AI-Based Fault-Proneness Metrics for Source Code Changes

Francesco Altiero[1], Anna Corazza[1], Sergio Di Martino[1], Adriano Peron[2] and Luigi L. L. Starace[1]

[1]Università degli Studi Di Napoli Federico II, Via Claudio, 21, 80125 Napoli, Italy
[2]Università degli Studi di Trieste, Via Alfonso Valerio, 12, 34127 Trieste, Italy

Abstract

In software evolution, some types of changes to the codebase (e.g.: a local variable renaming refactoring) are less likely to introduce faults than others (e.g.: changes involving control flow statements). Effectively estimating the fault-proneness of codebase changes can provide a number of advantages in the software process. For example, expensive and time-consuming regression testing, code review, or fault localization activities could be driven by fault-proneness, prioritizing the most critical changes to detect issues more rapidly. A number of works in the literature have focused on predicting the fault-proneness of software systems. Less work, however, has focused on the fault-proneness of evolutionary changes to a codebase, and existing approaches typically require project-specific historical data to be used effectively.

This paper presents a set of AI-based metrics designed to estimate the fault-proneness of source code changes. The proposed metrics are based on *Tree Kernel* functions and *Transformer* models, that have been largely and effectively used in the Natural Language Processing domain. Moreover, the proposed metrics can be used on any software project, and do not require fine-tuning with project-specific historical data. The effectiveness of the proposed metrics is assessed by applying them to a dataset of real-world source code evolution scenarios, and by comparing them against fault-proneness scores provided by a Software Engineering practitioner.

Results are promising and show that the proposed metrics are strongly correlated with human-defined fault-proneness scores, and could thus be used as a good proxy of costly human evaluations. The results also motivate further research on the application of these metrics to concrete scenarios such as regression testing.

Keywords

Software Metrics, Software Maintenance, Fault-proneness, Tree Kernels, Transformer Models

1. Introduction

Software evolution is an integral part of the software development lifecycle, encompassing the continuous process of maintaining software systems to meet changing requirements, fix bugs, and improve design and performance [1, 2]. During the evolution process, not all code changes are equal in terms of their impact on fault-proneness. Some changes, such as, for example,

IWSM–MENSURA 2023, September 14–15, 2023, Rome, Italy
francesco.altiero@unina.it (F. Altiero); anna.corazza@unina.it (A. Corazza); sergio.dimartino@unina.it (S. Di Martino); adriano.peron@units.it (A. Peron); luigiliberolucio.starace@unina.it (L. L. L. Starace)
https://luistar.github.io/ (L. L. L. Starace)
0000-0001-7090-4249 (F. Altiero); 0000-0002-9156-5079 (A. Corazza); 0000-0002-1019-9004 (S. Di Martino); 0000-0002-7111-3171 (A. Peron); 0000-0001-7945-9014 (L. L. L. Starace)

local variable renaming refactorings, are less likely to introduce faults, whereas more complex alterations involving control flow statements are more likely to introduce faults, thereby posing a higher potential risk to the overall stability and reliability of the system [3].

Effectively estimating the fault-proneness of codebase changes can significantly improve software development practices. By identifying the changes that are most likely to introduce faults, it is indeed possible to allocate limited resources more effectively, focusing testing and inspection efforts on the parts of the software that are more critically affected by changes [4]. Similarly, fault localization efforts could be directed towards the most critical areas, resulting in earlier localization (and thus fixing) of issues [5].

While a number of studies in the literature have explored the prediction of fault-proneness in software systems [6, 7], fewer works have investigated the fault-proneness of evolutionary changes to a codebase. Existing approaches often rely on project-specific historical data to construct prediction models, limiting their applicability to specific contexts and hindering their generalizability. Moreover, the time and effort required to collect and maintain project-specific data pose practical challenges for developers and researchers seeking to adopt these approaches.

To address these limitations, this paper presents a set of AI-based metrics for estimating the fault-proneness of source code changes. Inspired by the effectiveness of Tree Kernel [8] functions and Transformer models [9] in Natural Language Processing tasks, the proposed metrics leverage their capabilities to capture the structural and semantic characteristics of code changes. This approach offers several advantages over traditional methods, including the ability to work with any software project without the need for project-specific historical data, thus providing a more practical and scalable solution.

To evaluate the effectiveness of the proposed metrics, we conduct an empirical study leveraging a dataset of real-world source code evolution scenarios. In this study, we compare the fault-proneness scores generated by the metrics against expert evaluations provided by a Software Engineering practitioner. Results are promising and show that there exists a strong correlation between the proposed AI-based metrics and human-defined fault-proneness scores, suggesting that the proposed metrics could serve as a reliable proxy for costly human evaluations. Moreover, we make the employed dataset, consisting of more than 100 real-world software evolution scenarios, manually annotated by a Software Engineering practitioner, publicly available, to foster further research on fault-proneness estimation.

The remainder of this paper is organized as follows. In Section 2, we discuss related works on software fault-proneness estimation. Then, in Section 3, we describe in detail the proposed AI-based metrics. In Section 4, we describe the empirical procedure we defined to assess the effectiveness of the proposed metrics, while, in Section 5, we present the results and discuss the implications of our findings. Lastly, in Section 6, we draw some final conclusions and provide some recommendations for future research and applications of our findings.

2. Related works

Predicting the fault-proneness of software is a prominent topic in software engineering, due to its impacts on the whole software development process, as witnessed by a large number of studies in the literature [10, 11, 12].

Several Machine Learning approaches to predict the fault-proneness of software artefacts have been proposed in the literature. Code metrics are often used as input for ML models, as they can produce better performances and also be more intuitive compared to features derived from *PCA* analysis [6]. Other studies, such as [11, 13], focused on the impact of different sets of code metrics on the training of ML models for defect prediction. These studies showed that better performances can be achieved leveraging only on code change metrics, and the accuracy can be even higher when training the models with reduced sets of change-related features (i.e., metrics on the number of code units added, removed and edited in an evolutionary incremental step).

More recently, [10] proposed *Error-Type*, a set of metrics which included also patterns recurring in three common Java runtime errors, which were extracted by the application of formal modelling methods. In the study, it was empirically shown that the analysis of source code patterns could lead to models with a higher degree of accuracy in predicting software defects.

The study in [7] investigates the relationship between classical object-oriented software metrics and different *centrality measures* applied on structures which model the underlying software, i.e., *Static Dependency Graphs*. These kinds of models underline the connection between various code units (e.g., method calls) and centrality measures to try to estimate the most critical and fault-proneness modules in the software. The empirical evaluation performed in this study showed that this combination of metrics and centrality measures can be effectively used to predict both the number and the severity of faults.

In recent years, *deep learning* approaches to fault-detection and defect prediction have been broadly investigated. Several models use deep networks to extract features from the underlying source code and then classify the code units according to their probability to be faulty. [14] employed a *deep belief network* algorithm to extract useful features from code change information, and use such features as input of a logistic regression classifier. Lately, [15] leveraged on *convolutional neural network* to extract salient features both from code changes and from *GIT* commit messages, and a *fully-connected* shallow network to evaluate the probability of defects in the software update. Other approaches [16] applied *convolutional neural networks* to a structured representation of the source code (i.e. *Abstract Syntax Tree*) to identify program functionalities and to detect code patterns which can introduce faults. More recently, *embeddings* have also been employed in software defect prediction. *Cc2Vec* [17] produces vector embeddings of software patches using both patch log messages and a hierarchical representation of source code, paying also attention to the kind of modification, i.e., weighting differently added and removed code units. In their study, the authors use these embeddings to predict defective patches in the source code evolution. *Transformer* architecture, which retains the state-of-the-art performances in Natural Language Processing, has also been applied to source code analysis. One of the most used transformer-based models is *CodeBERT* [9], which leverages both natural language data (such as comments, documentation, or other descriptions of programs) and source code to produce embeddings that were originally used for the task of generating code documentation and natural language code search.

Dynamic approaches to fault detection need to be performed after the execution of test cases. They usually leverage on *program spectra* [18, 5, 19], i.e. a characterization of the behaviour of a program, which summarizes the trace of the program in exercising different constructs, such as branches or statements [20]. The information obtained by comparing the traces in successful

and faulty scenarios can be used to predict and localize faults. These approaches are useful to help developers locate faults *a-posteriori*, i.e. after the execution of test cases. They however incur in an increased effort to be applied due to the instrumentation and the execution of test cases to record their trace.

Fault-proneness estimation has been used also in the field of Regression Testing, in particular in Regression Test Prioritization and Selection. These approaches, which are particularly important in scenarios in which test execution requires a significant amount of time and resources, such as End-to-End Testing [21] or co-simulation-based testing [22], aim at easing the testing efforts by re-arranging the test execution order or selecting a subset of test cases to execute, respectively [23, 24]. Different studies embedded the evaluation of fault-prone changes to which a software was subjected to estimate the importance of re-execution of test cases exercising these changes. In particular, [4, 25] leverage on software metrics to assign a fault-proneness score to modules or source files and use the assessed scores in the evaluation of the criticality of a test to be executed. Static source code analyzers to build statistical models that can predict source code defects have also been proposed in [26]. These models are then employed along with *code coverage* or other test case information to weigh the likelihood of a test case to expose a fault. Other information, such as the bug history obtained from previous test execution [27], has often been applied to estimate the fault-proneness of source code areas and to rank the test cases accordingly. Other approaches also exploited meta-heuristic frameworks relying on source code change metrics to prioritize test cases [28].

Static source code analysis has also been applied in several branches of software engineering, such as *code clone detection* and *vulnerability identification* [29, 30]. Techniques in the field adopt different levels of source code analysis, such as textual [31, 32] or structural comparison, typically through *Abstract Syntax Trees* or *Dependency Graph* representation of the source code [33, 34]. The study in [35] successfully applies *Tree Kernel* function to evaluate the structural similarity of ASTs related to pairs of Java methods in order to detect similar patterns and to classify how much they are likely to be code clones.

To the best of our knowledge, very few studies focused on the estimation of fault-proneness of code changes with respect to human-evaluated scores. A human analysis of code changes during an evolutionary step of software maintenance is costly both in time and in resources, as it includes several complex tasks such as code comprehension. With the spread of different techniques to source code analysis and defect prediction, in this study, we aim to assess the correlation between different features and metrics employed in the literature and human-generated feedback. We took into account several kinds of features: those related to code change metrics, which have been proven to be the best metrics for fault-proneness evaluation; source code features, both leveraging on textual representation and on more-refined tree-based representations (i.e., *Tree Kernels*); and features produced by embedding vectors, in particular those obtained through the application of *CodeBERT*, which obtained the state-of-the-art performances in software comprehension related tasks.

3. Proposed Metrics

In this section, we describe the novel AI-based fault-proneness metrics we defined.

```
public int min(int a, int b) {
  if(a < b)
    return a;
  return b;
}
```

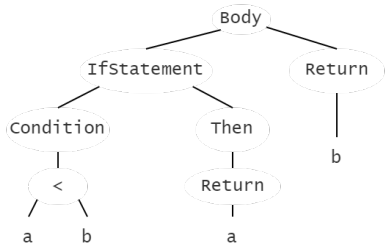

Figure 1: An *Abstract Syntax Tree* representation for a Java method which evaluates the minimum between two integers.

3.1. Tree Kernels

Tree Kernels are a family of functions which evaluates the similarity between tree-based structures. They have been extensively used in the Natural Language Processing domain and they have been successfully applied in several software engineering tasks, such as *code clone detection* and *security testing* [35, 36]. Tree Kernels are typically used in software engineering to assess the structural similarity between the *Abstract Syntax Tree* (*AST*) representation of the source code. This representation highlights the hierarchy of the source code through a tree model. Each tree node is labeled with the semantic of the programming language construct (e.g., *statement*, *expression*, *for* loops, ...), while relationships between nodes give information on the structure of the code (e.g., an expression which is part of a statement, or a statement belonging to a for loop). The leaves of the tree are related to the tokens in the source code, such as names of variables or literal values. An example of an AST is shown in Figure 1.

Generally, Tree Kernels evaluate the similarity between two trees by counting the number of *fragments* the trees have in common. The similarity is produced as a real, non-negative number in the $[0, +\infty[$ range. The type of considered fragments defines the particular Tree Kernel function. Among all Tree Kernels, *Sub-tree Kernel*, *Subset-Tree Kernel* and *Partial-Tree Kernel* are the most used in literature [8]. Given two nodes of two different trees, the *Sub-tree Kernel* (*STK*) considers as fragments the whole sub-trees rooted in these nodes, up to the leaves. If these sub-trees are equivalent (i.e., have the same label and the same structure), the fragments contribute to the overall similarity. *Subset-Tree Kernel* (*SSTK*) analyzes the subsets of common nodes, i.e., the sub-trees at any possible depth from the root nodes. While the former TKs consider the whole sequence of children when evaluating two nodes, the *Partial-Tree Kernel* relaxes this constraint by taking into account also partial sequences of child nodes in both trees.

To shift from the similarity score evaluated by TKs to a measure of *dissimilarity*, we used two methods: the first employs the *normalized* Tree Kernel similarity [37], while the second leverages *Kernel Distance* [38].

A *Normalized Tree Kernel* is evaluated by dividing the TK similarity of the two trees by the square root of the product between the TK similarity of each tree with itself. This ensures a similarity score in the range $[0, 1]$, where 0 means that the trees are completely different, while a score of 1 is assessed when the two trees are identical. To evaluate the *dissimilarity*, we just evaluate the complement to 1 of the similarity measure. More formally, given a Tree Kernel

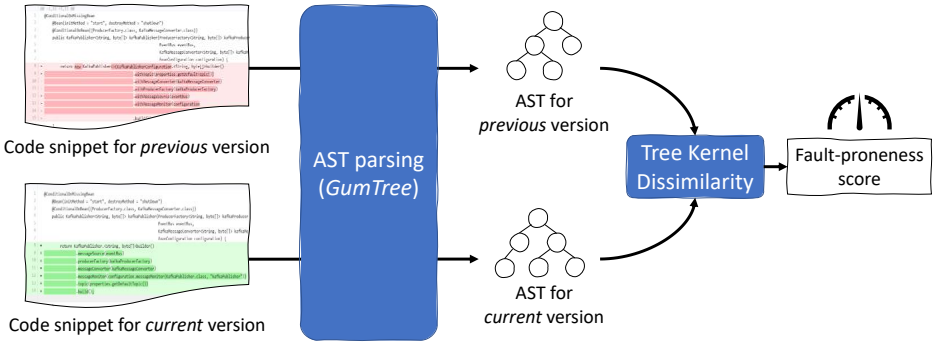

Figure 2: Computation pipeline for the *Tree Kernel* metrics. The dissimilarity evaluation process is the same for each type of Tree Kernel, and with both *normalized* dissimilarity and *Kernel distance*.

function K and two ASTs T_1, T_2, the normalized Tree Kernel *dissimilarity* is computed as:

$$d_{norm}(T_1, T_2) = 1 - \frac{K(T_1, T_2)}{\sqrt{K(T_1, T_1) \cdot K(T_2, T_2)}} \tag{1}$$

Kernel Distance is a distance measure evaluated leveraging the similarity score produced by the kernel and provides a dissimilarity score in the range $[0, +\infty[$. It can be evaluated as:

$$d_{dist}(T_1, T_2) = K(T_1, T_1) + K(T_2, T_2) - 2 \cdot K(T_1, T_2) \tag{2}$$

where T_1 and T_2 are two trees and K one of the Tree Kernel function discussed above.

We employed these two measures of dissimilarity to also investigate the effects of method sizes in our evaluation. In fact, Equation (1) can be seen as the *relative* dissimilarity between the two trees, measuring the relative amount of structural changes two methods have. On the other hand, the Kernel distance measures the *absolute* amount of structural changes. To this end, consider two methods, one with 100 statements and the other with 10 statements. Suppose that 50 statements are changed in the first method due to an evolutionary step and that 5 statements are changed in the second one. The evaluation of the normalized Tree Kernel score between the two versions of both methods could lead to close dissimilarity values. The kernel distance, instead, will assign a higher dissimilarity score to the larger method, as the structural changes it underwent are more prominent.

To evaluate a dissimilarity score between two methods or functions, we produce each method's AST by processing its source code through *GumTreeDiff* [39], a library widely used in the literature to produce Abstract Syntax Tree models. We then evaluate the similarity score using a specific Tree Kernel function (STK, SSTK, PTK) and calculate the dissimilarity using one of Equations (1) or (2). For the implementation of Tree Kernel functions we relied on *KeLP*[1], a consolidated Machine Learning framework which has been used in several studies in the field of NLP [40, 41]. In the remainder of the paper, *STK*, *SSTK* and *PTK* refer to a particular Tree Kernel function which uses the distance as dissimilarity metric, while the suffix *Normalized*

[1]http://www.kelp-ml.org/, accessed on 19/05/2023.

relates to the specific Tree Kernel with the normalized dissimilarity function. Figure 2 presents the steps to evaluate dissimilarity using tree kernels.

3.2. Transformer-based metrics

Transformer models are a class of deep learning models, introduced by Vaswani et al. in [42], that have revolutionized the field of natural language processing (NLP), rapidly becoming state-of-the-art models for various NLP tasks such as machine translation or sentiment analysis. The architectural innovation of self-attention mechanisms allows this kind of models to capture long-range dependencies between tokens in the input sequence, effectively learning vector representations that capture the semantic and syntactic structure of the input. Transformer models often employ pre-training on large-scale unlabelled corpora, using unsupervised learning objectives such as masked language modelling or next-sentence prediction, and can then be fine-tuned for specific tasks.

In this paper, to measure the fault-proneness of a code change, we leverage CodeBERT [9], a specialized language model, based on the Transformer architecture, designed to process and analyze source code as well as natural language. CodeBERT has been proven to be effective in a number of Software Engineering tasks, including the generation of commit messages for given code changes [43], code clone detection [44], or the automatic generation of fixes for bugs [45]. In particular, we leveraged a pre-trained instance of CodeBERT, which was trained on CodeSearchNet data [46] using an unsupervised Masked Language Modelling task. We use the pre-trained CodeBERT model to extract a vector representation (*embedding*) for both the previous version of a method and the current one. CodeBERT can indeed be used to map snippets of code to 768-sized vectors. Then, we estimate the fault-proneness of a method evolution by measuring the *cosine distance* between the two embeddings, under the assumption that changes that are more fault-prone have a greater impact on the syntactic and semantic structure of code, which are captured by CodeBERT in the embedding representation, thus resulting in embeddings that are farther away in the vector space. The cosine distance between the two CodeBERT embeddings is a value in the range [0,1], with 1 representing the highest possible diversity between the two embeddings, and thus the most fault-prone change. In the remainder of this paper, we will refer to the cosine distance between two subsequent versions of a source code snippet as *CodeBERT-distance*. An overview of the computation process of the *CodeBERT-distance* is depicted in Figure 3.

4. Empirical Evaluation

4.1. Goals

To assess the effectiveness of the proposed metrics in capturing the fault-proneness of a method evolution as perceived by a software engineering practitioner, we consider the following research question.

RQ1: *To what extent do the considered metrics correlate with fault-proneness scores that are manually defined by a Software Engineering practitioner?*

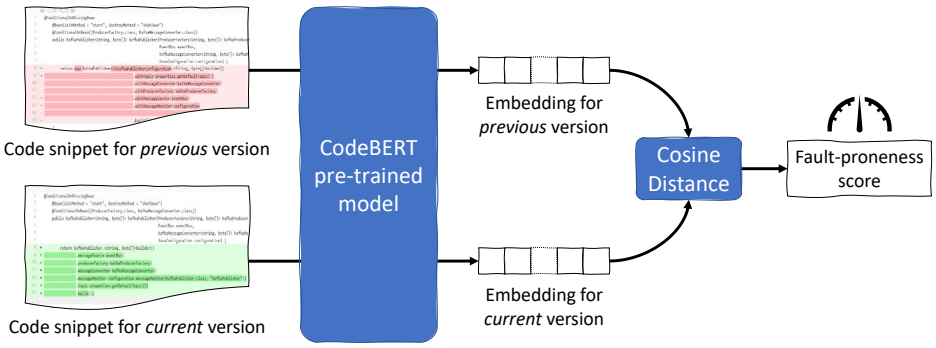

Figure 3: Overview of the computation of the *CodeBERT-distance* metric

Moreover, we also investigate the reliability of the fault-proneness perception of a software engineering practitioner by considering the following additional research question.

RQ2: *How subjective are manually defined fault-proneness ratings produced by a Software Engineering practitioner?*

4.2. Employed Data

To assess the effectiveness of the proposed metrics for estimating the fault-proneness of source code changes, we considered a set of real-world software evolution steps collected from open-source projects. More in detail, we based our analysis on software projects included in a recently-presented dataset [47], consisting of 114 pairs of subsequent software versions from open-source Java projects. By analysing these 114 version pairs, we extracted more than 1k real-world method-level evolution scenarios from 19 different projects. Each of these scenarios consists in two subsequent versions of the same method, in which the more recent version has been impacted by evolutionary changes. Since our study aims at comparing the proposed fault-proneness metrics with a manually-defined score produced by a Software Engineering practitioner, and manually annotating each method evolution is a time-consuming process, we randomly sampled a subset of 108 method evolution scenarios. More in detail, we performed a stratified sampling based on the belonging to a specific project, and randomly selected at most 7 method evolution scenarios from each project. For projects for which we collected less than 7 method evolution scenarios, all the available scenarios were selected. In Table 1, we report additional statistics on the dataset of method-level evolution scenarios we built.

In particular, for each of the 19 projects, we report the number of considered method evolution scenarios, the average, minimum and maximum number of Lines of Code (LOCs) of the considered methods (considering both versions), and the average, minimum and maximum number of *changed* LOCs in each method evolution scenario. The number of changed LOCs is defined as the sum of removed and added LOCs between the two versions of the method. These figures highlight that the sampled evolutionary scenarios cover a broad set of situations, including methods that vary in size, as well as in the extent of the source code evolution.

Table 1
Overview of the considered dataset

Project	# Method Evolutions	# LOCs per method			# Changed LOCs per method		
		Avg.	Min.	Max.	Avg.	Min.	Max.
Mapper	7	32	10	81	5	2	12
LittleProxy	7	29	4	70	18	1	69
shiro-redis	3	21	5	34	12	2	33
fastjson	7	153	3	542	37	7	101
sofa-rpc	5	34	13	97	11	2	41
incubator-dubbo	7	30	4	58	11	1	32
rocketmq	7	84	7	383	11	2	57
AxonFramework	7	21	4	57	11	2	33
demoiselle	7	14	4	31	7	2	21
dynjs	7	25	4	44	6	1	16
elastic-job-lite	7	10	5	17	2	2	4
hsweb-framework	5	16	4	46	6	1	13
JsonUnit	1	41	38	43	5	5	5
pf4j	5	30	5	54	4	3	6
pippo-java	1	61	59	63	8	8	8
rapidoid	7	10	3	38	6	2	27
consul-client	4	21	17	28	3	2	4
sismics-reader	7	79	40	126	10	4	24
titan	7	40	5	95	14	7	24
Aggregate	**108**	**40**	**3**	**542**	**11**	**1**	**101**

4.3. Baseline

As a baseline against which to compare the proposed AI-based techniques, we considered the percentage of changed LOCs in a method evolution step, a well-known metric used to estimate the fault-proneness of changes in regression testing scenarios [48]. More formally, given two subsequent versions V_a and V_b of a method, the percentage of changed LOCs in the method evolution is defined as

$$\frac{\#\ added\ LOCs + \#\ removed\ LOCs}{\#\ LOCs\ V_a + \#\ LOCs\ V_b},$$

where # *added LOCs* (resp., # *removed LOCs*) is the number of lines of code that are *added* (resp., *removed*) in the evolution from V_a to V_b, and # *LOCs* V_a (resp., # *LOCs* V_b) is the number of lines of code in V_a (resp., V_b).

4.4. Procedure

After preparing the dataset, each of the considered metrics was computed for each method evolution scenario. The LOC-based baseline metrics were computed leveraging the well-known *git diff* tool. The Tree Kernel-based metrics were computed by leveraging the *GumTreeDiff* tool to extract an AST representation of each method version, and by leveraging the implementation

of the considered Tree Kernels available in the *KeLP* framework [49], which has been used in several previous studies in Natural Language Processing and Software Engineering [50]. As for the *CodeBERT-distance* metric, we implemented it in Python using the *transformers* library and a pre-trained version of the CodeBERT model, which was made publicly available by Microsoft[2]. Since some of the methods in the considered dataset contain more tokens than the maximum input length supported by the pre-trained CodeBERT model (which amounts to 512 tokens), we had to deal with those cases specifically. In particular, when the input sequence was longer than the maximum processable length, we split it every 500 tokens and computed an embedding for each slice of the original sequence. Then, we concatenate all the embeddings and zero-pad them when necessary to compute the cosine distance.

The dataset was also manually annotated by a Software Engineering practitioner, to whom we asked to assign, according to their own sensibility and experience, a fault-proneness score in the range 1-10 to each of the method evolutions, where 1 is the least and 10 is the most fault-prone. Note that the recruited practitioner is not among the authors of this paper. To this end, the practitioner was provided with a human-readable diff, similar to those shown for commits on the GitHub platform, for each of the method evolution scenarios. These human-readable diffs were obtained using the *diff2html* tool. When carrying out this task, the practitioner was given no other metric to consider.

Independently, one of the authors of this paper also annotated the dataset and assigned his own fault-proneness scores using the same scale as the practitioner. This annotation was performed *before* any of the considered metrics was computed, and before the practitioner's scores were available to the authors.

5. Results and Discussion

5.1. RQ1: Correlation between metrics and fault-proneness scores

The first research question aims at investigating the effectiveness of the considered metrics in approximating the fault-proneness perception of a Software Engineering practitioner. To this end, we first carried out an analysis of the collected data and computed metrics. Figure 4 features histograms showing the value distributions for each of the considered metrics and for the manually defined fault-proneness scores, using 10 equally-sized bins.

The distribution analysis highlights that, in the considered dataset, the fault-proneness of the method evolutions is not uniformly distributed, but is skewed towards minor, less fault-prone changes. This is witnessed by the distribution of the absolute and relative number of changed LOCs, as well as by the manually-defined annotations. This is also in line with the evolutionary patterns observed in real-world software projects, as reported by a number of empirical studies such as [51, 52]. To verify these visual observations on the non-normality of the distributions, we also performed Shapiro-Wilk tests [53], which confirmed that all the metrics and the fault-proneness scores featured non-normal distributions with very high confidence ($p\text{-}value \ll 0.05$).

[2]https://huggingface.co/microsoft/codebert-base

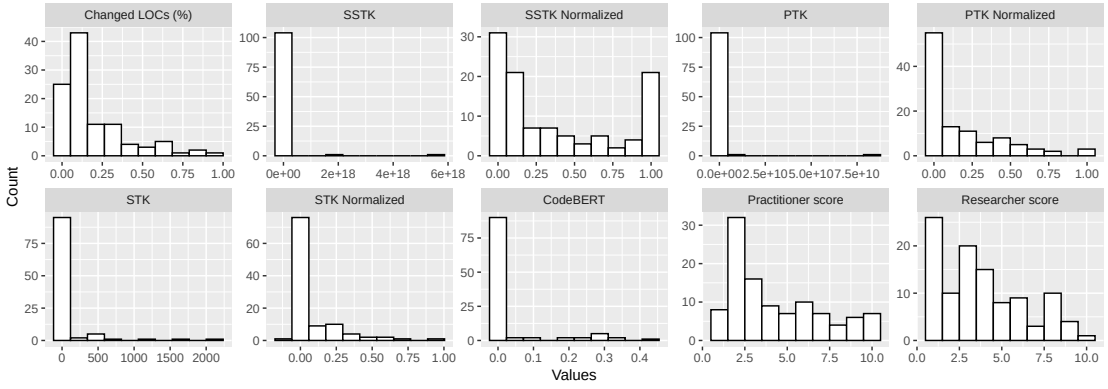

Figure 4: Histograms representing the distributions of the computed metrics and manually defined fault-proneness scores

Table 2
Spearman's Rank Correlations between each of the considered metrics and the Practitioner's fault-proneness scores

Metric	Spearman's correlation coefficient	*p-value*	Grading
STK	0.61	6.24E-05	Strong
CodeBERT-distance	0.52	1.69E-08	Strong
PTK Normalized	0.49	1.90E-08	Moderate
Changed LOCs (%)	0.43	4.05E-06	Moderate
PTK	0.40	5.18E-06	Moderate
SSTK Normalized	0.39	9.46E-06	Moderate
STK Normalized	0.37	7.70E-13	Moderate
SSTK	0.31	4.43E-04	Moderate

Since the data distribution is not normal, we computed correlation coefficients using Spearman's rank correlation coefficient [54]. In this correlation analysis, we considered the correlation between any pair of the considered metrics, as well as their correlations with the practitioner's scores. The computed Spearman's rank correlation scores are reported in the correlogram in Figure 5. Moreover, detailed correlation coefficients w.r.t. the practitioner score, as well as the *p-values* for the Spearman's rank correlations and grading for the correlations according to [55], are reported in Table 2. These results show that the *STK* and *CodeBERT-distance* metrics are the ones that correlate the most with the Practioner's fault-proneness scores, with correlation coefficients of 0.61 and 0.52, respectively, indicating a *strong* correlation, according to the interpretation table presented in [55]. All the other metrics are also positively correlated with the Practitioner's scores, even though to a minor extent, with correlation coefficients ranging from 0.49 to 0.31, indicating correlations of *moderate* entity [55]. Moreover, in all cases, *p-values* are very small ($\ll 0.05$, see Table 2), indicating that the results of the correlation analysis have a high statistical significance.

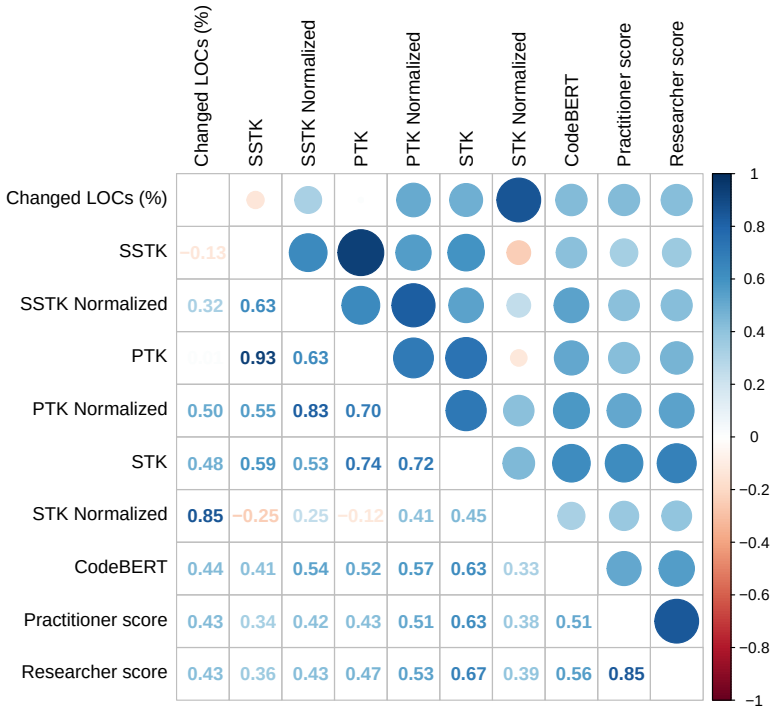

Figure 5: Correlogram representing the correlation between each pair of metrics and manually defined fault-proneness scores.

5.2. RQ2: Reliability of manually defined fault-proneness scores

The second research question aims at investigating the reliability of the manual fault-proneness ratings and, in a way, the degree to which different people subjectively perceive fault-proneness. As a first result in investigating this question, we observed that some subjective differences exist in the perception of fault-proneness. These differences can be observed in Figure 4, by comparing the rating distributions for both the practitioner's and the researcher's scores. Indeed, practitioner's scores exhibit a prevalence of "2" ratings, while researcher's scores show a prevalence of "1" and "3" ratings. In Table 3, we present a more detailed analysis of the entity and distribution of disagreements. The table shows that perfect agreement happens only in 22% of the cases, but, in 94% of the cases, the differences in fault-proneness ratings do not exceed 2 points on the rating scale. In no case, the disagreements exceeded 5 points on the rating scale. To better quantify the Inter-rater agreement level between the annotation produced by the Software Engineering practitioner and those produced by one of the authors, we used the Weighted Cohen's Kappa statistic with Quadratic Weights [56]. This approach takes into account both the level of agreement between raters and the potential for agreement due to chance. By assigning higher weights to disagreements farther apart on the rating scale, quadratic weights capture the importance of larger discrepancies, providing a more nuanced evaluation of agreement. The computed Weighted Kappa statistic is 0.84, indicating a *near-perfect* agreement level [57]. This

Table 3
Extents of disagreements between Practitioner and Researcher scores

Entity of disagreement	Percentage of occurrence	Cumulative Percent. of occurrence
0 (perfect agreement)	22	22
1	56	79
2	16	94
3	2	96
4	1	97
5	3	100
greater than 5	0	100

suggests that the human-assigned fault-proneness ratings are generally reliable.

6. Conclusions

Within software evolution, effectively estimating the fault-proneness of codebase changes can greatly enhance software development practices. Indeed, the availability of reliable fault-proneness estimations allows developers and organizations to optimize resource allocation, concentrating testing and inspection efforts on the parts of the software that are most significantly impacted by these alterations, and can also streamline debugging and fault localization activities.

Several studies in the literature have focused on predicting the fault proneness of entire software systems and/or components [6]. Fewer works, however, have studied the fault-proneness of changes made to a codebase over time. Most current approaches typically rely on historical data specific to a particular project in order to create prediction models, which restricts their use to specific situations and limits their overall applicability. Furthermore, the collection and upkeep of project-specific data present practical difficulties for developers and researchers aiming to adopt these methods.

To address these issues, in this paper, we propose a set of novel metrics designed to quantify the fault-proneness of codebase changes. The proposed metrics can be applied to any software evolution scenario, do not require training on historical data, and are based on Tree Kernels and Transformer models, which proved to be effective in Natural Language Processing and in Software Engineering tasks.

To assess the effectiveness of the proposed metrics, we conducted an empirical evaluation involving a dataset of 108 real-world source code evolution scenarios, each manually annotated by a Software Engineering practitioner who assigned it a fault-proneness rating according to its own sensibility and experience. Results show that some of the proposed metrics are strongly correlated with the human-defined fault-proneness scores, and suggest that these approaches could be used as effective automated proxies of costly human assessments.

In future works, we plan to further refine the proposed techniques, by investigating the feasibility of implementing novel Tree Kernel functions, specifically tailored towards the fault-proneness estimation task. Similarly, we plan to investigate the possibility of carrying out

fine-tuning on the transformer models, possibly leveraging the dataset we constructed within the present study, to further improve performances. Future research should also investigate the extent to which the proposed metrics correlate also with the actual introduction of faults, leveraging publicly available datasets of fault-inducing commits. Lastly, the promising results obtained in this study motivate future investigations on the effectiveness of the proposed metrics (or combinations thereof) in software engineering tasks such as regression testing optimization or fault localization.

Data Availability Statement

The considered dataset, including the computed metrics and manually defined fault-proneness scores, is publicly available in the replication package at https://doi.org/10.5281/zenodo.7998416. The replication package also includes all the code necessary to compute the proposed metrics, and data analytics scripts we used to analyze the raw data and produce plots and results discussed in this paper.

Acknowledgments

We acknowledge financial support from the PNRR MUR project PE0000013-FAIR.

References

[1] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, W.-G. Tan, Types of software evolution and software maintenance, Journal of software maintenance and evolution: Research and Practice 13 (2001) 3–30.

[2] E. Battista, S. Di Martino, S. Di Meglio, F. Scippacercola, L. L. L. Starace, E2E-Loader: A Framework to Support Performance Testing of Web Applications, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2023, pp. 351–361.

[3] F. Altiero, A. Corazza, S. Di Martino, A. Peron, L. L. L. Starace, Inspecting code churns to prioritize test cases, in: Testing Software and Systems: 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9–11, 2020, Proceedings 32, Springer, 2020, pp. 272–285.

[4] S. Hafez, M. Elnainay, M. Abougabal, S. Elshehaby, Potential-fault cache-based regression test selection, Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 0 (2016). doi:10.1109/AICCSA.2016.7945658.

[5] X. Xie, B. Xu, Spectrum-based fault localization for multiple faults, Essential Spectrum-based Fault Localization (2021) 83–91. URL: https://link.springer.com/chapter/10.1007/978-981-33-6179-9_8. doi:10.1007/978-981-33-6179-9_8.

[6] I. Gondra, Applying machine learning to software fault-proneness prediction, Journal of Systems and Software 81 (2008) 186–195.

[7] A. Ouellet, M. Badri, Combining object-oriented metrics and centrality measures to predict faults in object-oriented software: An empirical validation, Journal of Software: Evolution and Process (2023) e2548.

[8] A. Moschitti, Making tree kernels practical for natural language learning, in: Conference of the European Chapter of the Association for Computational Linguistics, 2006.
[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.
[10] K. Phung, E. Ogunshile, M. Aydin, Error-type—a novel set of software metrics for software fault prediction, IEEE Access 11 (2023) 30562–30574.
[11] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, Proceedings - International Conference on Software Engineering (2008) 181–190. doi:10.1145/1368088.1368114.
[12] N. E. Benton, M. Neil, A critique of software defect prediction models, IEEE Transactions on Software Engineering 25 (1999) 675–689. doi:10.1109/32.815326.
[13] Y. A. Alshehri, K. Goseva-Popstojanova, D. G. Dzielski, T. Devine, Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them, Conference Proceedings - IEEE SOUTHEASTCON 2018-April (2018). doi:10.1109/SECON.2018.8478911.
[14] X. Yang, D. Lo, X. Xia, Y. Zhang, J. Sun, Deep learning for just-in-time defect prediction, Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015 (2015) 17–26. doi:10.1109/QRS.2015.14.
[15] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, N. Ubayashi, Deepjit: An end-to-end deep learning framework for just-in-time defect prediction, IEEE International Working Conference on Mining Software Repositories 2019-May (2019) 34–45. doi:10.1109/MSR.2019.00016.
[16] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for programming language processing, Proceedings of the AAAI Conference on Artificial Intelligence 30 (2016) 1287–1293. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10139. doi:10.1609/AAAI.V30I1.10139.
[17] T. Hoang, H. J. Kang, D. Lo, J. Lawall, Cc2vec: Distributed representations of code changes, Proceedings - International Conference on Software Engineering (2020) 518–529. URL: https://dl.acm.org/doi/10.1145/3377811.3380361. doi:10.1145/3377811.3380361.
[18] R. Abreu, P. Zoeteweij, A. J. C. v. Gemund, Spectrum-based multiple fault localization, ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering (2009) 88–99. doi:10.1109/ASE.2009.25.
[19] M. A. Kabir, M. M. Islam, S. H. Mahmud, M. F. Elahe, Spectrum impact analysis of fault proneness statement for improved fault localization, in: Proceedings of the 2nd International Conference on Computing Advancements, 2022, pp. 59–66.
[20] M. J. Harrold, G. Rothermel, R. Wu, L. Yi, An empirical investigation of program spectra, ACM SIGPLAN Notices 33 (1998) 83–90. URL: https://dl.acm.org/doi/10.1145/277633.277647. doi:10.1145/277633.277647.
[21] S. Di Martino, A. R. Fasolino, L. L. L. Starace, P. Tramontana, Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing, Software Testing, Verification and Reliability 31 (2021) e1754.
[22] F. Basciani, V. Cortellessa, S. DiMartino, D. Di Nucci, D. DiPompeo, C. Gravino, L. L. L. Starace, Adas verification in co-simulation: Towards a meta-model for defining test scenarios, in: 2023 IEEE International Conference on Software Testing, Verification and

Validation Workshops (ICSTW), IEEE, 2023, pp. 28–35.

[23] S. Elbaum, G. Rothermel, J. Penix, Techniques for improving regression testing in continuous integration development environments, Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering 16-21-November-2014 (2014) 235–245. URL: https://dl.acm.org/doi/10.1145/2635868.2635910. doi:10.1145/2635868.2635910.

[24] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification & Reliability 22 (2012) 67–120. URL: https://dl.acm.org/doi/10.1002/stv.430. doi:10.1002/STV.430.

[25] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, P. McMinn, An empirical study on the use of defect prediction for test case prioritization, Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019 (2019) 346–357. doi:10.1109/ICST.2019.00041.

[26] S. Wang, J. Nam, L. Tan, Qtep: Quality-aware test case prioritization, Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering Part F130154 (2017) 523–534. URL: https://dl.acm.org/doi/10.1145/3106237.3106258. doi:10.1145/3106237.3106258.

[27] M. Mahdieh, S.-H. Mirian-Hosseinabadi, M. Mahdieh, Test case prioritization using test case diversification and fault-proneness estimations, Automated Software Engineering 29 (2022) 50.

[28] F. Altiero, G. Colella, A. Corazza, S. D. Martino, A. Peron, L. L. Starace, Change-aware regression test prioritization using genetic algorithms, Proceedings - 48th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2022 (2022) 125–132. doi:10.1109/SEAA56994.2022.00028.

[29] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on Software Engineering 33 (2007) 577–591. doi:10.1109/TSE.2007.70725.

[30] A. Kaur, R. Nayyar, A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code, Procedia Computer Science 171 (2020) 2023–2029. doi:10.1016/J.PROCS.2020.04.217.

[31] S. Ducasse, M. Rieger, S. Demeyer, Language independent approach for detecting duplicated code, Conference on Software Maintenance (1999) 109–118. doi:10.1109/ICSM.1999.792593.

[32] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (2002) 654–670. doi:10.1109/TSE.2002.1019480.

[33] J. Krinke, Identifying similar code with program dependence graphs, Reverse Engineering - Working Conference Proceedings (2001) 301–309. doi:10.1109/WCRE.2001.957835.

[34] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, Conference on Software Maintenance (1998) 368–377. doi:10.1109/ICSM.1998.738528.

[35] A. Corazza, S. Di Martino, V. Maggio, G. Scanniello, A tree kernel based approach for clone detection, in: 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–5. doi:10.1109/ICSM.2010.5609715.

[36] A. Avancini, M. Ceccato, Security oracle based on tree kernel methods, in: A. Moschitti,

B. Plank (Eds.), Trustworthy Eternal Systems via Evolving Software, Data and Knowledge, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 30–43.

[37] D. Haussler, Convolution Kernels on Discrete Structures, Technical Report UCS-CRL-99-10, University of California at Santa Cruz, Santa Cruz, CA, USA, 1999. URL: http://citeseer.ist.psu.edu/haussler99convolution.html.

[38] B. Schölkopf, The kernel trick for distances, in: T. Leen, T. Dietterich, V. Tresp (Eds.), Advances in Neural Information Processing Systems, volume 13, MIT Press, 2000.

[39] J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, 2014, pp. 313–324. URL: http://doi.acm.org/10.1145/2642937.2642982. doi:10.1145/2642937.2642982.

[40] S. Filice, G. Castellucci, D. Croce, R. Basili, Kelp: a kernel-based learning platform for natural language processing, in: Proceedings of ACL-IJCNLP 2015 System Demonstrations, Association for Computational Linguistics and The Asian Federation of Natural Language Processing, Beijing, China, 2015, pp. 19–24. URL: http://www.aclweb.org/anthology/P15-4004.

[41] S. Filice, G. Castellucci, G. D. S. Martino, A. Moschitti, D. Croce, R. Basili, Kelp: a kernel-based learning platform, Journal of Machine Learning Research 18 (2018) 1–5. URL: http://jmlr.org/papers/v18/16-087.html.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[43] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, et al., Codereviewer: Pre-training for automating code review activities, arXiv preprint arXiv:2203.09095 (2022).

[44] S. M. Rabbani, N. A. Gulzar, S. Arshad, S. Abid, S. Shamail, A comparative analysis of clone detection techniques on semanticclonebench, in: 2022 IEEE 16th International Workshop on Software Clones (IWSC), IEEE, 2022, pp. 16–22.

[45] E. Mashhadi, H. Hemmati, Applying codebert for automated program repair of java simple bugs, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 505–509.

[46] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search, arXiv preprint arXiv:1909.09436 (2019).

[47] F. Altiero, A. Corazza, S. Di Martino, A. Peron, L. L. Starace, Recover: A curated dataset for regression testing research, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 196–200.

[48] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, S. Kanduri, Understanding the effects of changes on the cost-effectiveness of regression testing techniques, Software testing, verification and reliability 13 (2003) 65–83.

[49] S. Filice, G. Castellucci, D. Croce, R. Basili, Kelp: a kernel-based learning platform for natural language processing, in: Proceedings of ACL-IJCNLP 2015 System Demonstrations, 2015, pp. 19–24.

[50] A. Corazza, S. Di Martino, A. Peron, L. L. L. Starace, Web application testing: Using tree kernels to detect near-duplicate states in automated model inference, in: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and

Measurement (ESEM), 2021, pp. 1–6.
[51] C. Kolassa, D. Riehle, M. A. Salim, A model of the commit size distribution of open source, in: SOFSEM 2013: Theory and Practice of Computer Science: 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindleruv Mlỳn, Czech Republic, January 26-31, 2013. Proceedings 39, Springer, 2013, pp. 52–66.
[52] L. P. Hattori, M. Lanza, On the nature of commits, in: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops, IEEE, 2008, pp. 63–71.
[53] B. W. Yap, C. H. Sim, Comparisons of various types of normality tests, Journal of Statistical Computation and Simulation 81 (2011) 2141–2155.
[54] J. H. Zar, Significance testing of the spearman rank correlation coefficient, Journal of the American Statistical Association 67 (1972) 578–580.
[55] G. W. Corder, D. I. Foreman, Nonparametric statistics for non-statisticians, 2011.
[56] J. L. Fleiss, B. Levin, M. C. Paik, Statistical methods for rates and proportions, john wiley & sons, 2013.
[57] M. L. McHugh, Interrater reliability: the kappa statistic, Biochemia medica 22 (2012) 276–282.