Test Case Prioritization based on Neural Network Classification with Artifacts Traceability

Ioana-Claudia Rotaru
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
ioana.rotaru@stud.ubbcluj.ro

Andreea Vescan
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
andreea.vescan@ubbcluj.ro

***Abstract*—Regression testing is an important factor in ensuring software system reliability once new changes are introduced, but maintaining complex testing suites in continuous integration environments is challenging. Test case prioritization techniques are a potential solution to this problem by computing a reordered testing suite that can provide better fault detection capabilities. However, current methods rely on manually providing artifact dependencies (requirements to code, code to test cases, test cases to faults) as input.**

The purpose of this paper is to minimize the gap between automatic dependency computation and test case prioritization by analyzing how Behavior-Driven Development (BDD) practices affect the two tasks. Thus, the first contribution of this paper is related to the design and implementation of an automatic traceability component to retrieve dependencies based on BDD artifacts (requirements, source code, test cases, and faults). The second contribution refers to the integration of the discovered traces as features in a neural network classification model for test cases for further prioritization.

Various architectures were used for the neural network classification model. Two real-world BDD projects were used for the validation of the models, comparing the best-performing models with a baseline test case prioritization technique to assess their fault-detection capabilities.

Our approach achieved promising fault detection rates that demonstrate the efficiency of automatic traceability and may lead to future applicability to large-scale projects.

***Index Terms*—Regression Testing, Test Case Prioritization, Artifact Traceability, Behavior-Driven Development.**

I. Introduction

Regression testing [1] plays a key role in software maintenance processes to ensure system reliability, as it evaluates whether new changes do not affect existing components and modules. Recently, there has been a growing interest in improving the quality of a regression test suite [2], [3], [4]. This is because regression testing in complex software systems takes a long time to complete, which can negatively impact continuous integration practices. In addition to its interest in better regression testing techniques, the software industry is investing in behavior-driven development practices [5] that make it easy to specify needs and transform them into executable tests.

Research into better regression testing techniques has received great results by studying methods for minimizing, selecting, or prioritizing test suites based on various artifact dependencies, but there is little work that considers automatically retrieving traces between the artifacts. Furthermore, behavior-driven development practices seem to facilitate the recovery of requirement-to-test and implementation traces. Starting from this assumption, the investigation of BDD artifacts could lead to potential solutions that minimize the gap between the requirements to test cases to code traceability and test case prioritization techniques.

In this paper, it is investigated how behavior-driven development practices could contribute to better test case prioritization techniques that would require less human guidance. Therefore, a solution is proposed to automatically retrieve requirements-tests-code traces from projects using BDD starting from the idea presented in [6]. Then, we further aim to analyze and transform the resulting dependencies so that they can be integrated as part of a dataset for the task of test case prioritization.

Toward the traceability objective, a traceability graph is built starting from the BDD artifacts. Our solution applies a series of Natural Language Processing (NLP) techniques to parse the data into graph nodes. Further, the approach integrates a custom language model for dependencies detection based on cosine similarity.

The achievement of the test case prioritization objective is accomplished through an Artificial Neural Network (ANN) classifier trained on a dataset generated starting from the retrieved traces. Our approach experiments with various network architectures validated against real-world BDD projects, with the end goal of improving fault detection capabilities.

The main contributions of this paper are the following:

- A systematic literature review on the Test Case Prioritization in the context of BDD and CI and regarding approaches for traceability from tests to source code.
- An automatic solution for requirements-tests-code traceability based on data retrieved through natural language processing techniques applied to BDD artifacts.

This work was funded by the Ministry of Research, Innovation, and Digitization, CNCS/CCCDI - UEFISCDI, project number PN-III-P1-1.1-TE2021-0892 within PNCDI III.

- A dataset generation approach that extracts various metrics from the detected dependencies and aggregates them as training and test data for the test case prioritization task.
- Various ANN architectures were built for the prioritization task to validate the fault detection capabilities of our traceability solution.
- A set of experiments based on datasets generated from real-world projects that integrate BDD practices. The experiments include both within-project and cross-project validation.

The work is organized as follows: In Section II we present background concepts on the subject of test case prioritization and review the current state of the art in the context of prioritizing test cases in regression testing. Section III briefly describes our proposed approach, while Section IV reports the experiments carried out on our classification models together with an analysis of the results. Section V outlines the threats to validity, while Section VI states the conclusions and future work.

II. Test Case Prioritization Background and Related work

This section outlines the definition of Test Case Prioritization (TCP) followed by related work on TCP, focusing on existing approaches.

A. Definition of TCP

Graves defines TCP in his paper [7], as follows.

Definition 1. Test case prioritization [7]: Given a test suite, T, the set of permutations of T, PT; a function from PT to real numbers, f.

Problem: to find T' ∈ PT such that

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')] \tag{1}$$

The function f assigns a real value to a permutation of T according to the test adequacy of the particular permutation.

B. Related work on TCP

A Sytematic Literature Review (SLR) was performed following the methodology presented by Kitchenham et al. in [8]. In the following paragraphs, we aim to summarize the information extracted during the SLR, presenting our findings for each of the 13 selected papers (out of an initially larger set). The SLR investigation seeks to answer questions related to current research on TCP in the context of BDD and CI and regarding approaches for traceability from tests to source code.

Analyzing the distribution of the selected publications, we observed that there is an increased interest in the last years, as 10 out of 13 papers have been published since 2019.

The set of selected papers can be further grouped by their main investigated topic. Using this approach, we grouped the papers into two categories, that is, one group that deals with the test case prioritization or optimization task and one group that investigates the traceability task. Moreover, the papers can be clustered based on the testing technique they address, i.e., black box, based only on the test suite and does not require any implementation or source code details, or white box.

Regarding test case prioritization, we identified 9 scientific research papers. The approaches are commonly based on a Greedy strategy [4], [9], on search strategies mainly focused on Genetic Algorithms (GA) [2] [10] [11], on Machine Learning (ML) solutions [12] [13] or on Model-based strategies using Finite State Machines (FSM) [14] [3]. Most of these studies focus on Java projects, both for industry (3) and academic purposes (6). Next, we outline several aspects of each approach.

The work of X. Wang et al. [9] introduces a black-box method to prioritize test cases for regression testing based on the associated requirements and historical fault detection data. For each requirement, a set of priorities are assigned by both the customer and the developers. An initial order of the test cases is then computed on the basis of the priorities of their corresponding requirements. Based on historical execution data and faults detected by each test case, the requirement priorities are adjusted, resulting in a reordering of the test suite. Their approach uses a Greedy strategy to select the test cases with the highest probability of fault detection, i.e. with the highest priority, given an execution budget. The solution was evaluated using the weighted average of the percentage of faults detected (APFD) and the fault detection rate (FDR) against the baseline provided by Random Prioritization. The evaluation was performed on an industrial Java system, obtaining an APFD of 71%.

A Greedy approach is also implemented by Xu et al. in [4], however, applied in the context of behavior-driven development with the selection based on similarities between requirements. The main assumption of the authors is that similar requirement specifications also share similar code structures. Thus, a new requirement will most likely introduce code changes in areas of a project that implement requirements having similar natural language specifications. The approach uses the computed cosine similarity between requirements, which can be further used as a criterion to select the most relevant acceptance tests and their corresponding unit tests that fit into a given budget. To evaluate their technique, the authors analyzed the inclusion and precision in a real industry project. However, the analyzed metrics were computed against human-selected requirements which are project-dependent and may introduce a bias factor.

In black box testing, prioritization based on computed similarities is widely adopted with evolutionary strategies, Cruciani et al. [13] providing such a solution, considered as a baseline in further research. His work introduces a family of solutions for test suite reduction and prioritization, called FAST-R based on similarity measures borrowed from the big data domain. The FAST-R solutions model the problem as a clustering task, using the k-means algorithm [15]. Distances are computed in terms of Euclidean distances, since test cases are represented as points in D-dimensional space. To transform each test case into its corresponding coordinates, the author uses the vector space model by mapping the textual representation of the test to an n-dimensional point according to the term frequencies. The approach was validated against 5 C programs from the Software Infrastructure Repository (SIR) [16] and 5 Java programs from the Defects4J database [17]. To evaluate the solutions, the Fault Detection Loss (FDL), Preparation, and Execution Time were computed based on the results obtained for a given reduction budget ranging from 1% to 30% achieving promising results.

Arrieta et al. [10] also described a suite of strategies, however, focused on seedings for multi-objective selection. Her work proposes three strategies based on randomization integrated with a Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [18]. The seeding strategies, i.e., Static and Dynamic Test Suite Size-based Random Seeding and Adaptive Random Population Generation are integrated as an initialization step with the goal of having higher diversity between the individuals of the initial population. The experiments focused on Simulink models, including industry setups as well. The evaluation was performed by analyzing the statistical significance of the solutions against a non-seeding baseline, resulting in a positive effect on the selection task.

The most recent work on black-box test case selection was introduced by [2] which is based on test case similarity applied in both GA and NSGA-II. The technique uses Abstract Syntax Trees (AST) to model test code and applies tree-based traversing algorithms in computing similarities, i.e., bottom-up, top-down, combined, and tree edit distances. The authors evaluated their method against the previously mentioned FAST-R baseline and random selection technique on 16 Java projects from the Defects4J dataset [17]. The metrics considered focused mainly on the fault detection rate (FDR), preparation, and execution time, achieving an average of 82% within 1.1 to 4.3 average hours of execution.

Although it might provide less scalability due to its need for implementation and source code details, white-box test case prioritization is widely studied. Research focuses on improving test case prioritization techniques and automatically achieving better requirements to code traceability. Besides greedy and search-based solutions, white-box testing also enables the use of model-based techniques built on Finite State Machines (FSM). Both [3] and [14] describe a model-based solution, although their approach is fairly different.

In [14] the authors aim to generate a ranking model using machine learning, starting from a system described as an Extended Finite State Machine (EFSM). To train the model, multiple faulty versions of the system were created by injecting faults into the EFSM. The faulty versions were then fed through a series of heuristic algorithms to generate a set of ranking results that were used as training data. The model was evaluated using APFD on 5 protocol EFSMs obtaining an average of 83%.

On the other hand, the authors of [3] aim to maximize coverage and reduce the set of inputs in the automated testing of mobile apps by implementing a model-based approach that integrates multiple testing strategies. Their solution combines and extends program analysis approaches to generate an extended window transition graph (EWTG) so that it can capture user interface differences and transitions triggered between different versions of applications. The solution was tested in different versions of a set of Android applications and was shown to significantly reduce testing time, that is, around 30% with a coverage of around 60 to 70%.

The work of [12] models test case prioritization as a classification task solved using a neural network classifier. The implemented approach incorporates both static and dynamic data, that is, requirement dependencies, faults discovered, and testing cost. Experiments include neural network architectures and configurations, all of which are analyzed against objects selected from SIR. Performance-wise, the approach was evaluated using the APFD metric and model-specific metrics, achieving a maximum accuracy of 98.96 and an APFD of 18%.

The work of Marchetto et al. [11] details the approach both for the task of prioritization and traceability. Their research introduced a test case prioritization technique that incorporates low- and high-level test case data together with automatically recovered traceability links. The test case data are collected based on a set of source code and requirements metrics (e.g.: size, complexity, cohesion, coupling, maintainability index), while traceability is achieved using Latent Semantic Indexing and similarity computation. All the extracted features are also used in the multi-objective optimization task implementing NSGA-II [18]. The technique was validated on 21 Java projects using the APFD metric, among others, proving it is able to early identify both technical and non-technical relevant defects.

In the task of traceability, the most common approaches are based on information retrieval (IR) techniques and natural language processing (NLP) applied to various code artifacts. Such examples are provided by [6] and [19] that aim to recover traceability through fine-grained relations between requirements and source code (classes, method calls) or between cross-commit artifacts. The work of [19] infers trace links in an unsupervised setting using word embedding similarities. The approach is based on the assumption that requirement and code artifacts express a single cohesive semantics that can be used to recover traces in the following manner: transform artifacts to word embeddings, compute similarities between elements, and aggregate them to recover trace-links. The analysis of results was performed on datasets provided by the Center of Excellence for Software & Systems Traceability (CoEST) [20] using accuracy, precision, recall, and F1 score with a maximum of 49.5%. However, a semantic gap between artifacts still remains but might be reduced by incorporating language models. We provide further details on each of these studies.

Yang [6] addresses the traceability task with respect to behavior-driven development. His work aims to identify traceability links between *.feature* files and source code by analyzing and further predicting changes between multiple commits. To achieve this, the author investigates common keywords and similarities between cross-commit files, then approaches the prediction task as a binary classification problem solved by implementing three techniques. Among these, the best results are obtained using a random forests algorithm when tested against 133 Java projects crawled from GitHub.

In the context of BDD, Fazzolino [21] investigated the potential use of operational profiles to generate trace links to support the prioritization of tests. Operational Profiles (OP) consider the probability of an artifact being executed to provide information about how a system is used. The authors' approach in generating this profile is based on statically and dynamically retrieved information. The retrieved data are classified by artifact, each artifact corresponding to a profile level that can be aggregated into an operational graph. The technique was evaluated on a real-world project, yet the evaluation was mainly empirical.

Lucassen et al. [22] introduced another BDD approach to traceability. Differently from the previously presented techniques, their approach does not make use of IR techniques. The solution presented by the authors implements a Tracer used to build a Traceability Matrix by recording the methods called by each step of a scenario. The proposed matrix relates the code artifacts to a single scenario step, a scenario, and an entire user story or requirement. The approach was evaluated in an open-source project, but the authors do not provide quantitative results.

Our findings, based on the SLR, are provided next in relation to TCP research and traceability. Although we identified state-of-the-art solutions in test case prioritization, implementing various techniques with promising results, *research on this task in the context of BDD or CI is still limited*. However, by analyzing the work presented above, we were able to define a baseline for our research and identify areas of potential improvement. The majority of solutions address the traceability task through information retrieval or natural language processing algorithms focused on semantic analysis, however, with little applicability tested. Despite the fact that new traceability solutions may lack quantitative and practical experiments, throughout the analysis step, we have seen an increasing interest in integrating new traceability solutions into the task of test case prioritization. Our proposal links the requirements to the source code and to the test cases (which are also linked to the faults) based on the data retrieved through natural language processing techniques applied to BDD artifacts, whereas other existing approaches are based on test case similarity by using various methods to model the test code.

III. TCP using Neural Network classification with Artifacts Traceability

The proposed approach for the generation of complete traceability from requirements to faults and test case prioritization based on neural network classification is outlined in this section. First, the automatic traceability graph construction process is presented along with the step descriptions. In addition, the test case prioritization problem is defined, incorporating the discovered traces as features of the neural network classification model. The metrics used for the evaluation of the proposed approaches are also provided.

A. Requirements-to-Code Traceability

The work of [6] introduces the concept of automatic traceability in the context of behavior-driven development (BDD). Our approach builds on this concept, we aim to assess traceability by identifying keywords in *.feature* and source code files and analyzing the similarity between each of them.

Our solution generates a dependency graph to better model dependencies between various components to better model dependence. Each component (requirement, test case, source code, and fault) is represented as a single graph node as part of its corresponding cluster. Dependencies between types are represented as weighted edges between nodes of two different clusters, where the weight describes the similarity score. To produce the final traceability graph, our solution involves the following three steps that may be visualized in Figure 1.

In the *File Type Parsing* step each project file is being parsed based on its corresponding type, i.e. *.feature* file, Java source code, or Java test file, and irrelevant, type-specific keywords are ignored. At the end of this phase, new graph nodes are created, each one containing the relevant information for its type. The exact steps of this phase are illustrated in the first part of Figure 1.

In the *Keyword Identification* step a node clean-up process further trims the node data by applying a series of natural language processing (NLP) techniques. The flow corresponding to this phase is detailed in the second container in Figure 1.

The *Dependency Identification* step represents the phase that creates the edges between the different types of nodes and assigns their weights based on the cosine similarity. This phase is also detailed in the last part of Figure 1.

The File Parsing phase processes only the Gherkin (*.feature*) files and Java (*.java*) source files, the others being considered irrelevant for traceability in the context of regression testing. However, other file types can be easily integrated and processed by adding a new parser as each of the existing ones implements a common interface and has a dedicated processing method. The parsing logic uses a dedicated Python library for each file type, as language-specific words are ignored by default. As a result of the parsing process, each parsed file is transformed into a corresponding graph node. Yet, Gherkin files follow a slightly different logic as they are split into both requirement nodes and test case nodes due to the existence of scenarios, that is, each scenario is considered a different test case, while the description of the feature is considered as part of a requirement.

Given the list of relevant data extracted from the previous step, the identification of keywords aims to further clean the data to achieve a standard format that can be fed to a language model. To achieve this, we apply a series of NLP techniques [23], i.e., tokenization, part-of-speech identification, and lemmatization. Similarly to the solution proposed by Yang et al. [6], we used the pipeline provided by the Stanford CoreNLP toolkit [24] and selected only nouns and

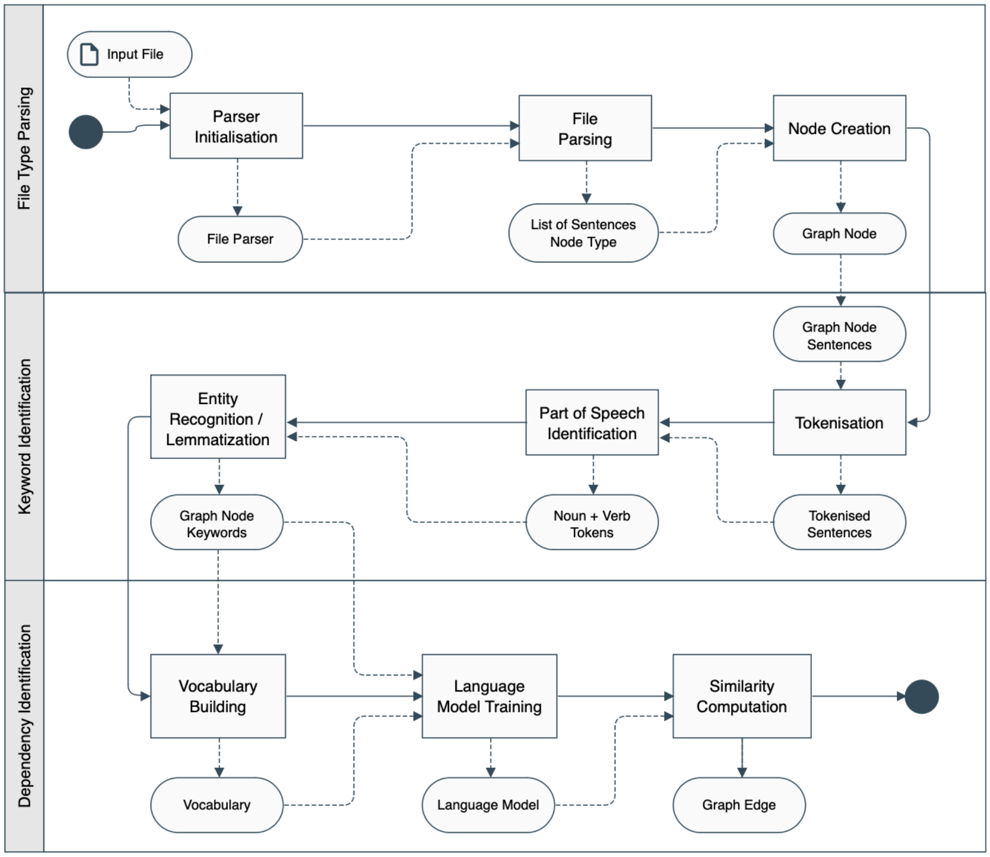


Fig. 1. Automatic Traceability Graph Process - Overview

verbs as relevant attributes. Furthermore, since Gherkin files [5] often include named entities as examples, we also included the named entity recognition step as part of the pipeline.

As the last part of the traceability graph building flow, dependencies between nodes of different types are inferred by computing the cosine similarity. To calculate it, we embedded the keywords of each node through a custom-trained Word2Vec [25] model. We also analyzed the option of using pre-trained models such as those provided by Stanford University in GloVe [26], but decided on custom training considering the large amount of potential unseen words introduced on a project-by-project basis as variable, method, or class names.

Regarding the traceability preservation it is important to mention that since requirements or source code or test cases may be changed, some parts of the mappings should be re-do. In the generated versions, the mapping is the same and only a subset of requirements was considered as changed in the experiments.

B. Test Case Prioritization

The TCP solution is built on top of the work of [12], formalizing prioritization from requirements-to-code dependencies. The TCP task is modeled as a multi-label classification problem. Thus, target labels represent the level of importance of a test case within the reordered test suite, while feature vectors are represented by transforming the dependencies obtained from the traceability module. That is, each test case t_i in a given test suite $T = (t_1, t_2, ..., t_n)$ gets a priority label assigned that is used further as an ordering criterion. In the end, we achieve an ordered test suite T' representing a potential prioritization of T.

1) Features: Our feature collection is based on the results obtained with our traceability module. The following attributes were included as part of the feature vector for each test case t:

- NR_t - number of requirements covered by the test case;
- NC_t - number of source code files covered by the test case;
- NF_t - number of faults detected by the test case;
- NR'_t - number of requirements covered by the test case, given a subset of modified requirements.

As the dependencies resulting from the traceability module are weighted by cosine similarity, a requirement or source code file is considered covered if its corresponding similarity exceeds a given threshold (or confidence level). The threshold for each dependency type was set upon analyzing the incipient results and the type characteristics. Thus, we achieved more granular control over the dependencies and minimize the risk of introducing irrelevant data into the final dataset.

The target labels for each test case are defined upon analyzing the data on the covered requirements, source code, and discovered faults as a weighted sum. Furthermore, the test suite is broken down into three priority classes so that the resulting labels are uniformly distributed.

2) Neural Network Model: Following the work of [12], our TCP solution is modeled as a classification task solved through an artificial neural network (ANN). As a general architecture, the ANN [27] consists of three main parts, each of which contains numerous control units for the computation known as neurons. The three components include the input layer, a set of hidden layers that represent the core of the classification process, and an output layer. Synaptic weights act as a bridge, carrying the input signal from the input layer to the first hidden layer. Activation functions decide whether the value of the weighted sum of inputs in the neurons of the hidden layer should be sent to the next layer.

In our case, the number of units on the input layers corresponds to the number of features considered for a test case, whereas the number of units on the output layer corresponds to the total number of priority classes. In addition, in the pre-processing phase, a normalization layer is integrated into our models. For the neurons of the hidden layers, a ReLu activation mechanism was set up, whereas the output units used a Softmax activation function.

The designed models, illustrated in Table I, are built by variating a series of parameters: number of hidden layers (2,3), number of neurons per hidden layer (40, 60), and the optimization algorithms used in the training phase. For the optimization algorithms, we decided on the Stochastic Gradient Descent (SGD) [28] and the Adam optimizer [28].

TABLE I
DETAILED NEURAL NETWORKS ARCHITECTURES

Model No.	Hidden Layers	Neurons	Optimizer
Model 1	2	40	Adam
Model 2	2	60	Adam
Model 3	2	40	SGD
Model 4	2	60	SGD
Model 5	3	40	Adam
Model 6	3	60	Adam
Model 7	3	40	SGD
Model 8	3	60	SGD

3) Dataset: The performance of both the traceability module and the test case prioritization module was analyzed on the same datasets to better understand the fault detection capabilities of our solution: the *trivial-graph* (https://github.com/akollegger/trivial-graph/tree/master)) and the *springmvc-router* (https://github.com/bclozel/springmvc-router) projects.

The data generation schema is depicted in Figure 2. Our traceability solution initially processed the 2 projects and the resulting files have subsequently been analyzed for feature extraction.

- the *trivial-graph*,
  https://github.com/akollegger/trivial-graph/tree/master
- the *springmvc-router*,
  https://github.com/bclozel/springmvc-router.

Our traceability solution initially processed these projects and the resulting files have subsequently been analyzed for feature extraction and data generation (Figure 2). For the feature extraction part, the dependency files were analyzed, and only the edges having similarity above a defined threshold were selected for further processing. That is, we consider a test case as relevant for a certain requirement or source code file depending on the comparison between its similarity and the threshold.

After defining the major features, i.e. NR_t, NC_t and NF_t, the datasets have been enhanced to simulate several different versions of a system being tested by generating an array of modified requirements, each corresponding to a new version. The version enhancement updated the dataset to include a newly extracted NR'_t feature. The target vectors were defined by manually assigning one of the three priority classes: High (0), Medium (1), and Low (2) such that the final labeling follows a uniform distribution.

The final datasets listed in Table II included 5 versions per dataset, 1 to 3 requirements changed per version, cumulating a total of 135 examples for *trivial-graph*, respectively 222 examples for *springmvc-router*. The generated files for the traceability steps are available in this Figshare link [29].

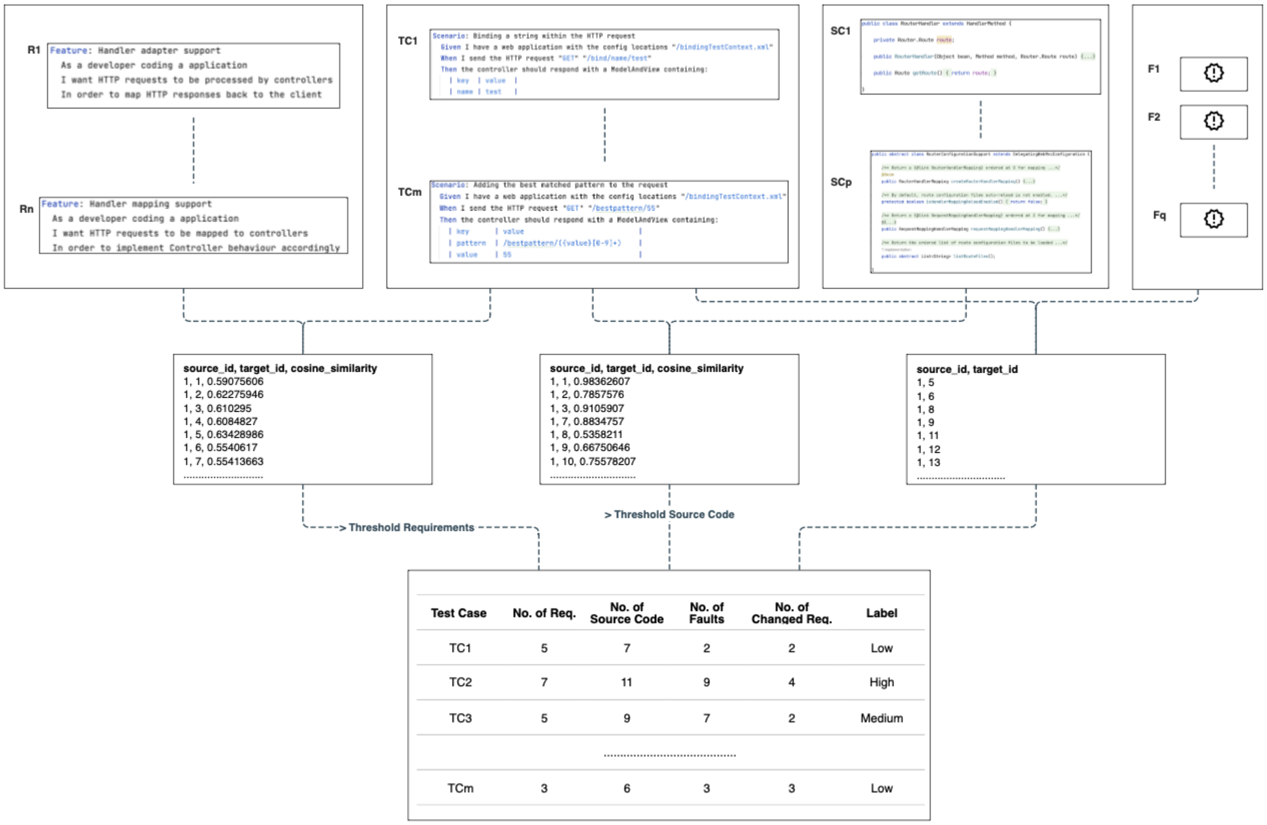
Regarding time execution, no such data was considered in the TCP analysis since were not considered important for the two projects (the time was similar among the executed test cases), thus time (cost) not considered in the APFD computation.

TABLE II
STRUCTURE OF SELECTED DATASETS

Dataset	Req.	Test	Code	Fault	Version	E.g.
trivial-graph	6	31	22	34	5	135
springmvc-router	5	45	14	116	5	222

4) Metrics: The performance of our traceability and prioritization solution was evaluated using both metrics related to neural networks and the Average Percent of Faults Detected (APFD) metric. The model-related metrics we incorporated are Accuracy, Precision, Recall, and F1-Score, each defined as follows (TP = Total Positive, TN = Total Negatives, FP = False Positives, FN = False Negatives):

- Accuracy
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$
- Precision
$$Precision = \frac{TP}{TP + FP}$$
- Recall
$$Recall = \frac{TP}{TP + FN}$$
- F1-Score
$$F1 = \frac{2 * TP}{2 * TP + FP + FN}$$



```
Feature: Handler adapter support
  As a developer coding a application
  I want HTTP requests to be processed by controllers
  In order to map HTTP responses back to the client
```

```
Feature: Handler mapping support
  As a developer coding a application
  I want HTTP requests to be mapped to controllers
  In order to implement Controller behaviour accordingly
```

```
Scenario: Binding a string within the HTTP request
  Given I have a web application with the config locations "/bindingTestContext.xml"
  When I send the HTTP request "GET" "/bind/name/test"
  Then the controller should respond with a ModelAndView containing:
    | key  | value |
    | name | test  |
```

```
Scenario: Adding the best matched pattern to the request
  Given I have a web application with the config locations "/bindingTestContext.xml"
  When I send the HTTP request "GET" "/bestpattern/55"
  Then the controller should respond with a ModelAndView containing:
    | key     | value                          |
    | pattern | /bestpattern/({value}[0-9]+)   |
    | value   | 55                             |
```

```
public class RouterHandler extends HandlerMethod {

    private Router.Route route;

    public RouterHandler(Object bean, Method method, Router.Route route) {...}

    public Route getRoute() { return route; }

}
```

```
public abstract class RouterConfigurationSupport extends DelegatingWebMvcConfiguration {

    /** Return a {@link RouterHandlerMapping} ordered at 0 for mapping ...*/
    @Bean
    public RouterHandlerMapping createRouterHandlerMapping() {...}

    /** By default, route configuration files auto-reload is not enabled. ...*/
    protected boolean isHandlerMappingReloadEnabled() { return false; }

    /** Return a {@link RequestMappingHandlerMapping} ordered at 1 for mapping ...*/
    @{...}
    public RequestMappingHandlerMapping requestMappingHandlerMapping() {...}

    /** Return the ordered list of route configuration files to be loaded ...*/
    1 implementation
    public abstract List<String> listRouteFiles();

}
```

```
source_id, target_id, cosine_similarity
1, 1, 0.59075606
1, 2, 0.62275946
1, 3, 0.610295
1, 4, 0.6084827
1, 5, 0.63428986
1, 6, 0.5540617
1, 7, 0.55413663
............................
```

```
source_id, target_id, cosine_similarity
1, 1, 0.98362607
1, 2, 0.7857576
1, 3, 0.9105907
1, 7, 0.8834757
1, 8, 0.5358211
1, 9, 0.66750646
1, 10, 0.75578207
..............................
```

```
source_id, target_id
1, 5
1, 6
1, 8
1, 9
1, 11
1, 12
1, 13
..............................
```

Test Case	No. of Req.	No. of Source Code	No. of Faults	No. of Changed Req.	Label
TC1	5	7	2	2	Low
TC2	7	11	9	4	High
TC3	5	9	7	2	Medium
		....			
TCm	3	6	3	3	Low

Fig. 2. Dataset Generation Process - Overview

The performance of our solution in comparison to the Random prioritization technique chosen as a baseline was examined using the APFD metric. Unlike the model performance metrics which can be applied to any collection of labeled test cases, The APFD must be applied to the entire rearranged test suite. The metric is defined as

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{m^2} + \frac{1}{2 * n}$$

where *m* represents the total number of faults, *n* represents the total number of test cases and TF_i represents the position of the test case that discovered the fault *i* within the re-ordered test suite.

IV. Experiements and Results

This section presents the results of our prioritization approach together with the details of the designed experiments. The experimental setup is first described, followed by the results grouped based on the training and test datasets. This section also illustrates a comparison between the various models considering both the model performance metrics and the APDF against a baseline prioritization technique.

A. Experimental Setup

The focus directions of the designed experiments are as follows:

i) observing the performances of the solution when used across multiple versions within the same project;

ii) observing the performances in a simulated real-world environment where the solution is trained on a sample set of projects and applied to new, unrelated data.

Therefore, both within-project validation and cross-project validation were considered in the experiments. Additionally, all the 8 ANN architectures described in Section III were included when running the experiments.

Our setup uses the hold-out method for within-project validation, with a 80%-20% train-test split by version to ensure a uniform distribution of data. We opted for this splitting method, as it simulated the real-world need for a test suite rerun once new requirements or source code changes occur.

For cross-project validation, the experiments included both datasets as training and test data such that the model is trained on an entire dataset and validated on several sampled versions of the remaining one. Therefore, the experiments included two variations: *i)* training on *trivial-graph* and testing on *springmvc-router* and *ii)* training on *springmvc-router* and testing on *trivial-graph*.

A random prioritization technique is also included as a baseline for comparison. Random permutations have been calculated at each successive experiment, and the result has been computed from a sample of 10 randomly selected configurations.

B. Within-Project Experiments

Next, the results of within-project experiments for both projects are provided.

1) Project springmvc-router: Table III lists the performance metrics that each model achieved using a 80% - 20% split of the *springmvc-router* dataset. As we can see, Model 1, which has two hidden layers with 40 units each and is set up to use the Adam optimizer, is our best-performing model.

TABLE III
HOLD-OUT VALIDATION RESULTS ON *springmvc-router* DATASET

Model No.	Accuracy	Precision	Recall	F1 Score
Model 1	**86.71**	90.75	**86.71**	**85.29**
Model 2	82.23	90.8	82.23	79.53
Model 3	67.78	77.52	67.78	64.38
Model 4	72.22	83.74	72.22	69.72
Model 5	84.45	**90.88**	84.45	81.05
Model 6	84.45	**90.88**	84.45	81.05
Model 7	78.89	84.74	78.89	73.23
Model 8	64.65	62.58	64.65	58.55

We have calculated the APFD for a test suite based on our best-performing model, with regard to performance against Random Prioritization Baseline. A mean APFD of 79.5% has been obtained by reordering a test set, while the random technique achieved an overall APFD of 43.33%.

2) Project trivial-graph: The performance metrics our models achieved on a 80%-20% split for the *trivial-graph* dataset are presented in Table IV. All the models configured to use the Adam optimizer outperform the models configured on SGD, although the best results were achieved by Model 1, i.e. a model using 2 hidden layers with 40 units.

TABLE IV
HOLD-OUT VALIDATION RESULTS ON *trivial-graph* DATASET

Model No.	Accuracy	Precision	Recall	F1 Score
Model 1	**98.38**	**98.53**	**98.38**	**98.39**
Model 2	95.96	97.02	95.69	95.65
Model 3	61.29	48.68	61.29	53.94
Model 4	67.74	50.61	67.74	57.33
Model 5	96.77	97.58	96.77	96.77
Model 6	95.48	97.45	95.48	95.18
Model 7	67.74	50.61	67.74	57.33
Model 8	64.51	47.25	64.51	54.29

As the Random obtained an APFD of 41,74%, the differences between the best-performing model and the Random permutation are less significant compared to the previous dataset, although it achieved an APFD of 42.85%.

C. Cross-Project Experiments

Next, the results of cross-project experiments for both projects are provided.

1) Train on trivial-graph, test on springmvc-router: Table V illustrates the results we obtained on *springmvc-router* dataset when the *trivial-graph* project was used for training. Most SGDs are performing better than the Adam configured models, compared to other experiments conducted on the *springmvc-router*. The best results are achieved on Model 4 and Model 8 using 60 unit layers, with 2 or 3 hidden layers.

TABLE V
CROSS-PROJECT VALIDATION RESULTS
TRAINED ON *trivial-graph*, VALIDATED ON *springmvc-router*

Model No.	Accuracy	Precision	Recall	F1 Score
Model 1	55.56	42.07	55.56	44.95
Model 2	55.56	38.48	55.55	45.40
Model 3	60.00	46.26	60.00	49.55
Model 4	**62.23**	44.89	**62.23**	51.8
Model 5	55.55	38.27	55.55	45.32
Model 6	55.56	38.83	55.56	45.53
Model 7	40.00	27.67	40.00	32.71
Model 8	62.22	**51.18**	62.22	**53.15**

The best-performing model achieved an APFD of 65.6% outperforming the Random configuration that scored an average of 43.33%, although it obtained a lower APFD than the models trained within the same project (79.5%).

2) Train on springmvc-router, test on trivial-graph: The results obtained on the *trivial-graph* after training the models on the *springmvc-router* are shown in Table VI. Models 2 and 5 obtained the best performances, i.e. architectures with 60 neurons, configured to use the Adam optimizer, although no significant differences were observed when compared to SGD models.

TABLE VI
CROSS-PROJECT VALIDATION RESULTS
TRAINED ON *springmvc-router*, VALIDATED ON *trivial-graph*

Model No.	Accuracy	Precision	Recall	F1 Score
Model 1	67.74	53.54	67.74	58.87
Model 2	**70.96**	**85.48**	**70.96**	**67.09**
Model 3	67.74	53.54	67.74	58.87
Model 4	64.51	52.63	64.51	57.23
Model 5	**70.96**	**85.48**	**70.96**	**67.09**
Model 6	67.74	84.71	67.74	61.98
Model 7	64.51	52.63	64.5	57.23
Model 8	**70.96**	83.4	**70.96**	66.3

Regarding fault detection capabilities, the APFD of the best-performing model (42. 62%) slightly exceeded the Random prioritization technique (41.74%). Moreover, the obtained APFD is close to the results of the experiment carried out within-project.

Both the within-project and cross-project experiments illustrated robust performances of our proposed models. The SGD optimization method proves to be more suitable for industrial use, as the real-world environment comes closer to our cross-project setup. Furthermore, our solution outperformed the Random prioritization technique, indicating that the traceability component can be efficiently used as a part of a test case prioritization solution.

V. Threats to validity

A number of factors, which could influence the validity of the results are inherent in our approach to this problem, as with all other solutions. Two different categories can be used to identify these factors: internal validity which relates to any potential bias that may have occurred in the study and external validity which deals with the ease of generalization of results.

To minimize internal validity threats, the datasets used for validation were labeled and further split to ensure uniform distribution. However, the empirical evaluation of the traceability module, together with the empirical selection of the dependency confidence level, and the further manual labeling process, could influence the validity of our results. To mitigate the impact, an enhancement of both the traceability module and the TCP technique will be considered in a forthcoming study so that the proposed solution can be validated against standard datasets for such tasks.

The two datasets we used to validate our approach might have an impact on the external validity of our solution. However, a series of different selection criteria were applied when selecting the real-world projects used for validation to try to minimize this influence. In addition, we will look at the possibility of adding new features and generalizing the traceability module to allow it to be applied to projects that are not currently integrating BDD, such as larger-scale industry projects and standard TCP datasets.

VI. Conclusions

In this paper, a test case prioritization approach is proposed that incorporates a traceability module to automatically retrieve requirements-to-code traces. The traceability component was developed in the context of behavior-driven development, but we plan to add support for other different approaches as well.

The traceability solution achieved promising results after validation in two real-world BDD projects. Furthermore, the results and implications of traceability were analyzed by integrating the retrieved traces as training data for various ANN architectures designed for the prioritization of test cases. This analysis consisted of investigating a set of model-related and fault detection metrics through multiple experiments, indicating better reliability of our approach compared to a baseline prioritization technique.

In subsequent work, our aim is to validate the effectiveness of our solution on larger-scale data in order to make a better assessment of their efficiency. Furthermore, we are planning on independently testing traceability and test case prioritization components on more metrics and different features for better understanding and improving their fault-detection capabilities.

Acknowledgment

This work was funded by the Ministry of Research, Innovation, and Digitization, CNCS/CCCDI - UEFISCDI, project number PN-III-P1-1.1-TE2021-0892 within PNCDI III.

References

[1] P. Ammann and J. Offutt, Introduction to Software Testing, 2nd ed. Cambridge University Press, 2016.

[2] R. Pan, T. A. Ghaleb, and L. Briand, "Atm: Black-box test case minimization based on test code similarity and evolutionary search," 2022. [Online]. Available: https://arxiv.org/abs/2210.16269

[3] C. D. Ngo, F. Pastore, and L. Briand, "Automated, cost-effective, and update-driven app testing," ACM Trans. Softw. Eng. Methodol., vol. 31, no. 4, jul 2022. [Online]. Available: https://doi.org/10.1145/3502297

[4] J. Xu, Q. Du, and X. Li, "A requirement-based regression test selection technique in behavior-driven development," in 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), 2021, pp. 1303–1308.

[5] J. Smart, BDD in Action: Behavior-driven development for the whole software lifecycle. Manning, 2014. [Online]. Available: https://books.google.ro/books?id=2BGxngEACAAJ

[6] A. Z. H. Yang, D. A. da Costa, and Y. Zou, "Predicting co-changes between functionality specifications and source code in behavior driven development," in Proceedings of the 16th International Conference on Mining Software Repositories, ser. MSR '19. IEEE Press, 2019, p. 534–544. [Online]. Available: https://doi.org/10.1109/MSR.2019.00080

[7] T. Graves, M. Harrold, J. Kim, A. Porters, and G. Rothermel, "An empirical study of regression test selection techniques," in Proceedings of the 20th International Conference on Software Engineering, 1998, pp. 188–197.

[8] B. A. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University and Durham University Joint Report, Tech. Rep. EBSE 2007-001, 07 2007. [Online]. Available: https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf

[9] X. Wang and H. Zeng, "History-based dynamic test case prioritization for requirement properties in regression testing," in Proceedings of the International Workshop on Continuous Software Evolution and Delivery, ser. CSED '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 41–47. [Online]. Available: https://doi.org/10.1145/2896941.2896949

[10] A. Arrieta, J. A. Agirre, and G. Sagardui, "Seeding strategies for multi-objective test case selection: An application on simulation-based testing," in Proceedings of the 2020 Genetic and Evolutionary Computation Conference, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1222–1231. [Online]. Available: https://doi.org/10.1145/3377930.3389810

[11] 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, 2016.

[12] C.-M. Tiutin and A. Vescan, "Test case prioritization based on neural networks classification," in Proceedings of the 2nd ACM International Workshop on AI and Software Testing/Analysis, ser. AISTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 9–16. [Online]. Available: https://doi.org/10.1145/3536168.3543300

[13] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 419–429.

[14] Y. Huang, T. Shu, and Z. Ding, "A learn-to-rank method for model-based regression test case prioritization," IEEE Access, vol. 9, pp. 16 365–16 382, 2021.

[15] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[16] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empirical Software Engineering, vol. 10, pp. 405–435, 10 2005.

[17] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[18] 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.

[19] T. Hey, F. Chen, S. Weigelt, and W. F. Tichy, "Improving traceability link recovery using fine-grained requirements-to-code relations," in 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 12–22.

[20] "Center of excellence for software & systems traceability (coest)." [Online]. Available: http://www.coest.org/

[21] R. Fazzolino and G. N. Rodrigues, "Feature-trace: Generating operational profile and supporting testing prioritization from bdd features," ser. SBES '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 332–336. [Online]. Available: https://doi.org/10.1145/3350768.3350781

[22] G. Lucassen, F. Dalpiaz, J. M. E. van der Werf, S. Brinkkemper, and D. Zowghi, "Behavior-driven requirements traceability via automated acceptance tests," in 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), 2017, pp. 431–434.

[23] D. Jurafsky and J. Martin, Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 02 2008, vol. 2.

[24] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in Association for Computational Linguistics (ACL) System Demonstrations, 2014, pp. 55–60. [Online]. Available: http://www.aclweb.org/anthology/P/P14/P14-5010

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[26] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162

[27] S. Russel and P. Norvig, Artificial intelligence: a modern approach. Englewood Cliffs, N.J. : Prentice Hall, 1995.

[28] S. Ruder, "An overview of gradient descent optimization algorithms," 2017.

[29] I.-C. Rotaru and A. Vescan, "Test case prioritization based on neural network classification with artifacts traceability," figshare, 2023. [Online]. Available: https://doi.org/10.6084/m9.figshare.23769129.v1