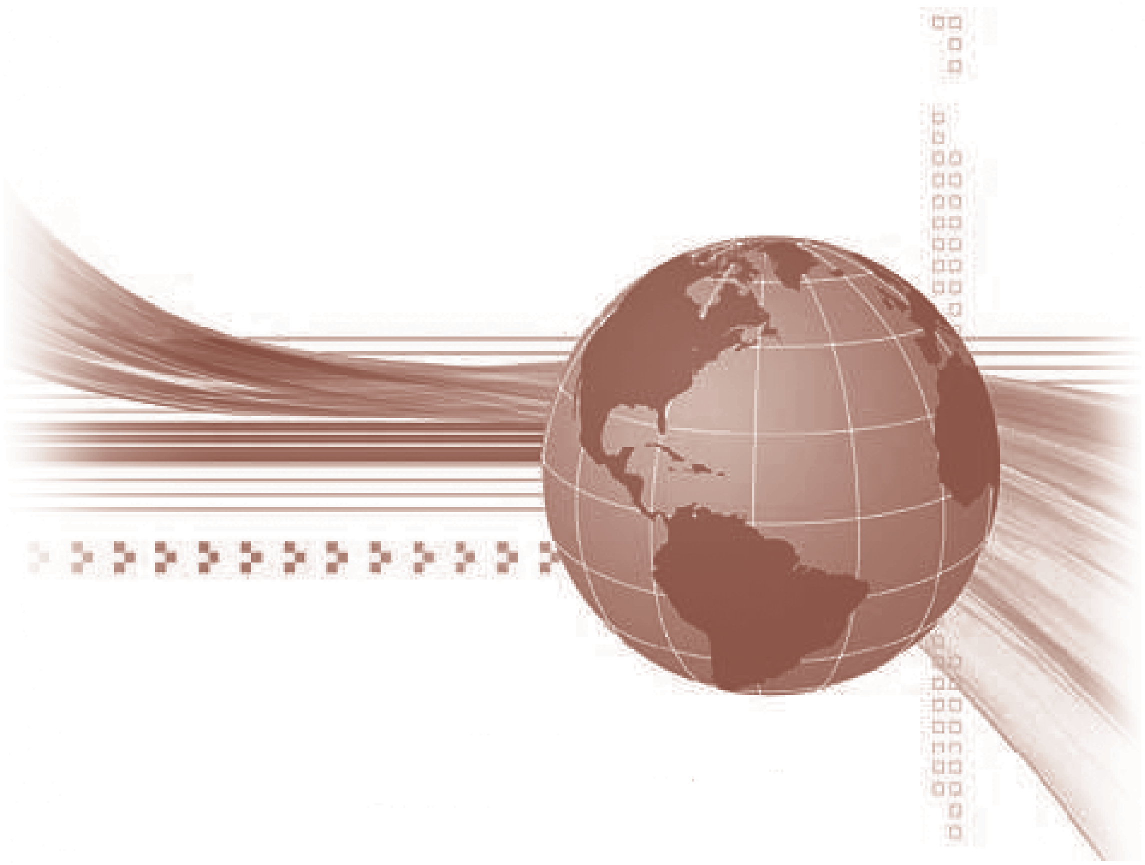
STUDIA UNIVERSITATIS
BABEŞ-BOLYAI

UNIVERSITATEA BABEŞ-BOLYAI
BABEŞ-BOLYAI TUDOMÁNYEGYETEM
BABEŞ-BOLYAI UNIVERSITÄT
BABEŞ-BOLYAI UNIVERSITY
TRADITIO ET EXCELLENTIA



INFORMATICA

2/2023

STUDIA

UNIVERSITATIS BABEŞ-BOLYAI
INFORMATICA

No. 2/2023

July - December

ISSN (online): 2065-9601; ISSN-L: 2065-9601

EDITORIAL BOARD

YEAR **Volume 68 (LXVIII) 2023**
MONTH **DECEMBER**
ISSUE **2**

S T U D I A
UNIVERSITATIS BABEȘ-BOLYAI
INFORMATICA

2

EDITORIAL OFFICE: M. Kogălniceanu 1 • 400084 Cluj-Napoca • Tel: 0264.405300

SUMAR – CONTENTS – SOMMAIRE

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 2, 2023
DOI: 10.24193/subbi.2023.2.01

DEOBFUSCATING JAVASCRIPT CODE USING CHARACTER-BASED TOKENIZATION

ALEXANDRU-GABRIEL SÎRBU

Abstract. The JavaScript code deployed goes through the process of minification, in which variables are renamed using single character names and spaces are removed in order for the files to have a smaller size, thus loading faster. Because of this, the code becomes unintelligible, making it harder to be analyzed manually. Since JavaScript experts can understand it, machine learning approaches to deobfuscate the minified file are possible. Thus, we propose a technique that finds a fitting name for each obfuscated variable, which is both intuitive and meaningful based on the usage of that variable, based on a Sequence-to-Sequence model, which generates the name character by character to cover all the possible variable names. The proposed approach achieves an average exact name generation accuracy of 70.53%, outperforming the state-of-the-art by 12%.

1. Introduction

Over the years, naming variables have proven to be one of the most challenging steps during programming that developers face. Choosing a poor variable name decreases the readability and understanding of the code, since their purpose and meaning is not reflected directly by the label assigned [3]. Thus, programmers communicate their intentions via suggestive names, which can serve as a form of documentation within the code itself, helping other developers understand the code without relying heavily on external comments or documentation.

JavaScript is a widely used programming language, primarily used for web development. JavaScript code is typically embedded directly into HTML documents or included as an external script, running client-side, meaning that

Received by the editors: 31 July 2023.

2010 *Mathematics Subject Classification.* 68T05, 68T50.

1998 *CR Categories and Descriptors.* I.2.6 [**Learning**]: Subtopic – *Connectionism and neural nets*; I.2.7 [**Natural Language Processing**]: Subtopic – *Language generation.*

Key words and phrases. JavaScript deobfuscation, variable name prediction, Deep Learning, Recurrent Neural Network, Abstract Syntax Tree.

it executes in the user's web browser, enabling dynamic content and interactivity without requiring server-side processing. Since JavaScript is a scripting language, meaning that the its code is interpreted, rather than compiled, and because the JavaScript code is ran directly into the user's web browser, this code is visible directly in the web browser, thus allowing users to visualize, analyze it and possibly learn from it.

In order to make the JavaScript code more difficult to understand, developers opt to modify the source code in order to make it less readable, while preserving its functionality. Usually, these methods are based on a mapping from the initial variable and function names to short, arbitrary, non-meaningful identifier names, by using code minification tools, mapping which is available only to the developers of the initial JavaScript code. As the name of the tools' type suggests, the main idea behind such a tool is to reduce the size of the JavaScript file, in order to reduce its loading time [13], thus increasing the performance of the web page, while also providing a layer of code obfuscation.

While code obfuscation is often used on the web pages in order to protect the intellectual property of the code, sometimes its deobfuscation is crucial. For example, deobfuscation can be valuable for security analysis and vulnerability assessment, since obfuscated code can hide potential security risks. Thus, by deobfuscating the code, security professionals can more easily identify and understand the underlying security issues, enabling them to assess the risks accurately and propose appropriate mitigation strategies. Moreover, deobfuscation can also be used in the educational process, allowing developers to learn from and understand obfuscated code. In general, experienced programmers can easily understand obfuscated code, but those who lack the experience require deobfuscation tools in order to gain some insights into advanced programming techniques and algorithms.

Although there are multiple deobfuscation techniques, such as Cloning, Static Path Feasibility Analysis and a combination between Static and Dynamic Analyses [14], our main focus will be to rename the obfuscated variables in order to facilitate the analysis of JavaScript files.

In this paper we propose a deep learning approach to deobfuscate JavaScript source files, which reverses the process of code minification by inferring meaningful variable names based on both their initial assignment and their further usages, using character-based tokenization. A Sequence-to-Sequence model will be used for generating the name character by character to cover all the possible variable names. Thus, our trained model does not rely on the labels that it has been initially trained on, offering a solution which gives decent naming suggestions even on unseen training data, while also being able to suggest better names for constant variables. Experiments will be conducted on a data set containing real world JavaScript code, and the results obtained by our model will be compared to state-of-the-art approaches.

In short, the work aims to answer the following research questions:

RQ1. How to correctly pick a name for a variable using a deep learning approach, without encountering it beforehand in the training data?
RQ2. How would a generative model compare as opposed to a simple, classification model for the problem of JavaScript code deobfuscation?

The rest of the paper is organized as follows. Section 2 will discuss previous ways of generating variable names, including state-of-the-art techniques and how well they performed. Section 3 will present our approach of choosing the best suited variable name, by using a Sequence-to-Sequence architecture, how the data is handled and how to evaluate our constructed model. Section 4 directly compares our approach to the state-of-the-art approach in the literature, while also discussing our achieved results. The conclusions of the paper and directions for future work are outlined in Section 5.

2. Related work

A deep learning approach is proposed by Bavishi et al. [1], which is compared to two state-of-the-art tools JSNice [11] and JSNaughty [15], against a large corpus of real-world JavaScript code, achieving 47.5% name prediction accuracy, outperforming or performing really close to the tools aforementioned. This approach tokenizes the JavaScript code and uses the context of a variable in order to provide a fitting name for it. Thus, for each occurrence of a local variable, it extracts the q preceding tokens and the q following tokens, concatenating them into the context of that respective variable. Since this approach generates highly redundant usage summaries, since sub-sequences of tokens occur repeatedly, an auto-encoder is used in order to compress the given vector. For prediction, a predefined vocabulary of possible variable names is constructed, and the the authors use a Recurrent Neural Network with a single Long Short-Term Memory layer in order to learn a mapping from the variable context to the variable name. The Recurrent Neural Network is used in order to solve conflicts between variable names: if the predicted variable name is a keyword or if it overshadows a name from its parent scope, then another name prediction is made.

In order to improve the performance of code related tasks, Roziere et al. [12] introduced a new pre-training objective based on deobfuscation and outperforms Masked Language Modeling objectives, such as BERT, on tasks such as code search, code summarization and unsupervised code translation, besides deobfuscating fully obfuscated source files. The pre-training objective used is represented by replacing class, function and variable names with special tokens, after which the model has to recover the original names, similarly to how Masked Language Modelling's objective is to predict a word based on its context. The deobfuscation objective is realized with a seq2seq model, which is trained to map the obfuscated code into a dictionary represented as a sequence of tokens. This model manages to recover 45.6% of the initial identifier names on the Google BigQuery data set. By solving the task of deobfuscation, the authors showed that this pre-trained model achieves better performances than the BERT model on clone detection, code summarization, natural language to code and on code translation from Python to Java and vice-versa.

The problem of variable name generation also arises when copy-pasting code, and the copied code has to be modified in order to match the context into which it was pasted. Liu et al. [9] have discussed this problem of code adaptation, whose importance rises from the adaptation bugs, raised by inconsistent control flow, inconsistent renaming, inconsistent data flow and redundant operations. To solve this task, the authors collect a data set from GitHub repositories with at least 20 stars. Their goal is to compare their approach with various pre-training objective introduced for code related tasks. Thus, the authors compared three Masked Language Modeling approaches by training a RoBERTa model, the aforementioned model in the previous approach and a CodeT5 model, which is supposed to outperform prior methods on understanding tasks such as code defect detection, clone detection and translation tasks. The model the authors proposed is a transformer with two possible implementations: a uni-decoder transformer, which names a variable based on previously predicted names, and a parallel-decoder transformer, which calculates the individual probability without taking into consideration the other predicted symbols. The second type of transformers predicts a name independently from the rest, factorizing the output distribution per-symbol. The results are somehow predictable, as the uni-decoder transformer achieves higher performance than the pre-trained objective-based models, and has slightly better results than the parallel-decoder transformer.

Jaffe et al. [6] have approached this problem in the form of assigning meaningful variable names for decompiled code. Their solution is based on aligning the line-by-line translation of the decompiled code into meaningful code, using a Statistical Machine Translation, Moses. In decompiled code, variables are automatically given general names, such as $v1$ and $v2$, and Statistical Machine Translation model should rename these variables, while keeping the rest of the code identical. The main idea behind such a model is that the model tries to learn the probability distribution of a sentence in target language to be the translation of a sentence in the source language. Moses is an open-source Statistical Machine Translation toolkit, which automatically estimates the language and translation models given a sentence-aligned parallel corpus, which, in our case, is the decompiled code against the initial pre-compiled code, which contains meaningful names for variables. This approach achieved a 28.6% exact match for the name of each variable.

3. Methodology

This section introduces our methodology for deobfuscating JavaScript code, using a Sequence-to-Sequence model in order to generate the variable names character-by-character, based on their initial value and their usages, with the goal of answering RQ1.

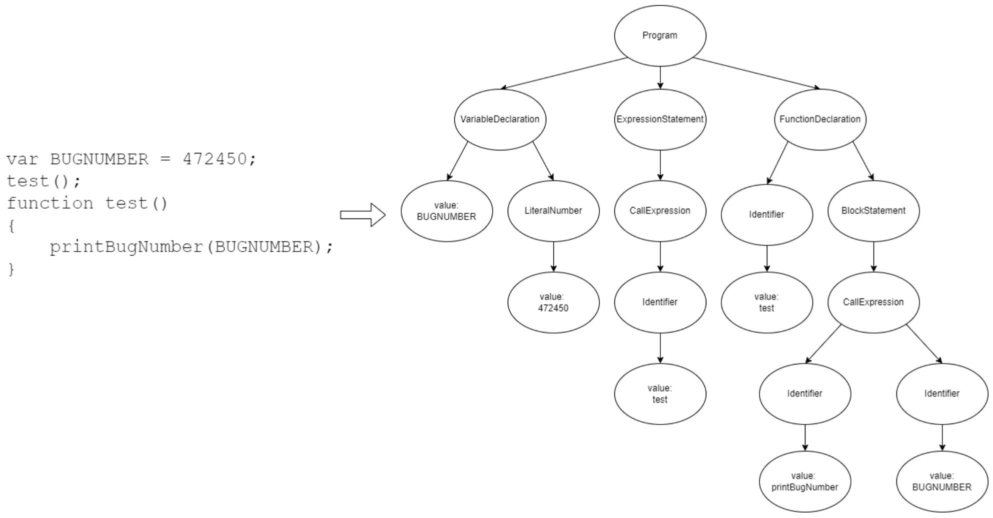
3.1. **Problem statement.**

Our problem can be formulated as follows: given a JavaScript file, rename a variable in such a way that its new name reflects intuitively its purpose. For this task, we will convert the JavaScript code into an Abstract Syntax Tree, after which we will extract that variable's assignment and usages and, based on that, the model will suggest a fitting name for it.

The data set for this problem can be constructed directly based on any available source code, be it online or offline, depending if the target is to write more general code or specialized code in some issue, respectively. Once the code has been selected, there are various tools, such as UglifyJS which, although are intended to be used as minification tools, they usually obfuscate variables to reduce the size of the JavaScript file, making them load faster in order to improving a website's performance. Thus, UglifyJS renames variables and function names to shorter, often single-letter names, removes unnecessary white spaces and comments, and perform other transformations to make the code more compact, without altering its functionality. Since we are later converting the obfuscated code directly into an Abstract Syntax Tree, none of the other operations done by UglifyJS should impact the information we gain through that conversion.

3.2. **Proposed approach.**

The first step in our approach is to convert the given code into an Abstract Syntax Tree. An Abstract Syntax Tree is a tree-like data structure that represents the abstract syntactic structure of a program, specific to a programming language, which are commonly used when constructing compilers and when analyzing code. This tree representation, when compared to the simple string interpretation of the input, provides a more structured representation of the code, that captures the hierarchical relationships between different elements of the code, such as functions and loops.



```
var BUGNUMBER = 472450;
test();
function test()
{
    printBugNumber(BUGNUMBER);
}
```

FIGURE 1. JavaScript code example converted into an Abstract Syntax Tree format

Thus, a machine learning algorithm would better understand the code's logical flow and dependencies, which would make it easier to extract meaningful patterns and features. Moreover, Abstract Syntax Trees typically have a lower dimensionality than raw text strings, especially for complex code, whose reduction can lead to both faster training times and better model performance due to reduced complexity [17]. An example of a JavaScript code conversion from plain-text code into an Abstract Syntax Tree format can be seen in Figure 1, where a variable is assigned, a function is called, after which that function is declared.

Usually, when naming a variable, choosing an appropriate and meaningful name is essential for writing clear, maintainable and understandable code. Thus, the chosen name has to clearly describe the meaning and purpose of the variable. The meaning is usually represented by the type of that variable, usually inferred by its initialization. As for the purpose of the variable, it can be inferred directly based on that variable's usages. Programmers tend to rename variables when the purpose of those variables is changed, or if their initial type is completely different. As for experts in code deobfuscation, they can intuitively guess a variable's purpose based on its usages, but still choose to rename them in order to aid them for further deobfuscation. Because of this, we will extract from the constructed Abstract Syntax Tree the initialization of our current variable, and its usages i.e., lines of code where the variable has been used, where the initial variable name is replaced by a special token.

3.2.1. *Deep learning model.* In our task, since we will generate the sequence of characters that determine the name of the variable, we will use a Sequence-to-Sequence model. This model is a type of deep learning model, which is composed on an Encoder and a Decoder. The Encoder is an Recurrent Neural Network which processes the input sequence and generates a fixed-length context vector, also known as the encoding. The Encoder works by reading the input sequence step by step and encoding the information into a context vector, aiming to capture the semantic representation of the input sequence. The second component, the Decoder, is also a Recurrent Neural Network, which takes the context vector produced by the encoder as its initial hidden state, then it generates the output sequence step by step, one token at a time. The Sequence-to-Sequence model architecture allows to handle input and output sequences of different lengths, by compressing the variable-length input into a fixed-length context vector, after which the output sequence is generated token by token based on that context vector.

In our approach, variable names and values are encoded in character-level, in order to accommodate for new values, unseen in our training data, to be handled correctly by the model constructed [10]. Moreover, using a character-level encoding, it is possible to capture sub-word information of, for example, a class name and its characteristics, and may allow the model to understand prefixes, suffixes and stems, thus giving the model the ability to understand word inflections and grammatical variations better. Another advantage is represented by the removal of noise in the form of typos, and the model may recognize similar words based on their character-level similarity, even if there might be some minor spelling differences, while also being more memory efficient because the vocabulary size is substantially reduced. When encoding Abstract Syntax Tree Nodes, we will be liniarizing each node using Breadth First Search in order to maintain the structural equivalence, while also keeping the model relatively more lightweight [5]. Thus, each node type will have a specific label, which will be stored in the vocabulary specific to the JavaScript code, alongside the ASCII characters for the value of these nodes.

The proposed model will follow the Sequence-to-Sequence architecture, which is composed of two components: the Encoder and the Decoder, whose architecture can be seen in Figure 2. The Encoder receives the input tokenized and converts it into a more dense and continuous representation, via the Embedding layer, whose purpose is to capture the semantic relationships between tokens. As a regularization technique, a Dropout will be used, in order to prevent overfitting and to improve the generalization of the model. The Gated Recurrent Unit, which computes the hidden state of the input and forwards it to the decoder as a context, by using both the output of the Decoder and

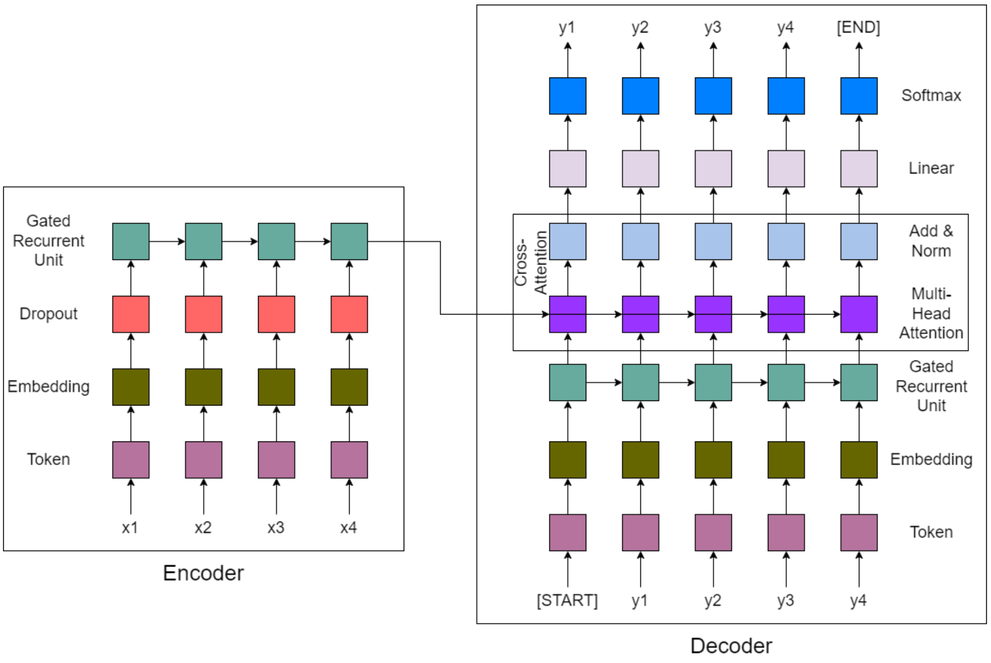


FIGURE 2. Sequence-to-Sequence deep learning model architecture

the output of previous Gated Recurrent Units. The Decoder follows a similar data flow, but this time, the previous output result will be the input to the Embeddings layer, forwarded to a Gated Recurrent Unit. After that, the output of the Gated Recurrent Unit, together with the context resulted from the Encoder will go through a Cross Attention layer, whose purpose is to allow the Decoder to focus on relevant parts of the source sequence, while generating each word of the target sequence [16]. A further processing of the results is done through the Linear layer, and a Softmax layer extracts the best next character for our resulting variable name.

The formula based on which the character at position t is generated is given by:

$$c_t = softmax(f_{seq2seq}([init : usages], s_{t-1})),$$

where $f_{seq2seq}$ is the function the Sequence-to-Sequence model tries to approximate, $init$ represents the embedding of the initialization of the variable, $usages$ is the embedding of the usages of that variable, [:] denotes vector concatenation, and s_{t-1} is the previous hidden decoder step.

3.2.2. *Performance evaluation.* For evaluating our model's performance, we will apply k-fold cross-Validation, splitting the data set into a two components: one for training, and the other for validation and testing. The latter component will be split in half, resulting in 80% of the data set being used for training, 10% for validation and the other 10% for testing, when choosing $k = 5$. Thus, we will be able to give a proper confidence interval, which should give a better performance measure grasp over the data set used [4].

For our task, two evaluation metrics will be used on a testing data set: one evaluation metric which computes the accuracy of each character prediction, since we are using Recurrent Neural Networks, and another evaluation metric which scores the accuracy of each name predicted. The second metric helps us compare to other approaches, in order to see how well the model proposed by us fares against the other approaches proposed.

4. Experimental results

This section presents the experimental results obtained by evaluating the performance of the approach introduced in Section 2 for deobfuscating JavaScript code. In addition, a direct comparison to Context2Name [1], JSNice [11] and JSNaughty [15] is conducted, in order to answer RQ2. Section 4.1 will describe the data set that we are working on, then the experimental setup and the parameters employed for the deep learning model are presented in Section 4.2. A discussion on the obtained results and a comparison to related work is further conducted in Section 4.3.

4.1. **Data set.**

At this step we will construct a data set similar to the one described by Bavishi et al. [1], to be able to compare our model with the state-of-the-art. Thus, from all the files from a publicly available data set[1] of JavaScript programs, which contains 150 thousands non-minified JavaScript files, the duplicate files, the ones very large and the ones that cannot be processed will be removed. After that removal, 97979 files remain which contain 2667804 total variables and 239007 unique names. The minification of JavaScript code is done using UglifyJS, which reduces the file size of those files by renaming variables and removing spaces.

As it can be seen in Figure 3, the most frequent names that the variables have are those with a more generic value, such as *len* or *length*, which usually describe the size of an object or array, *result*, which is usually the returned variable from a function call or operation, and *self*, which referred to the current browser window. There are also many variables names that are either

[1]https://www.sri.inf.ethz.ch/js150

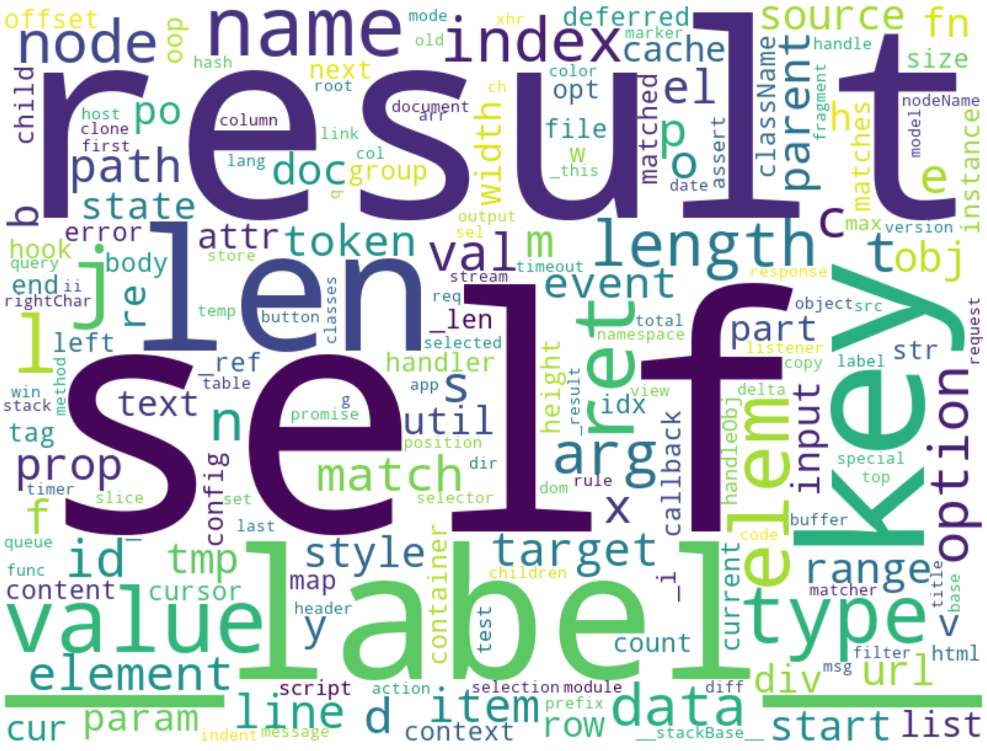


FIGURE 3. Word cloud over the most frequent variable names found in the dataset

highly generic, such as x, or whose name describes perfectly what it refers to, such as *style* or *error*.

The Recurrent Neural Network architecture highly depends on the size of the output, i.e., in our case, the length of the variables' name. Thus, as it can be seen in Figure 4, although the length can vary infinitely, most of the cases, a name has 4 characters, and the variable names with a length higher than 6 follow a standard exponential distribution.

4.2. **Experimental setup.** The input for our model is represented by both the context of the variable, i.e. its initialization and usages, and the previous hidden state of the Decoder's Gated Recurrent Unit, used for generating the next character for the mentioned variable. Thus, we will require two vocabularies: one for the code component, and one for the name of the variable. From our tests, the vocabulary for code has a size of 158, while the size for the name's vocabulary will be 77. In the data set, all non-ASCII characters have

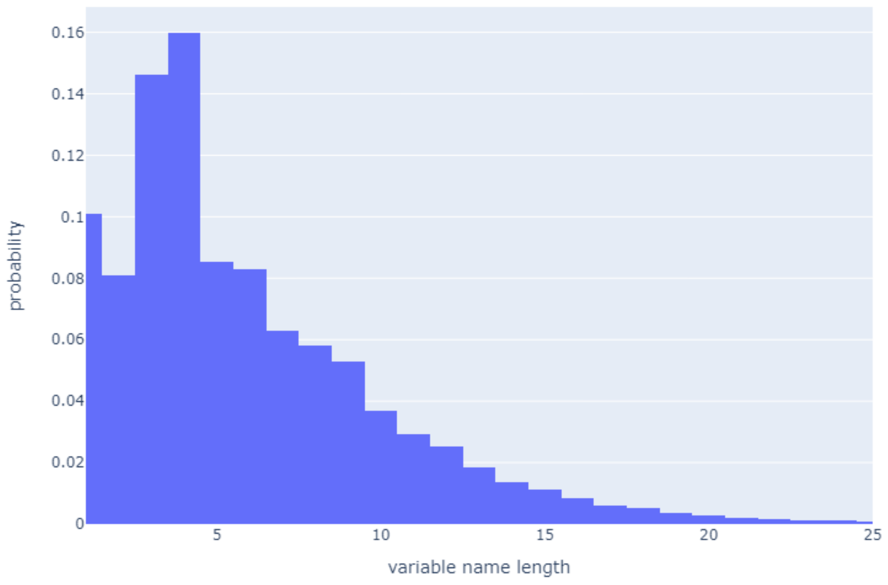


FIGURE 4. The distribution of the variable name length

been removed, since they would hundredfold these sizes, making them more difficult to align and the model more complex.

In our experiment, the architecture used has been described in Section 3.2.1. After constructing the vectors for the initialization and the usages, we decided to keep the first 300 tokens from them, while also picking only the first 3 usages, and concatenating everything, thus obtaining a vector with a size of 1200. The Encoder's Embeddings layer will have an input equal to the size of the code's vocabulary, and an output of 256. It will be followed by a Bidirectional Gated Recurrent Unit, which increases the output size to 512. The Decoder's Embeddings layer will have an input size equal to the size of the name's vocabulary and an output of 256, being followed by an Unidirectional Gated Recurrent Unit, which has the output equal to 256. Thus, the Cross-Attention layer receives a question having the size of 512, and the key equal to 256, having as an output a vector of size 512. The Decoder's Linear layer has an input of size 512 and an output equal to the name's vocabulary size. Thus, the total number of parameters that have to be trained is 738,381.

4.3. **Results and discussion.** After applying a 5-fold cross-validation as described in Section 3.2.2, we obtained an average character generation accuracy of 96.52% and an average name match of 70.53% on our test date sets. For each iteration of the cross-validation algorithm, we obtained the following name match accuracies: 70.81%, 81.16%, 61.70%, 66.30% and 72.68%. Thus, we obtain $[64.13, 76.93]$ name match accuracy percentage as a confidence interval, with a confidence of 95%. The 95% confidence interval (CI) [2] has been computed by using the formula $[\mu - \alpha, \mu + \alpha]$, where μ is the mean of the accuracies obtained during the 5-fold cross-validation and α is the margin of error, computed as

$$\alpha = 1.96\frac{\sigma}{\sqrt{5}}$$

(σ is the standard deviation of the obtained accuracies). Although our obtained 95% CI (6.4%) is large enough, the *name match accuracy* metric employed is more restrictive than the standard character generation accuracy, mostly because of the variable sequence length of both the input and the output, which might lead the model to learn certain parts of target name at different epochs. Moreover, although our goal is to match the generated name fully to the one predetermined, partial matches might still be valuable, even if the final output does not perfectly match the target sentence, which is not considered by our metric.

A direct comparison to other state-of-the-art approaches can be seen in Table 1, where our approach and its results are marked with bold. The table depicts the name match accuracies for our **Seq2Seq** model compared to the state-of-the-art tools Context2Name [1], JSNice [11] and JSNaughty [15]. This comparison has been made using the results available in the previous work [1], which have been computed on the exact same data set and testing methodology. During the training of our model, validation loss has decreased in conjunction with the training loss, which can be seen in Figure 5, which shows that the model was not overfitted. Moreover, in Figure 6, we can see a direct comparison between the output generated from UglifyJS, where each variable name is obfuscated using a single letter. The output of JSNice, as compared to the output generated from our model provides a level of confusion regarding the variables fb and $Helper$. In this case, our model generates wrongfully only one name, i.e. *adddataService*, as opposed to the instance found in the dataset *addDataService*, which proves that the model correctly learned that lowercase letters have the same meaning as capital letters.

As it can be seen in Figure 7, both most frequent and the most inaccurately predicted type is represented by function calls, whose assigned name can be challenging for programmers, thus the inconsistencies in the dataset, which might lead to incorrect-generated names. One interesting case is represented

Seq2Seq	Context2Name	JSNice	JSNaughty
70.53%	58.1%	56.0%	47.7%

Table 1. Results compared to state-of-the-art approaches

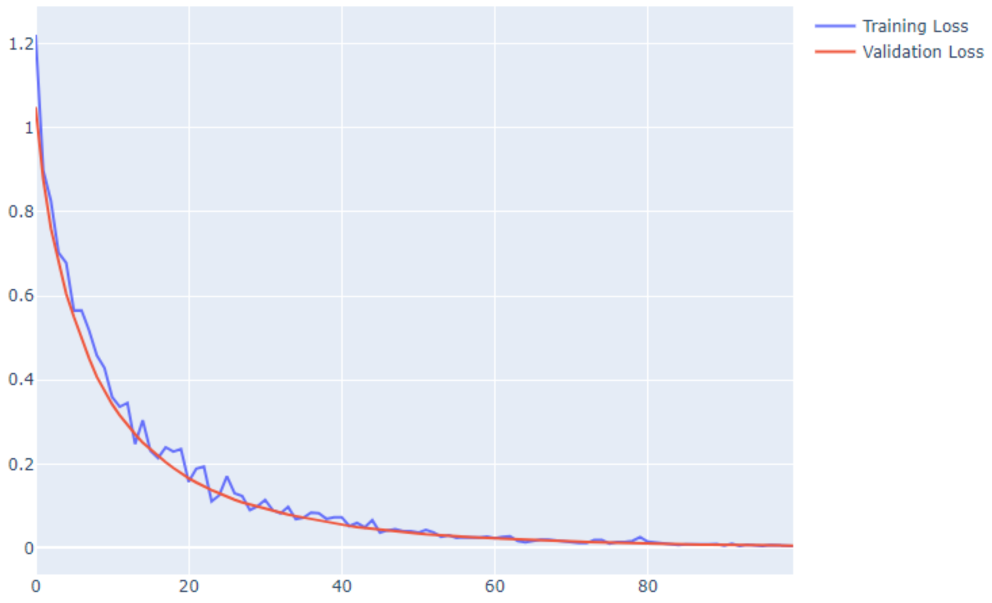
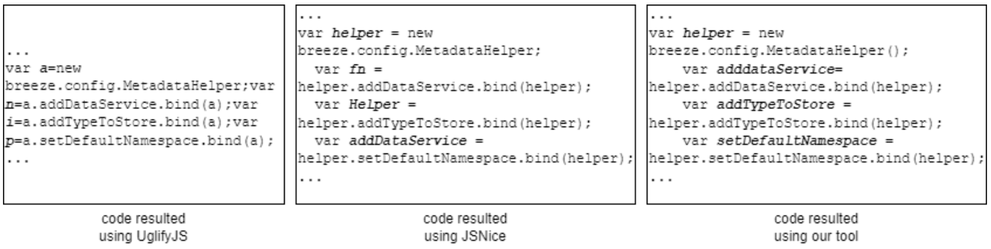


Figure 5. Training and validation losses



```
...
var a=new
breeze.config.MetadataHelper;var
n=a.addDataService.bind(a);var
i=a.addTypeToStore.bind(a);var
p=a.setDefaultNamespace.bind(a);
...
```

```
...
var helper = new
breeze.config.MetadataHelper;
  var fn =
helper.addDataService.bind(helper);
  var Helper =
helper.addTypeToStore.bind(helper);
  var addDataService =
helper.setDefaultNamespace.bind(helper);
...
```

```
...
var helper = new
breeze.config.MetadataHelper();
    var adddataService=
helper.addDataService.bind(helper);
    var addTypeToStore =
helper.addTypeToStore.bind(helper);
    var setDefaultNamespace =
helper.setDefaultNamespace.bind(helper);
...
```

Figure 6. Comparison between code generated by UglifyJS, JSNice and the output of our proposed model

by function expressions, which are similar to functions lambda functions assigned with a name, and string literals assignments, where there are fewer and more varied examples to learn from, which might cause the high error rate.

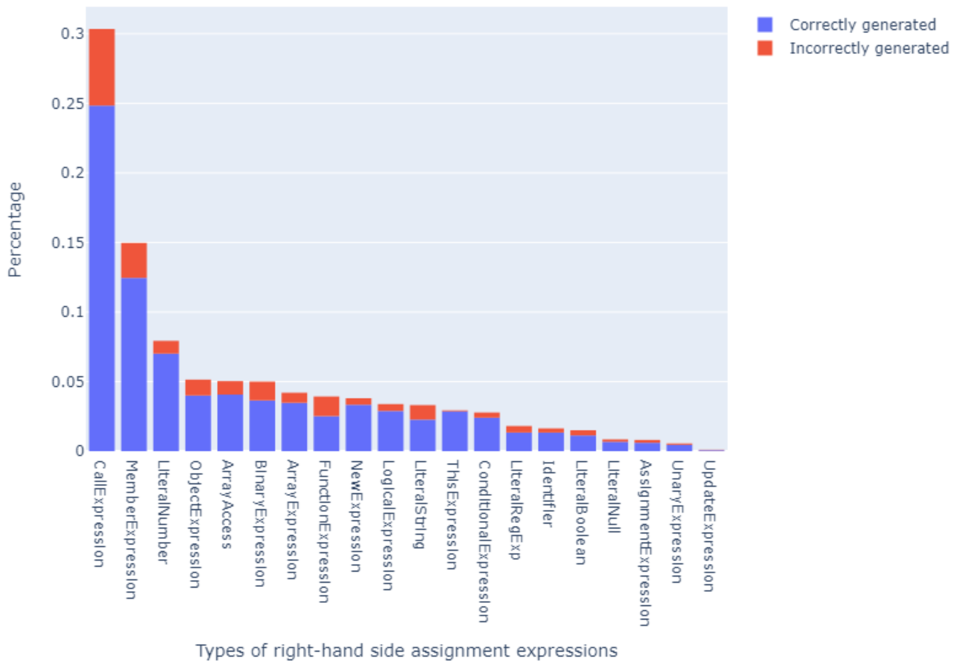


FIGURE 7. Histogram over the variable name types correctly and incorrectly predicted, sorted based on their frequency

The approach introduced in the paper presents both advantages and disadvantages which have to be considered. The main difference between our approach and the previous work in this field is the character-level encoding of non-fixed nodes in the Abstract Syntax Tree, such as the values of strings and function names. This allows us to build an open label vocabulary, which would be much smaller than others and uses strictly the possible types of nodes and the ASCII characters. Thus, the model is able to generate names from input never-before seen, and might generate an output related to that input. Moreover, the code encoding based on Abstract Syntax Tree allows the model to easily learn the relations between nodes and their values as opposed to the traditional token representation of each line of code. As for the disadvantages, as opposed to other methods, this approach cannot handle name collisions: in the previous work presented, name collisions were solved by picking the label with the highest probability, which is not yet existent in the current scope.

Such an approach to name collisions would not provide well-generated labels, but a random sequence of non-intelligible characters.

5. Conclusions and Future Work

This paper addressed the problem of JavaScript code deobfuscation, more generically the problem of assigning names to variables. We proposed a deep-learning generative model, which constructs a fitting name for a variable character-by-character, using their initialization and usages, encoded as Abstract Syntax Trees. After evaluating the model on a data set containing real world JavaScript code, we achieved 70.53% name match accuracy, outperforming state-of-the-art approaches.

Overall, the code obfuscation is a devious task, while also making the code harder for the user to understand it, and sometimes, making him completely unaware of the code that is running on his machine. Code obfuscation was used for malware to propagate through the internet [8], yet there are still organizations that try to protect their code and intellectual property, whose code should be technically safe. There are many other methods of obfuscation besides renaming variables, such as adding code sequences that, when executed, it will have no effect. This technique is used for generating polymorphic code, usually used in malicious code [7]. These techniques provide no real protection from stealing the intellectual property because a professional developer will eventually understand the code, but the described techniques make the whole process more difficult.

To conclude, these results prove that the names in variables are more than a simple label, and they provide a meaning, and their name's characters are similar to words in a sentence.

As for the future work, the presented approach cannot handle well name collisions, as opposed to any of the previous work presented. This approach could be enhanced by adding the current generated name up to that point, but would also require a data set where variables are annotated with multiple possible names. Thus, picking the character with the second-best probability would be a valid solution to this problem, since the model could generate a name properly by using it in future character generations.

The task of generating variable names, in the presented approach, can be adapted to other programming languages as well, where code obfuscation at the level of variable names is predominant, such as decompiled Java and C++. Thus, the only component which has to be changed would be the one that converts the code into the language-specific Abstract Syntax Tree, which can be an area worth experimenting in. Moreover, the task of variable name generation can be integrated in other code-related Natural Language Processing tasks, such as code generation, where suggestive names have to be recommended based on already-generated logic, or into a code linter, which suggests meaningful variable names based on pre-defined projects to set a naming standard as company policy.

References

[1] Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

[2] George W. Burruss and Timothy M. Bray. Confidence intervals. In Kimberly Kempf-Leonard, editor, *Encyclopedia of Social Measurement*, pages 455–462. Elsevier, New York, 2005.

[3] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on software engineering*, 36(4):546–558, 2009.

[4] Tadayoshi Fushiki. Estimation of prediction error by using k-fold cross-validation. *Statistics and Computing*, 21:137–146, 2011.

[5] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[6] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, pages 20–30, 2018.

[7] Xufang Li, Peter KK Loh, and Freddy Tan. Mechanisms of polymorphic and metamorphic viruses. In *2011 European intelligence and security informatics conference*, pages 149–154. IEEE, 2011.

[8] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 47–54. IEEE, 2009.

[9] Xiaoyu Liu, Jinu Jang, Neel Sundaresan, Miltiadis Allamanis, and Alexey Svyatkovskiy. Adaptivepaste: Code adaptation through learning semantics-aware variable usage representations. *arXiv preprint arXiv:2205.11023*, 2022.

[10] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 8(67), 2012.

[11] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.

[12] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*, 2021.

[13] Steve Souders. High-performance web sites. *Communications of the ACM*, 51(12):36–41, 2008.

[14] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10–pp. IEEE, 2005.

[15] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 683–693, 2017.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[17] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

Babeș-Bolyai University, Faculty of Mathematics and Computer Science, 1 Mihail Kogălniceanu, Cluj-Napoca 400084, Romania

Email address: `alexandru.gabriel.sirbu@stud.ubbcluj.ro`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 2, 2023
DOI: 10.24193/subbi.2023.2.02

SOFTWARE MAINTAINABILITY AND REFACTORINGS PREDICTION BASED ON TECHNICAL DEBT ISSUES

L. BERCIU AND V. MOLDOVAN

Abstract. Software maintainability is a crucial factor impacting cost, time and resource allocation for software development. Code refactorings greatly enhance code quality, readability, understandability and extensibility. Hence, accurate prediction methods for both maintainability and refactorings are vital for long-term project sustainability and success, offering substantial benefits to the software community as a whole. This article focuses on prediction of software maintainability and the number of needed code refactorings using technical debt data. Two approaches were explored, one compressing technical debt issues per software component and employing machine learning algorithms such as ExtraTrees, Random Forest, Decision Trees, which all obtained a high accuracy and performance. The second approach retained multiple debt issue entries and utilized a Recurrent Neural Network, although less effectively. In addition to the prediction of the requisite number of code refactorings and software maintainability for individual software components, a comprehensive analysis of technical debt issues was conducted before and after the refactoring process. The outcomes of this study contribute to the advancement of a dependable prediction system for maintainability and refactorings, presenting potential advantages to the software community in effectively managing maintenance resources. From all the employed models, the ExtraTrees model yielded the most optimal predictive outcomes. To the best of our knowledge no other approaches of using ML techniques for this problem have been reported in the literarture.

1. Introduction

In the last decades, software has known a continuous growth facing increasingly demanding expectations and requirements. Consequently, the software development process must aim for optimal efficiency. The objective is to create software systems that are bug-free, easily modifiable and updatable, and

Received by the editors: 19 June 2023.
2010 *Mathematics Subject Classification.* 68N99.
1998 *CR Categories and Descriptors.* D2.0 [**Software Engineering**]: General – *Standards*; D2.9 [**Software Engineering**]: Management – *Software Quality Assurance*.
Key words and phrases. Software Quality, Sonarqube, Refactoring, Code Smells.

capable of accommodating new features seamlessly. The maintenance phase of software development is a substantial part of its life cycle. This phase is of utmost importance as it spans the entire lifespan of the software product, commencing immediately after the completion of the development process. Numerous studies have emphasized the significance of software maintainability [9], [5], [21].

There is a strong connection between software systems' maintainability and technical debt issues. There are several types of software issues that can be detected by performing a static analysis of the software systems. Without actually running the code, software static analysis looks for potential problems and enhances code quality. It specifically looks at a program's source code to find potential problems such syntax mistakes, security flaws, performance problems, and maintainability concerns. A small number of issues implies a higher maintainability and vice versa. One of the code's problems found in the static analysis process is represented by the technical debt.

As presented in [2], [16], and [27], there is a strong connection between the software systems' detected issues, its maintainability and the refactorings performed on that software. Refactoring aims to lower technical debt while increasing readability, maintainability, and scalability of the software system. When the maintainability is too low and there are too many issues which overcomplicate the development process, a refactor is needed. After successfully and correctly performing the refactor, the number of issues should decrease and the software maintainability should increase, improving the quality of the software.

With the growing interest in the field of software development, there has been a significant rise in the development of tools aimed at enhancing the software development process. These tools offer various capabilities, including the ability to measure code metrics, conduct static analysis to identify technical debt issues, and provide suggestions for code improvements. Among the widely recognized tools in this domain are SonarQube [1], PyLint [2], ESLint [3], cppcheck [4], CheckStyle [5], FindBugs [6]. There are also tools that aim to detect

[1]Sonarqube. https://www.sonarqube.org/.
[2]PyLint, https://pypi.org/project/pylint/
[3]ESLint, https://eslint.org/
[4]cppcheck, https://cppchecksolutions.com/
[5]CheckStyle, https://github.com/checkstyle/checkstyle
[6]FindBugs, https://spotbugs.github.io/

code refactorings between different code versions, such as RefactoringMiner [7], RefactoringCrawler [8], RefDiff [9], and several others.

The goal of this paper is to predict the number of refactorings that need to be performed, and based on this number to classify the maintainability of each software component, aiming to obtain a performance as high as possible. To accomplish this, multiple intelligent algorithms have been employed and analyzed to identify the most suitable algorithm for this specific task. In addition to predicting software maintainability and number of needed code refactorings, an analysis of the technical debt issues before and after the refactoring process has been conducted.

The problem of predicting software maintainability and number of needed refactorings based on technical debt issues was addressed firstly as a classification problem, and then as a regression problem. The classification task referred to classifying each software component into one of the following maintainability classes: "Great", "Good", and "Poor", while the regression task referred to predicting the number of needed refactorings by each software component.

The subsequent sections of this paper are organized as follows: Section 2 outlines the related work, Section 3.1 presents the data preparation, and Section 3.2 expounds on the architectural configurations of the employed machine learning algorithms. A comparative analysis is presented in Section 4, followed by a more comprehensive examination of the SonarQube issues in Section 5. Ultimately, Section 6 offers the drawn conclusions and points towards potential directions for future work.

2. Related work

Ensuring the correctness and efficiency of modern software systems is essential. Software maintainability, being a crucial quality factor, plays a vital role in ensuring the long-term sustainability of software systems and mitigating technical debt. It facilitates effective bug fixing, enables seamless software updates and improvements, fosters collaboration within development teams, and supports the overall adaptability and evolution of software systems. By prioritizing maintainability, organizations can streamline their software development processes, enhance productivity, and deliver reliable, high-quality software products that meet the evolving needs of their customers and stakeholders.

The topic of software maintainability prediction has received a significant interest, leading to numerous studies dedicated to addressing this problem.

[7]RefactoringMiner, https://github.com/tsantalis/RefactoringMiner

[8]RefactoringCrawler, http://dig.cs.illinois.edu/tools/RefactoringCrawler/

[9]RefDiff, https://github.com/aserg-ufmg/RefDiff

A comprehensive review study conducted by Elmidaoui et al. [8] examined 77 research studies published between 2000 and 2018 that aimed to predict software maintainability based on various software quality metrics. The review presented in [8] provides a detailed analysis of the employed maintainability prediction techniques, validation methods, accuracy criteria, overall accuracy of machine learning (ML) techniques, and the techniques offering the best performance.

In [26], Van Koten and Gray found that ML techniques, including Bayesian networks [23], outperformed regression-based models in prediction accuracy. The study showed that ANNs [24] capture complex non-linear relationships and approximate any measurable function, SVM/R [6], [7] excels in learning classification and regression tasks, especially with high-dimensional data, DT [4] offer a straightforward and comprehensible approach and FNF methods [13] handle limited or missing data, while RA [20] is a simple and reliable technique, particularly useful with multiple independent variables. This represented a reason for which, in this approach, machine learning algorithms were chosen to be used in the detriment of regression-based models.

The prediction techniques utilized in these studies can be broadly classified into two main groups: ML techniques and statistical techniques. Statistical techniques encompassed various approaches, such as regression analysis (RA), probability density function (PD), Gaussian mixture model (GMM), discriminant analysis (DA), weighted functions (WF), and stochastic model (SM). In contrast, ML techniques encompassed artificial neural networks (ANN), case-based reasoning (CBR), regression and decision trees (DT), Bayesian networks (BN), evolutionary algorithms (EA), support vector machine and regression (SVM/R), fuzzy and neuro fuzzy (FNF), inductive rule-based (IRB), ensemble methods (EM), and clustering methods (CM).

The review study [8] showed that the statistical techniques, more popular from 2000 until 2007, are only effective when a linear or predetermined relationship exists between the dependent and independent variables. With the advent of ML techniques, researchers started exploring both statistical and ML approaches to assess their predictive capabilities for maintainability. In [14], Kaur and Kaur emphasized that traditional parametric statistical data analysis methods may be insufficient and suggested that the utilization of ML algorithms or pattern recognition approaches, which are inherently nonparametric, could lead to improved prediction accuracies.

The process of refactoring holds an important significance owing to its capacity to enhance code quality, improve maintainability, and foster collaboration among developers. By eliminating code smells, mitigating technical debt, and optimizing code performance, refactoring contributes to the development of robust and scalable software systems. Consequently, numerous research studies have been conducted to comprehensively analyze the relationship between refactoring and software maintainability, as well as to explore predictive methods for anticipating the need for refactorings.

In [12], Hegedus et al. presents an enhanced dataset comprising verified refactoring data pertaining to open-source systems. The study reveals that refactoring is frequently employed on entities exhibiting low maintainability, signifying developers' proactive efforts to address deteriorated code. Metrics associated with size, complexity, and coupling exhibit notable increases in refactored elements, indicating developers' focus on improving these aspects. However, the analysis suggests that metrics related to code clones have a comparatively lesser impact.

In [1], an investigation on the prediction of software refactoring by employing Support Vector Machine (SVM) and optimization algorithms is presented, exploring the relationship between code coverage and the effectiveness of the test suite in an evolutionary context. The authors examine the application of SVM in conjunction with genetic algorithms to forecast refactoring at the class level. Utilizing a dataset derived from open-source software systems, the study achieves promising levels of accuracy, ranging from 84% to 93%. The performance is further enhanced by integrating SVM with the optimization algorithms.

3. Experiment and study plan

The experiment was designed in the following manner: firstly, the data was gathered, then the dataset was prepared The last included several steps: Preparing the technical debt dataset: computing the number of issues per software component, then associating numerical values to severity (BLOCKER: 5, CRITICAL:4, MAJOR:3, MINOR:2, INFO:1) and type (CODE SMELL:1, BUG:2, VULNERABILITY:3), removing the N/A entries and finally computing for each software component the mean of the severity, debt and type values; Preparing the refactoring dataset implied computing the number of refactorings per software component. Finally, the dataset was created after merging the technical debt dataset with the refactoring dataset. After that, data was split into training (70%) and testing (30%). The models were trained, and then the testing data was used to evaluate them. The last step was represented by analyzing the obtained results and drawing conclusions.

3.1. **Data Preparation.**

In order to predict the software maintainability and number of needed refactorings based on technical debt issues, several steps need to be performed. The first element needed is represented by data. In this research investigation, a comprehensive analysis was conducted on three open-source Java projects, namely jEdit [10], FreeMind [11], and TuxGuitar [12]. The study encompassed two distinct categories of data pertaining to these projects. Firstly, the technical debt issues encountered in jEdit version 5.5, FreeMind version 1.0.1, and TuxGuitar version 1.5.2 were examined. Secondly, the refactorings performed between these versions and subsequent versions of each project, specifically jEdit 5.6, FreeMind 1.1.0, and TuxGuitar 1.5.3, were taken into account. The dataset [17] containing all technical debt issues information about the jEdit project version 5.5, FreeMind project version 1.0.1 and TuxGuitar project version 1.5.2, has been previously used in other studies as well, such as [18].

The data needed for the Technical Debt analysis step of the experiment was collected by running SonarQube on the three projects and extracting the issues data found by the tool, such that the details provided by the tool will be considered as attributes. While the tool execution for jEdit and FreeMind was straightforward, namely successfully compiling the projects and running the Sonarqube tool by using the Sonar Scanner version matching the build system used (ANT in both cases), the data collection for TuxGuitar proved to be more difficult, as TuxGuitar is divided in a multitude of individual projects compiled by Maven build system, more exactly 65 individual projects. From those, we selected the base TuxGuitar project, TuxGuitar Android Resource, TuxGuitar AudioUnit, TuxGuitar CoreAudio, TuxGuitar Editor Utils, TuxGuitar GM Utils, TuxGuitar Lib and TuxGuitar UI toolkit. SonarQube was executed on each individual project and then the issues fetched and merged together.

In this research study, a selective approach was adopted regarding the attributes of the technical debt issues under consideration. After running the chosen static analysis tool, a report was obtained that contained a list of all detected issues, each issue associated with the name of the software component in which it's located, and several other attributes such as severity, debt, type, creation date, rule, update date, and others. Specifically, the severity, debt, and type of each issue were thoroughly investigated. This decision was based on the notion that certain attributes, such as the key, did not provide significant or pertinent information for the classification or regression model. Furthermore, some attributes required more intricate examination and preprocessing, which warranted their inclusion in future research endeavors. In addition to the severity, debt, and type of each issue, the computation of the number of issues per component was performed and subsequently utilized in the prediction process.

[10]jEdit, http://www.jedit.org/

[11]FreeMind, https://freemind.sourceforge.net/

[12]TuxGuitar, https://sourceforge.net/projects/tuxguitar/

Regarding the final representation of the technical debt issues data, two distinct approaches were pursued:

- The first approach involved compressing the technical debt issues for each project into a single instance per software component. To obtain the final values of severity and type for each component, these attributes were mapped to numerical values using the following scheme: Severity = INFO: 1, MINOR: 2, MAJOR: 3, CRITICAL: 4, BLOCKER: 5 and Type = CODE SMELL: 1, BUG: 2, VULNERABILITY: 3. After the mapping process, the mean values of severity and type were computed by summing their respective values for each component and dividing the sum by the number of issues associated with that component. When calculating the mean for the debt attribute, instances with a value of N/A were excluded from consideration, and the associated issues were removed from the total count per component. By performing these operations, each software component was represented by a single instance.
- The second approach did not involve compressing the issue instances into a single instance per class/component. Instead, it focused on removing issues that had an N/A value for the debt attribute. This decision was made to enhance interpretability for the model, as N/A values posed difficulties in interpretation. Additionally, this approach also resulted in a decrease in the number of issues per component, which had been previously computed.

The data utilized in this study comprised information related to the refactorings conducted between two versions of each project, which was obtained through the employment of the RefactoringMiner tool. This tool facilitated a comparison between two project versions and generated a report detailing the refactorings executed during this transition. The provided refactoring information encompassed the type of each refactor, a concise description of its purpose, and the specific component on which the refactoring was performed. Additionally, the number of refactorings for each software component was computed. These attributes provided valuable insights into the code's condition and offered suggestions for improving its maintainability. To avoid challenges associated with high-dimensional data, such as overfitting, computational complexity, and data sparsity, only the count of performed refactorings per software component was considered in this study.

To create the final dataset, the technical debt issues data and the refactorings data were merged. The software components served as the common element between these datasets, enabling the addition of a new column in the technical debt issues dataset that represented the number of refactorings performed on each specific component. Consequently, the analyzed data fed into the intelligent algorithm contained information pertaining to the severity, debt, type, and count of technical debt issues for each software component, alongside the number of refactorings conducted on that particular component.

The problem in this study was initially formulated as a classification task and later as a regression task. In the regression setting, the predicted output was the estimated number of refactorings required for each component. In the classification problem, the output classes were defined as follows:

- "Great" category: Corresponded to components that had fewer than 5 refactorings performed on them.
- "Good" category: Associated with software components that underwent between 5 and 20 refactorings.
- "Poor" category: Assigned to software components that had more than 20 refactorings performed on them.

3.2. **Architectural Configurations of Employed Machine Learning Algorithms.**

Various architectural configurations were evaluated for the prediction of software maintainability and the required quantity of refactorings in the analyzed projects. Employing the initial dataset, which comprised a single entry for each software component, multiple models were trained and achieved commendable performance.

For the classification approach, the LazyClassifier from the *lazypredict.Supervised* library was employed, and several models were studied, including the Extra Tree Classifier [10], LGBM Classifier [15], Random Forest Classifier [3], and K-Neighbors Classifier [11]. Cross-validation with 5 folds and the *f1_macro* scoring metric were applied using the *cross_val_score* function from the *sklearn.model_selection module.* Additionally, the MLPClassifier [25] from the *sklearn.neural_network* module was evaluated with different configurations, such as varying the number of neurons (100, 150, and 200), considering activation functions like *Relu* and *logistic sigmoid*, and utilizing the $lbfgs$ and *adam* solvers as optimization methods. The results are presented in Table 1.

For the regression approach, the *LazyRegressor* from the *lazypredict.Supervised* library was utilized, and several models were examined, including the Extra Tree Regressor, K-Neighbor Regressor, Hist Gradient Boosting Regressor, Random Forest Regressor, and Decision Tree Regressor. Similar to the classification approach, cross-validation was conducted using a KFold of 5, and the negative mean squared error was used as the scoring metric. The multi-layer perceptron (MLP) was also applied for regression, with the number of neurons set to 20, 25, and 30, and the activation functions and solvers remaining the same as those used in the classification task.

TABLE 1. Results obtained for Lazy Classifier models and MLP and RNN models

Model	Accuracy	Recall	Precision	F1-Score
ExtraTreesClassifier	**0.92**	**0.92**	**0.92**	**0.92**
RandomForestClassifier	0.90	0.90	0.90	0.90
LGBMClassifier	0.88	0.88	0.88	0.88
DecisionTreeClassifier	0.88	0.88	0.88	0.88
ExtraTreeClassifier	0.82	0.81	0.83	0.82
KNeighborsClassifier	0.77	0.76	0.79	0.77
SVC	0.70	0.70	0.71	0.70
MLP	0.69	0.65	0.69	0.67
LogisticRegression	0.63	0.63	0.62	0.62
LinearSVC	0.60	0.59	0.58	0.58
Perceptron	0.56	0.56	0.56	0.56
RNN	0.57	0.58	0.56	0.56

The second dataset, which included multiple entries per component corresponding to detected technical debt issues, was used for training a recurrent neural network (RNN). The RNN model consisted of the following components:

- An Embedding layer with a length equal to the training data's length.
- A LSTM (long-short term memory) layer followed, utilizing Relu activation functions for both regression and classification tasks, and sigmoid activation function solely for classification.
- A Dropout layer with a dropout rate of 0.2.
- Another LSTM layer with the same activation functions as the previous LSTM layer
- A subsequent Dropout layer with a dropout rate of 0.2
- A Dense layer with Relu activation function for regression and classification, and sigmoid activation function only for classification,
- Another Dropout layer, identical to the previous two. This layer helps prevent overfitting and introduces noise, making the network more robust to dependencies on specific features.
- The last layer was a Dense layer with either 3 output channels for classification (corresponding to the defined maintainability classes) or 1 output channel for regression. The activation function used was Softmax for classification and no activation function for regression.

The model was compiled using the sparse categorical cross entropy loss function for the classification algorithm and mean squared error loss function for the regression algorithm. The optimizer used for both algorithms was adam. The results are presented in Table 2.

Table 2. Results obtained for Lazy Regressor models, MLP and RNN models

Model	R-Squared	Adjusted R-Squared	RMSE	MAE
ExtraTreesRegressor	**0.92**	**0.91**	**2.75**	**1.77**
RandomForestRegressor	0.90	0.89	3.12	2.36
ExtraTreeRegressor	0.81	0.81	2.88	2.54
DecisionTreeRegressor	0.88	0.88	3.01	1.87
LGBMRegressor	0.88	0.88	3.07	2.18
KNeighborsRegressor	0.75	0.75	2.87	1.99
SVR	0.65	0.65	3.88	3.28
LinearSVR	0.59	0.58	3.97	3.02
MLP	0.58	0.58	3.85	3.25
RNN	0.50	0.50	4.21	3.25
LinearRegression	0.49	0.48	4.69	3.88

4. Comparative analysis

In Table 1, the results obtained for LazyClassifier models, MLP and RNN models are presented. The best performing model was represented by ExtraTreesClassifier using the first approach of handling the data, while the lowest performance was obtained by the RNN model using the second data approach. The accuracy of the ExtraTreesClassifier was 0.92, also having the same value for the recall, precision and F1-Score metrics. This suggests that that the classification model is performing at a high level, making correct predictions, and effectively capturing positive cases. As for the RNN model, it achieved an accuracy of 0.57, a recall of 0.58 and a precision and a F1-Score of 0.56, indicating that model has some level of predictive ability, but the performance is quite low comparative to the other employed models. The RNN model might be making correct predictions for a portion of the data, but there are also instances where it's struggling to provide accurate results. This needs to be further investigate in order to be improved.

The regression results presented in Table 2 are similar to the ones from the classification task presented in Table 1, the ExtraTrees algorithms being the most performant one. The RNN model behaved better than LinearRegression, but its performance still needs to be further analysed and improved. The RNN model obtained a R-Squared and and Adjusted R-Squared of 0.50, suggesting that he model explains about 50% of the variability in the target variable, being a moderate fit. A RMSE of 4.21 implies that the model's predictions have an average error of around 4.21 units, while a MAE of 3.25 suggests that, on average, the model's predictions are off by about 3.25 units from the actual values.

Following the execution of the ExtraTrees classifier, the outcomes for each maintainability class are displayed in Table 3. The analysis reveals that the "Good" class achieved the highest performance, while conversely, the "Poor" class exhibited the lowest performance. This observation aligns with the characteristics of the initial dataset, which exhibited an imbalance prior to undergoing data augmentation. Specifically, the dataset contained a smaller number of instances classified as "Poor" compared to instances classified as "Great" and "Good."

Table 3. Results obtained for ExtraTrees model

Output Class	Accuracy	Recall	Precision	F1-Score
Great	0.92	0.91	0.93	0.92
Good	0.94	0.94	0.94	0.94
Poor	0.90	0.91	0.89	0.90
Total	0.92	0.92	0.92	0.92

The study focused on comparing the obtained results of software maintainability with the values of the maintainability index [22], which is a popular measurement method. The maintainability index categorizes software into three classes: Bad, Satisfactory, and Acceptable. The jEdit 5.5 project was classified as "Satisfactory," FreeMind 1.0.1 as "Bad," and TuxGuitar 1.5.2 as "Satisfactory" based on their maintainability index values. To perform a fair comparison between the results and the maintainability index, the mean value of the maintainability index for each project was computed.

The prediction model showed high performance based on the training data, with the lowest metrics observed for the "Poor" class. However, the observed maintainability index values for the three projects did not fully align with this finding. The mapping between software components and maintainability classes based on the number of needed refactorings did not match the mapping based on maintainability index values. This might suggest that there is no direct relationship between the number of refactorings or technical debt issues and the maintainability index value [19]. The discrepancy between the study's findings and the maintainability index can be attributed to the limitations of metrics used in the maintainability index, which do not fully capture the complexities of object-oriented software. Since the studied projects were developed using the object-oriented paradigm, it is expected that the results may not align perfectly with the maintainability index. Thus, our study confirms one more time that maintainability index does not accurately characterize object oriented systems.

5. A deeper dive into SonarQube issues

In the previous sections, we provided a high level overview on how maintainability index and technical debt metrics exhibit variation between two successive versions of a project's release timeline. On the Technical Debt experiment side, we leveraged the SonarQube issues and the refactorings data from the provided datasets, selected the data most suitable for the experiment from both sources, merged it together and fed it to artificial intelligence tools. While this proved to be effective in computing the results of the study and providing a general answer on the aforementioned research questions, we decided to further refine and improve our research findings by following a particular path: diving deeper into SonarQube issues.

Throughout this section, we will present the particularities of SonarQube issues found inside each project by doing a classification of their types, severities and numbers, see how they compare between versions and offer a final comparison with the initial results from the previous section.

5.1. **JEdit 5.5 and 5.6.**

JEdit proved to have the highest number of issues found for both versions, from all projects under study. For version 5.5, we have extracted a total of 139.905 issues, from which 134.901 were labeled as CODE_SMELLS and 4004 were labeled as BUG. For Version 5.6, the number of total issues was 63899, from which 57773 were labeled as CODE_SMELL and 6126 were labeled as BUG. Interestingly, the latter version also reported 8 issues explicitly labeled as vulnerabilities. The sonar rules spanned multiple file types, such as .java, .html and .xml. The general data can be visualised in Table 4, while Table 5 shows the data for Java only files.

Table 4. General issues comparison between jEdit versions

Project	**CODE_SMELL**	**BUG**	**VULNERABILITY**
jEdit 5.5	135901.0	4004.0	-
jEdit 5.6	57773	6126	8

To properly show how the issues fluctuated between the two releases, tables 6 and 5.1 show how the percentages between issue types changed. We can

Table 5. Java issues comparison between jEdit versions

Project	**CODE_SMELL**	**BUG**	**VULNERABILITY**
jedit-5-5.csv	29093.0	441.0	-
jedit-5-6.csv	20355	541	8

observe a decrease in code smells from 97.14% to 90.40% when it comes to total code smells reported to the other issues and a staggering numeric decrease from 135901 to 57773. While this looks like an improvement on a first glance, if we take BUG issues into consideration, we can observe an actual increase from 2.86% to 9.59% between versions, more specifically, from 4004 to 6126 issues reported as bugs. An extra 8 vulnerabilities were also found. While the total number of issues may have decreased, we can clearly observe that their severity increased, as the number of bugs increased by 50% and bugs having a higher severity than code smells in general. We can conclude that, at least for now, the quality of the code decreased through the versions.

Table 6. Distribution of general issues between jEdit versions

Project	**CODE_SMELL**	**BUG**	**VULNERABILITY**
jEdit 5.5	97.14%	2.86%	-
jEdit 5.6	90.40%	9.59%	0.01%

Table 7. Distribution of Java issues between jEdit versions

Project	**CODE_SMELL**	**BUG**	**VULNERABILITY**
jEdit 5.5	20.79%	0.32%	-
jEdit 5.6	31.85%	0.85%	0.01%

5.2. **Freemind 1.0.1 and 1.1.0.** For Freemind, we extracted a total of 12653 issues, from which 12349 labeled as code smells, 302 labeled as bugs and 2 labeled as vulnerabilities. As opposed to jEdit, the number of Java issues comprises the majority of general issues, with a number of 12549, from which 12269 code smells, 278 bugs and 2 vulnerabilities. Data can be visualised in Tables 8 and 9.

From a percentages point of view, we can observe that the ratio is similar for both general issues and java issues, with an approximate 97% and 2% percent of code smells and bugs holding between releases 10 and 11. The improvement here can be observed from the number of actual issues between the two versions, with a clear decrease of both code smells by 25% and bugs

Table 8. General issues comparison between Freemind versions

Project	Code Smell	Bug	Vulnerability
Freemind 1.0.1	12349	302	2
Freemind 1.1.0	9633	249	2

Table 9. Java issues comparison between Freemind versions

Project	Code Smell	Bug	Vulnerability
Freemind 1.0.1	12269	278	2
Freemind 1.1.0	9547	225	2

by 20%. While explicit vulnerabilities did not change, their number are too few to be relevant in this context.

We can say that the quality of code improved by 20% between releases, as opposed to jEdit, where the severity increased.

Table 10. Distribution of general issues between Freemind versions

Project	Code Smell (%)	Bug (%)	Vulnerability (%)
Freemind 1.1.0	97.60	2.39	0.02
Freemind 1.0.1	97.46	2.52	0.02

Table 11. Distribution of Java issues between Freemind versions

Project	Code Smell (%)	Bug (%)	Vulnerability (%)
Freemind 1.0.1	96.97	2.20	0.02
Freemind 1.1.0	96.59	2.28	0.02

5.3. **TuxGuitar 1.5.2 and 1.5.3.** TuxGuitar analysis shows a total of 3296 total issues for version 1.5.2, with a number of 3012 code smells, 258 bugs and 26 vulnerabilities. For version 1.5.3, we registered a total of 2930 issues, from which 2746 code smells, 184 bugs and 18 vulnerabilities. We can already observe an improvement from v1.5.2 to v1.5.3, as all categories of issues had a clear decrease. Data is shown in Table 12. A main difference from the previous two inspections shows that, for TuxGuitar, our test produced only java issues, meaning that the TuxGuitar projects that we analysed did not include other types of resources such as .html and .xml files that could have been analysed by SonarQube in the way we ran the tool. Hence, we did not publish a second table as the data between general issues and Java only issues is not different.

Table 12. General issues comparison between TuxGuitar versions

Project	Code Smell	Bug	Vulnerability
TuxGuitar 1.5.2	3012	258	26
TuxGuitar 1.5.3	2746	184	18

The distribution percentage of issues between versions has a similar ratio, while showing a slight increase in code smells and a slight decrease in bugs. This shows an improvement in the overall code base and a decrease in severity, as the percentage of bug issues is smaller than code smells when reported to the total number of issues. The improvement is better than in Freemind's case, where, even though every category of findings improved, the percentage of bugs related to the total issues became higher in the newer version. Table 13 contains the distribution of TuxGuitar issues.

Table 13. Distribution of issues between TuxGuitar versions

Project	Code Smell (%)	Bug (%)	Vulnerability (%)
TuxGuitar 1.5.2	91.38	7.83	0.79
TuxGuitar 1.5.3	93.15	6.24	0.61

5.4. **Comparison between the three datasets.** All three datasets showed improvements in the number of issues found between versions. This indicates that continuous development and refactorings decreased the number of total issues for each project. Even though the total number of issues has shown improvement, the quality of the changes was different:

- jEdit introduced more bugs than before, more specifically an addition of 2122 SonarQube BUG rule violations
- Freemind kept a similar ratio between code smells and bugs, with bugs slightly taking more space in the newer version
- TuxGuitar had the best result, with a similar ratio between code smells and bugs, and bugs also decreasing from a distribution point of view

Relating the above results with the maintainability index computed from Section 4, we can conclude the following:

- Starting with jEdit as the first analyzed project, we concluded that the overall code quality decreased between releases, even though the total number of issues has improved. This is backed by the fact that the number of bugs introduced in a newer release were superior (from 2% to 9%) than the older release and the fact that bugs hold a higher severity than code smells. The flow of development in this case seems to have brought down the overall quality of the project, asking the question of what might have happened in the development process. There may be a correlation between a big number of issues that the project has and the difficulty of maintaining and developing a complex application, hence the decrease in code smells and the increase in bugs in vulnerabilities. This can mark the subject of future research.

- With Freemind, the development process showed linear progress in the quality of the code. Both code smells and bugs issues have a similar percentage between releases, with an overall improvement of the project quality. Vulnerabilities were not taken into account due to their low number. This improvement may signify a better management in development processes and a greater attention to detail than the other projects when it comes to solving issues. The lower number of issues than jEdit may also pose a reason for this result, bringing into consideration the possible correlation from the previous point. Against all evidence, the maintainability index classified the project as "Bad", showing that the metric may not be generally applicable to empirical studies on refactorings.
- Finally, it can be observed that TuxGuitar's shift from version 1.5.2 to 1.5.3 not only exhibited a decrease in overall problems but also reflected an improvement in its maintainability index. The software obtained a "Satisfactory" rating in this area. The success of its refactoring endeavors is evident in the positive trajectory, which is characterized by a decrease in bugs and vulnerabilities. Simultaneously, the increase in code smells highlights the significance of ongoing emphasis on refining coding practices, in a way that both maintainability and overall code quality are maintained in future iterations.

6. Conclusions and future work

This study focuses on the analysis of software source code and its impact on software maintainability, considering factors such as cost and time allocation. The use of artificial intelligence has gained prominence in analyzing software problems. The study explores the relationship between software maintainability, technical debt issues, and code refactorings. The objective is to develop high-performing approaches for predicting software maintainability and the number of code refactorings based on technical debt issues.

To ensure accurate prediction of software maintainability and the number of required code refactorings, a comprehensive dataset was essential. This study focused on three open-source Java projects: jEdit, FreeMind, and TuxGuitar. Technical debt issues were obtained by performing static analysis on specific versions of these projects (jEdit 5.5, FreeMind 1.0.1, and TuxGuitar 1.5.2). Refactoring data was obtained by using the RefactoringMiner tool to compare different versions of each project. The technical debt data from the SonarQube tool was combined with the RefactoringMiner data and processed accordingly.

Two different approaches were considered. The first approach consolidated the technical debt issues to a single issue per software component, while also including the count of technical debt issues per component as an additional feature. Data augmentation techniques, such as Noise Injection, were applied to balance the dataset for increasing the generalization capacity of the models by simulating different variations in the data through the addition of random noise. Classical machine learning algorithms, including the Multi-layer Perceptron, Decision Trees, Random Forest, and Support Vector Machine, were employed as intelligent algorithms. The ExtraTrees algorithm yielded the best results in both the classification and regression tasks, achieving an accuracy of 0.92, F1-Score of 0.92 (classification), R-Squared of 0.92, and RMSE of 2.75 (regression).

The second approach did not modify the technical debt issues associated with each software component, allowing for the possibility of multiple entries for the same component in the dataset. A Recurrent Neural Network (RNN) was employed as the intelligent algorithm for this approach. However, the RNN model did not perform as well as the ExtraTrees algorithm in the first approach, achieving only an accuracy of 0.57 (classification) and R-Squared of 0.50 (regression).

The proposed methodology employs technical debt data to forecast the maintainability of software and the required number of code refactorings in three open-source Java projects. However, it is important to acknowledge a potential threat to the validity of the obtained outcomes due to the specific use of Java projects. Consequently, when applied to projects developed in different programming languages, the reliability of the results and the model's performance may be compromised.

Furthermore, certain aspects were not considered in the current approach that warrant exploration. These aspects include specific details pertaining to technical debt issues (e.g., message content) and refactorings (e.g., refactoring type and description). Integrating the message content of issues and the type or description of refactorings into the prediction model has the potential to enhance accuracy and provide more insightful information to end users. Additionally, the current implementation solely relies on technical debt data, and future enhancements could involve incorporating additional software metrics.

Such metrics offer valuable insights into the code's state and their inclusion could improve the predictive model.

Another area for investigation involves refining the performance of the second proposed approach, as it does not condense technical debt data to a single entry per software component. Similarly, in the first approach, reducing technical debt data to a single entry per component is achieved by calculating the mean of all values. However, it is essential to note that this may not be the most optimal reduction method. Therefore, this presents an additional area that requires further investigation and refinement.

The attained results represent a step towards building a strong predictive model for software maintainability and the necessary number of code refactoring. Additionally, these outcomes can be easily integrated into a web application, ensuring convenient access. This advancement has the potential to aid the software community in improving their assessment of software maintainability, thereby contributing to reduced resources needed for the maintenance phase.

References

[1] Akour, M., Alenezi, M., and Alsghaier, H. Software refactoring prediction using svm and optimization algorithms. *Processes 10*, 8 (2022).

[2] Arisholm, E., Briand, L. C., and Johannessen, E. B. An empirical study on the relationship between software maintainability and bug-proneness. In *2010 IEEE International Symposium on Software Metrics (METRICS)* (2010), IEEE.

[3] Biau, G., and Scornet, E. A random forest guided tour. *TEST 25* (2016), 197–227.

[4] Breiman, L. Classification and regression trees. In *Decision forests for computer vision and medical image analysis* (2017), Springer, pp. 19–38.

[5] CAST. 2018 software intelligence report. Tech. rep., CAST, 2018.

[6] Cortes, C., and Vapnik, V. Support-vector networks. *Machine Learning 20*, 3 (1995), 273–297.

[7] Drucker, H., Burges, C. J., Kaufman, L., Smola, A. J., and Vapnik, V. Support vector regression machines. *Advances in neural information processing systems 9* (1997), 155–161.

[8] Elmidaoui, S., Cheikhi, L., Idri, A., and Abran, A. Machine learning techniques for software maintainability prediction: Accuracy analysis. *Journal of Computer Science and Technology 35*, 5 (2020), 1147–1174.

[9] Ernst, N. A., and Eichmann, D. A. The future of software maintenance. *IEEE Software 16*, 1 (1999), 44–50.

[10] Geurts, P., Ernst, D., and Wehenkel, L. Extremely randomized trees. *Machine Learning 63*, 1 (2006), 3–42.

[11] Guo, G., Wang, H., Bell, D., Bi, Y., and Greer, K. Knn model-based approach in classification. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE* (Berlin, Heidelberg, 2003), R. Meersman, Z. Tari, and D. C. Schmidt, Eds., Springer Berlin Heidelberg, pp. 986–996.

[12] Hegedűs, P., Kádár, I., Ferenc, R., and Gyimóthy, T. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology 95* (2018), 313–327.

[13] Jang, J.-S., Sun, C.-T., and Mizutani, E. *Neuro-fuzzy and soft computing: a computational approach to learning and machine intelligence.* Prentice Hall, 1997.

[14] Kaur, A., and Kaur, K. Statistical comparison of modelling methods for software maintainability prediction. *International Journal of Software Engineering and Knowledge Engineering 23*, 6 (2013), 743–774.

[15] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc.

[16] Marinescu, R. An empirical study of the relationship between code smells and refactoring. *Empirical Software Engineering 9*, 4 (2004), 429–462.

[17] Molnar, A.-J. Collection of technical debt issues in freemind, jedit and tuxguitar open source software.

[18] Molnar, A.-J., and Motogna, S. Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (New York, NY, USA, 2020), ESEM '20, Association for Computing Machinery.

[19] Molnar, A.-J., and Motogna, S. A study of maintainability in evolving open-source software. In *Evaluation of Novel Approaches to Software Engineering* (Cham, 2021), R. Ali, H. Kaindl, and L. A. Maciaszek, Eds., Springer International Publishing, pp. 261–282.

[20] Montgomery, D. C., Peck, E. A., and Vining, G. G. *Introduction to linear regression analysis.* John Wiley & Sons, 2012.

[21] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report NISTIR 6859, National Institute of Standards and Technology, 2002.

[22] Oman, P., and Hagemeister, J. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992* (Nov 1992), pp. 337–344.

[23] Pearl, J. Probabilistic reasoning in intelligent systems: Networks of plausible inference. *Morgan Kaufmann* (1988).

[24] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323*, 6088 (1986), 533–536.

[25] Taud, H., and Mas, J. *Multilayer Perceptron (MLP).* Springer International Publishing, Cham, 2018, pp. 451–455.

[26] van Koten, C., and Gray, A. R. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology 48*, 1 (2006), 59–67.

[27] Wahler, M., Drofenik, U., and Snipes, W. Improving code maintainability: A case study on the impact of refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016), pp. 493–501.

Departyment of Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Email address: liviu.berciu@ubbcluj.ro, vasilica.moldovan@stud.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 2, 2023
DOI: 10.24193/subbi.2023.2.03

ON COMPOSING ASYNCHRONOUS OPERATIONS

RADU LUPŞA AND DANA LUPŞA

Abstract. Asynchronous operations are very useful for actions that wait for an external event or work for a long time, to avoid blocking the thread that launches them. Unfortunately, whether they report their termination via callbacks or via completing a future, composing several asynchronous calls is difficult and error prone. The continuations mechanism (provided, for example, in C# Task Parallel Library via `ContinueWith()`) offers limited support for scheduling a sequence of operations. In this paper we try to improve this mechanism with better support for sequencing operations and exceptions, and with support for conditionals and loops, while covering the specifics of a C++ implementation. The most recent version of our source code is at [14].

1. Introduction

Asynchronous operations are operations that are started by a thread via a function call, but that continue after the initiating call returns.

They are essential in exploiting the parallelism, either between the CPU and the peripherals or external events, or between CPU cores.

Asyncronous communication and future objects are useful mechanisms to tolerate high latencies and improve overall performance. Automatic continuation and the mechanism used for updating result values can be used to further increase applicability and performance in different application and deployment scenarios [11] [15].

Research in the domain include: analysis of different strategies for updating future objects[15], proposal for architectures to support asynchronous messages using future objects while preserving the separation between the logic and the control aspects in the implementation [10], design for asynchronous

Received by the editors: 28 February 2023.

2010 *Mathematics Subject Classification.* 68N19, 68Q85.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: Programming Languages – *Language Constructs and Features*; D.1.3 [**Software**]: Programming Techniques – *Language Concurrent Programming*; D.2.2 [**Software**]: Software Engineering – *Design Tools and Techniques* .

Key words and phrases. asynchronous programming.

stream generators, extending previous facilities [6], new framework to mimic simple synchronous programming but able to achieve fullflow processing asynchronously [4].

Combining several asynchronous operations to form a program is a difficult task. The goal of this paper is to study how to create patterns similar to structured programming. We try to improve the continuation mechanism with better support for sequencing operations and exceptions, and with support for conditionals and loops, while covering the specifics of a C++ implementation.

The rest of the paper is organised as follows: section 2 reviews the mechanisms available for getting the results for asynchronous operations. Section 3 discusses in more details the futures with continuation mechanisms. Section 4 describes the proposed framework allowing to combine simpler asynchronous operations into a more complex ones in a way similar to structured programming. The paper ends with conclusions.

2. Asynchronous operations result and chaininig

For any asynchronous operation, the caller needs to have some mechanism allowing it find out when the asynchronous operation has finished, and to retrieve any data produced by that asynchronous operation.

There are two basic mechanisms used by frameworks that offer asynchronous operations:

callbacks: When the asynchronous operation ends, a callback provided by the application is called.

futures: The call that initiates the asynchronous operation returns an object (a future or something that can be used as such) that can be polled by the application to find out if the asynchronous operation ended and, possibly, to wait until the operation ends.

2.1. Callbacks.

It should be noted that there are lots of ways in which various libraries offer the callback mechanism. For some, the callback is given as an argument to the function that starts the asynchronous operation, for others the callback is registered ahead of time to be called when any operation of a certain type completes. Also, for some libraries, the callback receives as an argument the result of the just finished asynchronous operation; for others, the callback is supposed to call some function to retrieve the asynchronous operation result and also to free any resources associated to the asynchronous operation.

Using callbacks is similar to *goto*-based programming. It is extremely flexible and can be used to build structures like sequence, *if-then-else*, or loops, but it is tedious and error-prone to use directly.

Another issue with callbacks is that they execute sometimes on some thread spawn by the library providing the asynchronous operations, and sometimes from within functions of that library called by the application. In either case, the thread on which the callback is called may hold some mutexes or may have to do some more work inside the library after the callback returns. As a result, there may be restrictions upon what library functions may be called from within the callback; calling forbidden functions may lead to the call being rejected, to a deadlock, or even to undefined behavior.

2.2. **Futures.** Futures were introduced by Liskov and Shrira [9]; they call them promises. They are available in most mainstream languages and recommended for asynchronous operations. Meyers [12] gives a very good explanation of the futures mechanism in C++11/14, while [5] discusses various future related issues and their approach in Kotlin.

Futures are better fit for structured programming mechanisms, and the main pattern is to start several asynchronous operations, that will then proceed in parallel, and then wait for all to finish and gather and using their results.

Having several operations executing one after another or one operation executing several times in a cycle requires however a thread that waits for the future corresponding to the previous operation and then calls the next one. This requires a thread that gets blocked.

3. Futures with continuations

The *futures with continuations* mechanism was introduced in C#/.NET Task Parallel Library (TPL) [3], and is also available in Boost, in a C++ standard proposal [1] or in the `stlab` library[8].

Extending the futures mechanism with continuations in C++ standard library is a debated issue. There are proponents, like [13], which proposes this as an extension to the basic C++ `std::future` mechanism. There are also opponents, like [7], which argue that `std::future` should remain a simple, wait-only type that serves a concrete purpose of synchronously waiting on potentially asynchronous work and they find that it should not be extended with continuations.

Basically, with futures with continuations, the asynchronous function returns a future that will complete when the asynchronous operation completes. However, beside the possibility to check whether the future has completed or to wait for its completion, the caller can also set a callback to be called when the future completes. The callback typically executes on some thread pool, usually called an *executor*.

The basic feature of the *future with continuation* mechanism is that it decouples the registering of the callback (the continuation) from the asynchronous call itself. With the classical callback mechanism, the callback is supplied as an argument to the asynchronous call. With the *future with continuation*, the asynchronous call returns a future — called `Task` in C# TPL — which can be used in all 3 ways of getting the result from the asynchronous operation:

wait for completion: call `Wait()`,
poll: examine the `IsCompleted` property,
register a callback: call `ContinueWith()`.

Decoupling of the callback (continuation) from the asynchronous operation has several benefits.

First, the continuation is registered after the asynchronous operation is started, at any convenient time for the caller. The classical callback needs to be prepared beforehand and, in extreme cases, it might get executed even before the control returns from the call that initiates the asynchronous operation.

Second, the classical callback executes on a thread controlled by the framework providing the asynchronous operation. This usually poses some restrictions regarding which functions (especially, from the same framework) can be called from within the callback; disobeying those restrictions may lead to calls being rejected, deadlocks, or, in extreme cases, undefined behavior. The continuations, on the other side, execute on a thread in a thread pool provided by the *future with continuation* framework; thus, no restrictions exist with respect to which functions can be called from within the continuation.

Finally, continuations are more flexible, as their handling is independent of the asynchronous operation. It is possible to add multiple continuations to the same asynchronous operation (by calling `ContinueWith()` multiple times). It is also possible to add a continuation that is to be invoked when all operations from a set of asynchronous operations complete (by calling `WhenAll()`).

4. Composable asynchronous operations

The already existing mechanisms presented in the previous section evolved in a bottom-up manner, being provided *as-is*, and to be used as the programmer sees fit.

In this paper, we try a more systematic approach: we examine how to write an asynchronous function as a composition of smaller asynchronous functions. The construction should thus be similar to the way a classical, sequential, function is constructed by composing smaller functions.

So, our goal in this section is to create a framework allowing to compose asynchronous functions in the classical programming structures: sequence, conditional (*if-then-else*), and loop, as well as a *try-catch* mechanism. The result of composing asynchronous functions should be a new asynchronous function.

At the same time, we want not to lose the possibility of executing the asynchronous operations in parallel, when appropriate.

One trivial way to compose asynchronous functions is to treat them as synchronous: just call the function and then wait for the asynchronous operation to complete, blocking the execution of the current thread. This defeats the purpose of asynchronous operations. It needs to create a potentially large number of threads, and then, each time a thread blocks waiting for an asynchronous operation, it must switch to a new thread. Creation of a thread and switching from a thread to another are expensive operations because they involve going through an operating system call. So, instead of blocking threads, the finishing of each asynchronous operation needs to trigger a callback that would call the function that launches the next asynchronous operation.

Finally, the framework must be a pure library, without needing any special support from the programming language, such as coroutines. Coroutines are indeed very useful for describing an asyncronous function that calls other asynchronous function, and languages like C# and Python support this via the *async-await* mechanism: the asynchronous function is declared *async*, so that the compiler or interpreter knows it should be implemented as a coroutine, and, when the function calls some other asyncrounous function, its coroutine gets suspended until the called asynchronous operation completes, at which time the coroutine resumes. This allows the programmer to write code almost as if it were ordinary sequential code. However, some languages do not support coroutines and, even though recent C++20 does, there are systems where, for various reasons, upgrading to C++20 is not possible.

We choose C++ language for implementing the framework.

4.1. **Basic building blocks.** First, the basic building blocks will be asynchronous functions. Each asynchronous function will return a future, that completes when the asynchronous operation completes.

The future mechanism needs to allow the possibility to hook to a future a callback that gets executed when the future completes.

4.2. **Sequence.** In a sequence of asynchronous calls, each asynchronous operation would start after the previous one finishes. The full sequence becomes an asynchronous operation that completes when the last asynchronous operation of the sequence completes.

The basic support for a sequence is the future with continuation mechanism, described in section 3.

The C# `ContinueWith()` operation, or its equivalent `then()` in the C++ standard proposal, takes a future, representing the result of a first asynchronous operation, and a function and returns a future that will get the result of the second operation.

However, looking at the continuation enqueueing operation from the composability perspective, there are two aspects to be noticed.

First, the function for the second operation in the sequence takes as argument a future (a `Task`, in C#) instead of its value.

Second, the original `ContinueWith()` returns a future that completes when the function passed as argument returns. This works if the continuation is a synchronous function. However, if the function is asynchronous, the result of `ContinueWith()` is a future that completes when the second operation starts. The value of that future is a second (inner) future and its value is what the user code is interested in. C# provides a function called `Unwrap()` that creates and returns a future that completes when the inner future completes.

The C++ proposed `then()` function does the *unwrap* automatically, if the continuation function returns a future.

As an example, consider an asynchronous function that looks up in some database. For simplicity, let both the key and the value be of type `int`, and that we want the asynchronous equivalent of a synchronous code like `lookup(lookup(k))`. Then, the asynchronous version of lookup needs to be declared as

```
Task<int> AsyncLookup(int)
```

and the usage would be

```
Task<int> result = AsyncLookup(k).
    ContinueWith((Task<int> arg) => AsyncLookup(arg.Result)).
    Unwrap()
```

The last `Unwrap()` is needed because the future returned `ContinueWith()` completes when the second lookup starts. Its value is a second future, that completes when the second lookup completes, and the value of that inner future is the result of that second lookup — which is what we are interested in.

Java's `CompletableFuture` [2] offers two distinct functions for adding a continuation to a future, `thenApply()` and `thenCompose()`, the first behaving like C#'s `ContinueWith()` and the second like `ContinueWith()` followed by `Unwrap()`.

Standard C++ futures do not offer continuations, but there is a proposed `then()` function on a future that does an automatic `Unwrap()` if and only if the continuation function returns a future, thus being assumed to be an asynchronous function.

The equivalent implementation of the double lookup above using the C++ standard proposal would be:

```
std::experimental::future<int> result = AsyncLookup(k).
    then([](std::experimental::future<int> arg) {
        return AsyncLookup(arg.get());
    });
```

We propose here some small modifications that, while mostly cosmetic, emphasize on composability. Our continuation enqueueing operation is declared as:

```
template<typename R, typename Func, typename Arg>
Future<R>
        addAsyncContinuation(Executor& executor, Func func, Future<Arg> fArg)
```

Aside from the explicit specification of the thread pool used for executing the continuation (`executor`) and the fact that `addAsyncContinuation()` is a stand-alone function (not a class member), the differences are that `func` takes a simple value of type `Arg` (not `Future<Arg>`), and the returned `Future<R>` completes when the asynchronous operation completes.

The above example becomes:

```
Future<int> tmp = AsyncLookup(k);
Future<int> result = addAsyncContinuation<int>(executor, AsyncLookup, tmp);
```

4.3. **Conditional.** Implementing an *if-then-else* can be done in a straightforward way even without framework support. An asynchronous function implementing an *if-then-else* could have the following structure:

```
Future<int> conditional() {
  Future<int> f1 = foo();
  Future<int> f2 = addAsyncContinuation<int>(executor,
    [](int v) -> Future<int> {
      if(v>0) {
        return thenFunc(v);
      } else {
        return elseFunc(v);
      }
    }, f1);
}
```

4.4. **Loop.** Creating a loop around an asynchronous function is the main contribution of this paper. While the other structured programming constructs are relatively easy to obtain from the continuation enqueueing operation, creating a loop is much harder.

The need for loops arise in many places. For example, consider reading and parsing data coming via a connection. Suppose that reading bytes is provided as an asynchronous operation. Also, suppose that one needs to implement parsing as an asynchronous operation, that returns a parsed value (an integer, or some more complex object). To obtain that, the parsing operation would have a loop where it starts a read and, when the read completes, parses the read data and, if not complete, starts a new read and repeats.

As another example, handling a client from a server is also a loop where a (parsed) request is read, the response is sent, and the cycle repeats until the connection is closed or the request is to terminate the connection.

The basic loop construction needs a loop condition and a loop body.

Similarly with the case of the sequence, the body of the loop will be a function that launches an asynchronous operation and returns a future.

The asynchronous operation for each iteration is started after the asynchronous operation for the previous operation completes. Additionally, to pass information from one iteration to the next, the function acting as the loop body takes a value of some type and returns a future of the same type, whose value is passed to the loop body for the next iteration.

For the loop condition, we will use a synchronous function, taking as the value the value passed from one iteration to the next.

Putting all together, the result is a framework function declared as follows:

```
template<typename R, typename LoopFunc, typename PredicateFunc>
Future<R> executeAsyncLoop(Executor& executor, PredicateFunc loopingPredicate,
    LoopFunc loopFunc, R const& startValue)
```

The `startValue` argument is the initial value to be checked by the predicate function and to be passed to the loop body function for the first iteration. Consequently, `executeAsyncLoop()` creates an asynchronous function, taking a value — that is passed as the initial value for the loop body — and returning a future — that completes when the loop ends and receiving the returned value from the last iteration.

The following is a simple example of how a loop can be created. The function `delayedResult()` returns a future that is completed with the value given as the last argument after a time given as the second argument (1000) after it starts. The loop will thus count up to 10 with each step taking the given amount of time.

```
Future<int> f = executeAsyncLoop<int>(executor,
    [](int v)->bool {return v < 10;},
    [&alarmClock](int const& v)->Future<int> {
        return delayedResult(alarmClock, 1000, v + 1); },
    0);
```

To test in a slightly more realistic scenario, a small demonstrative server was implemented. The server repeatedly reads two integers (as sequences of digits) and responds with their sum. Below is a small excerpt that demonstrates the usage of the asynchronous loop:

```
Future<bool> executeOneRequest() {
  Future<int> fa = m_reader.readInt();
  Future<int> fb = addAsyncContinuation<int>(*m_pExecutor,
    [this](int a)->Future<int> {
      if(a > 0) return m_reader.readInt();
      return completedFuture<int>(0);
    }, fa);
    Future<bool> result = addAsyncContinuation<bool>(*m_pExecutor,
      [this,fa](int b) -> Future<bool> {
        if(fa.get() > 0) {
          int sum = fa.get() + b;
          std::shared_ptr<std::string> pSumStr =
                  std::make_shared<std::string>(std::to_string(sum) + "\n");
          return m_pSocket->send(pSumStr);
        } else {
          return completedFuture<bool>(false);
        }
    }, fb);
    return result;
}

Future<bool> run() {
  return executeAsyncLoop<bool>(*m_pExecutor,
    [](bool b){return b;},
    [this](bool b){return executeOneRequest();},
    true);
}
```

The the function `executeOneRequest()` launches an asynchronous operation that reads two integers over the socket and sends back their sum. It uses in turn other asynchronous functions for receiving and for sending data over the socket (`readInt()` and `send()`). The function immediately returns a boolean future that completes with *true* after the sum has been sent. The future completes with *false* if the client closes the connection or if an error occurs. Then, the `run()` function does the complete handling of a client: it returns a future that completes when the handling of the client is over (either because the client disconnects or because an error occurs.

The pattern solved by the `executeAsyncLoop()` is thus repeatedly calling an asynchronous operation that pulls data from a source (a connection, for instance) or pushes data into a sink.

Obtaining the same effect without `executeAsyncLoop()` is possible, but tedious. The implementation of the function `run()` from above, with C# TPL, would be (a helper function, `loopBody()`, is needed):

```
void loopBody(TaskCompletionSource result) {
  Task.Factory.StartNew(executeOneRequest)
    .ContinueWith((Task<bool> execResult) => {
      if(execResult.Result) loopBody(result);
      else result.SetResult();
    }
}
```

```
Task run() {
  TaskCompletionSource ret = new TaskCompletionSource();
  loopBody(ret);
  return ret;
}
```

One difficulty that should be noted about this example is that there are a lot of shared pointers. They are needed because of the asynchronous nature of the code. The actual functions usually only do some setup, so local variables will be long gone when the actual computation happens. Note that this is not a characteristic of the framework, but rather a common issue of asynchronous functions.

4.5. **Exceptions.** In regular programming, exceptions provide a mechanism for easily exiting from the structures.

Providing the same mechanism for asynchronous programming requires several elements which we will present below.

First, the futures can complete in two modes: with a value or with an exception.

Next, the sequence stops if a step completes with an exception. The next steps are skipped, but the sequence result completes with the same exception. Concretely, this means that, for `addAsyncContinuation()`, if the future given as argument completes with an exception, the returned future completes with that exception without the function argument being called.

Note that this behavior is quite distinct from what C# TPL is doing. In C#, if a `Task` completes with an exception and its continuations are set to execute only on normal completion, then the continuations resulting `Task`s complete as canceled. This means that, in order to get the exception, one needs to examine the `Task` corresponding to the failed operation; subsequent `Task`s have no information on the exception.

Similarly to `addAsyncContinuation()`, for `executeAsyncLoop()`, if the body completes with an exception, the loop stops and the future returned by `executeAsyncLoop()` completes with that exception.

Finally, the equivalent of the *try-catch* mechanism is also needed. Our framework provides a function declared as

```
template<typename T, typename CatchFunc>
Future<T>
        catchAsync(Executor& executor, CatchFunc catchFunc, Future<T> value);
```

Its `catchFunc` argument must be an asynchronous function taking an `std::exception_ptr` and returning `Future<T>` and gets the role of the *catch* block. The semantic of `catchAsync()` is the following:

- It immediately returns a future;

- If `value` completes normally, the future returned from `catchAsync()` completes with the same value;
- If `value` completes with an exception, `catchFunc()` is called with that exception as argument and the future returned by `catchAsync()` will complete with the value (or exception) returned (respectively thrown) by `catchFunc()`.

Using exceptions, the small server of the previous section can be re-written in a simpler way, eliminating the repeated checks that the state of handling the client is still normal, and instead relying on the "fast exit" mechanism of the exceptions:

```
Future<bool> executeOneRequest() {
    Future<int> fa = m_reader.readInt();
    Future<int> fb = addAsyncContinuation<int>(*m_pExecutor,
        [this](int a)->Future<int> {
            if(a < 0) {
                throw Flag::client_disconnected;
            }
            return m_reader.readInt();
        }, fa);
    Future<bool> result = addAsyncContinuation<bool>(*m_pExecutor,
        [this,fa](int b) -> Future<bool> {
            if(b < 0) {
                throw Flag::invalid_input;
            }
            int sum = fa.get() + b;
            std::shared_ptr<std::string> pSumStr =
                 std::make_shared<std::string>(std::to_string(sum)+"\n");
            return m_pSocket->send(pSumStr);
        }, fb);
    return result;
}
```

5. Conclusions

We developed, and presented here, a framework for using the futures with continuations mechanism in a way very similar to the classical, structured style, programming. This way, programs using asynchronous calls look reasonably similar to regular programs, and this is probably the best that can be achieved without language support like coroutines. Its central point is the asynchronous loop mechanism.

As a limitation, the lifetimes of the variables are not very obvious, and shared pointers are required almost everywhere because of this. This is not a limitation of the framework per se, but a result of the asynchronous work model. A study of possible improvements in this area may come as a future work.

Yet another future direction would be to attempt to use the looping mechanism to produce or to consume a stream of values, in the reactive programming style.

References

[1] C++ reference. extensions for concurrency. `https://en.cppreference.com/w/cpp/experimental/future/then`. Accessed: 2023.

[2] `Java Platform, Standard Edition 8 API Specification, CompletableFuture` . `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html`. Accessed: 2023.

[3] David Pine, e. a. Task parallel library (tpl). `https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl`, 2022. Accessed: 2022.

[4] Duan, J., Yi, X., Wang, J., Wu, C., and Le, F. Netstar: A future/promise framework for asynchronous network functions. *IEEE Journal on Selected Areas in Communications 37*, 3 (2019), 600–612.

[5] Elizarov, R., Belyaev, M., Akhin, M., and Usmanov, I. Kotlin coroutines: Design and implementation. Onward! 2021, Association for Computing Machinery, p. 68–84.

[6] Haller, P., and Miller, H. A reduction semantics for direct-style asynchronous observables. *Journal of Logical and Algebraic Methods in Programming 105* (03 2019).

[7] Howes, L., Grynenko, A., and Feldblum, J. Continuations without overcomplicating the future. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0783r0.html`, 2017. Accessed: 2022.

[8] Lab, A. S. T. stlab: Api documentation. futures. `https://stlab.cc/libraries/concurrency/future/`. Accessed: 2023.

[9] Liskov, B., and Shrira, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988), PLDI '88, Association for Computing Machinery, p. 260–267.

[10] Manolescu, D. A. Workflow enactment with continuation and future objects. 40–51.

[11] Marshall Cline, e. a. A unified futures proposal for c++. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1054r0.html`, 2018. Accessed: 2022.

[12] Meyers, S. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, 1st ed. O'Reilly Media, Inc., 2014.

[13] N., G., A., L., H., S., and S., M. A standardized representation of asynchronous operations, tecnical report n3538. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3558.pdf`, 2013. Accessed: 2022.

[14] Radu, L. futures-demo `https://github.com/rlupsa/futures-demo`, 2023.

[15] Ranaldo, N., and Zimeo, E. Analysis of different future objects update strategies in `ProActive`. In *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), pp. 1–7.

Computer Science Department, Babeş Bolyai University, Cluj-Napoca, Romania

Email address: `radu.lupsa@ubbcluj.ro, dana.lupsa@ubbcluj.ro`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 2, 2023
DOI: 10.24193/subbi.2023.2.04

FACILITATING MODEL TRAINING WITH AUTOMATED TECHNIQUES

BOGDAN-EDUARD-MĂDĂLIN MURSA, MÁTYÁS KUTI-KRESZÁCS, CRISTIANA MOROZ-DUBENCO, AND FLORENTIN BOTA

Abstract. Automating artificial intelligence (AI) model training has emerged as a significant challenge in the field of automation. The complete pipeline from raw data to model deployment poses the need to define robust processes that ensure the efficiency of the services that expose the models. This paper introduces a generic architecture for automating data preparation, training of models, selection of models, and deployment of models as web services for third-party consumption using Microsoft Azure Machine Learning's (AzureML) CI/CD tools. We conducted a practical experiment utilizing AzureML pipelines with predefined and custom modules, demonstrating its readiness for integration into any production application. We also successfully integrated this architecture into a real-world product designed for industrial forecasting. This practical implementation demonstrates the effectiveness and adaptability of our approach, indicating its potential to address diverse training needs.

1. Introduction

Over the past decade, Artificial Intelligence (AI) has evolved from a buzzword to a state-of-the-art technology, providing reliable solutions in various domains such as fraud detection, healthcare, predictive maintenance, energy management, and retail. AI applications are now being used as stand-alone solutions in businesses' core processes. However, as the demand for AI grows exponentially, traditional processes for training AI models, which were once characterized as academic experiments, have now become a logistical problem. There is a need to migrate to reliable processes that can automate all

Received by the editors: 29.06.2023.

2010 *Mathematics Subject Classification.* 68T01.

1998 *CR Categories and Descriptors.* D.2.11 [**SOFTWARE ENGINEERING**]: Software Architectures – *Patterns (e.g., client/server, pipeline, blackboard)*; I.2.1 [**ARTIFICIAL INTELLIGENCE**]: Applications and Expert Systems – *Industrial automation* .

Key words and phrases. Artificial Intelligence, Automation, Optimization.

the steps, from cleaning a raw dataset to training the model and exposing it to other services that can consume it in real-world scenarios [12, 14, 26].

Moreover, there has been an emphasis on the necessity of creating user-friendly interfaces for software developers without an AI background, which can facilitate the maintenance of the AI processes encapsulated in modules, following the principles of gray-box. These requirements have already been addressed in the domain of software engineering, particularly in web programming, using the Continuous Integration/Continuous Deployment (CI/CD) mechanism and web services. In this paper, we present the latest advancements in automating AI model training using AzureML's CI/CD tools. Our approach allows for the automation of data preprocessing, training various AI models and selecting the best performing one, deploying the model as a web service for third-party consumption, while ensuring high availability and scalability of the service.

Our proposed solution was implemented and validated through an academic-industrial partnership aimed at developing a forecasting platform for industrial indicators. Subsequently, we present a comprehensive review of the logistics and challenges associated with AI model training (Section 2), followed by an overview of state-of-the-art techniques and successful applications of automating the AI model training process in various industries (Section 3). In Section 4, the final section of this paper, we describe our proposed generic architecture to automate the model training and deployment process using AzureML.

2. Problem definition

Developing a machine learning based solution in an industrial environment consists of two steps: building the model and deploying it into production. While the first part might be more interesting, it is the second part that takes the longest - all the tasks that come after a model is built and optimized. According to Gartner [1], one of the top five factors due to which 85% of Big Data projects fail is the complexity to deploy them.

Algorithmia's "2021 State of Enterprise ML" [3] reports that 64% of all organizations need at least one month to deploy a machine learning model and that the data scientists from 38% of organizations spend more than 50% of time deploying ML models to production.

According to [4], the development of a machine learning based solution in an industrial environment can be split into four stages. Each of these stages can be split into smaller steps, and [20] presents various issues and concerns associated to each step, as explained below.

The first stage concerns data management - preparing the data. Data is collected, gathered and understood. Problems can appear especially in large production environments, where, using the principle of ”single responsibility” [16], applications are usually built as multiple services communicating to each other, and this can easily lead to data being stored in different locations and forms by different services. After collection, data has to be cleaned and, according to [13], data cleanliness is the main reason causing expert to doubt the quality of their work. Insufficient, biased, noisy, irrelevant or imbalanced data can lead to either the under- or overfitting of the model. Moreover, if multiple data sources have to be integrated into a single one, they might differ in schema, convention or the procedure of storing and accessing the data. Once the data is preprocessed, it might be necessary to be augmented because in real life, more often than not, data is unlabeled and, since supervised learning techniques require labeled data for training, the process of assigning labels to large volumes of data can be a tedious task. And, finally, the labeled data has to be analysed in order to discover biases or unexpected distribution shifts. One area that is especially challenging in this step is visualization for data profiling, since there are very few tools available for efficiently executing this task.

The following stage is about model learning. Firstly, the best-fitted model for the problem at hand needs to be chosen. As shown in [28], complexity is one of the most important factors when choosing a model to be used in an environment with resource constraints. In practice, simpler models, such as Decision Trees, Random Forests, Principal Component Analysis etc., are chosen instead of deep learning or reinforcement learning techniques because they require less resources and also lead to an decreased development and deployment time. Moreover, depending on the field of application, being able to interpret the output of a machine learning model can outbalance even its performance. Then, the model has to be trained. This step usually requires increased computational resources, which leads to increased costs. For instance, in [25], it is shown that the training of the BERT model [7] costs at least $50 K, which can be impossible to afford for many companies. Even more, challenges can arise while selecting hyper-parameters in order to find the optimal setting for the model. The size of the hyper-parameter optimization task can grow exponentially in the worst case, leading to tough computational challenges. Mainly when talking about deep learning techniques, this process can be extremely expensive and resource-heavy.

Once the model is trained, it must be verified. This stage involves defining requirements for the model, which should not solely prioritize increased model performance but also consider business-driven metrics for evaluating and monitoring the model in a production environment. Verification of all requirements is necessary, including adherence to business-defined regulatory frameworks, in addition to mathematical correctness or error bounds. Testing the model in a real-life setting is ideal for ensuring quality, but this can pose challenges regarding safety, security, and scaling.

And, finally, the last stage in the development of an ML-based solution in industrial environments is the deployment of the model. The model has to be implemented so that it can be consumed and an infrastructure for running the model needs to be built. Due to the fact that machine learning models can explicitly depend on external data, a lot of engineering anti-patterns, such as correction cascades, are widespread in software that uses ML [24]. Once the system is deployed to production, its maintenance intervenes. The system has to be monitored, but this process is still in the early stages within the ML community, and monitoring the overall performance of a machine learning model is still an open problem. Yet, the biggest challenge emerges when the existing models needs to be adapted to new data and the new model artifact delivered to production. While software engineering solves this problem using continuous delivery, things are more complicated with machine learning problems because, unlike regular software products that only have changes in code, ML solutions can be changed on three aspects: data, model, and code.

Therefore, we can state that the process of developing machine learning based solutions in production environments requires not only skills and time, but also new and emerging technologies that can ease the process.

3. Advances in Automated AI Training

Recently, AI has emerged as a cutting-edge technology with numerous use cases across diverse industries. AI models have been employed either as novel solutions to problems, as assisting systems, or as complete replacements for human intervention. However, this rise in demand has revealed certain challenges, such as the necessity for dependable model training, deployment, and real-time production consumption mechanisms. Large cloud service providers have promptly identified this demand and addressed it by offering a comprehensive suite of tools that facilitate large-scale AI model training and deployment. Furthermore, modern CI/CD mechanisms [15] have been integrated to ensure scalable solutions that meet the needs of high availability, performance, and logging.

AzureML [18], a cloud-based machine learning platform developed by Microsoft, provides a comprehensive suite of tools necessary for the implementation of artificial intelligence models for various use cases. With its ability to integrate with different AI model architectures, it has been utilized by several large companies as a trusted ecosystem for the entire pipeline from training to deployment. For instance, American Express uses AzureML to develop apps for fraud detection, Mediktor employs it for healthcare solutions to check symptoms, E-ON applies it to manage energy in solar panel farms and predict energy solutions, Belfius uses it to help detect fraud and money laundering, Cognizant and Claro personalize and improve users' learning experience with AzureML and Epiroc advances manufacturing innovation with its help [19].

On the other side, AzureML gains popularity in the scientific literature as well. [22] analyzes eight two-class and three multi-class machine learning algorithms for network intrusion detection using AzureML as a solution to the limitation of traditional models, which focus rather on the improvement of the attack detection rate and the reduction of false alarms rather than on time efficiency. The proposed algorithms are analyzed and evaluated not only on their performance related to the task, but also in terms of training and prediction time, concluding that the use of AzureML leads to saving computational resources and, thus, reducing excessive costs. Moreover, the study highlights the fact the AzureML is useful for large datasets handling, as it can successfully serve as an expedient Integrated Development Environment.

In [23], a data-driven machine learning workflow is proposed for forecasting the outcome of Business to Business (B2B) sales. The workflow was implemented and deployed to a B2B consulting firm's sales pipeline using the AzureML platform. This cloud-based solution was chosen because it can be easily integrated into existing Customer Relationship Management (CRM) systems allowing for more scalability than traditional solutions. What is more, AzureML supports the creation of models' endpoints on Azure Kubernetes Service, which ensures high scalability and low latency for request-response service, therefore being suitable for production-level deployments. The authors conclude that the AzureML-based workflow is also highly sustainable due to relying on cloud computing power, rather than on on-premise resources.

As a solution for the task of designing the complex systems that an electrical machine consists of, [21] employ AzureML as a means of optimization and best candidate selection. That is, the platform was used to compare two searching algorithms, namely Boosted Decision Tree and Multiclass Neural Network, in order to predict the best configuration of an electric motor according to the maximum efficiency. The advantages presented by the usage of the platform as compared to the tools used on the developers' local computers consist in the 10-times decrease in work time and the simplification of the project creation, taking 15 minutes instead of 3 days.

In [27], one of the main capabilities of the AzureML platform is presented in-depth: automated machine learning. In order to help developers with little ML-related knowledge to build solutions that employ ML models, AzureML provides the possibility of using automated ML, such that developers do not need to fully understand the processes of selecting the learning algorithm or tuning the hyper parameters. Being given a dataset and only a few configuration parameters, AzureML can provide a high-quality, already trained, ML model that can be used for predictions, without further modifications.

The paper [8] once again demonstrates the advantages of the automated machine learning capability of the AzureML platform. Being applied in medical environment, specifically, for predicting antimicrobial resistance and selecting appropriate treatment, the benefit of automated ML presents itself. This paper presents a procedure that is easy applicable and, most importantly, can be explained and even used by non-technical experts, leading to the conclusion that AzureML can be used as a decision tool for physicians, the deduced models proving good performance.

In this paper, we describe our process of training a scalable number of AI models for a generic forecasting platform. Alike explained in [22], with our approach, we also intend to reduce the computational costs and use a solution that easily handles large datasets. Since we need high scalability and low latency and, most importantly, an architecture that is suitable for production-level deployments, our choice is based on the same reasons as presented in [23]. Similar to the work presented in [21, 27, 8], we need to train multiple AI models and choose the one that yields the best results. And, finally, the most notable characteristic of the AzureML platform that motivates the choice of AzureML for all of the presented solutions, including our own, is the possibility of being used even by non ML-specialized developers. For all of these reasons, our proposed procedure is inspired by these successful stories of AzureML's deployment, combining and adjusting the parts that suit our needs.

4. Automating Model Training Process

The present section provides an overview of an experiment that involves utilizing the most advanced state-of-the-art guidelines to define and construct a sturdy pipeline comprising all necessary steps required for training a model, using one or more datasets, and deploying it onto an infrastructure to enable real-time consumption by clients. Our objective is to streamline the conventional process of training AI models, enabling individuals without specialized expertise to train and deploy models through an automated procedure to the greatest extent possible.

The following sections will elaborate on an anonymous methodology that our university team employed while collaborating with a company operating in the IT industry that sought assistance in the field of artificial intelligence. Given that our deliverable was intended for utilization in a live production environment, our aim was to incorporate, right from the beginning, techniques used within the software development industry, which enable principles such as scalability, versioning, monitoring, and high availability. By implementing such techniques, we intended to ensure the functionality and effectiveness of our deliverable in a demanding production environment.

Furthermore, within the scope of our collaboration, we agreed to deliver a preliminary product consisting of a series of trained neural networks, and in parallel, we developed a protocol for knowledge transfer that enabled the company to take ownership of the pipeline process that we had designed. As the company's team of developers lacked expertise in the field of artificial intelligence, we developed a grey-box mechanism that allowed them to not only implement future model training but also edit and enhance modules defined within the pipeline. To accomplish this, we proposed a generic process with easily generalized steps that could be assembled into a framework with user-friendly features. We initiated this approach from the very beginning of the project to ensure that the process could be readily integrated into the existing framework, and easily manipulated like a puzzle.

After brainstorming with the architect of the company, we reached an agreement on a solution that was compatible with their current technology stack. This solution entailed using Microsoft AzureML as a framework to define modules, which were coded in Python, to facilitate data storage, data cleaning and augmentation, model training, and model deployment. Through this solution, we aimed to create a cohesive and efficient pipeline for the company, which would facilitate the utilization of the current technology stack, while also enhancing the process of developing and implementing AI models.

4.1. **Pipelines architecture.** During the architecture brainstorming process, one of the most important considerations was the business requirement of receiving various datasets from clients and subsequently triggering the pipeline. These datasets were sourced from diverse warehouses and in different formats, including SQL databases, NoSQL databases, and files. Consequently, a layer was required to abstract the source of the training data and to allow for pipeline interaction, providing an interface that exposed simple data downloading operations, but also the uploading of the results. This approach enabled us to provide a seamless and flexible pipeline that could process a wide range of data sources, while also facilitating the integration of multiple data sources into the pipeline. In this particular case we can consider that these modules will receive the dataset as input, and output it with the data processing operations applied.

Figure 1 represents an overview of the proposed pipeline that represents a state-of-the-art mix between software engineering principles in CI/CD deployments and procedures of AI model training. After defining the data interface,

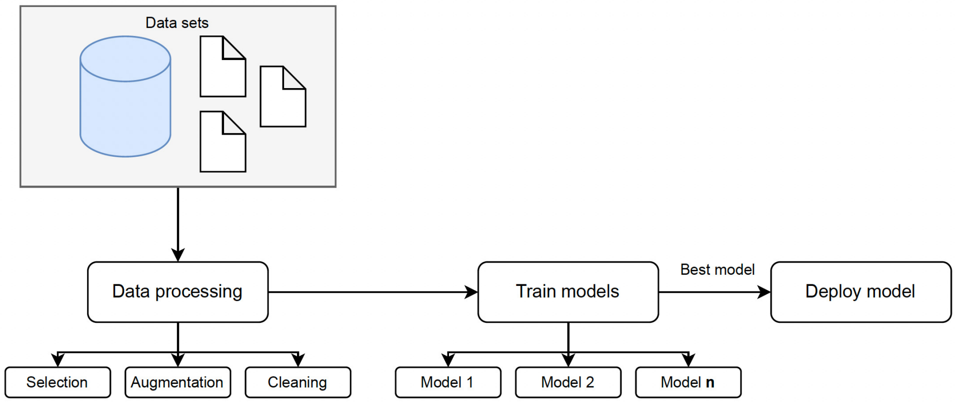


Figure 1. Generalized architecture of a model training pipeline

we proceeded to develop a series of modules that would code the state-of-the-art operations required in the process of training an AI model. Each module was defined through an interface, and each model had an input and output, enabling seamless assembly within the desired pipeline. To initiate the pipeline, we first defined modules for the most well-known operations of data processing, beginning with data cleaning (i.e., removing empty values, deduplication, and filling incomplete data), data augmentation (i.e., one-hot encoding of categorical columns or data normalization), and column selection (i.e., selecting only specific columns if needed). Through this process, we aimed to establish a flexible and scalable pipeline capable of performing various data processing operations to facilitate the effective and efficient training of AI models.

After the data cleaning process, the next step in our pipeline involved model training. This stage was comprised of an umbrella module that encapsulated various sub-models, each representing a different type of model such as neural networks, logistic regressions, decision trees, and many others. The module initiated the training of all these models and subsequently chose the model with the highest metric, which was determined based on its accuracy. The output of this pipeline module was a flexible binary file that could be deployed in the subsequent stage of model deployment.

To expose the trained binary model to other components of the company's software stack, we implemented a mechanism that allowed for the input of new data and retrieval of inferred results. Following best practices in web development, we decided to utilize microservices due to their capabilities in high availability, versioning, and scalability.

Following the architecture described, we propose a hands-on application of Microsoft AzureML pipelines and its features following the state-of-the-art CI/CD principles that will enable the desired automatization of the AI tasks [9, 11, 5].

4.2. **AzureML Pipelines.** The Microsoft AzureML platform represents a significant advancement in cloud-based technology, providing users with a range of functionalities for data processing, model training and versioning, and large-scale deployment. Given our architectural requirements and project needs, AzureML proved to be a highly suitable solution for our work.

AzureML's pipeline designer functionality is a low-code/no-code solution for developing machine learning pipelines both for model training and prediction. The visual components enable the configuration of the pipelines, which can be performed with limited knowledge of the underlying transformations and ML algorithms. Also, its pre-built infrastructure with curated environments allows us to run these pipelines with minimal maintenance overhead, while its versatile suite of features enabled us to develop the necessary modules for our project. The visual pipeline designer and the managed infrastructure with curated environments in the AzureML platform provide an innovative and efficient solution for organizations seeking to streamline their AI workflow and improve the efficiency of their operations.

Microsoft AzureML provides another crucial interface for managing the various datasets utilized by clients, through the implementation of Azure Datasets. This functionality allows for the loading of datasets in tabular format from diverse sources, including both internal and external data repositories. Moreover, an Azure Dataset can be versioned, and its creation can trigger the execution of associated pipelines. To illustrate the practical application of these concepts, Figure 2 presents a diagram of a simulated AzureML pipeline. While we are unable to provide detailed insights into the proprietary work conducted with external collaborators, we can attest that the principles guiding the construction of this pipeline align with the ones from the original experiments.

The demo pipeline depicted in Figure 2 begins with an *Azure Dataset* that serves as the primary input for subsequent data processing steps. Specifically, the pipeline executes a sequence of *data processing modules*, including column selection, missing data removal, and data splitting. The *data splitting module* divides the input dataset into separate training and testing sets, which are subsequently fed into a training module for model development. Each module contains customizable properties that can be adjusted according to specific user needs. For instance, the percentage of data allocated to the training and testing sets in the data splitting module can be configured as needed.

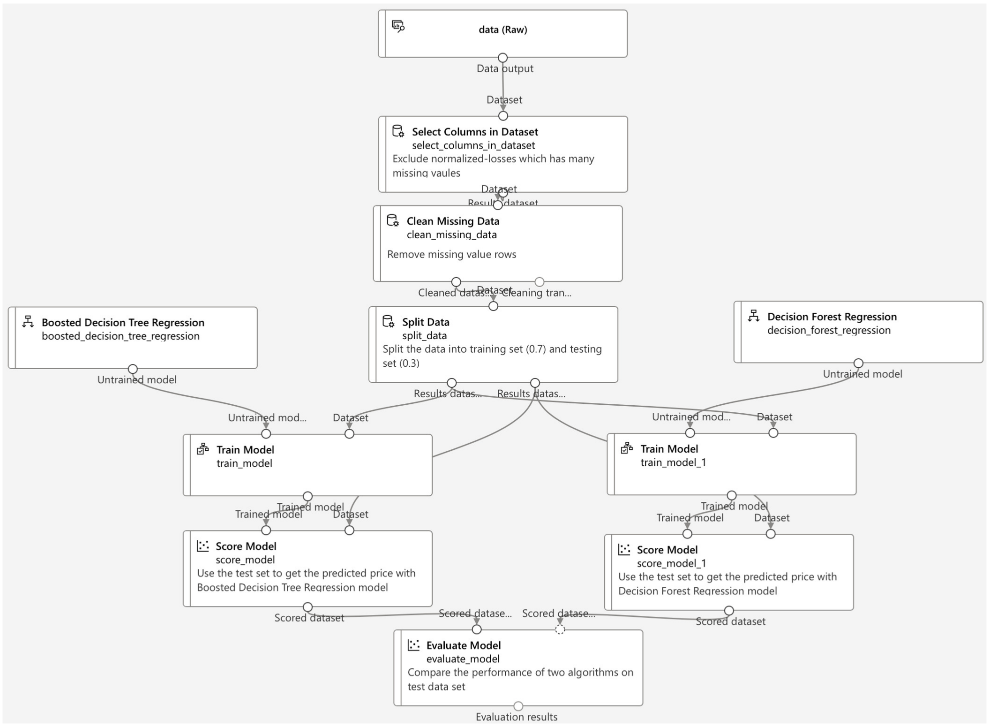

data (Raw)
Data output
Dataset
Select Columns in Dataset
select_columns_in_dataset
Exclude normalized-losses which has many missing vaules
Dataset
Result dataset
Clean Missing Data
clean_missing_data
Remove missing value rows
Cleaned datas...
Dataset
Cleaning tran...
Boosted Decision Tree Regression
boosted_decision_tree_regression
Untrained model
Split Data
split_data
Split the data into training set (0.7) and testing set (0.3)
Results datas...
Results datas...
Decision Forest Regression
decision_forest_regression
Untrained model
Untrained mod...
Dataset
Train Model
train_model
Untrained mod...
Dataset
Train Model
train_model_1
Trained model
Trained model
Dataset
Score Model
score_model
Use the test set to get the predicted price with Boosted Decision Tree Regression model
Trained model
Trained model
Dataset
Score Model
score_model_1
Use the test set to get the predicted price with Decision Forest Regression model
Scored dataset
Scored datase...
Scored datase...
Scored dataset
Evaluate Model
evaluate_model
Compare the performance of two algorithms on test data set
Evaluation results

FIGURE 2. Example of Microsoft AzureML pipeline implementing the generalized architecture

Following the initial data processing steps, the next group of modules in the pipeline pertains to the training of multiple AI models, specifically regression models from the family of decision trees. Each model is trained using the designated training set via the Train Model module, and subsequently tested using the testing set through the *Score Model module*. At the conclusion of this phase, the *Evaluate Model* module is employed to compare and assess the performance of each trained model, with the ultimate goal of identifying the optimal model. An established threshold can then be used to determine whether the best-performing model meets the requirements of the business case. If the model surpasses the predefined threshold, it may be deployed for use in a production environment. Thus, through the use of these modules, AzureML enables the efficient development, testing, and evaluation of multiple AI models, with the ultimate aim of identifying the best solution for a given task.

The flexibility and extensibility of AzureML is further exemplified through its open library of built-in components, maintained by a vibrant and supportive community. For many use cases, the built-in components satisfy the necessary requirements for developing machine learning models. However, in cases where specific and proprietary steps are required, developers may leverage the Python programming language and the *azureml* library to implement custom-built components. In the context of our collaboration with external clients, we designed and developed a series of neural networks (specifically, LSTM [10] and GRU models [6]) that were customized and optimized to suit the needs of our collaborators. These models were then deployed as custom components within the AzureML pipelines we developed. While the modules and components we implemented are tailored to our specific use case, they still follow the overarching principle of letting the models compete to determine a winner, based on their respective levels of accuracy. By providing the ability to use both pre-defined components and implement custom-built ones, AzureML affords significant gains in both performance and maintainability for AI development, even for non-specialized developers.

4.3. **Automating models training and deployment.** To achieve a robust ecosystem for model training and deployment, it is essential to have a well-structured and automated pipeline that can respond to a series of events, validate models against a predefined threshold, and initialize model deployment as a new version only when the threshold is exceeded. To ensure a smooth and efficient process, the proposed pipeline must adhere to the principles of CI/CD, which provide a set of protocols for training machine learning models in real-time and deploying them with high availability and scalability using the infrastructure provided by AzureML. By following the principles of CI/CD, the pipeline can be structured to streamline the model development process, reduce overhead costs, and maximize the return on investment for the AI development process. In general, the CI / CD principles are essential to ensure that the model training and deployment pipeline is robust and scalable, facilitating the rapid deployment of models with minimal interruption to the overall workflow.

Figure 3 depicts the proposed automated mechanism of model training using CI/CD in the Microsoft AzureML ecosystem. The CI/CD system is triggered whenever a change is made to the Azure Dataset associated with the pipeline, indicating the availability of a new dataset for training. The system then pulls the latest version of the dataset and initiates the pipeline, as described in the previous section. Notably, the components in the pipeline will be parallelized if they are on the same level in the hierarchy, allowing all model training to

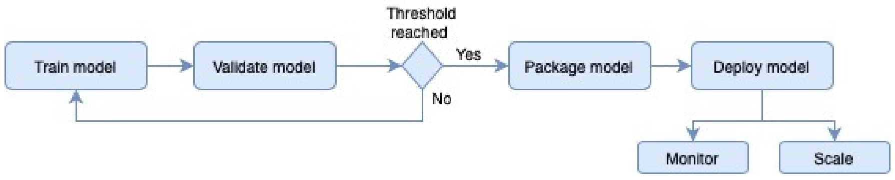


FIGURE 3. CI/CD proposed for the automatized model training pipeline.

be executed simultaneously, thus saving time and resources and, therefore, leading to cost savings.

In the post-training phase of the proposed CI/CD pipeline, the best model's accuracy will be compared to the threshold, which, if not reached, will trigger a notification to the administrators to perform additional fine-tuning on the training set or modify the components (e.g., by adding more cleaning methods). On the other hand, if the threshold is exceeded, the best model will be packaged as a binary model encapsulated into a Dockerized web microservice [17]. This microservice will be deployed on the AzureML infrastructure as a service, which will expose the model through an endpoint. This process ensures that the model is readily available to be consumed by other services.

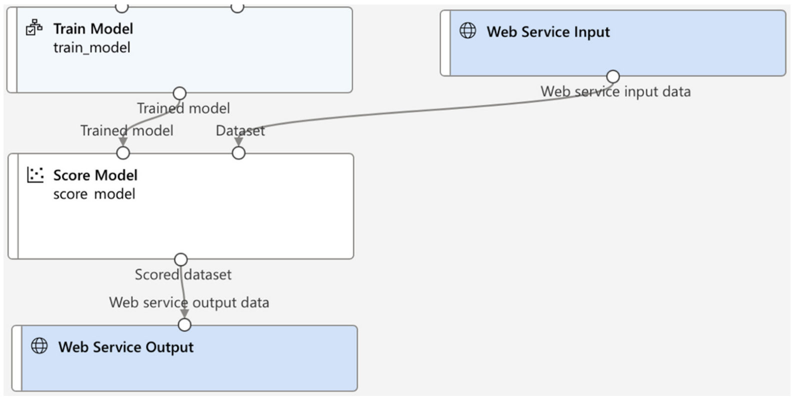


FIGURE 4. AzureML pipeline adjustment to package model as a webservice.

In the context of AzureML, the mechanism of packaging a trained model can be achieved by using two key components, namely the *Web Service Input* module and the *Web Service Output* module, as depicted in Figure 4. These components enable AzureML to recognize that the output of the Train Model section is intended to be encapsulated in a web service. The *Web Service*

Input component is responsible for receiving a set of input parameters, which are then passed on to the component responsible for handling the input and feeding it to the model for inference. The model output is then passed to the *Web Service Output* component, which in turn returns the result to the service that initiated the request.

Upon the completion of the CI/CD pipeline, the resulting web service can be located in the AzureML Endpoints section, which provides all the necessary details to interact with the model. In addition to supporting the deployment of microservices, AzureML incorporates a range of features that are particularly useful during the post-deployment phase, ensuring the high availability and performance of the model. We consider two of these built-in features to be extremely helpful: the fault tolerance - AzureML guarantees that at least one instance of the web service will be operational at all times - and the automated scaling - in situations where there is a high demand, the web service can be scaled to multiple instances managed through a load-balancer.

The underlying infrastructure used for model training, validation, and deployment can be configured to operate on either a CPU or a GPU. Typically, GPUs are essential during the training phase. While GPU virtual machines are considerably more expensive than those equipped with a CPU, weighing the time required for model training against the hourly cost may result in a cost-effective solution involving a GPU instance.

The proposed solution of using an AzureML pipeline for model training and deployment is an automated mechanism that starts with the upload of a new dataset version to the associated Azure Dataset and continues until the last step of model deployment. This approach provides the benefits of an autonomous and intuitive maintenance process, as every step of the pipeline is executed automatically based on the defined logic. The pipeline and its deployment mechanism are suitable for web applications used in live environments, fulfilling requirements such as availability, ease of maintenance and debugging, and fast execution. The monitoring system also enables quick identification and resolution of any issues. Thanks to all of these, the proposed solution offers a comprehensive and reliable solution for automating the model training and deployment process.

4.4. **Consuming trained models.** This section describes the deployment of a model in AzureML as a microservice and the standardization of web services or APIs using Swagger [2]. After a model is deployed, it can be accessed through HTTP requests, and all relevant information regarding the web service, such as the REST endpoint, authentication type, and monitoring logs, can be found in the AzureML Endpoints section. Additionally, the web service created via AzureML Pipelines includes its own documentation that provides details on existing web service endpoints, their expected input structure, and the expected output structure. Swagger is a popular standard for normalizing web services or APIs, which is implemented in AzureML to enable language-agnostic solutions. Therefore, any programming language that has an HTTP framework can make requests to the model and expect output following the structure described by its Swagger.

5. Conclusions

This paper provides an overview of the latest advancements in automating the training of AI models and presents a scalable CI/CD architecture for industrial forecasting utilizing AzureML. Our aim is to develop a user-friendly solution that is easily maintained by individuals with minimal experience in AI. To this end, we conducted a practical experiment utilizing AzureML pipelines with both pre-defined and custom modules, demonstrating its readiness for integration into any production application. While presenting a quantitative validation of our proposed solution may not be feasible, we deem the successful integration of the architecture into the products of our collaborators, along with their teams utilizing and maintaining the proposed automated pipelines for various training purposes, as a valuable qualitative validation. This practical implementation demonstrates the effectiveness and adaptability of our approach, indicating its potential to address various training needs.

Moving forward, we plan to enhance our solution by creating a separate pipeline architecture dedicated to deploying external custom modules. This will allow contractors with AI expertise to easily maintain and improve pipelines. When a new module is created, it will be automatically deployed via the dedicated CI/CD pipeline for custom modules and added to the list of pipelines for training models enabling the module for further use.

6. Acknowledgement

This research was partially supported by DataSEER project, financed through POC 2014-2020, Action 1.2.1, by European Commission and National Government of Romania (Project ID: 121004).

The authors express their gratitude to the industrial partner, OPTIMA GROUP SRL, for their collaboration and valuable information exchange.

References

[1] Gartner. `https://www.gartner.com/en`. Accessed: June 16, 2023.
[2] Swagger: The world's most popular framework for apis. `https://swagger.io`, 2022. Accessed: Feb. 7, 2022.

[3] Algorithmia. 2021 state pf enterprise ml. `https://info.algorithmia.com/hubfs/2020/Reports/2021-Trends-in-ML/Algorithmia_2021_enterprise_ML_trends.pdf`. Accessed: June 9, 2023.

[4] Ashmore, R., Calinescu, R., and Paterson, C. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *ACM Computing Surveys (CSUR) 54*, 5 (2021), 1–39.

[5] Boer, A., Koolen, M., van den Berg, J., and van der Werf, J. Continuous delivery pipelines: Best practices in safe. In *2018 IEEE 15th International Conference on e-Business Engineering (ICEBE)* (2018), IEEE, pp. 217–224.

[6] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

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

[8] Feretzakis, G., Sakagianni, A., Loupelis, E., Kalles, D., Skarmoutsou, N., Martsoukou, M., Christopoulos, C., Lada, M., Petropoulou, S., Velentza, A., et al. Machine learning for antibiotic resistance prediction: A prototype using off-the-shelf techniques and entry-level data to guide empiric antimicrobial therapy. *Healthcare informatics research 27*, 3 (2021), 214–221.

[9] Fischer, D., Jung, M., Kasparick, M., and Momm, C. Continuous integration and deployment for software of things using jenkins and docker: A case study. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)* (2019), IEEE, pp. 9–16.

[10] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[11] Huang, W. Best practices for continuous integration and delivery. In *2016 11th International Conference on Computer Science & Education (ICCSE)* (2016), IEEE, pp. 422–426.

[12] Hutter, F., Kotthoff, L., and Vanschoren, J. Automated machine learning: Methods, systems, challenges. *arXiv preprint arXiv:1908.02259* (2019).

[13] Kim, M., Zimmermann, T., DeLine, R., and Begel, A. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering 44*, 11 (2017), 1024–1038.

[14] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Automating neural architecture search using bayesian optimization and hyperband. In *International Conference on Learning Representations* (2018).

[15] Li, W., Chen, J., and Huang, W. A survey on continuous integration, delivery and deployment tools. *Journal of Systems and Software 147* (2018), 1–15.

[16] Martin, R. C. The single responsibility principle. *The principles, patterns, and practices of Agile Software Development* (2002), 149–154.

[17] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal 2014*, 239 (2014), 2.

[18] Microsoft. Azure machine learning. `https://azure.microsoft.com/en-us/services/machine-learning/`, 2021. Accessed: February 7, 2023.

[19] Microsoft. Microsoft customer stories. *Microsoft Azure Blog* (January 2022).

[20] Paleyes, A., Urma, R.-G., and Lawrence, N. D. Challenges in deploying machine learning: a survey of case studies. *ACM Computing Surveys 55*, 6 (2022), 1–29.

[21] Pliuhin, V., Pan, M., Yesina, V., and Sukhonos, M. Using azure maching learning cloud technology for electric machines optimization. In *2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T)* (2018), IEEE, pp. 55–58.

[22] Rajagopal, S., Hareesha, K. S., and Kundapur, P. P. Performance analysis of binary and multiclass models using azure machine learning. *International Journal of Electrical & Computer Engineering (2088-8708) 10*, 1 (2020).

[23] Rezazadeh, A. A generalized flow for b2b sales predictive modeling: An azure machine-learning approach. *Forecasting 2*, 3 (2020), 267–283.

[24] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. *Advances in neural information processing systems 28* (2015).

[25] Sharir, O., Peleg, B., and Shoham, Y. The cost of training nlp models: A concise overview. *arXiv preprint arXiv:2004.08900* (2020).

[26] Tan, M., and Le, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning* (2020), PMLR, pp. 6105–6114.

[27] Umamahesan, A., and Babu, D. M. I. From zero to ai hero with automated machine learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2020), pp. 3495–3495.

[28] Wagstaff, K. L., Doran, G., Davies, A., Anwar, S., Chakraborty, S., Cameron, M., Daubar, I., and Phillips, C. Enabling onboard detection of events of scientific interest for the europa clipper spacecraft. In *Proceedings of the 25th acm sigkdd international conference on knowledge discovery & data mining* (2019), pp. 2191–2201.

Faculty of Mathematics and Computer Science Babeş-Bolyai University, Cluj Napoca, România

Email address: `bogdan.mursa@ubbcluj.ro`
Email address: `matyas.kuti @ubbcluj.ro`
Email address: `cristiana.moroz@ubbcluj.ro`
Email address: `florentin.bota@ubbcluj.ro`