applied sciences

MDPI

Article

Automated Classification of Unstructured Bilingual Software Bug Reports: An Industrial Case Study Research

Ömer Köksal [1,*] and Bedir Tekinerdogan [2,*]

[1] ASELSAN Research Center, 06200 Ankara, Turkey
[2] Information Technology Group, Wageningen University and Research, 6706 KN Wageningen, The Netherlands
* Correspondence: koksal@aselsan.com.tr (Ö.K.); bedir.tekinerdogan@wur.nl (B.T.)

Abstract: Software bug report classification is a critical process to understand the nature, implications, and causes of software failures. Furthermore, classification enables a fast and appropriate reaction to software bugs. However, for large-scale projects, one must deal with a broad set of bugs from multiple types. In this context, manually classifying bugs becomes cumbersome and time-consuming. Although several studies have addressed automated bug classification using machine learning techniques, they have mainly focused on academic case studies, open-source software, and unilingual text input. This paper presents our automated bug classification approach applied and validated in an industrial case study. In contrast to earlier studies, our study is applied to a commercial software system based on unstructured bilingual bug reports written in English and Turkish. The presented approach adopts and integrates machine learning (ML), text mining, and natural language processing (NLP) techniques to support the classification of software bugs. The approach has been applied within an industrial case study. Compared to manual classification, our results show that bug classification can be automated and even performs better than manual bug classification. Our study shows that the presented approach and the corresponding tools effectively reduce the manual classification time and effort.

Keywords: software bug classification; text categorization; text mining; machine learning; natural language processing

Citation: Köksal, Ö.; Tekinerdogan, B. Automated Classification of Unstructured Bilingual Software Bug Reports: An Industrial Case Study Research. *Appl. Sci.* **2022**, *12*, 338. https://doi.org/10.3390/app12010338

Academic Editors: Bruno Baruque Zanón, Jose Luis Calvo-Rolle, Santiago Porras Alfonso and Petr Dolezel

Received: 14 November 2021
Accepted: 23 December 2021
Published: 30 December 2021

1. Introduction

Due to software systems' increased complexity and size, software failures are inevitable in software development projects. A large part of a software development project is therefore often dedicated to software verification and validation. Detecting the faults, diagnosing the cause of these faults, and resolving these faults is crucial to ensure the functional and quality requirements for software projects. Fault detections are typically described in bug reports, including textual descriptions of the problem and the steps that led to a failure [1]. The validity and performance of the verification tasks depend heavily on the quality of bug reports. Hence the accuracy and completeness of these bug reports are essential.

Bug reports are generally handled and tracked with the help of bug tracking software to manage maintenance activities and keep the bugs' details [2]. After detecting the bugs, the classification of software bugs is needed to understand the root causes and provide the proper steps for correcting the bugs. The primary benefit of bug classification is to reduce the bug fixing time by reducing the bug assignment time to the corresponding developer who can fix the bug in the least amount of time [3].

However, one must soon deal with a broad set of bugs from multiple types for large-scale projects. Manually classifying bugs then becomes quickly cumbersome, error-prone, and time-consuming. For example, Pingclasai et al. [4] pointed out that, in their work,

researchers spent 90 days manually classifying more than 7000 bug reports. Similarly, Jeong et al. [5] reported that, in big software projects, assigning a bug to the first developer who will fix the bug might take up to 180 days, and to the second developer might take an additional 250 days if the first developer cannot fix the bug. Moreover, bug reports are not always well-structured and are very often unstructured. Structured reports adopt a well-defined template for bug classification, whereas unstructured reports use free text, which can dramatically impede bug classification.

Several studies have focused on adopting the automation of bug classification activities to overcome the bug classification problem, often using machine learning approaches. However, these studies have focused mainly on academic case studies, open-source software, and unilingual text input. This paper presents the results and lessons learned of our developed bug classification approach and the corresponding tools, applied and validated in an industrial case study research. Furthermore, in contrast to existing studies, our study is applied to a commercial software system. Finally, bug classification is based on unstructured bilingual bug reports written in English and Turkish, derived from black-box testing results. The presented approach adopts and integrates machine learning (ML), text mining, and natural language processing (NLP) techniques to support the classification of software bugs. Our study shows that the presented approach and the corresponding tools effectively reduce the manual classification time and effort.

Furthermore, we have used commercial bug reports in this paper. However, Gomez et al. [6] stated that almost all studies in the literature use bug repositories of open source systems. Therefore, in this paper, we attempt to extend bug classification issues into commercial bug reports.

The contributions of this study are the following. First of all, we provide bug classification for a real industrial project. Since earlier studies mainly focused on open source projects, this study can provide additional insight into the topic. Secondly, the approach focuses on the usage of bilingual unstructured bug reports. Bug classification has been broadly addressed in the literature, but bilingual unstructured bug report classification still requires more in-depth research. Therefore, the current study adds to the existing knowledge. Finally, we provide an empirical case study research-based approach for evaluating automated bug classification. On the one hand, the study confirms the earlier conclusions regarding bug classification. However, on the other hand, it provides additional insight into bug classification for the particular situation of the case study, that is, automated bug classification based on unstructured, bilingual black-box bug reports.

The remainder of the paper is organized as follows. Section 2 provides the background for this study. Section 3 presents the bug classification process and the adopted bug classification schema. Section 4 describes the research questions and the research methodology. Section 5 presents the adopted case study. Section 6 describes the results of manual classification and the evaluation. Section 7 provides the discussion. Section 8 describes the related work on papers performing bug classification, emphasizing research trying to automate software defect classification using machine learning algorithms, and finally, Section 9 concludes the paper.

2. Background

2.1. Machine Learning

Machine learning enables computers to learn without being explicitly programmed for a specific task. Hence, machine learning focuses on developing computer programs that can access data and use it to learn for themselves. Due to advancements in computing, machine learning has gained momentum and is applied in many application domains.

Machine learning algorithms can be categorized into four main fields: supervised, semi-supervised, unsupervised, and reinforcement learning. In supervised learning, the machine is trained using labeled data. This means the correct answer of classification is already known for the training data. After the training phase, a new set of data is used for which the supervised learning algorithm will produce the outputs based on the analyses in

training. Supervised learning algorithms can be divided into two main categories, namely classification, and regression. In classification problems, the output of supervised learning is a category. In the regression problems, the output is a real value.

In unsupervised learning, the machine is not trained with labeled or classified data. Instead, the unsupervised learning algorithm decides the number of classes and which data belongs to which class without a preliminary training phase. Unsupervised learning can be divided into two main categories, which are clustering and association. In clustering problems, the algorithm discovers the inherent groupings in the data. The algorithm discovers the rules that describe the large chunks of data in the association rule learning problems.

In semi-supervised machine learning, data used for training includes both labeled and unlabeled samples. The goal of a semi-supervised learning algorithm and the supervised learning algorithm is the same. However, in semi-supervised learning, some unlabeled data are used to obtain a better model.

Reinforcement learning is about maximizing reward by taking suitable action or the best possible behavior in a particular situation. Unlike supervised learning, in reinforcement learning, there is no answer key. Instead, the algorithm can be trained with the correct answers. In reinforcement learning, the agent decides what to do in the given task by trying alternative actions and getting experience.

2.2. Text Mining and Text Classification

Text mining, also known as text analysis, is the automated process that uses natural language processing to transform unstructured text data into information that provides valuable insights. Text mining combines statistics, linguistics, and machine learning to create models that can predict results on new information based on their previous experience.

Different methods and techniques for text mining can be distinguished. The basic elementary methods include word frequency, collocation, and concordance. Word frequency defines the list of the most frequently occurring words or concepts in a given text. Collocation is used to help identify words that commonly co-occur. Concordance is used to identify the context and instances of words or a set of words. These basic approaches are often used in two advanced text mining techniques, including text classification and extraction.

Text classification is the process of assigning predefined tags or categories to unstructured text. Popular text classification tasks include topic analysis, sentiment analysis, language detection, and intent detection. Topic analysis aims to understand the main themes or subjects of a text. Sentiment analysis can be described as identifying the emotions that underlie any given text. The objective of language detection is to classify a text based on its language. Finally, intent detection aims to recognize the intentions or the purpose behind a text. Text classification is often performed using a rule-based system that uses linguistic rules to automatically detect the different linguistic structures and assign the corresponding tags. As a result, these rules typically consist of references to syntactic, morphological, and lexical patterns. In machine learning-based text classification, data is used to train a machine learning model. Hybrid systems combine rule-based systems with machine learning-based systems, typically to further increase the accuracy of the results.

2.3. Feature Extraction

In the feature extraction step, the preprocessed text is transferred into a numerical representation in the form of a vector, also called vectorization. A common approach for vectorization is the bag of words approach, whereby the vector represents the frequency of a word in a predefined dictionary of words. We have used the "term frequency-inverse document frequency" (TF-IDF) metric to have feature vectors. TF is the number of times a given term (word or phrase) occurs in the document. TF-IDF stands for term frequency-inverse document frequency and defines a statistical measure used to evaluate how important a word is to a document in a collection or corpus. TF-IDF method formulation is given in the below formula:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \log(N/\text{DF}(t)) \quad (1)$$

where 'd' is document and t (term) is the word in a document. TF(t) stands for term frequency, and 'N' is the number of documents in the corpus. Finally, DF(t) represents the number of documents in the corpus containing "t". The TF-IDF value increases as the number of times a word appears increases in the document. The TF-IDF method also takes into account the fact that some documents may be larger than others by normalizing the TF term (expressing relative term frequencies instead).

As the vocabulary size increases, so do the size of the representation. Therefore, in using these text representations, one of the most critical values is the size of the bag-of-words in the model. The bag-of-words size used affects the performance of the classification. To find the optimal bag-of-word size, we performed a grid search.

2.4. Word Embeddings

A more recent approach in feature extraction is using word embeddings. Word embeddings acquire the meaning of words or phrases mapped to vector representations, enabling similar text grouping in a new vector space. Hence, word embeddings might be more efficient than the BOW models. In the BOW models, the broadness of document collection and tagging at the index position causes data sparsity problems. However, word embeddings take the token's surrounding words into account to solve the data sparsity problem. The given text's information is transferred to the model to end up with dense vectors. In this continuous vector space representation, semantically alike words are close to each other. This deduction can either be ensured by utilizing neural networks for language modeling, predicting a word in a sentence given the nearby words as input, or capturing the training corpus's statistical properties. When predicting words in similar context inputs, neural networks generate similar predicted word outputs, resulting in a semantic representation space with the desired property.

The most common word embeddings libraries used in the literature are Word2Vec [7], Doc2Vec [8], GloVe [9], and FastText [10].

FastText

FastText treats each word as composed of character n-grams contrary to Word2Vec and GloVe. For rare words, FastText results in better word embeddings. Furthermore, it can generate a vector for an out of vocabulary word that does not exist in the training corpus. This feature is not possible with either Word2Vec or Glove.

2.5. Evaluation Metrics

Most of the evaluation metrics of classification tasks are built on the confusion matrix concept that reveals the prediction results of the classification model for the test set. In the confusion matrix, the columns present the instances of the predicted classes, whereas the rows represent values for the actual classes, as shown in Table 1.

Table 1. Confusion matrix.

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

This matrix indicates the ways our model is confused in predicting classes. TP presents the predicted positive instances whose actual classes are positive (i.e., no confusion). Similarly, TN shows the instances that are predicted as negative, which are also true. On the other hand, FP and FN show instances predicted as positive and negative, respectively, which are false. The next sub-sections present evaluation measures based on the confusion matrix.

Accuracy can be defined as dividing the sum of correct predictions (TP and TN) to all instances, as given in the equation below.

$$\text{Accuracy (A)} = (TP + TN)/(TP + TN + FP + FN) \quad (2)$$

Precision shows the accuracy of the positive class. Precision is the division of TP to the total positive predictions (TP and FP), as shown in the following formula:

$$\text{Precision (P)} = (TP)/(TP + FP) \quad (3)$$

Recall (or sensitivity) shows the ratio of correctly detected positive classes, and it is defined as TP divided by the sum of positive classes (TP and FN), as shown below:

$$\text{Recall (R)} = (TP)/(TP + FN) \quad (4)$$

F-Measure (or F1 Score) compares models having different Precision and Recall values with a single evaluation measure. It can be defined as the harmonic mean of Precision and Recall, as shown below:

$$F1 = 2 \times (P \times R)/(P + R) \quad (5)$$

3. Triaging and Bug Report Classification Process

In the previous section, we have presented the preliminaries for understanding the bug classification approach used in this paper. We will apply a machine learning-based text mining approach for classifying bugs. The adopted type of the bug report might differ from company to company or even within different departments in the same company. Depending on the project, methodology, and process types, the bug reports can generally be in two formats, structured or unstructured report types, and often used in combination. Unstructured bug reports include free-text descriptions of the bugs, whereas structured bug reports include more detailed and well-presented information about the bugs. The structured forms include specialized areas such as several check-boxes and standard questions to describe the bug formally. Since these specialized areas do not exist in unstructured forms, the only way to extract this information is to understand the free texts in unstructured reports. Independent of the adopted bug report, several well-defined steps are typically followed in the bug classification process. We have modeled the process as shown in Figure 1. The defined model in the figure shows the typical steps of the bug classification process and constitutes a baseline for the case study used in this paper.

First of all, a bug report is prepared based on the input from the adopted tracking system. Then, the bug report will be reviewed, and each identified bug will be checked whether it is a real bug or not. If the item is confirmed as a bug, it is checked whether it is a duplicate definition of the previously entered bug report. For example, the same error might be reported using different words, and it might be hard to understand the duplicate definition. In practice, this is often not a difficult task, yet it is time-consuming since the duplicate definitions need to be removed from the bug tracking systems. Subsequently, bugs are classified into predefined categories. Classification of the software bugs helps in assigning the most appropriate developer to fix the bug. Different classifications can be used. A popular bug classification schema was proposed by Seaman [11]. Seaman's schema uses historical data to guide future projects. IBM presented another popular taxonomy for software bugs called orthogonal defect classification (ODC) taxonomy [12], applied to improve software development processes. The classification of the bug impacts the subsequent activities, that is, triaging of the bugs and assignment to the developer.

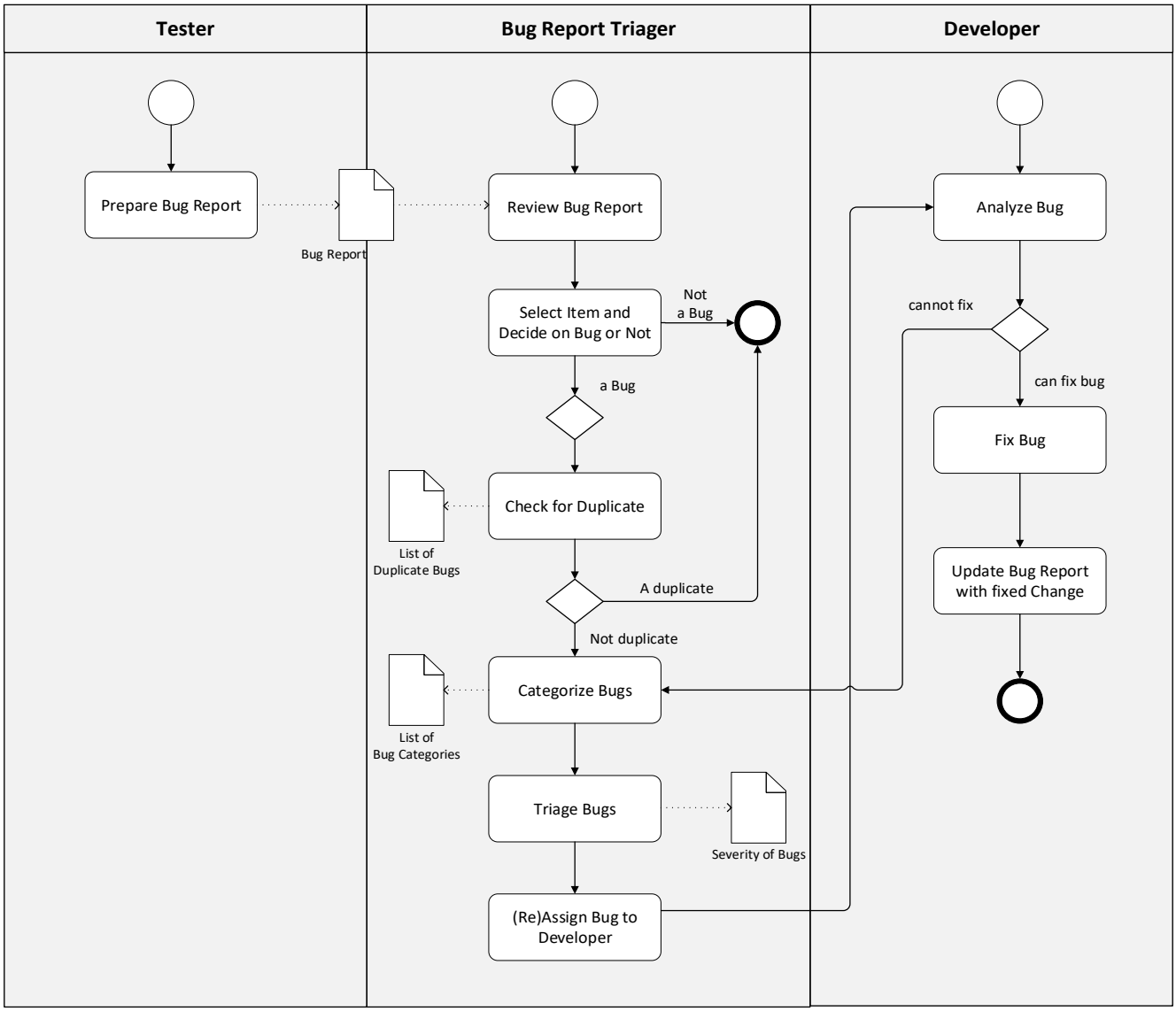

Figure 1. Bug classification process.

The triaging process aims to prioritize software bugs, thereby determining the bug severity. Bug severity is defined as the degree of bug impact on the software. Usually, the more severe bugs will be fixed earlier than those less severe bugs and thus can be resolved later. Assigning the severity of the bug is generally performed by a dedicated person who is called a triager. Although bug reports might include important information about the severity of the failure for the software developers, one of the most time-consuming parts of solving bugs is to understand the degree of severity. Therefore, several studies in the literature have focused on determining the severity of software bugs and report categorization [13–15].

If the bugs are not correctly classified and prioritized, the selected developer might not be right. This issue, named bug bouncing [5], causes the assigned developer to send the bug back, and another developer needs to be selected to fix it. Apparently, unnecessary iterations will increase the fixing time of the bug and, as such, increase the cost.

For small projects with small bug reports, bug classification could, to some extent, be conducted manually. However, it soon becomes less scalable and tractable. Several drawbacks of manual bug classification can be identified. First of all, bug classification is human-dependent and relies on the personal knowledge and experiences of the reviewer. Understanding both the category and the severity of the bugs are not easy. Hence, a more

experienced reviewer will classify and prioritize bugs better, making the bug classification a subjective activity. Furthermore, in the case of a massive number of bugs, the process of reviewing, classifying, and triaging bugs can take too much time that is often not available. It is thus essential to classify the bugs accurately, reducing the unnecessary time and effort spent.

4. Research Questions and Research Methodology

This study aims to assess the effectiveness of ML-based automated bug classification over manual bug classification by utilizing the F1 measure. To this end, we provide an automated bug classification methodology based on machine learning. The typical process for this is shown in Figure 2.

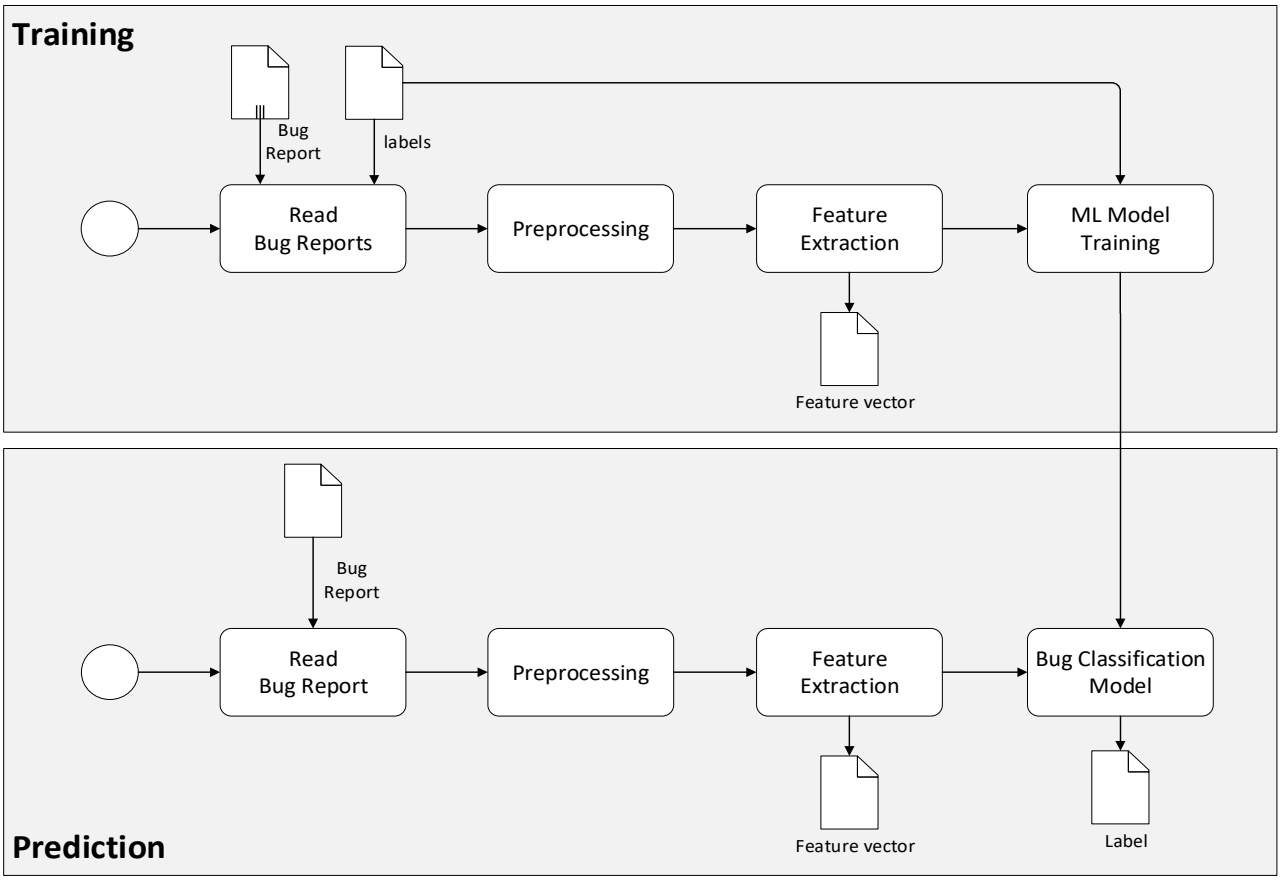

Figure 2. Machine learning-based bug classification process.

Here, two distinct processes are identified, that is, training and prediction. In the training activity, pre-labeled examples of bug classifications are used as training data, whereby multiple machine learning algorithms are used to develop a bug classification model. The preprocessing activity typically includes the steps of tokenization, stemming, and lemmatization. Feature extraction is the method used to transform the preprocessed text into a numerical representation in a vector. A common approach for feature extraction is the bag-of-words, where a vector represents the frequency of a word in a predefined dictionary of words. We will elaborate on this in a later section. Once the bug classification model is trained, it can be used in the prediction activity, which inputs a bug report and automatically classifies the bugs.

Apart from existing ML-based bug classification approaches, our approach distinguishes itself in the combination of the following characteristics:

- Commercial bug reports In contrast to many studies that focus on open-source, public non-commercial data sets, we have used commercial bug reports that reflect the bug report from a large-scale software company.
- Use of unstructured bug reports We rely on unstructured bug reports for categorizing the bugs, which are more difficult to classify than structured bug reports. In addition, unstructured bug reports generally consist of short text, which is sometimes even hard for a human being to understand the real cause of the problem. Therefore, additional preprocessing of the text is needed to prepare it for proper classification.
- Bug report based on black-box testing Apart from the general drawback of the unstructured bug reports, we use bug reports generated after black-box testing, i.e., software test engineers and developers might not know the root cause of the problem. In the adopted case study, testers develop the bug reports after black-box testing. Hence the code is not accessed by the testers, and thus the reports are considered black-box. Even if the bug is defined perfectly in the report, it might be required to investigate the source codes to classify the software bug type correctly.
- Bilingual bug reports Finally, bug definitions used in the case study are not written in a single language but in English and Turkish. Generally, English terms are used together with Turkish verbs and sentences. An example set of unstructured, bilingual bug reports entered in the bug tracking systems is shown in Table 2.

Table 2. A sample set of unstructured bilingual bug report definitions.

Original Bug Definition	Bug Definition in English
Bearing-Coordinate hesaplamaları gözden geçirilecek. Önceki projelerde yapılan kordinat hesaplamalarına göre formüller güncellenecek.	Bearing-coordinate calculations shall be checked. The formulas shall be updated using the coordinate calculations performed in the previous projects.
CommonLogger Log Seviyesi and Dosyaya Yazma Düzeltmesi. CommonLogger Sınıflarına Log Seviyesi olarak 'Error', 'Info', 'Warning', 'Debug' eklenecek. Sınıf **log'**larını dosyaya yazarken **queue** mantığı kullanacak.	CommonLogger log levels and correction of file write. 'Error', 'Info', 'Warning', and 'Debug' levels shall be added to the CommonLogger log levels. Queue logic shall be implemented while writing the class logs.
'Adaptive System Parameters' ekranı tüm değerleri XML dosyasından açılışta yükleyecek.	'Adaptive System Parameters' screen must load all values from the related XML file during the startup of the software.
'Adaptive System Parameters' ekranında search Adaptif parametre ekranındaki search özelliği çalışmıyor.	'Search Adaptive System Parameter' function in the 'Adaptive System Parameters' screen is not working.

This bilingual structure of the bug definitions makes classification even more complicated. To the best of our knowledge, no study has explicitly applied to the use of bilingual bug reports for bug classification. The issues in the classification of bilingual bug reports are given in Section 5.2.2.

The provided objective and the above combination of concerns make this study unique, thereby providing additional insight into machine learning-based bug classification. In this context, we focus on the following particular research questions:

- *RQ 1: To what extent is the presented automated bug classification approach effective?*
- *RQ 2: How does the automated bug classification approach compare to the manual bug classification approaches?*

For addressing these research questions, we adopt the research methodology steps as shown in Figure 3 and describe the industrial case study followed by the adopted bug classification schema.

Subsequently, we identify two basic steps that are executed in parallel. The activity of manual classification includes the activities for describing the case study and the manual

bug classification. The output of this activity is the manual bug classification report. The activity of automated bug classification includes steps for ML-Based classification and results in the automated bug classification report. This automated bug classification report will be compared to the manual bug classification report to evaluate the approach's effectiveness and the corresponding implementation of the process utilizing the F1 measure. In the following subsections, we will describe each of the above steps in more detail.

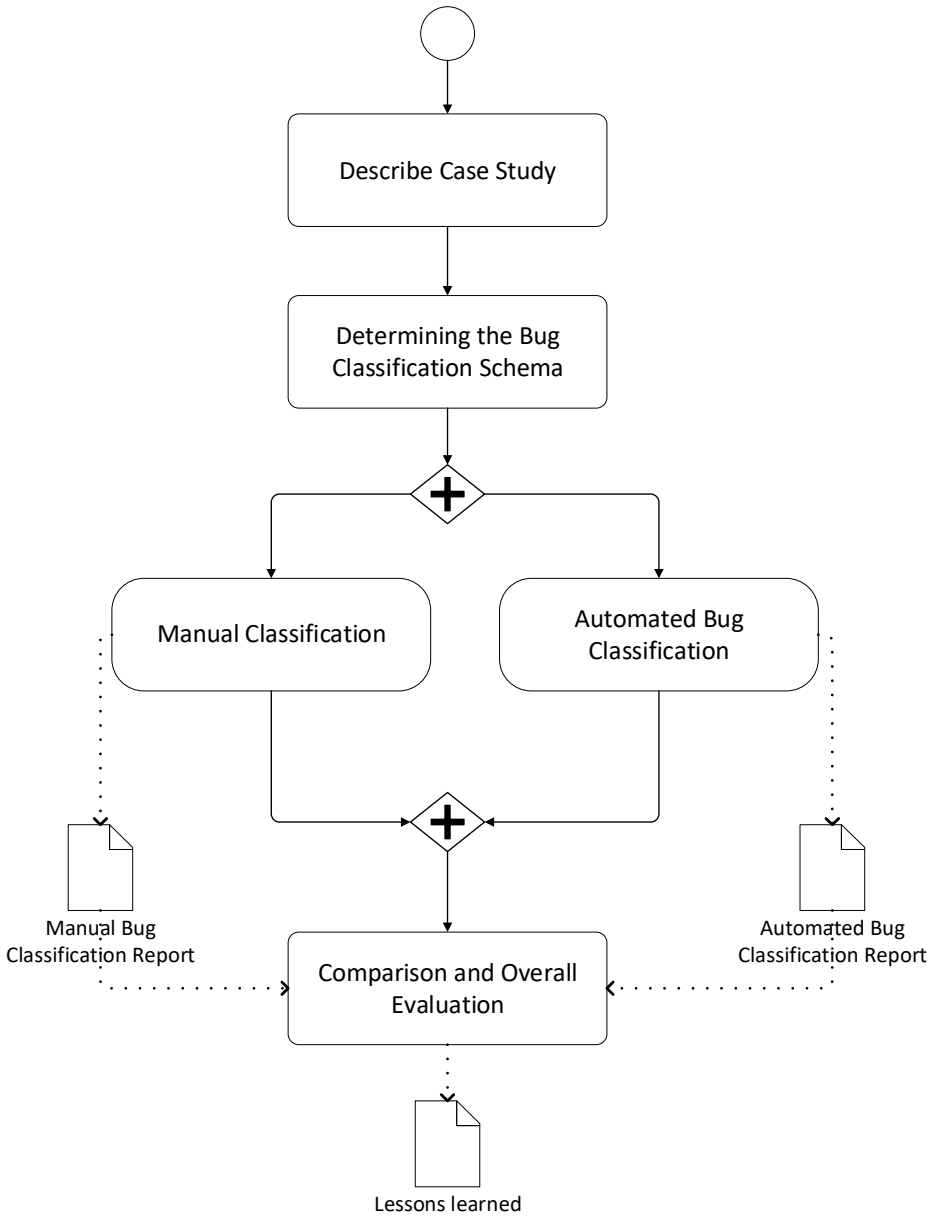

Figure 3. Adopted research methodology.

5. Case Study and Bug Classification

To validate our automated bug classification approach, we have adopted the case study empirical evaluation protocol discussed by Runeson and Höst [16]. The protocol consists of the following steps: (1) case study design, (2) preparation for data collection, (3) execution with data collection on the studied case, (4) analysis of collected data, and (5) case reporting. Table 3 present the case study design steps for the selected case study.

Due to confidentiality reasons, the adopted case study is named a commercial software project (CSP). Eight developers developed the project over 3 years. The programming languages used are C++ and Java. We have used and adapted the bug classification schema

defined by Seaman [11] for the bug classification. We have used four bug categories in our case study: (1) assignment/initialization, (2) external interface, (3) internal interface, and (4) other. The number of bugs entered into the bug tracking software in our case study is 504.

In the following sub-sections, we elaborate on the manual and automated bug classification approaches, followed by the experimental results and evaluation in the following section.

Table 3. Case study design of evaluation protocol adopted from [16].

Case Study Design Activity	Case Study
Goal	Assessing the effectiveness of ML-based automated bug classification over the manual bug classification utilizing F1 measure.
Research Questions	RQ 1: To what extent is the presented automated bug classification approach effective? RQ 2: How does the automated bug classification approach compare to the manual bug classification approaches?
Background and Source	Software testers Software developers Bug classifiers Bug reports Source code
Data Collection	Direct data collection of bug reports Indirect data collection based on source code analysis
Data Analysis	Qualitative and quantitative data analysis

5.1. Manual Bug Classification

In the manual bug classification approach, no automated tool is used, but the categorization relies entirely on the decision of human experts. According to the general flow of bug processing in Figure 1, the bug reports are first reviewed by a reviewer/triager. The reviewer/triager inspects the bug reports and classifies the bugs concerning the bug classification schema used in the project. In this classification, since the reviewer/triager does not know the root cause of the problem, the decision is based on the reports' explanations only. If the report type is an unstructured bug report, it is more challenging to make this classification, especially for the reports of black-box tests. As stated previously, we have classified 504 bugs manually before introducing machine learning algorithms. In our case study, the bugs were classified according to Seaman's bug categories [11].

The classification was performed independently by two expert senior software developers ESE1 and ESE2. Since ESE1 and ESE2 are fluent in English and native in Turkish, bilingual bug reports are not an issue in manual bug classification. Both ESE1 and ESE2 are senior software developers having more than ten years of experience in software development and six years of experience in the case study's domain. The manual classification of software bugs in the case study took five days for ESE1 and ESE2, but the overall time for the case study took longer. The other tasks for the overall case study included the preparation of the set-up (two days), developing and running the test cases (seven days), developing the bug reports (three days), and the overall reporting (three days), which took fifteen days in total. Two expert software engineers were involved in the bug classification, but more than twenty people were involved in the whole process, including project managers, developers, testers, and clients. In the paper, we have only focused on the time for classifying the bugs.

These were then compared with the ground truth classification derived after thoroughly checking the bugs in the source code and the joint discussion with all the experts. Finally, the results of manual classification are given in Table 4.

Table 4. Manual classification by two expert software engineers (ESE).

Bug Category	Ground Truth Classification	Classification of ESE1	Classification of ESE2
Assignment–initialization	54	19	20
External interface	245	244	200
Internal interface	67	43	16
Other	138	126	268
Total	504	504	504

To evaluate the results, we have considered two issues. First of all, to what extent are the classification of the expert software engineers aligned with each other? Secondly, to what extent are the results of these first bug classifications accurate and reliable?

To address the first question, which is the inter-rating differences, we performed a Kappa test. The Kappa (K) coefficient of agreement [17] measures inter-rater and intra-rater reliability for categorical items. The Kappa coefficient is more robust than a simple agreement calculation. K values are between 0 and 1, where higher K values indicate higher and lower values indicate lower agreement between raters. The formula for K is given as follows:

$$K = (Pr(a) - Pr(e))/(1 - Pr(e)) \quad (6)$$

Pr(a) is the relative observed agreement among raters in the above equation, and Pr(e) is the hypothetical probability of chance agreement.

The calculated K value for ESE1 and ESE2 based on the results of Table 4 is 0.63. To interpret this coefficient, we have used the evaluation table of Landis and Koch [18] given in Table 5. This evaluation table concluded that the classifications performed by ESE1 and ESE2 are defined as "agreement with significant importance".

Table 5. Interpretation of K measure [18].

K Measure	Interpretation
K < 0	No agreement
0.00–0.20	Agreement with less importance
0.20–0.40	Agreement with medium importance
0.41–0.60	Agreement with general importance
0.61–0.80	Agreement with significant importance
0.81–1.00	Agreement with excellent importance

Table 6 present the accuracy and F1 measure. As observed from Table 6, the accuracy and F1 measure are very close in each iteration, implying that multiple iterations are necessary to classify the bugs correctly. In turn, this will lead to an increased time for correcting bugs and the increased cost of the project.

Table 6. Accuracy and weighted F1 measure for manual classification by two ESE.

	Accuracy (%)	Weighted F1 Measure (%)
ESE1	60.32	58.48
ESE2	56.75	55.36

To sum up, even though expert software engineers classify the bugs, there seem to be differences in classifications. It should be noted that, as stated before, the classifications are based on bug reports that were derived from black-box testing. The source code was thus not visible. The differences in classifications and the low F1 measures in the above table could be explained from this perspective. Investigation of bug labels and explanations

showed that it is hard to remove the discrepancy. Labels and explanations are sometimes written shortly, and it might be hard to understand the details of the bugs even for the software developers.

Furthermore, bug labels and explanations may also include errors. Altogether we could conclude that the classification is subjective and, to some extent, error-prone and requires further improvement to reduce the time to classify the bugs and herewith the correction of the bugs. Some sample bug definitions and possible root causes are given in Table 7. As observed from this table, it is impossible to determine the error's root cause in these cases by just reading the bug definitions. Therefore, software engineers use their past experience to guess the type of bug for these definitions.

Table 7. Sample bug reports in CSP.

Bug Report Definition	Classification
ERR-1: Camera 1 is not working	The bug might be due to: - The interface between the camera and software might have an error (**Classification**: External Interface) - The camera control software component is not working (**Classification**: Internal Interface) - Software component developed for camera GUI is not working (**Classification**: Other) - The camera hardware is not working (**Classification**: Not a software bug)
ERR-2: Unable to initialize the software	The bug might be due to: - Error in software initialization components (**Classification**: Assignment/Initialization) - Error in multithreading components (**Classification**: Other) - Error in operating system settings during the test (**Classification**: Not a software bug)

5.2. Automated Bug Classification

The second step of the research method presents the automated bug classification approach and the corresponding results. The overall process is shown in Figure 4.

As observed from this figure, the process follows the general ML-based bug classification process, as shown before in Figure 2. In addition, however, it has been adapted to the specifically identified concerns, that is, unstructured, bilingual bug reports derived from black-box testing. We discuss the details of the steps in the following sub-sections.

5.2.1. Bug Classification Schema

Similar to the manual classification, the bugs were classified based on the four categories. We have already noted that we could use a limited set of bugs (504) identified based on black-box test reports. However, it is very hard for software developers and test engineers to determine the real type of bug from the free text of unstructured bug reports. In some cases, code inspection is required to find the class of the bug. In addition, bug definitions written in short texts make classification harder.

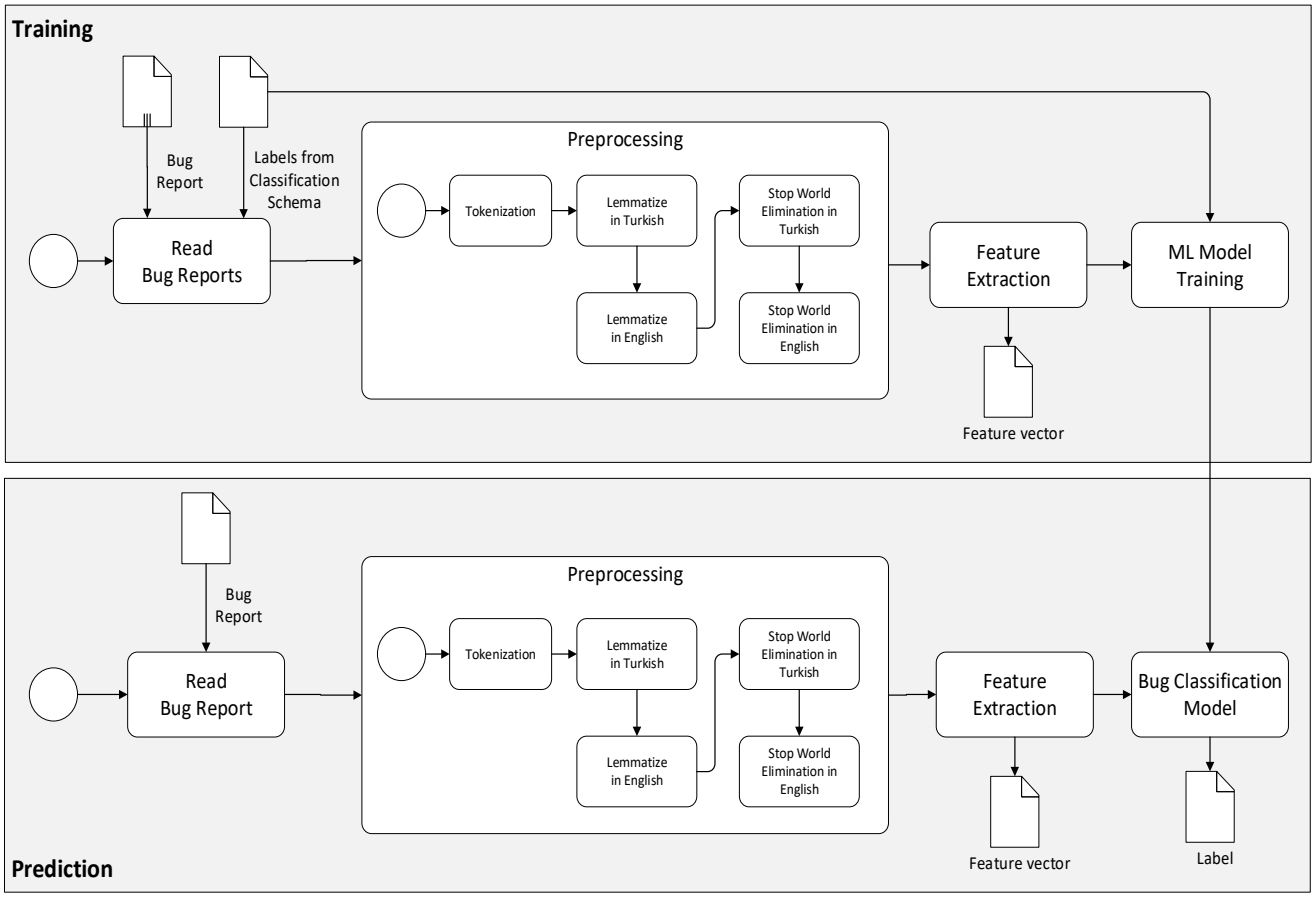

Figure 4. Proposed machine learning-based bug classification process for bilingual, unstructured bug reports.

Furthermore, in bug definitions, both English and Turkish languages were used simultaneously to define the bug that makes the classification complicated. The distribution of the bug definitions over bug categories is not uniform, and almost half of the bugs are classified as an "External Interface" type. The distribution of the bug data used in automatic classification is as given in the first column of Table 4.

5.2.2. Preprocessing of Bug Reports

In the preprocessing phase, we have first removed all punctuations and several non-ASCII characters and performed lower case conversion. Then, the preprocessing activity continued with the steps of tokenization, stemming, and lemmatization.

The tokenization step, that is, the process of breaking down text documents apart into those pieces for further processing, is the same as the conventional approach. However, the bug reports are mainly written in Turkish but also mixed with English words. Therefore, to deal with these bilingual texts, we have applied the lemmatizing step for both languages.

In automated bug classification, classifying bilingual bug reports requires extra care. Firstly, the English and Turkish words require different stemming/lemmatizing libraries. Furthermore, some words in both languages are written in the same letters but represent different meanings. For example, the English word 'on' also exists in Turkish but means 'ten'. Suppose the word 'on' is removed in the stop-word removal process from the bug reports. In that case, there will be a loss of meaning in the available data causing extra difficulty in classification. In addition, due to the agglutinative structure of the Turkish language [19,20], stop-word removal and lemmatizing processes are performed by considering the grammatical structure of the Turkish language.

Similarly, the stop-word elimination phase has been applied for both languages. Since some of the words could be valid both in English and Turkish (for example, the word "on" is a valid and frequent word in English as well as it means "ten" in Turkish), we have applied elimination considering this type of bilingual-equivocal definitions. This process is unique to the bilingual NLP applications, which we call MUSE "Multi-language Stop-word Elimination".

We have applied the same methodology for lemmatizing, Multi-language Lemmatizing (MULE) for the hybrid bug definitions. The definitions must be lemmatized in English first and then in Turkish (or vice-versa), considering that the same words might exist in both languages. In cases such as the above example, the user (the person creating the recommender system) shall check the words used in definitions and determine if the language will be lemmatized.

5.2.3. Machine Learning-Based Bug Classification

Once the text is transformed into vectors, it is fed into a machine learning algorithm and the expected predictions (tags), creating a classification model. Various ML classifiers can be used the bug classification. To experiment with and as such identify the feasible algorithms for this problem, we have selected Naïve Bayes (NB), support vector machine (SVM), k-nearest neighbor (KNN), logistic regression (LR), decision tree (DT), and random forest (RF) classifiers. We have used four kernels of the SVM classifier: linear, poly, radial basis function (RBF), and sigmoid kernels.

Besides the selection of ML algorithms, we have also considered the typical characteristics of the data set. In limited data sets, cross-validation techniques are typically used to improve performance. Cross-validation also prevents obtaining random results. Cross-validation is a random resampling procedure with a single parameter 'k' to refer to the number of subgroups that the data are split into. Cross-validation is used to estimate how the model is expected to perform when the data are not included during the training phase, i.e., each sample is given an opportunity to be used in the test phase and $k - 1$ times used in the training phase. Cross-validation is a popular method since it produces a less biased or less optimistic estimate of the models than a simple train/test split. In our case study, we have performed a grid search to find the optimal 'k' value.

5.2.4. Tuning Process

Our adopted machine learning training approach for the adopted case study is given in Figure 5. Firstly, we select a machine learning classifier and investigate the effect of bilingual stop-word elimination and lemmatizing and the effect of BOW size simultaneously. After performing optimization in preprocessing phase, we optimize the internal parameters of all selected machine learning classifiers. Then we perform N-fold cross-validation. In this phase, instead of applying a fixed N value for the N-fold cross-validation, we tried and checked the 'N' values for better results. Finally, we select the best classifier for our data concerning the results obtained in previous steps.

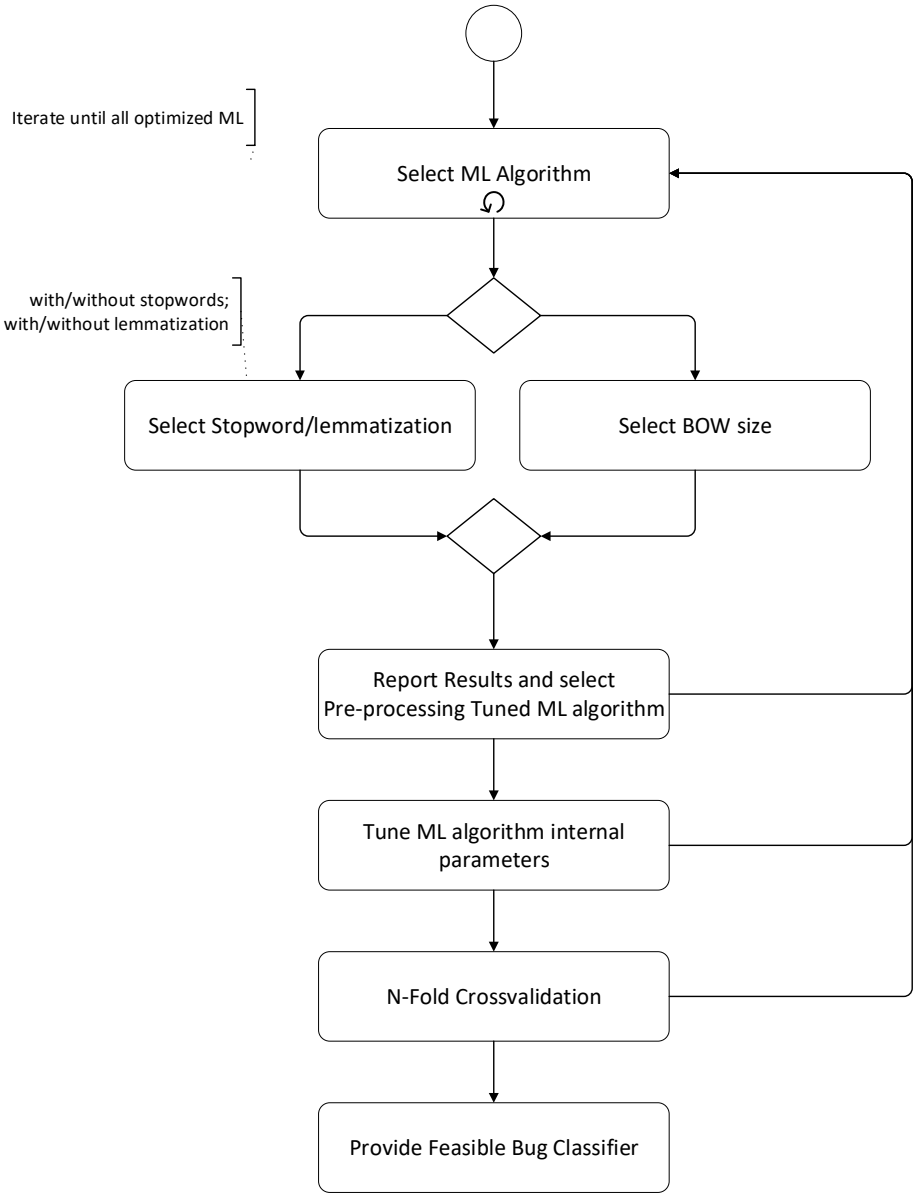

Figure 5. Adopted machine learning training approach for the case study.

6. Experimental Results and Evaluation

In this section, we present the results of the execution of the automated bug classification process for the provided case study with the 504 bugs and the selected ML algorithms. The answers to the first research question are provided in Sections 6.1–6.5. The answer to the second research question is provided in Section 6.6.

As described in the previous section, we follow the process and describe the optimization concerning the preprocessing parameters and internal ML algorithm hyperparameter values. The adopted detailed tuning process is shown in Figure 5. The separate steps will be explained in the following sub-sections. In Section 6.1, we provide the effect of preprocessing on the bug classification. In Section 6.2, we analyze the impact of the bag of words (BOW) size. In Section 6.3, we investigate the effect of tuning internal parameters on classification results. Section 6.4 describes selecting the feasible 'N' value for the 'N-fold cross validation'. Section 6.5 describes the effect of word embeddings. Finally, Section 6.6 provides a comparison with the manual bug classification approach.

6.1. Effect of Preprocessing on Bug Classification

The preprocessing stage includes tokenization, lemmatization, and removing stop-words for bilingual bug definitions. Bilingual data poses different challenges for NLP [21]. Although some researchers apply different parsing techniques for multilingual data [22], we use two-step lemmatization and two-step removal of the stop words, as shown in Figure 4. Details of this process are given in our previous study [23]. Since we are dealing with two different languages (Turkish, English), we have two different steps to eliminate stop words and two different steps for lemmatizing. Accordingly, we use two different lemmatizers, one for English and one for Turkish, executed for each word. In the automated bug classification process, we first use a Turkish lemmatizer followed by an English lemmatizer. During the experiments, we applied a particular stop-word removal process. Hence, applying this process means we have removed both Turkish and English words.

Similarly, not applying the stop-word removal process means that we did not remove stop-words in either language. The same is true for lemmatizing. For the bug classification using the various ML algorithms, we could use the alternative with or without removing stop-words and similarly with or without lemmatization. These combinations led us to four different alternatives, each of which had a different impact on the bug classification using the corresponding ML algorithm. The computed F1 measures of the decision for using or not using stop-word elimination and lemmatizing are given in Table 8.

Table 8. Effect of bilingual preprocessing using stop-word removal (SW) and lemmatizing (LEM) for the case study.

Classifier	F1-Measure (%)			
	No SW No LEM	**No SW LEM**	**SW No LEM**	**SW LEM**
Naïve Bayes	58.49	59.05	60.05	59.30
SVM-linear kernel	**70.92**	70.60	**71.66**	**71.30**
SVM-poly kernel	56.47	56.43	56.42	55.76
SVM- RBF kernel	67.92	68.97	69.48	68.71
SVM- sigmoid kernel	70.24	**70.88**	70.36	70.30
KNN	57.57	57.72	58.48	55.93
Logistic regression	63.11	64.24	65.63	65.15
Decision tree	55.04	55.78	58.03	54.40
Random forest	66.05	66.42	65.48	65.93

The experimental results showed that the most optimal ML algorithm appears to be the SVM linear kernel (71.66%). Subsequently, we wanted to examine the effect of stop word removal and lemmatizing in preprocessing process. We took the "no stop-words and no lemmatizing" case (first table column) as the baseline. Then we saw that the lemmatizing and stop-word removal processes affected the results of the classifiers positively or negatively slightly in any combination of these. Therefore, it is not possible to determine how these processes affect the classification results. However, this result is also consistent with the work of Cagataylı and Celebi [24]. In their work, they applied several preprocessing techniques and investigated the effect on the text classification results. They reported the effect of each technique separately, concluding that only lowercase conversion improves classification success in terms of accuracy and dimension reduction regardless of domain and language. Thus, no particular combination of preprocessing tasks results in better classification performance for any domain and language studied.

6.2. Effect of BOW Size on Bug Classification

In parallel with the focus on stop-words/lemmatization selection, we have also evaluated the impact of the BOW size for the selected ML algorithms; the results are shown in Figure 6. From the figure, we can observe that the SVM linear kernel performs the best.

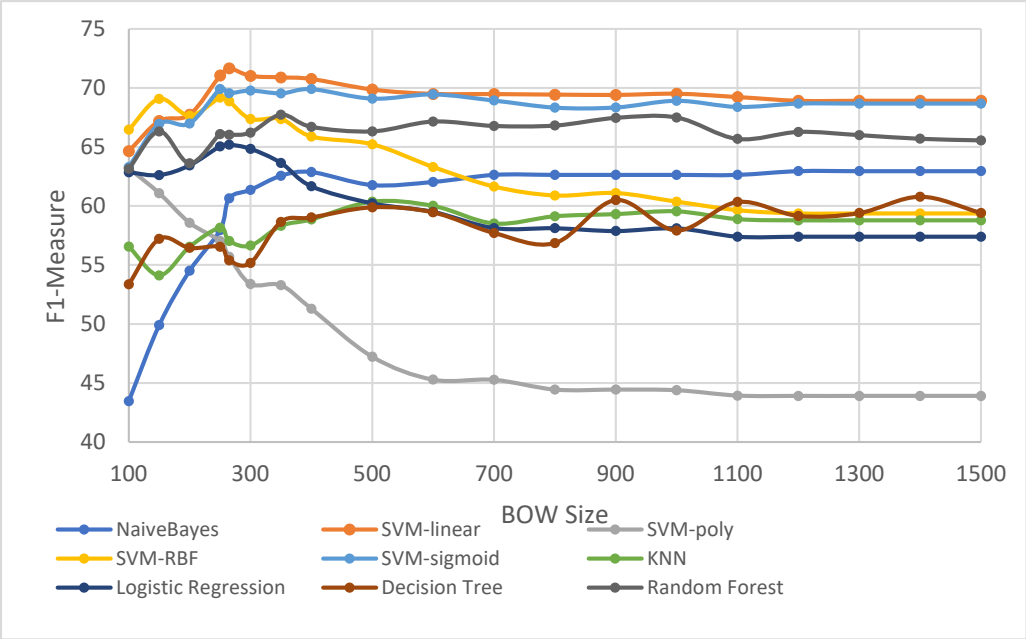

Figure 6. Effect of BOW size in classification for the selected machine learning algorithms.

The F1 measure values obtained for varying BOW size using SVM linear kernel are plotted in Figure 7. The vertical axis represents the F1 measure, while the horizontal value shows the BOW size. From the figure, we can conclude that the BOW size of 265 seems to be optimal. BOW sizes greater than 265 do not seem to have a further positive impact on the accuracy of the ML algorithm.

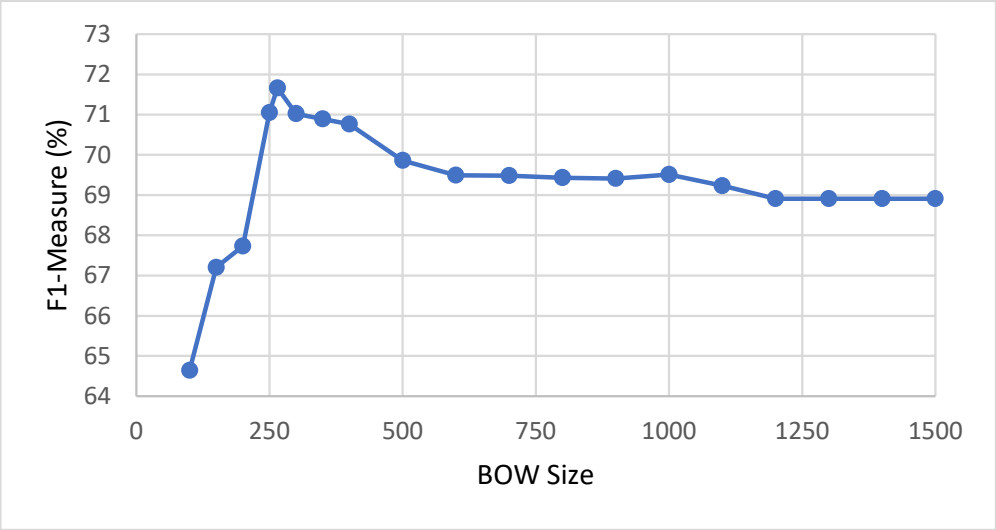

Figure 7. Effect of BOW size in classification (using SVM-linear kernel).

6.3. Tuning Internal Parameters of Selected Machine Learning Algorithms

We performed hyper-parameter tuning to enable better classification. We evaluated the effect of several parameters to have optimal results. The test ratio was always set to 20% during all the tests. Furthermore, we discussed the classification metrics in the following sections.

We used six classification algorithms in our case study. We have used Naïve Bayes, support vector machines (SVM), k-nearest neighbors (KNN), logistic regression, decision tree, and random forest. We have applied SVM with four kernels: linear, poly, RBF, and

sigmoid kernels. By applying different kernels of SVM to the same data, we investigated the differences between SVM kernels. These algorithms are widely used in classification problems. However, depending on the dataset used, one of the given algorithms might perform better. Therefore, instead of selecting some of the algorithms, we applied all of the given algorithms to our dataset. Finally, we selected the best classifier for our dataset. For tuning the ML algorithms, we checked each ML algorithm and its corresponding hyperparameters in detail. For the Naïve Bayes, we did not have any relevant hyperparameter. For SVM, as stated before, we adopted four different kernels. For the k-nearest neighbor algorithm, we optimized the K value to be used, namely, the number of neighbors used in calculations. For the logistic regression algorithm, we optimized the solver and iteration number. Finally, for the random forest algorithm, we optimized the number of estimators and criteria.

6.4. N-Fold Cross-Validation

Firstly, we have investigated the number 'N' in N-fold cross-validation. We have evaluated different N values to have optimal results. In this measure, the F1 mean value is crucial for us. The F1-max and F1-min are subject to more random results. Therefore, we have focused on the change of the F1-mean in the experiments. The change of the F1 measure concerning different 'N' values in N-fold cross-validation is plotted in Table 9.

Table 9. Impact of chosen N value for the N-fold cross-validation.

	F1-Measure (%)					
Classifier	**N = 5**	**N = 10**	**N = 15**	**N = 20**	**N = 25**	**N = 30**
Naïve Bayes	62.40	60.64	60.44	59.79	59.66	58.48
SVM-linear kernel	**69.98**	**71.66**	**68.73**	**70.27**	**68.89**	**67.56**
SVM-poly kernel	55.43	55.69	55.34	55.35	54.49	55.70
SVM-RBF kernel	66.54	68.87	68.04	67.36	68.27	66.03
SVM-sigmoid kernel	69.06	69.56	68.57	69.21	67.93	67.37
KNN	55.06	57.03	55.67	55.50	55.91	54.41
Logistic regression	63.64	65.19	65.64	64.70	65.55	64.42
Decision tree	55.96	56.38	54.97	55.04	57.26	56.40
Random forest	65.82	65.54	65.71	65.86	65.00	64.07

From Table 9, although we can observe that the F1 mean values are close for the selected N values, the best results are obtained using 10-fold cross-validation for the SVM linear kernel case.

The most feasible bug classifier can be selected and applied after training the ML models using the preprocessing and tuning hyperparameters. Moreover, we have obtained the best F1 measure using SVM with a linear kernel, as stated above. Hence, this bug classifier can thus be used for classifying the newly detected bugs following the steps of the prediction process as depicted in the lower part of Figure 4.

6.5. Bug Classification with Word Embeddings

In the above section, we used the BOW approach for vectorization. As the next step, we used word embeddings for feature extraction. Word embeddings libraries provide pre-trained word vectors that can be used with machine learning classifiers. This paper selects FastText since FastText overperforms other word embedding libraries in many NLP tasks [10,25]. Therefore, as an alternative to the TF-IDF approach, we have used FastText in feature extraction. We obtained the results provided in Table 10 using the same machine learning classifiers as in the previous case.

Table 10. Effect of bilingual preprocessing using stop-word elimination (SW) and lemmatizing (LEM) for the case study.

Classifier	F1-Measure (%)
FastText with Naïve Bayes	65.35
FastText with SVM-linear kernel	67.33
FastText with SVM-poly kernel	70.30
FastText with SVM-RBF kernel	69.31
FastText with SVM-sigmoid kernel	65.35
FastText with KNN	58.42
FastText with logistic regression	63.37
FastText with decision tree	50.00
FastText with random forest	**71.19**

In general, word embeddings might be more efficient than the BOW models in classification tasks. However, in our case study, we obtained a 71.19% F1 measure using FastText with random forest. However, this value is slightly less than we obtained with the BOW approach. We believe this is because FastText built-in vectors were trained using the Wikipedia database to provide a general-purpose corpus suitable for news classification or a product's sentiment analysis. But such a corpus appeared not to be a perfect match for classifying bug reports.

6.6. Comparison with Manual Bug Classification

The results of the overall comparison of automated bug classification with manual classification are given in Table 11. The accuracy and F1 measure values for automated bug classification outperform manual classification results performed by expert software engineers, given in Table 4.

Table 11. Accuracy and weighted F1 measures for manual and automatic classification.

	Accuracy (%)	Weighted F1 (%)
ESE1	60.32	58.48
ESE2	56.75	55.36
Automated classification	**73.70**	**71.66**

7. Discussion

Several studies have addressed the topic of automated bug classification, indicating that the manual classification of bugs is cumbersome and time-consuming. Moreover, since the size and complexity of software keep increasing, manual bug classification is not scalable and tractable anymore for human experts. As a result, the chance of making more errors in the bug classification process can increase, and herewith also the process for handling and resolving the bugs.

Although many studies in the literature investigated the automated bug classification problem applying machine learning techniques; our approach distinguishes from these studies in focus on the combination of various aspects: including the usage of commercial software, use of bilingual bug reports (Turkish and English), use of unstructured bug reports, and use of black-box bug reports. Each of these aspects by themselves implies an additional complexity in the bug classification process. Together, these aspects even further impede the manual bug classification process and the automated bug classification process. The problem is thus inherently complicated. Therefore, the first research question focused on searching the answer for whether the adoption of an automated bug classification approach and, in particular, the approach that we have presented would be effective.

Despite the inherent difficulties, we can indeed state that the presented approach is effective. The presented approach adopts and integrates machine learning, text mining, and natural language processing techniques to support the classification of software bugs.

All of these techniques were necessary and had to be adequately integrated. Hence, we have also discussed a well-defined and preplanned process in this paper. We have adopted proven methods in text mining to analyze the bug reports lexically by tokenization, derive the semantic meaning by lemmatization, and extract features using TF-IDF and word embeddings. Then, we mapped these to vectors (vectorization) which machine learning algorithms can process. With this, we took into account and also processed words in the two different languages. We have not limited ourselves to a few machine learning algorithms but adopted Naïve Bayes (NB), support vector machine (SVM), k-nearest neighbor (KNN), logistic regression (LR), decision tree (DT), and random forest (RF) classifiers in our experiments. We have also used all four kernels of the SVM classifier: linear, poly, RBF, and sigmoid kernels. It should be stated that the process required some iterations to fine-tune the hyperparameters of the ML algorithms.

In addition to the BOW approach, we have also adopted the more recent approach in feature extraction using word embeddings. In general, word embeddings might be more efficient than the BOW models in classification tasks. However, in our case study, we obtained a 71.19 % F1 measure using FastText with random forest. This value is slightly less than we obtained with the BOW approach and SVM with linear kernel. We believe this is because FastText built-in vectors were trained using the Wikipedia database to provide a general-purpose corpus suitable for news classification or a product's sentiment analysis. Despite this, such a corpus appeared not to be a perfect match for classifying bug reports.

We used the F1 score and accuracy metrics for the evaluation metrics, similar to the earlier studies described in the survey on automated bug classification of Gomes et al. [6]. For the automated bug classification, the precision and recall metrics were considered important. Since F1 covered both metrics, these were not separately reported. We might have added other metrics independent of most of the practices published in the related studies. This topic will be further explored and considered in our future studies.

Nevertheless, with this approach, we could process bug reports in an automated manner without the need for a human expert. This result was an important observation for the practical context whereby mainly manual classification by experts was applied. Still, as we stated before, it became more and more cumbersome, time-consuming, and costly, which brought us to the second question. Automated bug classification seems to work and with good results too. However, was it also better than manual bug classification? Our answer to this question is also confirmative. Our experimental results showed that automated bug classification, as we have implemented, outperforms the human expert.

To evaluate this process, we have adopted a well-defined case study research [16] in which we have carefully followed and executed the protocol. As a result, we first checked the results of manual bug classification by two experts and compared these to the known results (ground truth). The experiments also confirmed our hypothesis that bug classification is subjective even for experts. Different explanations can be given for this, such as the fact that bug classification is based on bug reports derived from black-box testing, the relatively short texts in the bug reports, and the bilingual aspects. The investigation of the reasons for the subjectivity could be one of the further research topics of this study. After the manual bug classification, we thoroughly assessed the impact of each machine learning algorithm that required some fine-tuning and effort. In parallel, we also checked the impact of the BOW size and eventually derived the most feasible situation that is effective in classifying bugs. Due to confidentiality reasons, we could only use a data set of 504 bug reports. However, the size of the dataset is not large enough to use in deep learning techniques that require large datasets for training. Furthermore, since this is a proprietary dataset, we did not have a chance to increase the data size.

Several other text mining techniques can be identified, among which we can distinguish text summarization, which is mainly used to shorten longer texts in the text data. The aim of this process is to obtain a short, coherent, and fluent summary by outlining major points in the longer text data. However, in our case, we have short text in our bug repository. Hence, text summarization is not a suitable approach to be used in short text

classification. Similarly, other techniques such as semantic similarity and abbreviation discovery can be identified, which will be considered in our future work.

Since the data set was insufficient for training neural network architectures, we could not apply deep learning models directly. We adopted artificial neural networks (ANN) and convolutional neural networks (CNN) in earlier experiments. However, the F1 scores were as expected, less than the machine learning approaches (ANN < 55% and CNN < 65%) that we eventually applied.

As in this case study, the lack of data for commercial bug reports is still an obstacle. Moreover, due to the business-critical aspects, the provided data is often confidential and not publicly accessible. In case of the availability of larger commercial data sets, we will experiment with the application of various deep learning models for bug classification.

Another issue was the classification schema that we used for classifying bugs. We used and adapted the bug types of Seaman [11], whereby some categories were taken together to align this with the industrial practice. In the adopted case study, we had to cope with an imbalanced dataset which thus implies different results for different bug types of the category. Hence the result of an imbalanced categorization of the bugs was expected. We could have adopted a different bug taxonomy, but this would not have changed the imbalance of the data.

Altogether, we can state that the approach and the corresponding tools were effective and valuable for the automated bug classification of bilingual, unstructured bug reports derived from black-box testing.

8. Related Work

Several studies in the literature have focused on bug classification, which has addressed different issues, including finding duplicate bugs, checking a report that includes a bug, and bug triaging. In this section, we briefly provide general issues in the bug classification domain and then give details regarding bug classification studies.

8.1. Bug Classification

One of the common issues in the bug classification domain is the selection of bug categories. One of the oldest works to determine the software bugs was performed by Beizer [26]. According to Beizer, many different sources are influencing the taxonomy. He offers more than 100 bug categories, 10 of which are base categories. Chillarege et al. (IBM) offered one of the earliest and most famous bug taxonomy Orthogonal Defect Classifications (ODC) [12], which consists of 13 bug categories. Freimut et al. (Hewlett-Packard) provided an approach for defining, introducing, and validating customized software bug classification schemes in the industrial context [27]. In this context, bugs are defined by their origin, mode, and type. The 'origin' of the bug is the activity in which the defect was introduced. The 'mode' of the bug describes the scenarios leading to the bug. Plosky et al. surveyed the literature that provides quantitative data on categories of software faults [28]. Seaman et al. [11] offer categorization schemes to use the historical data by guiding future projects. Seamon et al. offer three inspection areas for software defects: "Requirements inspection", "Design and source code inspection", and "Test plan inspection" defect types.

It has been observed that bug classification depends on the quality of bug reports, and several studies have shown that bug reports are widely different in quality. In [27,28], the authors report on the survey results to find the critical information that needs to exist in bug reports.

Zimmermann et al. [29] and Bettenbutg et al. [30] claim that the duplicate bug definitions are not harmful as opposed to the common belief. On the other hand, many studies [31] indicate that duplicate bug definitions might cause extra maintenance costs.

Several studies have focused on bug triaging [13,14] to assign software bugs to the most suitable developer. Zhang et al. [15] conducted a thorough overview of the conducted research in the context of bug triaging. Gomes et al. [6] provided a survey on severity

prediction in open-source software. They provided a comprehensive mapping study review of recent research efforts on automatic bug report severity prediction. The paper concludes that unstructured text features, traditional machine learning algorithms, and text mining methods have played a central role in the most proposed methods in literature to predict bug severity levels. The authors also conclude that there is room for improving prediction results using state-of-the-art machine learning and text mining algorithms and techniques. Furthermore, the paper shows that all of the papers have used open-source software, and none of these have focused on commercial software as we did in this study.

In [32], the authors proposed a new severity prediction approach using a linear combination of stack trace similarity and categorical features similarity on open-source datasets.

8.2. Automatic Bug Classification with Machine Learning Algorithms

Antoniol et al. [2] investigated the automatic classification of bug reports by utilizing conventional text mining techniques dealing with the reports' description part (free text). They have reported the misclassification of bug reports and claimed that more than half are unrelated to software bugs. Therefore, they have proposed an ML-based classification approach to distinguish software bugs from other issues in the bug tracking systems. Similarly, Herzig et al. [33] concluded that "every third bug is not a bug", stating that alternatively, manual classification is time-consuming.

Pingclasai et al. [4] proposed using topic modeling techniques, namely, latent dirichlet allocation (LDA), to classify bug reports automatically. Their work claimed that Naïve Bayes is the most efficient classification model, applying LDA yields the F-Measure between 0.65 and 0.82. On the other hand, Limsettho et al. [34] applied the nonparametric Hierarchical Dirichlet Process (HDP) as an alternative topic modeling technique. They claimed that HDP performance is comparable with LDA where parameter tuning is required while their data set, for both LDA and HDP, suffers from a lack of data and imbalance dataset problems. Further, Zhou et al. [35] proposed a multi-stage classification hybrid approach with data grafting by combining data mining and text mining with structured data (priority, severity, etc.) to automate the prediction process. As a result, Zhou et al. claimed that they achieved a reasonable enhancement between 81.7 and 93.7 by applying their technique to the same data on the three large-scale open-source projects. Finally, Terdchanakul et al. [36] proposed an alternative solution by using N-gram IDF to detect if bug reports contain software bugs or non-bugs, achieving an F-Measure between 0.62 and 0.88.

Catalino et al. [1] indicated the research community deeply investigated the bug triaging process. In contrast, only a few studies are available in the literature to support the developers in understanding the type of a reported bug which is the most time-consuming step of fixing software bugs. Therefore, they built a taxonomy for several popular open-source projects such as Mozilla, Apache, and Eclipse, providing automatic support for labeling bugs according to their type. Then they evaluated the automated classification model utilizing the defined taxonomy, achieving an F-Measure of 64%.

In their recent work, Hernández-González et al. [37] provided a solution for classifying the impact of bugs using ODC schema [12]. As a result, they claimed that they had reached enhanced performance regarding the majority voting. Similarly, Huang et al. [38] used ODC by casting as a supervised text classification problem and proposed AutoODC obtaining an overall accuracy of 80.2% when using manual classification as the baseline.

Thung et al. [39] manually analyzed and classified 500 defects according to the ODC. They offer an automatic classification approach to categorize bug defects into three super categories of ODC: control and data flow, structural, and non-functional.

In his more recent work, Thung et al. [40] proposed an active semi-supervised defect categorization approach to reduce the cost of manually labeling a huge number of examples using the same 500 defects as a benchmark. Authors claim that this approach outperforms the baseline approach of Jain and Kapoor [41] by incrementally refining the training model and achieving an F-Measure of 0.623 and AUC of 0.710.

Xia et al. [42] proposed a fuzzy set-based future selection algorithm and categorized defects into fault trigger categories by analyzing the natural language descriptions of bug reports and utilizing text mining. They reported that they analyzed 809 bug reports and their approach achieved an F-Measure of 0,453 by improving the Mandelburg F-Measure, which is the best performing baseline, by 12.3%.

Lopes et al. [43] used ODC for classifying software defects, indicating that it requires one or more experts to categorize each defect in a reasonably complex and time-consuming process. They evaluated the applicability of a set of machine learning algorithms (NB, KNN, SVM, Recurrent Neural Networks (RNN), Nearest Centroid, and RF) for performing automatic classification of software defects based on ODC and using unstructured text bug reports. The authors claimed that the difficulties in automatically classifying certain ODC attribute solely using unstructured bug reports. Hence, they suggested that the use of larger datasets to improve overall classification accuracy.

To compare our study with the related works in this section, we have listed the various approaches in Tables 12 and 13. Thus, our study builds on and complements the above studies. In addition, our study is unique since it deals with commercial bug reports instead of other researchers dealing with bug reports of open-source software, as given in Table 12. Furthermore, Table 13 show the techniques and classifiers used in the related works.

Table 12. Comparison of automatic bug classification studies.

Ref.	Paper	Year	No. of Bugs	Commercial/Open Source	Dataset(s)
[2]	Antoniol et al.	2007	1800	Open Source	Mozilla, Eclipse, and JBoss
[39]	Thung et al.	2012	500	Open Source	Mahout, Lucene, and OpenNLP
[4]	Pingclasai et al.	2013	1940	Open Source	HTTP Client, Jackrabbit, and Lucene
[34]	Limsettho et al.	2014	2718	Open Source	HTTP Client, Jackrabbit, and Lucene
[35]	Zhou et al.	2014	3220	Open Source	Mozilla, Eclipse, JBoss, Firefox, and OpenFOAM
[42]	Xia et al.	2014	809	Open Source	Linux, Mysql, Apache HTTPD, and AXIS
[40]	Thung et al.	2015	500	Open Source	Mahout, Lucene, and OpenNLP
[36]	Terdchanakul et al.	2017	3356	Open Source	HTTP Client, Jackrabbit, and Lucene
[37]	Hernández et al.	2018	1444	Open Source	Compendium and Mozilla
[1]	Catalino et al.	2019	1280	Open Source	Mozilla, Apache, and Eclipse
[43]	Lopes et al.	2020	4096	Open Source	MongoDB, Cassandra, and HBase
-	Our work	2021	504	Commercial	Proprietary Bug Dataset

Table 13. Comparison of automatic bug classification studies.

Ref.	Techniques Used	Classification/Clustering Algorithms
[2]	ML and model feature selection	NB, LR, and alternating decision tree
[39]	ML and text mining using three super categories of ODC defect types	C4.5, NB, SVM, and LR
[4]	LDA topic modeling	NB, LR, and alternating decision tree
[34]	HDP topic modeling	NB, LR, and alternating decision tree
[35]	Data mining and text mining with structured data, multi-stage classification with data grafting	Multinominal NB, LR, and alternating decision tree
[42]	Fuzzy-based text selection, text mining, and natural language descriptions with a fuzzy eet-based feature selection [USES] algorithm.	USES, multinominal NB, NB, SVM, LR, and RBF network
[40]	Hybrid ML approach with semi-supervised defect categorization, active learning, and clustering	K-means clustering and SVM
[36]	N-gram with inverse document frequency (IDF)	LR and RF
[37]	Bayesian network classifiers, K-means clustering and expectation-maximization (EM) strategy with majority voting and ODC defect types	NB, tree augmented NB (TAN), and K-dependence Bayesian network classifier (KDB)
[1]	Taxonomy building by classifying according to the defined taxonomy with TF-IDF, Word2Vec, and Doc2Vec	NB, LR, SVM, and RF
[43]	ML and RNN using ODC defect types	NB, SVM, KNN, RNN, nearest centroid, and RF
Our Work	ML, NLP, information retrieval, and multi-language pre-processing with Seaman's defect types	NB, SVM, KNN, LR, DT, and RF

9. Conclusions

The proper management of bugs is a significant factor in decreasing the cost of software projects. Different activities are needed here, including reporting the bug, determining if it is a bug, finding duplicate bugs, bug prediction, bug classification, and bug triaging. This article focused on and provided an approach for automated bug classification in a real industrial context. Two main questions were posed: Is automated bug classification effective, and how does this compare to manual bug classification?

We have shown that manual classification, even by performed by experts, is complex and can lead to subjective results and interpretation; thus it is unnecessary to extend the time to handle the bug. Bug classification becomes even more difficult if we deal with unstructured bug reports derived from black-box testing, in which more than one language is used. We have provided the method and the corresponding tools for automated bug classification. We could build on existing text mining approaches but had to refine these processes for the bilingual aspects. After the preprocessing, we applied several different machine learning algorithms, fine-tuned the hyperparameters, compared the results for different BOW sizes, and derived the most feasible algorithm. The SVM classifier that we eventually used appeared to be effective. Our first conclusion was thus that the automated bug classification that we have developed was undoubtedly effective.

We followed a systematic case study research protocol to answer the second question of whether the presented automated bug classification is better than manual bug classification. We checked the manual bug classification by experts and compared this to the automated bug classification approach results. The results of the case study research showed that automated bug classification was indeed more effective than manual bug classification.

Despite the overall performance of the approach, we concluded that preprocessing in our bilingual case study did not result in significantly better classification performance. This conclusion is in alignment with the outcome of earlier studies on the impact of preprocessing.

The presented automated bug classification approach will be integrated into the company's bug tracking and handling process and system. In our future work, we will further experiment and enhance the approach using deep learning and NLP. For this, we will also consider larger case studies. In addition, we will also explore solutions for imbalanced datasets which had an impact on the use of bug classification schema.

Author Contributions: Conceptualization, Ö.K. and B.T.; methodology, Ö.K. and B.T.; software, Ö.K.; validation, Ö.K. and B.T. Writing—original draft preparation, Ö.K. and B.T.; writing—review and editing, Ö.K. and B.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Catolino, G.; Palomba, F.; Zaidman, A.; Ferrucci, F. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *J. Syst. Softw.* **2019**, *152*, 165–181. [CrossRef]
2. Antoniol, G.; Ayari, K.; Di Penta, M.; Khomh, F.; Guéhéneuc, Y.G. Is it a bug or an enhancement? A text-based approach to classify change requests. In Proceedings of the 2008 Conference of the Center for Advanced Studies (CASCON'08), Richmond Hill, ON, Canada, 27–30 October 2008.
3. Bhattacharya, P.; Neamtiu, I.; Shelton, C.R. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *J. Syst. Softw.* **2012**, *85*, 2275–2292. [CrossRef]
4. Pingclasai, N.; Hata, H.; Matsumoto, K.I. Classifying bug reports to bugs and other requests using topic modeling. In Proceedings of the Asia-Pacific Software Engineering Conference (APSEC), Bangkok, Thailand, 2–5 December 2013; IEEE Computer Society: Hoboken, NJ, USA, 2013; Volume 2, pp. 13–18.

5. Jeong, G.; Kim, S.; Zimmermann, T. Improving bug triage with bug tossing graphs. In Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'09), Amsterdam The Netherlands, 24–28 August 2009; ACM Press: New York, NY, USA, 2009; pp. 111–120.
6. Gomes, L.A.F.; da Silva Torres, R.; Côrtes, M.L. Bug report severity level prediction in open source software: A survey and research opportunities. *Inf. Softw. Technol.* **2019**, *115*, 58–78. [CrossRef]
7. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. In Proceedings of the 1st International Conference on Learning Representations, ICLR 2013—Workshop Track Proceedings, Scottsdale, AZ, USA, 2–4 May 2013.
8. Le, Q.V.; Mikolov, T. Distributed Representations of Sentences and Documents. In Proceedings of the 31st International Conference on Machine Learning (ICML), Bejing, China, 22–24 June 2014; Volume 32, pp. 1188–1196.
9. Pennington, J.; Socher, R.; Manning, C. GloVe: Global Vectors for Word Representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; Association for Computational Linguistics (ACL): Doha, Qatar, 2014; pp. 1532–1543.
10. Bojanowski, P.; Grave, E.; Joulin, A.; Mikolov, T. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguist.* **2017**, *5*, 135–146. [CrossRef]
11. Seaman, C.; Shull, F.; Regardie, M.; Elbert, D.; Feldmann, R.L.; Guo, Y.; Godfrey, S. Defect categorization: Making use of a decade of widely varying historical data. In Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'08), Kaiserslautern, Germany, 9–10 October 2008; pp. 149–157.
12. Chillarege, R.; Bhandari, I.S.; Chaar, J.K.; Halliday, M.J.; Ray, B.K.; Moebus, D.S. Orthogonal defect classification—A concept for in-process measurements. *IEEE Trans. Softw. Eng.* **1992**, *18*, 943–956. [CrossRef]
13. Čubranić, D.; Gail, M. Automatic bug triage using text categorization. In Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE), Banff, AB, Canada, 20–24 June 2004; pp. 92–97.
14. Neelofar; Javed, M.Y.; Mohsin, H. An automated approach for software bug classification. In Proceedings of the 6th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), Palermo, Italy, 4–6 July 2012; pp. 414–419.
15. Zhang, T.; Jiang, H.; Luo, X.; Chan, A.T.S. A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions. *Comput. J.* **2016**, *59*, 741–773. [CrossRef]
16. Runeson, P.; Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* **2008**, *14*, 131–164. [CrossRef]
17. Cohen, J. A Coefficient of Agreement for Nominal Scales. *Educ. Psychol. Meas.* **1960**, *20*, 37–46. [CrossRef]
18. Landis, J.R.; Koch, G.G. The Measurement of Observer Agreement for Categorical Data. *Biometrics* **1977**, *33*, 159. [CrossRef] [PubMed]
19. Oflazer, K.; Saraçlar, M. Turkish and Its Challenges for Language and Speech Processing. In *Turkish Natural Language Processing*; Springer: Cham, Switzerland, 2018; pp. 1–19.
20. Oflazer, K.; Çetinoğlu, Ö.; Say, B. Integrating morphology with multi-word expression processing in Turkish. In Proceedings of the Workshop on Multiword Expressions: Integrating Processing, Barcelona, Spain, 26 July 2004; Association for Computational Linguistics (ACL): Doha, Qatar, 2004; pp. 64–71.
21. Yirmibeşoğlu, Z.; Eryiğit, G. Detecting Code-Switching between Turkish-English Language Pair. In Proceedings of the 2018 EMNLP Workshop W-NUT: The 4th Workshop on Noisy User-Generated Text, Brussels, Belgium, 1 November 2018; Association for Computational Linguistics (ACL): Doha, Qatar, 2019; pp. 110–115.
22. Hall, J.; Nilsson, J.; Nivre, J.; Eryiğit, G.; Megyesi, B.; Nilsson, M.; Saers, M. Single malt or blended? A study in multilingual parser optimization. In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), Prague, Czech Republic, 28–30 June 2007; Springer: Dordrecht, The Netherlands, 2007; pp. 933–939.
23. Koksal, O. Tuning the Turkish Text Classification Process Using Supervised Machine Learning-based Algorithms. In Proceedings of the International Conference on INnovations in Intelligent SysTems and Applications (INISTA), Novi Sad, Serbia, 24–26 August 2020; IEEE: Hoboken, NJ, USA, 2020; pp. 1–7.
24. Cagataylı, M.; Celebi, E. The effect of stemming and stop-word-removal on automatic text classification in Turkish language. In *Proceedings of the ICONIP 2015, Istanbul, Turkey, 9–12 November 2015*; Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Springer: Cham, Switzerland, 2015; Volume 9489, pp. 168–176.
25. Kilimci, Z.H.; Yoruk, H.; Akyokus, S. Sentiment Analysis Based Churn Prediction in Mobile Games using Word Embedding Models and Deep Learning Algorithms. In Proceedings of the 2020 International Conference on INnovations in Intelligent SysTems and Applications (INISTA), Novi Sad, Serbia, 24–26 August 2020; Institute of Electrical and Electronics Engineers Inc.: Hoboken, NJ, USA, 2020.
26. Beizer, B. *Software Testing Techniques*; Van Nostrand Reinhold: New York, NY, USA, 1990.
27. Freimut, B.; Denger, C.; Ketterer, M. An industrial case study of implementing and validating defect classification for process improvement and quality management. In Proceedings of the International Software Metrics Symposium, Como, Italy, 19–22 September 2005; Volume 2005, pp. 165–174.

28. Ploski, J.; Rohr, M.; Schwenkenberg, P.; Hasselbring, W. Research issues in software fault categorization. *ACM SIGSOFT Softw. Eng. Notes* **2007**, *32*, 6. [CrossRef]
29. Zimmermann, T.; Premraj, R.; Bettenburg, N.; Just, S.; Schröter, A.; Weiss, C. What makes a good bug report? *IEEE Trans. Softw. Eng.* **2010**, *36*, 618–643. [CrossRef]
30. Bettenburg, N.; Just, S.; Schröter, A.; Weiss, C.; Premraj, R.; Zimmermann, T. What makes a good bug report? In Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Atlanta, GA, USA, 9–14 November 2008; pp. 308–318.
31. Sun, C.; Lo, D.; Khoo, S.C.; Jiang, J. Towards more accurate retrieval of duplicate bug reports. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, KS, USA, 6–10 November 2011; pp. 253–262.
32. Sabor, K.K.; Hamdaqa, M.; Hamou-Lhadj, A. Automatic prediction of the severity of bugs using stack traces and categorical features. *Inf. Softw. Technol.* **2020**, *123*, 106205. [CrossRef]
33. Herzig, K.; Just, S.; Zeller, A. It's not a bug, it's a feature: How misclassification impacts bug prediction. In Proceedings of the International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 392–401.
34. Limsettho, N.; Hata, H.; Matsumoto, K.I. Comparing hierarchical dirichlet process with latent dirichlet allocation in bug report multiclass classification. In Proceedings of the 2014 IEEE/ACIS 15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Las Vegas, NV, USA, 30 June–2 July 2014; Institute of Electrical and Electronics Engineers Inc.: Hoboken, NJ, USA, 2014.
35. Zhou, Y.; Tong, Y.; Gu, R.; Gall, H. Combining text mining and data mining for bug report classification. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME), Victoria, BC, Canada, 29 September–3 October 2014; Institute of Electrical and Electronics Engineers Inc.: Hoboken, NJ, USA, 2014; pp. 311–320.
36. Terdchanakul, P.; Hata, H.; Phannachitta, P.; Matsumoto, K. Bug or not? Bug Report classification using N-gram IDF. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; Institute of Electrical and Electronics Engineers Inc.: Hoboken, NJ, USA, 2017; pp. 534–538.
37. Hernández-González, J.; Rodriguez, D.; Inza, I.; Harrison, R.; Lozano, J.A. Learning to classify software defects from crowds: A novel approach. *Appl. Soft Comput. J.* **2018**, *62*, 579–591. [CrossRef]
38. Huang, L.G.; Ng, V.; Persing, I.; Geng, R.; Bai, X.; Tian, J. AutoODC: Automated generation of Orthogonal Defect Classifications. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, KS, USA, 6–10 November 2011; pp. 412–415.
39. Thung, F.; Lo, D.; Jiang, L.; Le, X.B.D.; Lo, D. Automatic defect categorization. In Proceedings of the Working Conference on Reverse Engineering (WCRE), Kingston, ON, Canada, 15–18 October 2012; IEEE Computer Society: Hoboken, NJ, USA, 2012; Volume 2015-Augus, pp. 205–214.
40. Thung, F.; Le, X.B.D.; Lo, D. Active Semi-supervised Defect Categorization. In Proceedings of the IEEE International Conference on Program Comprehension, Florence, Italy, 18–19 May 2015; IEEE Computer Society: Hoboken, NJ, USA, 2015; Volume 2015-Augus, pp. 60–70.
41. Jain, P.; Kapoor, A. Active learning for large multi-class problems. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; Institute of Electrical and Electronics Engineers (IEEE): Hoboken, NJ, USA, 2009; pp. 762–769.
42. Xia, X.; Lo, D.; Wang, X.; Zhou, B. Automatic defect categorization based on fault triggering conditions. In Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Tianjin, China, 4–7 August 2014; Institute of Electrical and Electronics Engineers Inc.: Hoboken, NJ, USA, 2014; pp. 39–48.
43. Lopes, F.; Agnelo, J.; Teixeira, C.A.; Laranjeiro, N.; Bernardino, J. Automating orthogonal defect classification using machine learning algorithms. *Futur. Gener. Comput. Syst.* **2020**, *102*, 932–947. [CrossRef]