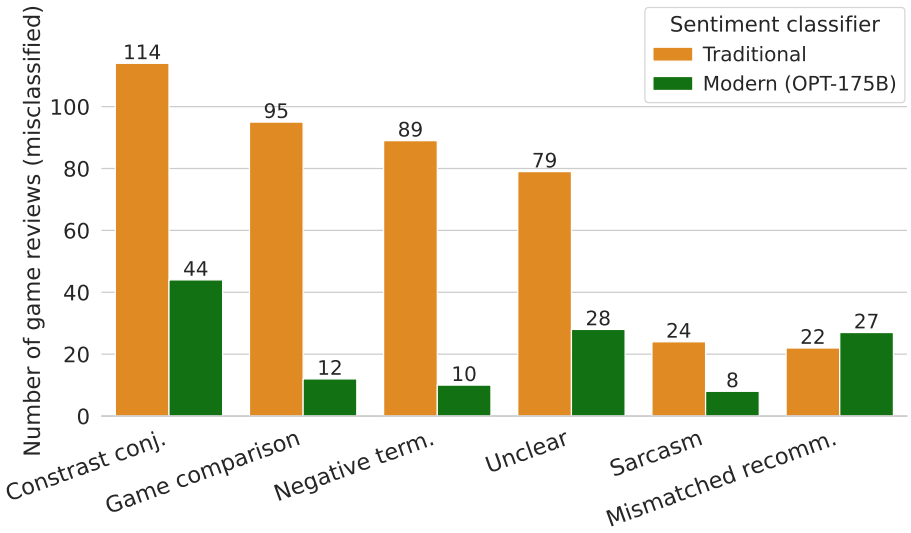



Figure 6.4: Comparison of the number of misclassified game reviews with traditional sentiment classifiers (SentiStrength, NLTK, and Stanford CoreNLP) and a modern classifier (OPT-175B).

addition, game reviews for which we could not clearly identify the misclassification reason (*unclear*) and for which there is a mismatch between the review and the recommendation (*mismatched recommendation*) are also common (with 28 and 27 reviews, respectively). Problems that significantly affect traditional classifiers, such as reviews with comparisons between different games (*game comparison*) and reviews with negative terminology (*negative terminology*) are mostly addressed by OPT-175B since only a few game reviews with those characteristics were misclassified by OPT-175B.

To exemplify how the OPT-175B model overcomes one of the issues that affect traditional classifiers, Table 6.2 shows an example of a game review that contains sarcasm. The review was wrongly classified by all the three traditional classifiers but correctly classified by OPT-175B.

Our second manual analysis of 100 game reviews correctly classified by all the traditional classifiers but wrongly classified by OPT-175B did not show any new reason that affects OPT-175B that we had not identified before. *Contrast conjunctions* is still the most common issue that affects OPT-175B in that analyzed sample, appearing in 52% of the misclassified reviews as shown in Figure 6.5. Furthermore, reviews with an

Table 6.2: Example of a game review with sarcasm wrongly classified by traditional classifiers but correctly classified by OPT-175B.

| **Game review** | **True sentiment about the game** | **Sentiment (trad. classifiers)** | **Sentiment (OPT-175B)** |
|---|---|---|---|
| "The game was really fun, it was crashing endlessly" | Negative | Positive | Negative |

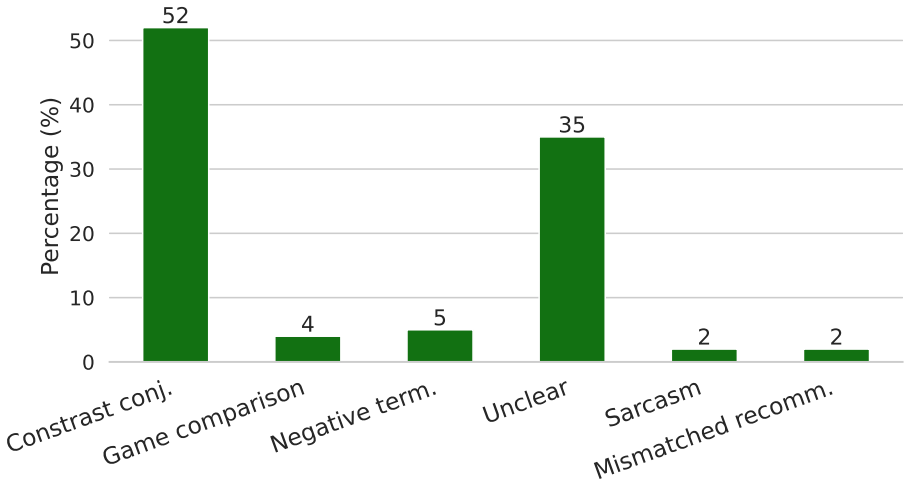



Figure 6.5: Percentage of root causes for sentiment misclassifications with OPT-175B.

*unclear* reason occurred in 35%, while *negative terminology*, *game comparison*, and *sarcasm* appeared only in 5%, 4%, and 2% of the sample, respectively. In a further analysis, we also observed that for reviews with *contrast conjunctions*, OPT-175B often (incorrectly) assigns a positive sentiment, which occurred in 71% of the reviews with *contrast conjunctions*. For comparison purposes, traditional sentiment classifiers (incorrectly) assign a positive sentiment to approximately 55% of game reviews with *contrast conjunctions*.

We found that the sentiment classification task with OPT-175B is affected by game reviews with contrast conjunctions. However, contrast conjunctions have a much smaller impact in the sentiment classification with OPT-175B compared to traditional sentiment classifiers. We also found that challenges with the sentiment analysis of game reviews, such as *game comparison* and *negative terminology*, have been mostly solved by OPT-175B.

## 6.7 Threats to Validity

A threat to the **external validity** is regarding the generalizability of our findings. Our findings are based on the evaluation of a single Large Language Model (OPT-175B) on game reviews from the Steam platform. Future studies should investigate other pre-trained language models and use other data. A threat to the **internal validity** is the time limit that we set to collect the sentiment classifications made by OPT-175B. OPT-175B is a very large model and computationally expensive, therefore, we set a limit of one week to execute it. Future work should execute OPT-175B for a longer time to obtain the sentiment classifications for a larger number of game reviews. Another internal threat is the prompt that we used to obtain the sentiment of a game review. Even though we performed an informal, minimal evaluation with other prompts and found that the used prompt achieved the best results, it is necessary to perform a more thorough evaluation using other prompts for sentiment classification.

## 6.8 Conclusion

In this chapter, we presented a research study on the performance of the OPT-175B Large Language Model on game reviews from the Steam platform. We also investigated the challenges that affect the performance of the sentiment classification task using that model. We showed that OPT-175B presents a high performance on the sentiment classification, with an F-measure of 0.93 (an increase of 72% compared to traditional sentiment classifiers) and an AUC of 0.91 (an increase of 30% compared to traditional sentiment classifiers). We also found that most issues that affect traditional classifiers, such as reviews with game comparisons and negative terminology, have been addressed by OPT-175B. This demonstrates that a model like OPT-175B can be used to successfully obtain the sentiment of players about games. In the next chapter, we conclude the thesis and discuss future research directions.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

Despite recent advancements in test automation, manual testing is still a prevalent practice in the gaming industry. And although manual testing is challenging and costly, mainly in a scenario of short game release cycles, prior research on how to improve and optimize manual game testing is scarce. In this thesis, we conducted five research studies to investigate how manual game testing can be improved and optimized from different perspectives. In the first research study (Chapter 2), we explored several unsupervised approaches to identify similar natural language test cases, which helps to reduce the redundancy in the test suite. In the second research study (Chapter 3), we proposed an automated framework to analyze test case descriptions and suggest improvements to QA engineers to reduce the manual testing effort of QA engineers by improving the quality of test case descriptions. In the third research study (Chapter 4), we investigated how we could optimize the execution of game test cases without source code by prioritizing test cases that cover game features which are used by players the most. This helps to avoid bugs that could affect a very large number of players. In the fourth research study (Chapter 5), we performed a large-scale study to investigate how traditional sentiment classifiers perform on game reviews and what are the issues with those classifiers. Finally, in the fifth research study (Chapter 6), we investigated how a state-of-the-art model (the OPT-175B Large Language Model)

performs on the sentiment classification of game reviews and how that model's performance compares to the performance of traditional sentiment classifiers. We also investigate if OPT-175B solves the issues that affect traditional classifiers. This study provides important insights to game developers and researchers about whether we can obtain players' sentiment about games and integrate that information in the game testing process. We briefly discuss the approach, main findings and contributions of our research studies below:

- In Chapter 2, we leveraged text embedding, text similarity, and clustering techniques to identify groups of similar test cases that are specified in natural language. Our findings show that we can identify similar test cases with a high performance (an F-score of 86.13%) using an ensemble of different embedding techniques (e.g., Word2Vec and Sentence-BERT), the cosine similarity metric, and K-means for clustering. A validation through an industrial case study showed that our approach has different uses cases in practice, such as to identify and remove redundant test cases.

- In Chapter 3, we leveraged statistical and neural language models to automatically analyze and suggest improvements to the description of test cases. In addition, we used a frequent itemset mining algorithm to suggest potentially missing test steps in a test cases. Our results show that we can successfully suggest improvements to test case descriptions (with an accuracy of up to 88%) and identify missing test steps with an average accuracy of 98%. Our automated framework can provide important and actionable recommendations that support the creation and maintenance of a high-quality, more consistent and more standardized test suite. In addition, our framework can be particularly useful for new employees who do not yet have much knowledge about the existing test suite.

- In Chapter 4, we proposed an approach to (1) identify the game features cov-

ered by test cases and (2) prioritize the execution of test cases based on the game features that they cover. We used an ensemble of zero-shot models to automatically identify the game features covered by test cases and the NSGA-II genetic algorithm for the multi-objective optimization of the test case ordering to maximize the coverage of highly-used features while minimizing the cumulative execution time. Our findings show that we can identify the covered game features with a high performance (an F-score of 76.1%). In addition, we could find test case orderings that cover highly-used game features early in the test execution with a practical test execution time. QA engineers can use our approach to prioritize test cases that cover game features which are more relevant to players.

- In Chapter 5, we performed a large-scale study to better understand how three traditional sentiment classifiers (SentiStrength, Stanford CoreNLP, and NLTK) perform on game reviews, the causes for the wrong classifications and the impact of each cause on the overall performance. We found that traditional sentiment classifiers do not perform well on game reviews, with a maximum AUC of 0.70 (obtained with NLTK). Furthermore, we found four main issues that affect traditional classifiers, such as reviews that compare the game under review with other games and reviews with negative terminology (such as the words "shoot" and "kill", which are commonly used in reviews of action games).

- In Chapter 6, due to major advancements in the NLP field since we performed the study discussed in Chapter 5, we investigated how a state-of-the-art technique, the OPT-175B Large Language Model, performs on the sentiment classification of game reviews. We also investigated the challenges that might affect the performance of OPT-175B. We found that OPT-175B achieves a (far) better performance compared to traditional sentiment classifiers. For instance, OPT-175B achieved a median F-measure of 0.93 (which is 72% higher than

the median F-measure of the traditional classifiers) and a median AUC of 0.91 (which is 30% higher than the median AUC of the traditional classifiers). Furthermore, most of the challenges faced by traditional sentiment classifiers have been solved by OPT-175B.

## 7.2 Future Work

Although we have thoroughly investigated several ways of improving manual game testing in this thesis and proposed effective and practical approaches, the research presented here can be extended in different ways. Below we present possible future research directions:

- **Combining clustering with topic analysis techniques to improve the discovery of redundant test cases in natural language.** In Chapter 2, we investigated unsupervised approaches to identify similar test cases in natural language in two stages: test step clustering and test case similarity detection. Future studies should investigate whether combining clustering with other techniques, such as topic analysis, can improve the quality of the clusters of test steps in the first stage. For instance, (1) a topic analysis technique could be applied to test steps to identify steps that belong to the same topic and then (2) a clustering technique would be applied to identify similar test steps within the same topic. This might improve the quality of the obtained clusters of test steps and, consequently, improve the test case similarity detection approach.

- **Exploring other techniques to improve the description and quality of natural language test cases for game testing.** In Chapter 3, we proposed an automated framework to automatically analyze and provide recommendations to improve natural language test cases. Future research should investigate other NLP techniques that can help to further improve test cases. For instance, word sense disambiguation techniques can help to make test case descriptions

less ambiguous, which makes the testing process more effective and efficient. In addition, graph-based techniques can improve the discovering of test step occurrence patterns across test cases and be used to recommend potentially missing steps in a new test case.

- **Evaluating few-shot techniques and fine-tuned models to identify game features covered by test cases.** One major step of our proposed prioritization approach in Chapter 4 consists of automatically identifying the game features that are covered by test cases. Our solution, which is a zero-shot technique, does not require labeled data or training. However, in case there are a few labeled examples for each label, few-shot techniques might work better than zero-shot. In addition, if a large amount of labeled examples is available, fine-tuning a pre-trained model might also improve the classification performance. We encourage future works to investigate how few-shot and fine-tuned models perform on the identification of game features covered by test cases.

- **Evaluating other techniques to prioritize natural language test cases.** In Chapter 4, we adopted the widely-used NSGA-II genetic algorithm to perform the multi-objective optimization of test case ordering. Future works should explore other genetic algorithms and other approaches for the test case prioritization, such as Reinforcement Learning, which has shown to be effective to prioritize test cases.

- **Integrating players' sentiment about game features in the game testing process.** In Chapter 6, we showed that a state-of-the-art model (OPT-175B) performs very well in the sentiment classification of game reviews and that it overcomes most issues that affect traditional classifiers that were widely-used in the past. Future research should investigate whether OPT-175B or other Large Language Models can be used for aspect-based sentiment analysis to capture players' sentiment about specific game features of interest. This

information can be used in the game testing process in a few ways, such as (1) to prioritize test cases that cover features that players like the most (e.g., to avoid harming the player experience or satisfaction with the game) or (2) to investigate potentially existing bugs on game features that players dislike the most to improve the overall players' experience and satisfaction with the game.

# Bibliography


[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, IEEE, 2007, pp. 89–98.

[3] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. J. Passonneau, "Sentiment analysis of Twitter data," in *Proceedings of the Workshop on Language in Social Media (LSM 2011)*, pp. 30–38.

[4] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *ACM sigmod record*, ACM, vol. 22, 1993, pp. 207–216.

[5] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo, "Fast discovery of association rules," *Advances in knowledge discovery and data mining*, vol. 12, no. 1, pp. 307–328, 1996.

[6] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *IEEE sixth international conference on software testing, verification and validation*, IEEE, 2013, pp. 312–321.

[7] M. Araújo, P. Gonçalves, M. Cha, and F. Benevenuto, "Ifeel: A system that compares and combines sentiment analysis methods," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, pp. 75–78.

[8] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[9] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*, Springer, 2011, pp. 33–47.

[10] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[11] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, 2021.

[12] D. Bamman and N. A. Smith, "Contextualized sarcasm detection on Twitter," in *Ninth International AAAI Conference on Web and Social Media*, 2015.

[13] B. Bazelli, A. Hindle, and E. Stroulia, "On the personality traits of StackOverflow users," in *2013 IEEE International conference on software maintenance*, pp. 460–463.

[14] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *The Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.

[15] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE'07)*, IEEE, 2007, pp. 85–103.

[16] S. K. Bharti, K. S. Babu, and S. K. Jena, "Parsing-based sarcasm sentiment recognition in Twitter data," in *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 1373–1380.

[17] BigScience, *Bigscience language open-science open-access multilingual (bloom) language model*, International, 2022.

[18] G. Biondi, V. Franzoni, and V. Poggioni, "A deep learning semantic approach to emotion recognition using the IBM Watson bluemix alchemy language," in *International Conference on Computational Science and Its Applications*, Springer, 2017, pp. 718–729.

[19] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: Analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[20] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.

[21] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[22] M. Bouazizi and T. Ohtsuki, "Opinion mining in Twitter: How to make use of sarcasm to enhance sentiment analysis," in *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 1594–1597.

[23] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean, "Large language models in machine translation," 2007.

[24] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[26] C. Buck, K. Heafield, and B. Van Ooyen, "N-gram counts and language models from the common crawl.," in *LREC*, vol. 2, 2014, p. 4.

[27] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering*, 2016, pp. 975–980.

[28] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1352–1382, 2018.

[29] F. Calefato, F. Lanubile, and N. Novielli, "Emotxt: A toolkit for emotion recognition from text," in *In 2017 7th International Conference on Affective Computing and Intelligent Interaction Workshops and Demos)*, IEEE, pp. 79–80.

[30] L. V. G. Carreño and K. Winbladh, "Analysis of user comments: An approach for software requirements evolution," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, pp. 582–591.

[31] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Céspedes, S. Yuan, C. Tar, Y.-H. Sung, B. Strope, and R. Kurzweil, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018.

[32] C. Chambers, W.-c. Feng, S. Sahu, and D. Saha, "Measurement-based characterization of a collection of on-line games," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, 2005, pp. 1–1.

[33] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Special issue on learning from imbalanced data sets," *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 1–6, 2004.

[34] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2011, pp. 1–10.

[35] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*, PMLR, 2020, pp. 1597–1607.

[36] N. Chetouane, F. Wotawa, H. Felbinger, and M. Nica, "On using k-means clustering for test suite reduction," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2020, pp. 380–385.

[37] C. Chiu, R.-J. Sung, Y.-R. Chen, and C.-H. Hsiao, "App review analytics of free games listed on Google play," in *Proceedings of the 13th International Conference on Electronic Business, Singapore*, 2013.

[38] T. Y. Chong, R. E. Banchs, and E. S. Chng, "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 and Hybrid Approaches to Machine Translation*, 2012, pp. 30–37.

[39] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[40] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[41] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[42] N. Cui, Y. Jiang, X. Gu, and B. Shen, "Zero-shot program representation learning," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 60–70. DOI: 10.1145/3524610.3527888.

[43] K. Deb, "Multi-objective optimisation using evolutionary algorithms: An introduction," in *Multi-objective evolutionary optimisation for product design and manufacturing*, Springer, 2011, pp. 3–34.

[44] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[45] P. Devine and K. Blincoe, "Unsupervised extreme multi label classification of stack overflow posts," 2022.

[46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[47] T. G. Dietterich, "Ensemble methods in machine learning," in *International workshop on multiple classifier systems*, Springer, 2000, pp. 1–15.

[48] N. DiGiuseppe and J. A. Jones, "Semantic fault diagnosis: Automatic natural-language fault descriptions," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–4.

[49] L. Dong, F. Wei, M. Zhou, and K. Xu, "Adaptive multi-compositionality for recursive neural models with applications to sentiment analysis," in *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[50] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification and scene analysis*. Wiley New York, 1973, vol. 3.

[51] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. CRC press, 1994.

[52] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 234–245.

[53] I. Etikan, S. A. Musa, and R. S. Alkassim, "Comparison of convenience sampling and purposive sampling," *American journal of theoretical and applied statistics*, vol. 5, no. 1, pp. 1–4, 2016.

[54] T. Finin, W. Murnane, A. Karandikar, N. Keller, J. Martineau, and M. Dredze, "Annotating named entities in Twitter data with crowdsourcing," in *Proc. of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, pp. 80–88.

[55] Z. P. Fry and W. Weimer, "Fault localization using textual similarities," *arXiv preprint arXiv:1211.2858*, 2012.

[56] *Gaming market size, share covid-19 impact analysis, by game type (shooter, action, sports, role-playing, and others), by device type (pc/mmo, tablet, mobile phone, and tv/console), by end-user (male and female), and regional forecast, 2021-2028*, 2021 (last visited: November 4, 2022). [Online]. Available: https://www.fortunebusinessinsights.com/gaming-market-105730.

[57] T. Gao, A. Fisch, and D. Chen, "Making pre-trained language models better few-shot learners," *arXiv preprint arXiv:2012.15723*, 2020.

[58] V. Garousi, S. Bauer, and M. Felderer, "NLP-assisted software testing: A systematic mapping of the literature," *Information and Software Technology*, vol. 126, p. 106 321, 2020.

[59] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[60] W. H. Gomaa and A. A. Fahmy, "A survey of text similarity approaches," *International Journal of Computer Applications*, vol. 68, no. 13, pp. 13–18, 2013.

[61] R. González-Ibánez, S. Muresan, and N. Wacholder, "Identifying sarcasm in Twitter: A closer look," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Short Papers-Volume 2*, Association for Computational Linguistics, 2011, pp. 581–586.

[62] M. Goul, O. Marjanovic, S. Baxley, and K. Vizecky, "Managing the enterprise business intelligence app store: Sentiment analysis supported requirements engineering," in *2012 45th Hawaii International Conference on System Sciences*, IEEE, pp. 4168–4177.

[63] J. M. Guarte and E. B. Barrios, "Estimation under purposive sampling," *Communications in Statistics-Simulation and Computation*, vol. 35, no. 2, pp. 277–284, 2006.

[64] J. L. Guerrero, J. García, L. Martí, J. M. Molina, and A. Berlanga, "A stopping criterion based on kalman estimation techniques with several progress indicators," in *Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*, 2009, pp. 587–594.

[65] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," *arXiv preprint arXiv:2004.10964*, 2020.

[66] E. Guzman, O. Aly, and B. Bruegge, "Retrieving diverse opinions from app reviews," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–10.

[67] E. Guzman, D. Azócar, and Y. Li, "Sentiment analysis of commit comments in GitHub: An empirical study," in *Proc. of the 11th Working Conference on Mining Software Repositories*, ACM, 2014, pp. 352–355.

[68] E. Guzman and W. Maalej, "How do users like this feature? A fine grained sentiment analysis of app reviews," in *2014 IEEE 22nd International requirements engineering conference (RE)*, pp. 153–162.

[69] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, "How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1281–1291.

[70] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM sigmod record*, ACM, vol. 29, 2000, pp. 1–12.

[71] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data mining and knowledge discovery*, vol. 8, no. 1, pp. 53–87, 2004.

[72] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.

[73] H. Hemmati, Z. Fang, and M. V. Mantyla, "Prioritizing manual test cases in traditional and rapid release environments," in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[74] H. Hemmati, Z. Fang, M. V. Mäntylä, and B. Adams, "Prioritizing manual test cases in rapid release environments," *Software Testing, Verification and Reliability*, vol. 27, no. 6, e1609, 2017.

[75] H. Hemmati and F. Sharifi, "Investigating nlp-based approaches for predicting manual test case failure," in *Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 309–319.

[76] M. R. Hess and J. D. Kromrey, "Robust confidence intervals for effect sizes: A comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances," in *annual meeting of the American Educational Research Association*, Citeseer, vol. 1, 2004.

[77] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[78] J. Howarth, *How many gamers are there? (new 2022 statistics)*, 2022 (last visited: November 8, 2022). [Online]. Available: https://www.fortunebusinessinsights.com/gaming-market-105730.

[79] A. Huang, "Similarity measures for text document clustering," in *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, vol. 4, 2008, pp. 9–56.

[80] J. Huang and C. X. Ling, "Using AUC and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[81] R. Huang, D. Towey, Y. Xu, Y. Zhou, and N. Yang, "Dissimilarity-based test case prioritization through data fusion," *Software: Practice and Experience*, vol. 52, no. 6, pp. 1352–1377, 2022.

[82] M. R. Islam and M. F. Zibran, "A comparison of software engineering domain specific sentiment analysis tools," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 487–491.

[83] M. R. Islam and M. F. Zibran, "Leveraging automated sentiment analysis in software engineering," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 203–214.

[84] M. R. Islam and M. F. Zibran, "Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text," *Journal of Systems and Software*, vol. 145, pp. 125–146, 2018.

[85] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 422–446, 2002.

[86] T. Joachims, "A probabilistic analysis of the rocchio algorithm with TF-IDF for text categorization.," Carnegie-mellon univ, Pittsburgh, PA - Dept of computer science, Tech. Rep., 1996.

[87] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, 1972.

[88] R. Jongeling, S. Datta, and A. Serebrenik, "Choosing your weapons: On sentiment analysis tools for software engineering research," in *2015 IEEE International Conference on Software Maintenance and Evolution*, pp. 531–535.

[89] R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik, "On negative results when using sentiment analysis tools for software engineering research," *Empirical Software Engineering*, vol. 22, no. 5, pp. 2543–2584, 2017.

[90] D. Jurafsky and J. H. Martin, "Speech and language processing," 2009.

[91] J. D. Knowles and D. W. Corne, "Approximating the nondominated front using the pareto archived evolution strategy," *Evolutionary computation*, vol. 8, no. 2, pp. 149–172, 2000.

[92] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *arXiv preprint arXiv:2205.11916*, 2022.

[93] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, PMLR, 2015, pp. 957–966.

[94] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *In Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2016, pp. 361–368.

[95] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[96] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning (ICML)*, PMLR, 2014, pp. 1188–1196.

[97] D. Lee, D. Lin, C.-P. Bezemer, and A. E. Hassan, "Building the perfect game–an empirical study of game modifications," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2485–2518, 2020.

[98] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[99] B. Li and L. Han, "Distance weighted cosine similarity measure for text classification," in *International conference on intelligent data engineering and automated learning*, Springer, 2013, pp. 611–618.

[100] K. Li, Z. Liu, T. He, H. Huang, F. Peng, D. Povey, and S. Khudanpur, "An empirical study of transformer-based neural language model adaptation," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 7934–7938.

[101] L. Li, Z. Li, W. Zhang, J. Zhou, P. Wang, J. Wu, G. He, X. Zeng, Y. Deng, and T. Xie, "Clustering test steps in natural language toward automating test automation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1285–1295.

[102] Z. Li, Y. Bian, R. Zhao, and J. Cheng, "A fine-grained parallel multi-objective test case prioritization on GPU," in *International Symposium on Search Based Software Engineering*, Springer, 2013, pp. 111–125.

[103] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?" In *2018 IEEE/ACM 40th International Conference on Software Engineering*, pp. 94–104.

[104] C. Lin and Y. He, "Joint sentiment/topic model for sentiment analysis," in *Proceedings of the 18th ACM conference on Information and knowledge management*, 2009, pp. 375–384.

[105] D. Lin, C.-P. Bezemer, and A. E. Hassan, "An empirical study of early access games on the steam platform," *Empirical Software Engineering*, vol. 23, no. 2, pp. 771–799, 2018.

[106] D. Lin, C.-P. Bezemer, Y. Zou, and A. E. Hassan, "An empirical study of game reviews on the Steam platform," *Empirical Software Engineering*, vol. 24, no. 1, pp. 170–207, 2019, ISSN: 1573-7616.

[107] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE)*, 2020, pp. 473–485.

[108] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *arXiv preprint arXiv:2107.13586*, 2021.

[109] J. D. Long, D. Feng, and N. Cliff, "Ordinal analysis of behavioral data.," 2003.

[110] J.-F. Lu, J. Tang, Z.-M. Tang, and J.-Y. Yang, "Hierarchical initialization approach for k-means clustering," *Pattern Recognition Letters*, vol. 29, no. 6, pp. 787–795, 2008.

[111] F. Macklon, M. R. Taesiri, M. Viggiato, S. Antoszko, N. Romanova, D. Paas, and C.-P. Bezemer, "Automatically detecting visual bugs in HTML5 canvas games," in *2022 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[112] M. Mahdieh, S.-H. Mirian-Hosseinabadi, and M. Mahdieh, "Test case prioritization using test case diversification and fault-proneness estimations," *Automated Software Engineering*, vol. 29, no. 2, pp. 1–43, 2022.

[113] X. P. Mai, F. Pastore, A. Göknil, and L. Briand, "A natural language programming approach for requirements-based security testing," in *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2018.

[114] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[115] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proc. of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[116] A. Marchetto, M. M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 918–940, 2015.

[117] M. Marcińczuk, M. Gniewkowski, T. Walkowiak, and M. Bedkowski, "Text document clustering: Wordnet vs. TF-IDF vs. Word embeddings," in *Proceedings of the 11th Global Wordnet Conference*, 2021, pp. 207–214.

[118] L. Martí, J. García, A. Berlanga, and J. M. Molina, "An approach to stopping criteria for multi-objective optimization evolutionary algorithms: The mgbm criterion," in *2009 IEEE congress on evolutionary computation*, IEEE, 2009, pp. 1263–1270.

[119] S. Masuda, F. Iwama, N. Hosokawa, T. Matsuodani, and K. Tsuda, "Semantic analysis technique of logics retrieval for software testing from specification documents," in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2015, pp. 1–6.

[120] S. Masuda, T. Matsuodani, and K. Tsuda, "Automatic generation of UTP models from requirements in natural language," in *Proceedings of the 9th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2016, pp. 1–6.

[121] P. D. McNicholas, T. B. Murphy, and M O'Regan, "Standardising the lift of an association rule," *Computational Statistics & Data Analysis*, vol. 52, no. 10, pp. 4712–4721, 2008.

[122] F. McSherry and M. Najork, "Computing information retrieval performance measures efficiently in the presence of tied scores," in *European conference on information retrieval*, Springer, 2008, pp. 414–421.

[123] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[124] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*,

[125] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 222–232.

[126] P. Mishra, R. Rajnish, and P. Kumar, "Sentiment analysis of Twitter data: Case study on digital India," in *2016 International Conference on Information Technology*, IEEE, pp. 148–153.

[127] T. K. Moon, "The expectation-maximization algorithm," *IEEE Signal processing magazine*, vol. 13, no. 6, pp. 47–60,

[128] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model," in *International Workshop on Artificial Intelligence and Statistics*, 2005, pp. 246–252.

[129] A. Mudinas, D. Zhang, and M. Levene, "Combining lexicon and learning based approaches for concept-level sentiment analysis," in *Proceedings of the 1st International workshop on issues of sentiment discovery and opinion mining*, 2012, pp. 1–8.

[130] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 1–11.

[131] M. P. Naik, H. B. Prajapati, and V. K. Dabhi, "A survey on semantic document clustering," in *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, IEEE, 2015, pp. 1–10.

[132] P. Nand, R. Perera, and R. Lal, "A HMM POS tagger for micro-blogging type texts," in *Pacific Rim International Conference on Artificial Intelligence*, 2014, pp. 157–169.

[133] S. Nguyen, T. Nguyen, Y. Li, and S. Wang, "Combining program analysis and statistical language model for code statement completion," in *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, 2019, pp. 710–721.

[134] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2015, pp. 58–68.

[135] N. Novielli, F. Calefato, and F. Lanubile, "A gold standard for emotion annotation in Stack Overflow," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 14–17.

[136] S. Omri and C. Sinz, "Learning to rank for test case prioritization," in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*, IEEE, 2022, pp. 16–24.

[137] M. Ortu, B. Adams, G. Destefanis, P. Tourani, M. Marchesi, and R. Tonelli, "Are bullies more productive?: Empirical study of affectiveness vs. issue fixing time," in *Proc. of the 12th Working Conference on Mining Software Repositories*, IEEE Press, 2015, pp. 303–313.

[138] B. Pang and L. Lee, "Opinion mining and sentiment analysis," *Foundations and Trends in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008.

[139] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can I improve my app? Classifying user reviews for software maintenance and evolution," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 281–290.

[140] L. Pascarella, F. Palomba, M. Di Penta, and A. Bacchelli, "How is video game development different from software development in open source?" In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, IEEE, 2018, pp. 392–402.

[141] H. Pei, B. Yin, M. Xie, and K.-Y. Cai, "Dynamic random testing with test case clustering and distance-based parameter adjustment," *Information and Software Technology*, vol. 131, p. 106 470, 2021.

[142] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 324–336.

[143] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[144] D. Pletea, B. Vasilescu, and A. Serebrenik, "Security and emotion: Sentiment analysis of security discussions on GitHub," in *Proc. of the 11th working conference on mining software repositories*, 2014, pp. 348–351.

[145] C. Politowski, F. Petrillo, and Y.-G. Guéhéneuc, "A survey of video game testing," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, IEEE, 2021, pp. 90–99.

[146] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "CBGA-ES+: A cluster-based genetic algorithm with non-dominated elitist selection for supporting multi-objective test optimization," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 86–107, 2018.

[147] J. A. do Prado Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Transactions on Software Engineering*, 2020.

[148] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," in *International Conference on Machine Learning*, PMLR, 2021, pp. 8748–8763.

[149] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[150] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 81–90.

[151] K. Raison, N. Tomuro, S. Lytinen, and J. P. Zagal, "Extraction of user opinions by adjective-context co-clustering for game review texts," in *International Conference on NLP*, Springer, 2012, pp. 289–299.

[152] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[153] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Nov. 2019. [Online]. Available: https://arxiv.org/abs/1908.10084.

[154] P. C. Rigby and A. E. Hassan, "What can OSS mailing lists tell us? A preliminary psychometric text analysis of the Apache developer mailing list," in *Fourth International Workshop on Mining Software Repositories*, IEEE, 2007, pp. 23–23.

[155] L. Rokach and O. Maimon, "Clustering methods," in *Data mining and knowledge discovery handbook*, Springer, 2005, pp. 321–352.

[156] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "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*, 2006, pp. 1–51.

[157] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[158] S. Ruder, M. E. Peters, S. Swayamdipta, and T. Wolf, "Transfer learning in natural language processing," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: Tutorials*, 2019, pp. 15–18.

[159] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 268–279.

[160] H. Saif, Y. He, and H. Alani, "Semantic sentiment analysis of Twitter," in *International semantic web conference*, Springer, 2012, pp. 508–524.

[161] G. Salton, "Developments in automatic text retrieval," *science*, vol. 253, no. 5023, pp. 974–980, 1991.

[162] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[163] T. Santos, F. Lemmerich, M. Strohmaier, and D. Helic, "What's in a review: Discrepancies between expert and amateur reviews of video games on Metacritic," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, pp. 1–22, 2019.

[164] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: A case study in software product lines," in *2013 35Th international conference on software engineering (ICSE)*, IEEE, 2013, pp. 492–501.

[165] T. Schick and H. Schütze, "It's not just size that matters: Small language models are also few-shot learners," *arXiv preprint arXiv:2009.07118*, 2020.

[166] S. Y. Shin, S. Nejati, M. Sabetzadeh, L. C. Briand, and F. Zimmer, "Test case prioritization for acceptance testing of cyber physical systems: A multi-objective search-based approach," in *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis*, 2018, pp. 49–60.

[167] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 309–320.

[168] V. K. Singh, R. Piryani, A. Uddin, and P. Waila, "Sentiment analysis of movie reviews: A new feature-based heuristic for aspect-level sentiment classification," in *2013 International Mutli-Conference on Automation, Computing, Communication, Control and Compressed Sensing)*, IEEE, pp. 712–717.

[169] P. H. Sneath and R. R. Sokal, *Numerical taxonomy. The principles and practice of numerical classification.* 1973.

[170] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proc. of the 2013 conference on empirical methods in natural language processing*, pp. 1631–1642.

[171] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 12–22.

[172] B. Stråååt and H. Verhagen, "Using user created game reviews for sentiment analysis: A method for researching user attitudes.," in *GHITALY@ CHItaly*, 2017.

[173] M. R. Taesiri, F. Macklon, and C.-P. Bezemer, "CLIP meets GamePhysics: Towards bug identification in gameplay videos using zero-shot transfer learning," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, IEEE, 2022, pp. 46–57.

[174] M. R. Taesiri, F. Macklon, Y. Wang, H. Shen, and C.-P. Bezemer, "Large language models are pretty good zero-shot video game bug detectors," *arXiv preprint arXiv:2210.02506*, 2022.

[175] S. H. Tan and Z. Li, "Collaborative bug finding for android apps," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1335–1347.

[176] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas, "Sentiment strength detection in short informal text," *Journal of the American Society for Information Science and Technology*, pp. 2544–2558, 2010.

[177] J. J. Thompson, B. H. Leung, M. R. Blair, and M. Taboada, "Sentiment analysis of player chat messaging in the video game StarCraft 2: Extending a lexicon-based model," *Knowledge-Based Systems*, vol. 137, pp. 149–162, 2017.

[178] M. D. C. Tongco, "Purposive sampling as a tool for informant selection," *Ethnobotany Research and applications*, vol. 5, pp. 147–158, 2007.

[179] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: On the importance of pre-training compact models," *arXiv preprint arXiv:1908.08962*, 2019.

[180] S. Varvaressos, K. Lavoie, S. Gaboury, and S. Hallé, "Automated bug finding in video games: A case study for runtime monitoring," *Computers in Entertainment (CIE)*, vol. 15, no. 1, pp. 1–28, 2017.

[181] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[182] M. Viggiato, D. Lin, A. Hindle, and C.-P. Bezemer, "What causes wrong sentiment classifications of game reviews?" *IEEE Transactions on Games*, vol. 14, no. 3, pp. 350–363, 2022. DOI: 10.1109/TG.2021.3072545.

[183] M. Viggiato, D. Paas, C. Buzon, and C.-P. Bezemer, "Identifying similar test cases that are specified in natural language," *IEEE Transactions on Software Engineering*, 2022.

[184] M. Viggiato, D. Paas, C. Buzon, and C.-P. Bezemer, "Using natural language processing techniques to improve manual test case descriptions," in *International Conference on Software Engineering-Software Engineering in Practice (ICSE-SEIP) Track.(May 8, 2022)*, 2022.

[185] T. Wagner, H. Trautmann, and L. Martí, "A taxonomy of online stopping criteria for multi-objective evolutionary algorithms," in *International Conference on Evolutionary Multi-Criterion Optimization*, Springer, 2011, pp. 16–30.

[186] B. Walter, M. Schilling, M. Piechotta, and S. Rudolph, "Improving test execution efficiency through clustering and reordering of independent test steps," in *IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2018, pp. 363–373.

[187] X. Wan, "Using bilingual knowledge and ensemble techniques for unsupervised chinese sentiment analysis," in *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, 2008, pp. 553–561.

[188] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[189] C. Wang, F. Pastore, A. Goknil, and L. Briand, "Automatic generation of acceptance test cases from use case specifications: An nlp-based approach," *IEEE Transactions on Software Engineering*, 2020.

[190] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 385–396.

[191] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 182–191.

[192] Y. Wang, L. Wang, Y. Li, D. He, W. Chen, and T.-Y. Liu, "A theoretical analysis of ndcg ranking measures," in *Proceedings of the 26th annual conference on learning theory (COLT 2013)*, vol. 8, 2013, p. 6.

[193] S. M. Weiss, N. Indurkhya, and T. Zhang, *Fundamentals of predictive text mining*. Springer, 2015.

[194] S. M. Weiss, N. Indurkhya, T. Zhang, and F. Damerau, *Text mining: predictive methods for analyzing unstructured information*. Springer Science & Business Media, 2010.

[195] S. Wijayanto and M. L. Khodra, "Business intelligence according to aspect-based sentiment analysis using double propagation," in *2018 3rd International Conference on Information Technology, Information System and Electrical Engineering*, IEEE, pp. 105–109.

[196] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Impediments for software test automation: A systematic literature review," *Software Testing, Verification and Reliability*, vol. 27, no. 8, e1639, 2017.

[197] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*, Springer, 1992, pp. 196–202.

[198] W. Witkowski, *Videogames are a bigger industry than movies and north american sports combined, thanks to the pandemic*, 2021 (last visited: November 8, 2022). [Online]. Available: https://www.marketwatch.com/story/videogames-are-a-bigger-industry-than-sports-and-movies-combined-thanks-to-the-pandemic-11608654990.

[199] Y. Woldemariam, "Sentiment analysis in a cross-media analysis framework," in *2016 IEEE International Conference on Big Data Analysis (ICBDA)*, pp. 1–5.

[200] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, *Transformers: State-of-the-Art Natural Language Processing*, Oct. 2020. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[201] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[202] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[203] Y. Xian, Z. Akata, G. Sharma, Q. Nguyen, M. Hein, and B. Schiele, "Latent embeddings for zero-shot classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 69–77.

[204] Z. Xiao, L Wang, and J. Du, "Improving the performance of sentiment classification on imbalanced datasets with transfer learning," *IEEE Access*, vol. 7, pp. 28 281–28 290, 2019.

[205] H. Xie and T. Virtanen, "Zero-shot audio classification via semantic embeddings," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 29, pp. 1233–1242, 2021.

[206] Y. Yang, X. Huang, X. Hao, Z. Liu, and Z. Chen, "An industrial study of natural language processing based test case prioritization," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2017, pp. 548–549.

[207] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, 2022.

[208] K. Yauris and M. L. Khodra, "Aspect-based summarization for game review using double propagation," in *2017 International Conference on Advanced Informatics, Concepts, Theory, and Applications*, IEEE, pp. 1–6.

[209] W. Yin, J. Hay, and D. Roth, "Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach," *arXiv preprint arXiv:1909.00161*, 2019.

[210] T. Yue, S. Ali, and M. Zhang, "Rtcm: A natural language based, automated, and practical test case generation framework," in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 397–408.

[211] J. P. Zagal, A. Ladd, and T. Johnson, "Characterizing and understanding game reviews," in *Proceedings of the 4th international Conference on Foundations of Digital Games*, 2009, pp. 215–222.

[212] J. P. Zagal and N. Tomuro, "Cultural differences in game appreciation: A study of player game reviews.," in *FDG*, 2013, pp. 86–93.

[213] J. P. Zagal, N. Tomuro, and A. Shepitsen, "Natural language processing in game studies research: An overview," *Simulation & Gaming*, pp. 356–373, 2012.

[214] M. Zalmanovici, O. Raz, and R. Tzoref-Brill, "Cluster-based test suite functional analysis," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 962–967.

[215] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[216] Y. Zhang, J. Lu, F. Liu, Q. Liu, A. Porter, H. Chen, and G. Zhang, "Does deep learning help topic extraction? A kernel K-means clustering method with word embedding," *Journal of Informetrics*, vol. 12, no. 4, pp. 1099–1117, 2018.

[217] Z. Q. Zhou, C. Liu, T. Y. Chen, T. Tse, and W. Susilo, "Beating random test case prioritization," *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 654–675, 2020.