Predicting the Change Impact of Resolving Defects by Leveraging the Topics of Issue Reports in Open Source Software Systems

MARAM ASSI, Queen's University, Canada
SAFWAT HASSAN, University of Toronto, Canada
STEFANOS GEORGIOU, Queen's University, Canada
YING ZOU, Queen's University, Canada

Upon receiving a new issue report, practitioners start by investigating the defect type, the potential fixing effort needed to resolve the defect and the change impact. Moreover, issue reports contain valuable information, such as, the title, description and severity, and researchers leverage the topics of issue reports as a collective metric portraying similar characteristics of a defect. Nonetheless, none of the existing studies leverage the defect topic, *i.e.*, a semantic cluster of defects of the same nature, such as *Performance, GUI and Database*, to estimate the change impact that represents the amount of change needed in terms of code churn and the number of files changed. To this end, in this paper, we conduct an empirical study on 298,548 issue reports belonging to three large-scale open-source systems, *i.e.*, Mozilla, Apache and Eclipse, to estimate the change impact in terms of code churn or the number of files changed while leveraging the topics of issue reports. First, we adopt the Embedded Topic Model (ETM), a state-of-the-art topic modelling algorithm, to identify the topics. Second, we investigate the feasibility of predicting the change impact using the identified topics and other information extracted from the issue reports by building eight prediction models that classify issue reports requiring small or large change impact along two dimensions, *i.e.*, the code churn size and the number of files changed. Our results suggest that XGBoost is the best-performing algorithm for predicting the change impact, with an AUC of 0.84, 0.76, and 0.73 for the code churn and 0.82, 0.71 and 0.73 for the number of files changed metric for Mozilla, Apache, and Eclipse, respectively. Our results also demonstrate that the topics of issue reports improve the recall of the prediction model by up to 45%.

CCS Concepts: • **Open source software system → Change impact prediction**.

Additional Key Words and Phrases: issue reports, topics of issue reports, defect fixing, fixing effort, change impact analysis, amount of change, code churn

ACM Reference Format:
Maram Assi, Safwat Hassan, Stefanos Georgiou, and Ying Zou. 2023. Predicting the Change Impact of Resolving Defects by Leveraging the Topics of Issue Reports in Open Source Software Systems. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (April 2023), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software systems have become an integral part of our daily activities. However, they are prone to defects. In the realm of software engineering, defects consume a considerable amount of the project budget; it is estimated that 113 billion

Authors' addresses: Maram Assi, maram.assi@queensu.ca, Queen's University, Ontario, Canada; Safwat Hassan, safwat.hassan@utoronto.ca, University of Toronto, Ontario, Canada; Stefanos Georgiou, stefanos.georgiou@queensu.ca, Queen's University, Ontario, Canada; Ying Zou, ying.zou@queensu.ca, Queen's University, Ontario, Canada.

Manuscript submitted to ACM

dollars are budgeted yearly in the US to identify and fix software defects [2] and 312 billion dollars are budgeted yearly globally fixing defects [1]. To ensure an efficient defect reporting process, issue reports are usually employed to describe software failures. Upon receiving a new issue report, developers start by investigating the nature of the software failure. In the literature, researchers support developers in fixing defects by automating various defect-related processes including defect assignment [52], duplicate defect detection [98], defect localization [59], change impact analysis [57], defect prioritization [5], and defect fixing time prediction [112].

Estimating the human labour and the change impact, i.e., amount of change, needed to fix a defect, plays a crucial factor in the efficiency of the defect assignment and prioritization [68, 135] given that there are limited human resources, *i.e.*, developers, available to work on a software project [54]. In the defect fixing effort prediction research, the existing work leverages the status of the issue reports and the textual information of an issue, such as the title and the description, to predict the fixing effort in terms of fixing time [37, 124, 128] and code churn [104]. Similarly, change impact analysis research [74, 126] predicts the amount of change needed to fix a defect in terms of code churn and the number of files changed, leveraging the historical data of the application source code and the textual information of issue reports.

Nevertheless, the predicted defect fixing time might not be accurate since the prediction relies on the status of the issue reports, which might not be updated on time to reflect the real fixing time [51, 127]. For example, developers might not start fixing the defect right after the corresponding issue report was assigned to them. Moreover, the calculated predicted time represents the calendar days or hours, not the working days or hours that reflect the real effort to fix a defect [9]. In addition, predicting the fixing effort, using the summary and the description of the issue report only, is unable to achieve a high accuracy of prediction, *i.e.*, AUC of 0.61 [104]. Moreover, none of the existing change impact techniques focusing on predicting the amount of change leverages the common characteristics of the issue reports to quantize the size of the change into small and large change impact categories without relying on the source code.

In this study, we assess the change impact that predicts the amount of fixing needed in terms of (1) the code churn size and (2) the number of files changed. Issue reports contain valuable information, such as the title, description and severity. In existing work, researchers leverage the collective knowledge of issue reports by identifying shared topics among defects and use them in defect assignments [79, 116, 117, 121]. In our work, we consider the collection of defects belonging to the same topic, which can provide common characteristics of the defects and leverage the topics of issue reports to estimate the change impact rather than relying on the content of individual issue reports. By leveraging the topics of issue reports we could improve the accuracy of the change impact prediction models. More specifically, we conduct our study in two steps. First, we automatically assign a topic to each issue report leveraging the state-of-the-art topic modelling technique, *i.e.,* Embedded Topic Model (ETM) [31]. Second, we train eight predictive models capable of distinguishing between defects requiring a small or large change impact while leveraging the identified topics.

We conduct an empirical study on 298,548 issue reports belonging to three known ecosystems, *i.e.,* Mozilla, Apache, and Eclipse. We predict the change impact along two dimensions, *i.e.,* the code churn size and the number of files changed. In addition, we investigate the most influential features affecting the prediction. We structure our study along by answering the following research questions (RQs):

RQ1: Can we accurately assign topics to issue reports of software systems using ETM?

Developers spend time manually investigating an issue report to understand the nature of a defect. Automatically identifying the topic of an issue report can save the developers' effort and time. In this RQ, we demonstrate that it is feasible to leverage ETM [31] to automatically assign topics to issue reports with a promising average accuracy of 79% over the three ecosystems.

RQ2: Can we predict the change impact of resolving defects in terms of code churn size?

What are the most influential metrics for predicting the change impact in terms of code churn?
Practitioners estimate the change impact of resolving a defect to prioritize the list of defects efficiently. The change impact can be measured by calculating the lines of code changes. In this RQ, we demonstrate that it is feasible to leverage the topics of issue reports to predict the change impact in terms of code churn with a high accuracy achieving an AUC score up to 0.84. We find that the number of attachments, the number of issues blocked, and the number of comments per developer are the most influential metrics for the change impact prediction.

RQ3: Can we predict the change impact of resolving defects in terms of the number of files changed?
Certain defects propagate to several source code locations and require a fix in several files. The number of modified files represents the change impact of resolving the defect. In this RQ, we demonstrate that it is feasible to predict the change impact in terms of the number of modified files with an AUC score up to 0.82.

The main contributions of our work are as follows:

(1) We adopt the topics of issue reports and various features extracted from issue reports to improve the accuracy of the change impact (*i.e.,* code churn size and the number of files) prediction models.
(2) We evaluate the proposed predictive models on a large dataset of 298,548 issue reports of three large-scale open-source systems.
(3) We identify the most important metrics related to the prediction of the change impact of resolving defects.

Paper organization. The remaining part of the paper is organized as follows. Section 2 presents the data collection and preprocessing processes. Section 3 provides a description of the ETM. Section 4 illustrates the motivation, approach, and findings of our research questions. Section 5 discusses the implication of our work. Section 6 describes the potential threats of this study. Section 7 discusses the related work. Finally, Section 8 concludes the study and discusses future work.

2 DATASET

In this section, we present the dataset and the steps to collect and prepare the metrics from issue reports and source code repositories. Figure 1 depicts the overview of our study.

Our study focuses on three large open-source software ecosystems, *i.e.*, Mozilla, Apache, and Eclipse. We specifically select issue reports belonging to these three ecosystems as they are widely used in the realm of software engineering [48, 134], and they are rich in issue reports.

Mozilla and Apache. For Mozilla and Apache, we utilize the 20-MAD dataset provided by Claes and Mäntylä [29] that encompasses 20 years of issue tracker and commit information existing between 1998 to January 2020. The corresponding dataset includes meta-data information about commits (*e.g.*, hash and commit date), issues (*e.g.*, summary, description, and status), and comments (*e.g.*, author and date created). In addition, several other comment-related metrics (*e.g.*, emoticons, sentistrength) processed with various NLP tools are included in the dataset. The data extracted is stored in Parquet files format. First, we convert the data to CSV format and filter out metrics unrelated to our study. As we aim to map the issue reports to their respective commits, we keep only a subset of the reports that represent defects and filter out all the reports whose *final resolution* is not "FIXED" and *status* not "CLOSED". Since the commit information is needed to identify the code churn, we exclude the issue reports that are not associated with any commit.

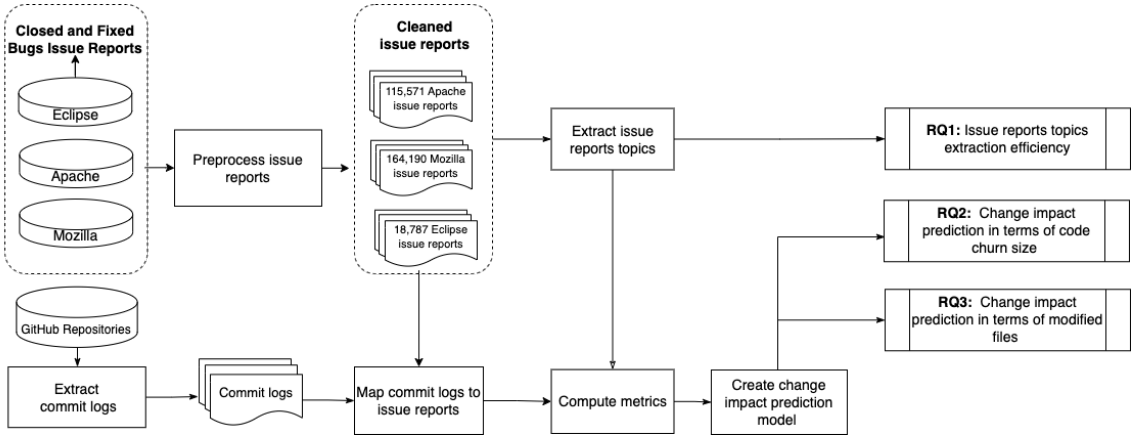

Fig. 1. Overview of our experiment.

Among the metrics used in our study and listed in Section 2.2, the number of attachments, the number of CC and the number of issues blocked do not exist in the 20-MAD dataset. Therefore, we parse issue report data found on the bug tracking systems of Mozilla[1] and Apache[2] and collect the missing metrics from the 20-MAD dataset.

Eclipse. For Eclipse, we download the issue reports belonging to five popular software projects (*e.g.*, Eclipse JDT and Eclipse Platform) from a web-based issue tracker, called Bugzilla.[3] The collected data spans 20 years, from October 2001 until February 2021. Similar to Mozilla and Apache, we only keep the reports representing defects, and that are "CLOSED" and "FIXED". Next, as shown at the bottom of Figure 1, we download the GitHub Repositories of the projects and extract their corresponding commit logs. Next, we download the source code repositories of the projects and their corresponding commit logs. For all the downloaded issue reports, we extract the issue ids and map the id value of each report to the commits, using the numerical id value in the commit log message. To realize the mapping, we adopt a heuristic that matches patterns in the commit messages that include the issue report number using regular expressions [34]. For example, we look for patterns such as *"Bug #1346"* or *"Fix for #6742"*. Then, we use GitPython library [4] to interact with the git repositories and extract the commit information, i.e., number of modified files, number of lines of code added and removed, and commit timestamp of the associated commit. Table 1 summarizes the statistics of the collected issue reports of the three ecosystems.

Table 1. Dataset Statistics.

Ecosystem	Issue reports
Mozilla	164,190
Apache	115,571
Eclipse	18,787
Total	298,548

[1]https://bugzilla.mozilla.org/home
[2]https://issues.apache.org/jira/
[3]https://bugs.eclipse.org/bugs/
[4]https://github.com/gitpython-developers/GitPython

2.1 Issue report labels inconsistency

Issue tracking systems, such as, Bugzilla and Jira, have a field known as keyword and label, respectively, dedicated to tagging issue reports with a meaningful keyword, including the issue report topic. While this field could be helpful to practitioners to better understand the defect, we find that only 13% out of 298,548 issue reports in our study contain at least one keyword. We manually investigate a statistical random sample (i.e., 288 reports in total, 96 from each ecosystem) from the 13% issue reports that have keywords. We notice that: keywords represent various dimensions that do not represent the defect type. For instance, for Eclipse issue reports, 64% of the keywords are irrelevant to the topics of issue reports and include keywords, such as “contributed”, “noteworthy”, “helpwanted”, “greatfix”, “bugday”. For Apache, 90% are irrelevant: "pull-request-available", "windows", "easyfix", "ready-to-commit", "iOS”. For Mozilla, 84% are irrelevant: “fixed1.9.1”, “regression”, “verified1.9.2”, “reproducible”.

2.2 Metrics collection

As shown in Figure 1, after we associate the issue reports to their commits, we compute the metrics. In total, we collect 11 metrics, *i.e.,* the code churn size from the associated commit and another 10 metrics from the issue reports. The 10 metrics from the issue reports constitute textual factors (*e.g.*, the title of issue reports) and characteristic factors (*e.g.*, the number of comments per developer).

Issue report textual metrics

- *The length of the issue report title:* is the count of words encompassed in the title field of the issue report. A longer title might represent a more complex defect that might have a larger change impact.
- *The length of the issue report description:* is the count of words encompassed in the description field of the issue report. Zhang et al. [127] explain that the length of the issue report might indicate the complexity of understanding the issue report. In addition, Huang et al. [48] explain that a more extended description of the issue report provides more elaborate information about the issue. Therefore, we assume that a complex defect that has a large change impact is more likely to have a lengthy issue report description.
- *The number of comments per developer:* is the ratio of the count of comments posted prior to fixing the defect and the count of developers involved in posting the comments on an issue report. Comments are an indication of increased communication [35]. Increased communication might indicate that the defect is complex and hence have a larger change impact [108].

Issue report characteristics metrics

- *The number of CC:* is the count of distinct developers added to the list of carbon copy (CC). A developer added to the list of CC is a developer interested in the progress of the defect. A large number of CC might indicate that the defect is a bottleneck in the maintenance process [107] which can consequently suggest that the defect might have a larger change impact.
- *The number of issues blocked:* is the count of issue reports that can be fixed only after the issue report in question is resolved. The larger the number of issue reports blocked by an issue, the more likely it is that the issue has more change impact [107].
- *The number of votes:* is the count of users who would like the issue report resolved. Developers vote for an issue report if they favour fixing the defect and consider it an important issue [110]. Also, the number of votes unveils the effort invested in discussing an issue [35]. Since the number of votes can be perceived as an indication of the

importance of the issue [67], an issue of high importance might affect a larger scale of the software product, i.e., several modules, and hence have a higher change impact [6]. Therefore, we assume that an issue report with a larger number of votes might have a larger change impact.

- *The number of attachments:* is the count of attachments added to the issue reports. Attachments can include testing cases, stack traces and screenshots. A larger number of attachments might be an indicator that the report may have a large change impact [15].
- *Has version:* is a boolean variable that describes whether the issue report has a version (*i.e.,* the version of the software). The tracking information is likely to affect the change impact. Related work [82] demonstrates that the version metric can influence the defect fixing time.
- *Has milestone:* is a boolean variable that describes whether the issue report has a milestone target (*i.e.,* target to fix). Similarly to *has version*, a report with more tracking-related information might affect the change impact.
- *Is severe:* is a boolean variable that describes whether the issue report is considered severe or not. We consider the defect severe (*i.e.,* value 1 is assigned) if the severity level specified in the issue report is *blocker*, *critical*, *major*, *urgent*. A more severe defect might have more change impact [76].

Code churn. A code churn is the number of modified lines of code which is the sum of the lines of code added and removed [60, 66, 78]. Every issue report is associated with one or more commits. Thus, for every report, we extract the number of lines of code modified (*i.e.*, added or removed), which is the sum of the changed lines as the code churn.

The number of changed files. The number of changed files represents the spread of the propagation of the code change [88, 127] in the file level. We calculate the total number of files changed for every issue report as the sum of the added, deleted, and modified files. If an issue report is associated with several commits, the total number of changed files equals the sum of the number of files changed in all the commits.

2.3 Data preprocessing

As depicted in Figure 1, we apply the same preprocessing steps to Mozilla, Apache and Eclipse issue reports resulting in cleaned issue reports.

Removing the noise. A manual investigation of the issue reports shows that the description field of an issue report could contain noise. For example, it is common for developers to include code fragments and log traces in the description. Therefore, we use regular expressions to identify the lines that contain noise. We exclude the code fragments by identifying the lines embedded in the `Code` or `noformat` tags by using the regular expression `(Code:).*(Code)` and `(noformat):.*(noformat)`. For the log traces, some lines start with timestamps of the forms `hh:mm:ss` and `hhmmss`. Hence, we identify the log lines using two regular expressions `^\d{2}[:]+\d{2}[:]+\d{2}.*` and `^\d{2}+\d{2}+\d{2}.*` and remove them from the descriptions of the issue reports. Some other log traces start with the expression `Caused by`. Therefore, we use another regular expression `(^Caused by)` to exclude them. In addition, we remove the hyperlinks and the automatically generated code related to automated tests using the following respective regular expressions `(https|http).*?[\t\s\n]` and `^(TEST-INFO|TEST-START|Build ID:|User Agent:)`.

Normalizing the text. We apply the following text normalization steps on the title and description of an issue report. We remove English non-description stop words (*e.g.*, "the", "was" and "of") using the `nltk` package [16] that contains

a defined corpus of English stop words. We also apply text tokenization to exclude punctuation and special symbols. Lastly, we bring the words to their ground forms by converting them to their stemmed and lemmatized versions. This step is essential for Topic Modeling as it maps the words "connections", "connecting" and "connected" to their basic form "connect" [69] which reduces the vocabulary size.

3 BACKGROUND

In this section, we discuss about ETM, a state-of-the-art topic modelling technique that we adopt to taxonomize the collected issue reports.

In 2003, Bengio et al. [12] introduce the concept of word embeddings, a distributed learned representation for the text that represents words with similar meanings in close proximity in a vector space. Word embedding plays a vital role in the realm of natural language processing. In particular, ETM adopts the continuous bag-of-words (CBOW) [72] word embeddings. Given a corpus of documents D with V unique words, let represent the n^{th} word in the d^{th} document. The CBOW likelihood of the word w_{dn} is:

$$w_{dn} \sim softmax(\rho^T \alpha_{dn})$$

where α_{dn} represents the transpose of the embedding matrix that contains the embedding representations of the vocabulary. α_{dn} represents the context embedding which is the sum of the vectors of the words surrounding w_{dn}.

In 2003, Latent Dirichlet Allocation (LDA) [17], a statistical model that generates topics, is developed. LDA considers that each topic is characterized by a full distribution over the vocabulary of a corpus. Each document is represented by a unique mixture of topics. However, despite its popularity, LDA suffers in learning interpretable topics when the vocabulary size becomes immense. To overcome the limitation mentioned above for large vocabulary and obtain good quality of topics, practitioners should omit words to reduce the vocabulary size. However, pruning the vocabulary could also threaten the quality of a model. To mitigate the limitations of LDA, Dieng et al. [31] propose ETM. ETM has been shown to outperform LDA by being robust to even large vocabularies.

ETM is a technique that combines properties from both topic modelling and word embedding. First, it relies on the topic model to identify interpretable latent semantics of the corpus. Second, it leverages word embedding to efficiently represent the meaning of the words in the vector space. Similar to LDA, ETM is a generative probabilistic model that represents each document as a probability of topics. Compared to LDA, ETM represents not only the words using embedding vectors but also the topics. For instance, in LDA, the k^{th} topic is a distribution over all the words in a vocabulary. In contrast, ETM represents the k^{th} topic as an embedding vector, *i.e.,* α_k, in the embedding space. Also, ETM presents an improvement over LDA in the process of reconstructing the words from an assigned topic. It relies on the topic embedding and the embeddings of the vocabulary to assign words to each topic using the CBOW likelihood. However, in ETM, the context embedding is selected from the document context instead of the surrounding words as in standard CBOW.

There are two parameters in ETM: the word embedding ρ and the topic embeddings α. In the fitting process, ETM is trained to maximize the marginal likelihood of the documents as follows:

$$\mathcal{L}(\alpha, \rho) = \sum_{d=1}^{D} \log p(w_d | \alpha, \rho)$$

where for a given corpus of documents $\{w_1,...,w_D\}$, w_d is a collection of N_d words. Thus, the word embedding and the topics are found concurrently by ETM. However, the word embeddings can be prefitted. In that case, the topics can be identified in a specific embedding space.

Similar to LDA, ETM represents each document as a probability of topics, and each topic is a distribution over words. ETM will assign each document one topic.

4 RESEARCH QUESTIONS

In this section, we evaluate the feasibility of clustering issue reports into topics and the viability of leveraging the topics of issue reports in predicting the change impact. Precisely, we discuss motivation, approach and findings for our research questions.

4.1 RQ1: Can we accurately assign topics to issue reports of software systems using ETM?

4.1.1 Motivation. In practice, developers manually investigate the issue reports to complete development activities such as defect assignment and prioritization [8, 87, 119]. Issue reports contain rich information that can help developers understand the nature of the defects and therefore save development effort and time. Lili et al. [64] demonstrate that defects of the same categories tend to have the same trend of fine-grained change operations, *e.g.,* if statement, while statement, assignment statement and function call statement, and the same frequency of fine-grained change operation use. The collective knowledge in issue reports, i.e., topics, can guide developers in the defect fixing activities and in predicting the change impact of fixing a defect. As depicted in Section 2.1, most issue reports do not contain keywords representing the topics of issue reports. Therefore, in this RQ, we leverage the state-of-the-art topic model, ETM, to automatically discover the hidden common topics across the issue reports. Having a high accuracy in identifying the topics provides valuable data to the predictor models to estimate the change impact. We also study the relationship between the identified topics and their severity.

4.1.2 Approach. Since the effectiveness of our proposed issue report classification approach relies on the accuracy of the adopted embedding-based topic model, we additionally conduct a quantitative evaluation of the effectiveness of ETM in extracting topics associated with every issue report in our dataset. We use ETM implementation[5] provided by its authors. Our approach consists of three steps.

Step 1: Non-textual data processing. To obtain optimal results, we use ETM authors' script[6] to filter out words with a *maximum document frequency* above 70% and remove low-frequency words appearing in only a few documents, referred to as the *minimum document frequency*. Setting a *minimum document frequency* to remove the low-frequency words helps eliminate the rare words that are not important to the topic model. It also reduces vocabulary size [31] and leads to better computing time. After varying the value for the minimum document frequency by increasing starting and 1, we set it to 15 as for the values higher than 15, only a small amount of improvement in the vocabulary pruning and the computing time can be achieved. Hence, we exclude the words that appear in less than 15 documents from the vocabulary. Then, we exclude the reports that have a combined length for description and summary of fewer than three words as previous studies find that short textual information rarely conveys meaningful information [11, 22, 62].

Step 2: Topic modeling hyperparameter tuning. Selecting the optimal number of topics plays a pivotal role in the quality of the topic modelling results. We rely on the topic coherence (*i.e.,* the interpretability of a topic [73]) and

[5]https://github.com/adjidieng/ETM
[6]https://github.com/adjidieng/ETM/tree/master/scripts

topic diversity (*i.e.,* unique words in the top 25 words of all topics as defined by the ETM authors) to quantitatively measure the quality of the model in respect to the selected topic number. Additionally, we rely on the human judgement that aligns with the quantitative metrics [100]. We run the model with different numbers of topics by increasing and decreasing the value of the topic number and finally set it to 15 as it provides the best quality of topics. We also set the number of epochs to 300.

Step 3: Classification accuracy. After applying ETM on the issue reports of all three ecosystems, we obtain 15 clusters of keywords, each representing an issue report topic. The first and third authors followed an open coding approach [90, 92] to manually and independently assign labels to the clusters generated by ETM. To evaluate if ETM can accurately assign topics to issue reports, we select for each ecosystem a statistically representative random sample of issue reports with a confidence level of 95% and a confidence interval of 10%. In total, we select 288 issue reports that belong to Mozilla, Apache and Eclipse. We perform the below three steps to evaluate the performance of the approach:

(1) The first and the third authors independently assign one topic from the 15 topics obtained by ETM to each of the 288 sample reports.
(2) The Cohen's kappa agreement score [70] is calculated on the annotated testing issue reports using the "irr" package[7] provided in R[8]. We achieve a score of 0.78, which indicates a substantial level of agreement. Next, the annotators resolve the disagreements after discussing the conflicts case by case. Finally, all the issue reports were assigned to one issue report topic.
(3) We calculate the final accuracy as the percentage of true positive (TP), where a TP represents a topic assigned correctly by ETM that matches the topic assigned by an annotator.

The three ecosystems adopt different severity level schemas, e.g., Apache adopts an 8-level severity schema, whereas Eclipse adopts a 5-level schema. Therefore, to conduct the severity analysis across the different topics, we map the various severity levels of the three ecosystems to a three-level severity schema, *i.e.*, *low*, *medium* and *high*, introduced by Thung et al. [105]. Table 2 exhibits the three-level severity schema mapping. Then we conduct the following statistical tests.

(1) To check if the severity is significantly different among the topics of issue reports, we conduct a comparison using the Kruskal-Wallis test [58], a non-parametric statistical test used to compare more than two samples of data. If we obtain a *p-value <= 0.05*, we reject the null hypothesis and conclude that not all the topics of issue reports have the same median severity.
(2) If the null hypothesis is rejected, we investigate if the differences among the topics are of strong significance by calculating the epsilon squared effect size [123]. The effect size represents the relationship of the variables, such as the topic of issue reports and severity, on a numeric scale. A value close to zero indicates a negligible effect, whereas a value close to 1 indicates a very strong effect. We refer to Table 3 to identify the effect size.
(3) To further differentiate the topics of issue reports with different severity levels without ambiguity, we conduct the Scott-Knott Effect Size Difference (SK-ESD) [102, 103]. The SK-ESD uses hierarchical clustering to compare the means in the dataset to form statistically distinct groups. The SK-ESD clusters the topics of issue reports in a way where the intra-group difference in severity level is negligible and the inter-group difference is non-negligible.

4.1.3 Results. **Our approach identifies 15 topics of issue reports commonly existing in the three ecosystems but with different distributions**. Table 4 provides a closer look at the topics obtained by ETM along with the top 10

[7]https://cran.r-project.org/web/packages/irr/index.html
[8]https://www.r-project.org/

Table 2. Three-level severity schema mapping.

Severity level	Mapping
trivial, minor, low, and normal	Low
major	Medium
blocker, critical, and urgent	High

Table 3. Epsilon square effect size interpretation reference.

Epsilon squared	Effect size
0.00 < 0.01	Negligible
0.01 < 0.04	Weak
0.04 < 0.16	Moderate
0.16 < 0.36	Relatively strong
0.36 < 0.64	Strong
0.64 < 1.00	Very Strong

Table 4. Inferred fifteen issue report topics with their representative keywords and an example of issue report

Topic	Keywords	Sample report title
Platform compatibility	'app', 'devic', 'sync', 'web', 'android'	Remove usage of nsIDOMWindowUtils.goOnline() in mobile's netError.xhtml
Testing	'test', 'fail', 'run', 'unit', 'expect'	Update .hgignore to ignore Loop unit test files
User experience	'open', 'page', 'step', 'tab', 'window'	Dialog opens up off the screen
File management	'file', 'packag', 'depend', 'instal', 'path'	Export bundle should create the description file
Build and deployment	'build', 'warn', 'compil', 'make', 'consol'	javaCompiler*.args generated just once per session
API related issues	'instanc', 'class', 'method', 'interfac', 'context'	SWIG interface doesn't support CASException thrown from some common APIs
Security	'user', 'password', 'client', 'group', 'permiss'	Password reset issues tokens w/ "&" in them, URL not escaped
Release and Update	'updat', 'version', 'releas', 'patch', 'branch'	Add license header to RELEASE_NOTES
Performance	'cach', 'memori', 'size', 'buffer', 'alloc'	Portable spark: thread/memory leak in local mode
Database	'tabl', 'data', 'queri', 'schema', 'record'	Delete table does not remove the table directory in the FS
Parallel event processing	'event', 'frame', 'process', 'thread', 'call'	Convert formSubmitListener.js to a process script instead of a frame script
General program related anomaly	'use', 'implement', 'code', 'want', 'work'	Implement automatic bookmarks backup for 1.1
GUI	'background', 'imag', 'font', 'style', 'display'	Scrollbar handle is not colored correctly when selected and dragged in gtk3
Server issues	'thread', 'run', 'connect', 'session', 'node'	Secondary socket of "tee" socket is not threadsafe
Interprocess communication (IPC)	'url', 'request', 'input', 'header', 'pars',	Olingo2's batch process generates the invalid request

keywords. We observe that some of the topics are considered common topics equally present in the three ecosystems. A few topics are system specific and are widely present in a specific ecosystem compared to the others. For example, as we

Table 5. The extracted topics of issue reports percentages distribution across the three ecosystems

Topics	Apache(%)	Mozilla(%)	Eclipse(%)
Platform compatibility	1	11	1
Testing	6	7	3
User experience	3	14	20
File management	12	2	21
Build and deployment	3	10	5
API related issues	12	2	14
Security	13	2	1
Release and Update	4	8	9
Performance	2	10	1
Database	15	2	2
Parallel event processing (PEP)	1	8	1
General program anomaly (GPA)	2	6	5
GUI	2	12	10
Server issues	16	2	3
Interprocess communication (IPC)	8	4	4

Table 6. Accuracy of ETM calculated on 288 manual labelled issue reports.

Ecosystem	Manually labelled issue reports	Accuracy
Mozilla	96	76%
Apache	96	77%
Eclipse	96	83%
Average	288	79%

can see in Table 5, *GUI* and *User experience* are predominantly present in Mozilla and Eclipse and represent combined more than 30% of the topics of issue reports as opposed to 4% for Apache. This is intuitive since the browser and IDE domains heavily rely on user interaction and navigation through a graphical interface. On the other hand, *Server issues*, *Security* and *Database* exceed 50% of the topics of issue reports in Apache which is common for web servers. *Performance* for instance is uniquely dominant for Mozilla, which can be explained by the importance of performance for web accessibility. Also, we observe that *Platform compatibility*, *Testing*, *Interprocess communication*, and *Release and update* have similar distribution across the three ecosystems. In fact, these topics are expected to be present in almost any kind of software.

Our approach achieves an accuracy of 83%, 72% and 77% for Eclipse, Mozilla and Apache. As shown in Table 6, our approach achieves high accuracy across the three ecosystems with an average of 79%.

Defects of the different topics have different median severity levels for the three ecosystems. Table 7 shows that the Kruskal-Wallis test's p-value <= 0.05 for Mozilla, Apache and Eclipse. Moreover, we observe that for Mozilla, the Epsilon squared effect size is moderate and weak for Apache and Eclipse. Table 8 shows the different severity groups obtained by the Scott-Knott test. We notice that *GUI* and *IPC* constantly belong to the groups representing the low severity across the three ecosystems. Figures 2, 3 and 4 show the distribution of severity levels across the different topics of issue reports for Mozilla, Apache and Eclipse, respectively.

By observing these figures, we come up with two findings:

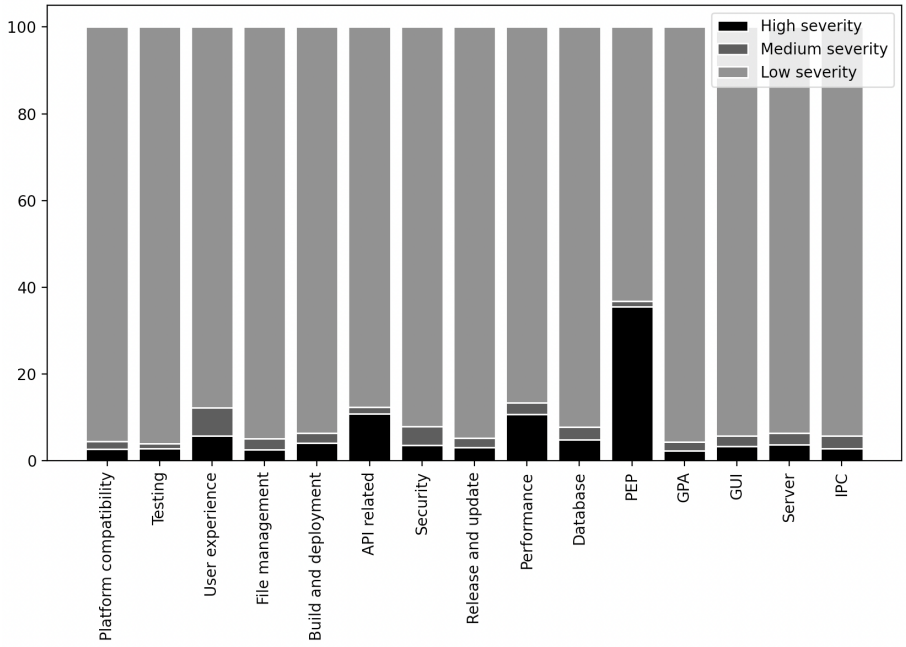

Fig. 2. The distribution of severity levels for Mozilla issue reports.

(1) We notice that different topics of issue reports have different severity levels within the same ecosystem. In Mozilla and Eclipse, for example, the number of defects with high severity belonging to the *Parallel event* topic is at least double the number of high-severity defects in other topics. For Apache, more than 20% of the *Security* and *Server* defects are highly severe, whereas less than 10% are severe for *API* and *GUI* topics.
(2) The same topics of issue reports have different severity levels distribution across the studied ecosystems. When considering Apache, we notice that *Security* and *Server* topics have the highest proportion of highly severe defects. This could be explained by the fact that Apache applications fall under the web server domain, in which reliability is crucial. As for Eclipse and Mozilla being IDE and client/browser applications respectively, *Parallel event* defects that represent processes and threads are more severe than others.

Table 7. Kruskal-Wallis p-value and Epsilon squared effect size

Ecosystem	p-value	Epsilon squared	Effect size
Mozilla	0.000000e+00	0.088	Moderate
Apache	0.000000e+00	0.024	Weak
Eclipse	3.004759e-41	0.012	Weak

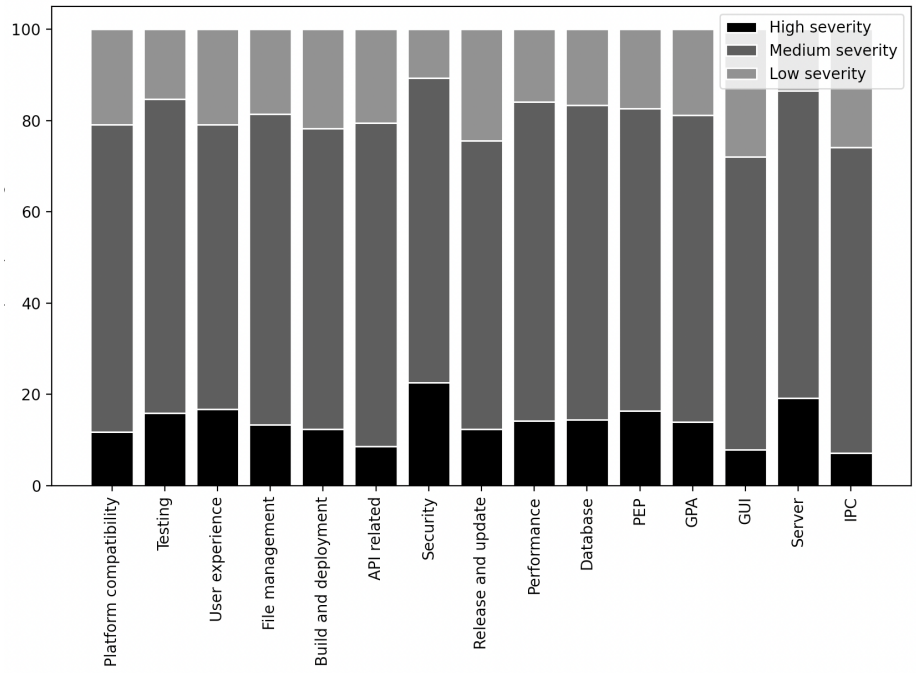

Fig. 3. The distribution of severity levels for Apache issue reports.

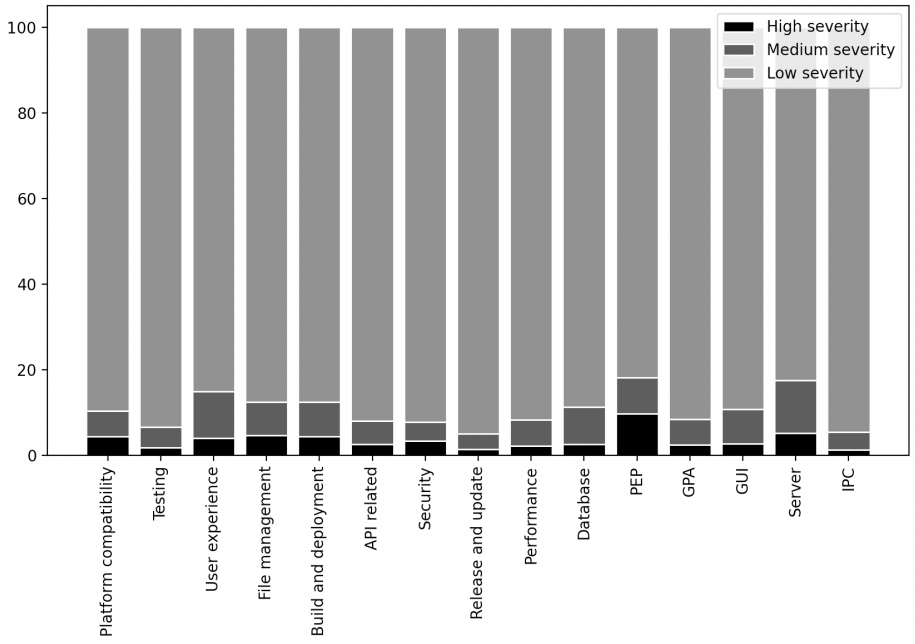

Fig. 4. The distribution of severity levels for Eclipse issue reports.

Table 8. The severity group raking obtained by SK-ESD.

Mozilla		Apache		Eclipse	
Group	**Topic**	**Group**	**Topic**	**Group**	**Topic**
G1	Parallel event	G1	Security	G1	Parallel event
G2	Performance	G2	Server		Server
	API related	G3	Testing	G2	User experience
G3	User experience		Parallel event		File management
G4	Database		Performance		Build and deployment
	Security		Database	G3	Platform compatibility
	Build and deployment	G4	User experience		Database
	Server		GPR		GUI
G5	GUI		File management		Security
	IPC	G5	Platform compatibility		GPR
	Release and update		Build and deployment		API related
	File management		API related		Performance
	Platform compatibility		Release and update		Testing
	Testing	G6	IPC		IPC
	GPR		GUI		Release and update

Summary of RQ **1**

ETM achieves a promising average accuracy of 79% on 288 manual annotated reviews. The extracted topics of issue reports can support the developers in better understanding the nature of the defect.

4.2 RQ2: Can we predict the change impact of resolving defects in terms of code churn size? What are the most influential metrics for predicting the change impact in terms of code churn?

4.2.1 Motivation. Given the time constraints and limited resources, during defects assignment, practitioners, *e.g.,* developers and project managers, estimate the defect fixing effort before assigning the defect to a developer. When prioritizing defects, practitioners take into consideration the effort level required to fix the defect [54]. The developer effort can be measured by the size of the required change, i.e., lines of code modified that is the code churn [18]. In this RQ, we want to predict the change impact of defects in terms of code churn size. We do not only use information extracted from issue reports but also the topics of issue reports extracted in RQ1. In addition, we identify the most influential metrics for predicting the change impact. This knowledge can help developers better understand the prediction results and guide them on which metric they should focus on to estimate the change impact.

4.2.2 Approach. Our goal is to predict the amount of change needed to fix the defect in terms of code churn using only the issue report information. Thus, we formulate the dependent variable of our prediction model (a.k.a., the prediction output Y) to be a boolean variable representing whether the issue report requires a large or a small change impact. If the report is classified to require a large change impact the prediction output Y = 1.

Dependent variable. In this RQ, we define the dependent variable as the change impact of fixing a defect measured in terms of code churn size (*i.e., small* and *large*). We apply two different steps to achieve the report classification to change impact. First, we apply the log transformation to the numerical code churn value collected from the commit to correct the skewness in the data. Second, we sort the issue reports by the increasing order of its code churn size and select the lower 10% as issue reports requiring *small* change impact and the upper 10% as the ones requiring a *large*

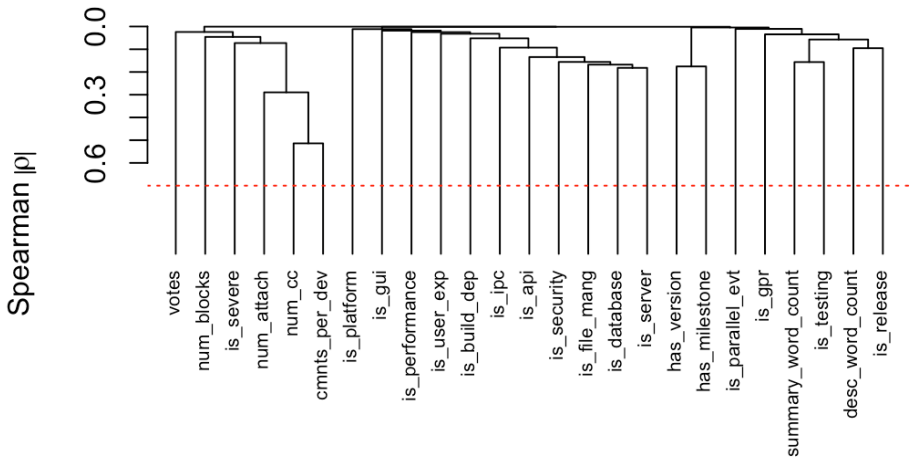

Fig. 5. The hierarchical clustering of independent variables for Apache. The dotted red line represents the threshold value of 0.7.

change impact. To obtain the dependent variable, we assign the value of 1 for the reports with *large* change impact and 0 otherwise.

Independent variables. In total, we have 25 independent variables. While 10 of these metrics are directly extracted from issue reports as explained in Section 2.2, we synthesize another 15 metrics that represent the topics of issue reports. In RQ1, we extracted 15 different topics. To leverage these extracted topics, we created 15 different boolean independent variables: *Is platform*, *Is testing*, *Is user experience*, *Is file management*, *Is build & deployment*, *Is API*, *Is Security*, *Is Release*, *Is Performance*, *Is Database*, *Is Parallel event*, *Is GPR*, *Is GUI*, *Is Server*, *Is IPC*. Each of the aforementioned variables represents an issue report topic for the collected report. As explained in Section 4.1, each report belongs to only one of the 15 topics. Therefore, for every report, only one of the 15 independent variables has the value 1 and the others is assigned 0.

Correlation and redundancy analysis. The presence of correlated metrics might affect the performance of the model [53]. Therefore, we apply *varclus*[9] function in R to detect the existence of highly correlated metrics in our dataset. We consider any pair of metrics that achieves a coefficient of 0.7 [71] and higher as highly correlated. Figure 5 illustrates the Spearman correlation of the metrics of Apache projects. We only present the correlation of one ecosystem due to space limitations. Mozilla and Eclipse correlation analyses give similar results, *i.e.,* the absence of highly correlated features; therefore, we keep all the metrics and exclude none.

Prediction models. We train eight different machine learning models to automatically classify the issue reports based on the *small* and *large* change impact they require. We use the following models that are widely utilized in the binary prediction and in the realm of Software Engineering [35, 48, 120]: *Logistic regression*, *Naive Bayes*, *SVM*, *Random Forest*, *XGboost*, *Catboost*, *LGBM* and *Multi-layer Perceptron*. All the ML models are implemented using the scikit-learn[10] library in Python. We adopt the 10-fold cross-validation approach to validate the models and ensure reliable performance. To configure the machine learning models, we use two automated parameter optimization techniques Random Search and Grid Search. Grid Search is a brute-force approach that finds the best hyperparameters for the machine learning

[9]https://search.r-project.org/CRAN/refmans/Hmisc/html/varclus.html
[10]https://scikit-learn.org/

model [13]. However, computing all possible combinations of parameters is time-consuming. Therefore, to tune our machine learning models efficiently, we first use RandomSearch, which is capable of testing a random wide range of parameters very fast. After obtaining the best values for every parameter through Random search, we adopt Grid search on a smaller search space of parameters. Grid Search identifies the best combination of the parameters after applying the cross-validation score. To achieve the best performance for our models, we adopt the RandomizedSearchCV and the GridSearchCV from sk-learn to tune the hyperparameters of models.

To provide a robust model evaluation and to imitate the real-life settings, we adopt a time-based train and test data splitting. Instead of randomly splitting the data into 80% training and 20% testing, we sort the issue report by their creation date, select the most recent 20% of reports as testing, and leave the rest for training. The Grid Search is performed on the training set with cross-validation. Once the optimal hyperparameters are obtained, we evaluate the model on the test set. We use the 25 metrics described above as dependent variables for all the models.

Evaluation metrics. To quantify the performance of the predictive models, we consider precision, recall, and F_1-Score (*i.e.,* the harmonic combination of precision and recall) as the evaluation metrics. Equations 1, 2, and 3 show the computation for precision, recall, and F_1-Score.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F_1\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

We also use Area Under the Receiver Operating Characteristic Curve (AUC) [65] to evaluate the effectiveness of our machine learning models. The AUC values range from 0 to 1. The performance of a prediction model is considered promising if the AUC-ROC is 0.7 and above [38, 80] and 1 denotes a perfect predictive power.

Sensitivity Analysis. Our prediction approach has one hyper-parameter, the small-large code churn change impact threshold. In this section, we select the prediction model that can achieve the best performance to conduct the sensitivity analysis. Then, we experiment with three additional thresholds, *i.e., 25%*, *40%*, and *50%*. The lower 25% reports are considered as defects with *small* change impact and the upper 25% as *large*. A similar classification applies to the *40%*, and *50%* thresholds.

Feature importance. To find the most influential metrics, first, we investigate if the 15 metrics, *i.e.,*, topics of issue reports, synthesized from the results obtained in RQ1 improve the prediction model. We construct a baseline model derived from the best performing model and best small-large code churn threshold that considers only 10 metrics, *i.e.,* 3 issue report textual metrics, 7 issue reports characteristic metrics. We exclude the 15 issue report topics metrics. We also conduct a complementary study that seeks to identify the most influential metrics among the 25 metrics for identifying the change impact. Second, since every ecosystem is made of several software projects, *e.g.,* HIVE, Firefox and JDT are three popular software projects belonging to Apache, Mozilla and Eclipse respectively, we create individual prediction models per software project. To avoid working on toy software projects, we randomly select 4 software projects from each ecosystem having at least 2,000 issue reports [95].

We employ the permutation feature importance[11] from sk-learn to detect the influence, a.k.a. importance, of every metric in our model. In the permutation feature technique, the metric importance (*i.e.,* feature importance) is calculated by considering the drop in the model performance score when the values of the metric in question are randomly shuffled [86]. Shuffling the values of the metric leads to breakage between the metric and the dependent variable and therefore indicates to what extent the model depends on this metric. As done in previous work [48], to attain a reliable result, we apply the permutation test on all 25 metrics repeatedly 10 times. To statistically identify the magnitude of the difference between the importance score of the metrics, we compute the SK-ESD test. The SK-ESD clusters the 25 metrics into groups, *i.e.,* ranks, based on their importance score.

4.2.3 Results. **It is feasible to predict the issue reports requiring large change impact in terms of code churn based on the information of the issue reports and the topics of issue reports.** Table 9 shows the performance in terms of precision, recall, F1-Score, and AUC of the eight constructed prediction models. As we can notice, XGBoost achieves the best performance. For instance, the XGBoost model achieves AUC values of 0.84 for Mozilla, 0.76 for Apache, and 0.74 for Eclipse ecosystems. There exists a minuscule difference between the AUC values of the gradient boosting tree-based models (*i.e.,* XGboost, Catboost, LGBM) and Multi-layer Perceptron model AUCs. This can be explained by the fact that all of these models are very efficient in interpreting the complex relationships in our issue reports tabular data [55].

Our approach is not sensitive to the threshold of the selection of the code churn classes (*i.e., small* and *large*). We select XGBoost, the best performing model, to conduct the sensitivity analysis. Table 10 show that for Mozilla, the AUC drops by 8% to 16% when the threshold is 25% (*i.e., XGBoost-25%*) and 50% (*i.e., XGBoost-50%*) respectively. Similarly, for Apache, a drop of 7% to 13% when the threshold is 25% and 50% respectively. As for Eclipse, the AUC drops by 6% to 13% when the threshold is 25% and 50%, respectively.

The model fit on the metrics, including the topics of issue reports, improves the base model's AUC by 5%. The model fit on all the metrics, including the topics of issue reports metrics extracted in RQ1, achieves an AUC value of 0.78 on average (AUC 0.84 for Mozilla, AUC 0.76 for Apache and AUC 0.73 for Eclipse) across the three ecosystems for classifying the issue report change impact. This obtained AUC of 0.78 outperforms the AUC of the based model *XGBoost-10-base* fit on the textual and characteristic metrics only, present in the issue report, that achieves an average of 0.73 (AUC 0.80 for Mozilla, AUC 0.71 for Apache and AUC 0.67 for Eclipse). This result suggests that the topics of issue reports can support developers in predicting the change impact of fixing a defect. For instance, we observe in Table 11, that *Is GUI* and *Is User experience* are among the top 5 influential metrics for Eclipse and *Is database* ranks 6th for Apache.

The number of attachments and the number of comments per developer are among the top 3 influential metrics across the three ecosystems. Table 11 depicts the top 6 influential metrics along their importance scores and ranks. The importance score is the average importance score of the corresponding metrics. **As we can notice, the top 6 metrics are ranked differently across the systems. However, the number of attachments and the number of comments per developer are among the top 3.** This suggests that issue reports with more attachments and developers in their discussions have a larger change impact. The presence of attachments, including test cases, may indicate the complexity of a defect, thus leading to a larger change impact. Similarly, a higher number of comments per developer in an issue may also be associated with a more complex defect that is harder to solve.

[11] https://scikit-learn.org/stable/modules/permutation_importance.html

Table 9. The performance of our prediction models for the considered datasets. The values shown represent the prediction of the issue reports that require a large change impact in terms of code churn size. Prec. represents the precision.

	Ecosystem	Prec.	Recall	F-score	AUC
Logistic Regression	Mozilla	0.93	0.64	0.76	0.80
	Apache	0.74	0.65	0.69	0.71
	Eclipse	0.77	0.72	0.74	0.70
Naive Bayes	Mozilla	0.86	0.44	0.58	0.68
	Apache	**0.80**	0.54	0.64	0.71
	Eclipse	0.77	0.47	0.58	0.63
SVM	Mozilla	**0.94**	0.62	0.74	0.79
	Apache	0.74	0.65	0.69	0.72
	Eclipse	**0.78**	0.71	0.74	0.70
Random Forest	Mozilla	0.93	0.73	0.81	0.83
	Apache	0.75	0.68	0.71	0.74
	Eclipse	0.74	0.74	0.74	0.68
XGboost	Mozilla	0.93	0.72	0.81	**0.84**
	Apache	0.74	0.68	**0.71**	**0.76**
	Eclipse	0.76	0.77	0.76	**0.73**
Catboost	Mozilla	0.93	0.73	**0.82**	0.83
	Apache	0.74	0.68	0.71	0.74
	Eclipse	0.75	0.75	0.75	0.71
LGBM	Mozilla	0.93	0.73	0.82	0.83
	Apache	0.74	0.69	0.71	0.73
	Eclipse	0.71	0.81	0.76	0.67
Multi-layer Perceptron	Mozilla	0.92	**0.74**	0.82	0.83
	Apache	0.72	**0.69**	0.71	0.72
	Eclipse	0.71	0.81	0.76	0.66

Table 10. AUC performance of the three ecosystems with three different thresholds for the code churn size.

	XGBoost-25%	XGBoost-40%	XGBoost-50%
Mozilla	0.78	0.73	0.70
Apache	0.67	0.64	0.61
Eclipse	0.67	0.63	0.60

We observe that the *number of issues blocked* ranks in second place for Mozilla. The association between the number of issues blocked and the change impact can be attributed to the fact that a defect that blocks several other defects may be used in several packages or in several modules, which may have a larger change impact. In fact, Valdivia-Garcia et al. [107] quantified the effect caused by blocking defects and found that blocking defects require between 1.2–4.7 more lines of code changes than non-blocking defects.

The severity of the issue report, which represents the importance of the issue report, is also an important metric for Apache projects. This result hints that severe defects may be treated meticulously where additional coding practices are implemented, thus leading to a larger change impact.

Similar to the ecosystem-level prediction, the performance of the software project-level change impact prediction model is improved when the issue report topics metrics are included. We perform a per software

Table 11. The top 6 most influential metrics in the XGBoost-10% model for Mozilla, Apache and Eclipse ranked by their importance. Imp. represents the importance score obtained by the feature permutation approach. The Rank column represents the clusters raking obtained by SK-ESD.

	Metric	Imp.	Rank
Mozilla	# of attachments	0.1424	1
	# of issues blocked	0.0271	2
	# of comments per developer	0.0269	3
	Has milestone	0.0114	4
	Title of issue report	0.0093	5
	Has version	0.0089	6
Apache	# of comments per developer	0.0559	1
	# of attachments	0.0323	2
	Is severe	0.0195	3
	Description of issue report	0.0193	4
	# of CC	0.0092	5
	Is database	0.0066	6
Eclipse	# of comments per developer	0.1086	1
	# of issues blocked	0.0365	2
	# of attachments	0.0107	3
	Is GUI	0.0090	4
	Is User experience	0.0072	5
	Description of issue report	0.0063	5

project prediction for the software projects of the sampled ecosystems shown in Table 12 using XGBoost-10%. Table 12 shows that the per-software project prediction model performance is improved by up to 5% in terms of AUC when the topics of issue reports are fed to the machine learning model. Including the topics of issue reports metrics improves the recall by up to 11% (*i.e.,* JDT and AMBARI). The results suggest that the topics of issue reports have a positive impact on the prediction model.

Summary of RQ 2

Our results propose that machine learning models such as XGBoost have the potential to predict the change impact in terms of code churn size with a high AUC of 0.84, 0.76 and 0.73 for Mozilla, Apache and Eclipse. The topics of issue reports could be leveraged to achieve higher accuracy for predicting the change impact for the three ecosystems. To benefit from the change impact prediction models, we suggest to the reporters to attach relevant documents to the issue reports, list the issues blocked by the defect in question and indicate the accurate severity of the defect.

4.3 RQ3: Can we predict the change impact of resolving defects in terms of the number of files changed?

4.3.1 Motivation. The number of files changed represents the amount of effort required to fix a defect [89]. Previous work demonstrates that the code churn size and the number of files changed in a defect fix are not highly correlated [42]. The amount of changed files is an indication of the propagation of a code change. Some commits impact many files and require a change in several locations, whereas others require a local change in a single function of one file.

Table 12. The performance measures of our XGBoost-10 prediction models per software project. The values shown represent the prediction of the issue reports that require a large change impact.

	Software project	Precision	Recall	F-score	AUC
Mozilla	Core	0.91	0.77	0.84	0.86
	Core (without defect types)	0.90	0.76	0.82	0.85
	Firefox	0.87	0.51	0.65	0.72
	Firefox (without defect types)	0.86	0.55	0.67	0.72
	Firefox OS	0.98	0.79	0.88	0.87
	Firefox OS (without defect types)	0.97	0.78	0.86	0.85
	Toolkit	0.90	0.64	0.75	0.80
	Toolkit (without defect types)	0.89	0.63	0.75	0.68
Apache	Ambari	0.60	0.78	0.68	0.61
	Ambari (without defect types)	0.71	0.33	0.45	0.60
	Hbase	0.86	0.78	0.81	0.83
	Hbase (without defect types)	0.85	0.78	0.81	0.82
	Hive	0.75	0.90	0.82	0.82
	Hive (without defect types)	0.69	0.94	0.79	0.78
	Spark	0.67	0.86	0.75	0.78
	Spark (without defect types)	0.58	0.76	0.65	0.69
Eclipse	Platform	0.73	0.67	0.70	0.72
	Platform (w/o defect types)	0.75	0.58	0.65	0.69
	JDT	0.76	0.87	0.81	0.83
	JDT (w/o defect types)	0.78	0.76	0.77	0.79
	PDE OS	0.71	0.51	0.59	0.66
	PDE OS (w/o defect types)	0.66	0.44	0.53	0.62
	Equinox	0.65	0.61	0.63	0.69
	Equinox (w/o defect types)	0.72	0.59	0.65	0.71

Furthermore, certain defect types might present fewer or more dependencies between files. Therefore, we predict the change impact in terms of the number of files changed in this RQ.

4.3.2 Approach. In this RQ, our goal is to predict the change impact in terms of the number of files changed. We follow the similar approach adopted in RQ2. We treat the problem as a binary classification. We assign 1 to the prediction output Y if the required change impact is large and 0 if small.

Dependent variable. The amount of files, *i.e., small* and *large*, needed to fix the defect is considered the output of the prediction. To obtain the dependent variable, we sort the issue reports by the number of changed files in ascending order. We assign the lower 10% issue reports a value of 0 to the change impact, meaning that it requires a small change impact. We assign to the upper 10% a value of 1.

Independent variables and prediction models. We use the same 25 independent variables (*i.e.,* textual metrics, characteristic metrics and topics of issue reports metrics) used in RQ2 as independent variables. We split the data into training and testing following the time-based approach. We train the eight machine learning models mentioned in the approach of RQ2, and we follow the same hyperparameter pipeline (*i.e.,* Random Search, Grid Search, and the 10-fold cross-validation) to attain the best model performance. To evaluate the models, we refer to the precision, recall, F1Score and AUC performance metrics.

Sensitivity Analysis. We run the best performant model, with different thresholds for the small-large binning of the number of files changed. We predict the change impact using three thresholds *25%*, *40%*, and *50%*.

4.3.3 Results. **It is possible to successfully predict the change impact in terms of the number of changed files by leveraging the issue reports information.** In table 13, we report the precision, recall, F1-Score, and AUC of the most performant model, XGboost. XGboost achieves an AUC of 0.71, 0.82, and 0.73 for Apache, Mozilla and Eclipse respectively.

Our approach is not sensitive to the threshold of the selection of the *small* and *large* the number of files changed. We observe in table 14 that the AUC drops when the upper and lower threshold increases from 25% to 50% by 7%, 5% and 6% for Mozilla, Apache and Eclipse, respectively.

Table 13. The performance of our prediction models for the considered datasets. The values shown represent the prediction of the issue reports that require a larger change impact in terms of the number of files changed. Prec. represents the precision.

	Ecosystem	Prec.	Recall	F-score	AUC
XGboost	Mozilla	0.92	0.71	0.80	0.82
	Apache	0.72	0.62	0.66	0.71
	Eclipse	0.84	0.74	0.79	0.73

Table 14. AUC performance of the three ecosystems with three different thresholds for the number of files changed.

Ecosystem	XGBoost-25%	XGBoost-40%	XGBoost-50%
Mozilla	0.75	0.71	0.68
Apache	0.66	0.64	0.61
Eclipse	0.67	0.64	0.61

Summary of RQ 3

It is feasible to predict the change impact of defects in terms of the number of files changed. XGboost achieves an AUC between 0.71 and 0.82 for the three ecosystems.

5 IMPLICATIONS

We discuss in this section the possible implications of our findings that could be useful to practitioners and researchers.

Leveraging the topics of issue reports. Identifying the topic of an issue report could help the researchers and practitioners in many ways:

- Researchers could benefit from the topics of issue reports to improve the defect fixing process. Some topics are easier to be fixed, and some others introduce challenges and require more developers' effort. For example, GUI defects might require manual investigations and validations. As shown in Table 11, for Eclipse, the *Is GUI* and *Is User experience* are among the top 5 influential metrics predicting the change impact. Thus, integrating the topics of issue reports in the process of the defect fixing could lead to more satisfying results.
- Practitioners could benefit from the topics of issue reports to improve the software project from a specific perspective (*i.e.*, defect topic). As shown in Table 5, for one specific system, certain topics might be more frequent

than others. Identifying the most frequent topics could guide developers in mitigating these frequent defects in the system and put in place test cases to detect these defects and hence improve the software.

- Practitioners could benefit from the topics of issue reports to improve the defect prioritization process. The relationship between the topics of issue reports and the severity level of the defect could help the developers recognize which topics should be treated with higher importance. For instance, as we observe in RQ1, *Server* or *Security* defects could possibly be given more priority than GUI defects in a software project belonging to the Web Server domain such as Apache.
- Practitioners could benefit from the topics of issue reports to improve the defect assignment process. Practitioners believe that defects belonging to the same topic tend to have similar solutions [135]. Therefore, practitioners could assign to developers defects that belong to the same topic which could consequently lead to a more efficient fixing process while avoiding the context switching overhead.

Leveraging the change impact. Predicting the change impact could help the researchers and practitioners in many ways :

- Researchers could benefit from the estimated *small* or *large* amount of change to improve the existing automated triage tools. Practitioners expressed their need for an automated triage tool that automatically assigns small defects, i.e., requiring *small* amount of change, and require manual intervention when the defect requires *large* amount of change [135]. Therefore, we encourage researchers to design an automated bug triage tool that incorporates both change impact prediction models, *i.e.*, code churn and the number of changed files, proposed in our work.
- Practitioners could benefit from the estimated change impact in terms of the number of files changed, to implement an efficient manual defect triage process. Practitioners could assign the defects that require a change in a large number of files (1) to developers that have knowledge of a large part of the codebase and its modules' dependency and (2) to developers that have access to the various codebases.
- Practitioners could benefit from the two change impact dimensions, *i.e.*, code churn and the number of files changed proposed in our work to design an efficient defect triage process that adeptly manages the limited human resources.

6 THREATS TO VALIDITY

Construct validity relates to a possible error in the data preparation. In RQ2, our results depend on the metrics extracted from the issue reports and the source code of the associated commits. First, we follow a widely used approach to map issue reports to commits [34, 56]. The used metrics have been extensively used in prior empirical software engineering research. Second, we estimate the change impact of defects in terms of code churn and the number of files changed. Although there are other ways of estimating the change impact (*e.g.*, the number of modules, operations, or classes changed), measuring the lines of code and the number of files has been considered a common way of estimating the amount of change [43, 75]. Therefore, we assert a strong construct validity.

To classify the issue reports as requiring small or large change impact, we select the lower 10% and upper 10% of the issue reports as requiring small and large change impact, respectively. While this hyperparameter might be dependent on the selected dataset, we demonstrate that our approach is not sensitive to the selected threshold

To automatically assign topics to issue reports, we adopt ETM, the state-of-the-art topic modelling. Dieng et al. [31] demonstrate that ETM, the state-of-the-art topic modelling, outperforms LDA and its variants in extracting topics in terms

of topic quality and predictive performance. LDA is one of the most used unsupervised topic modelling [26, 31, 40, 41]. Existing work in the realm of software engineering used LDA, and other variations of LDA to tackle research questions related to defect categorization [23, 27, 96], and bug report categorization [96, 133]. Therefore, we adopt ETM in our study.

Internal validity relates to the concerns that might come from the internal methods used in our study. The first concern comes from the manual assignment of topics to issue reports. To calculate the accuracy of ETM, we manually assign topics to a statistically representative sample of issue reports with a confidence level of 95% and a confidence interval of 10%, *i.e.,* 288 issue reports in total. To mitigate the possible human errors, we calculate Cohen's kappa agreement score between the two annotators, indicating a substantial agreement level. Although the annotators are not the owners of the issue reports, we highlight that the annotators have five years of work experience as software developers making them familiar with the reporting process and the topics of issue reports. The second concern is related to the textual free-text field of the issue reports, particularly the description. The description field might contain different noise types (*e.g.*, steps to reproduce, log execution and code fragment). To alleviate the noise, we manually check a sample of reports and ensure we exclude all unwanted patterns from the descriptions. Third, the performance of the predictive models might be influenced by the training and testing datasets. To alleviate this risk, we adopt the 10-cross-validation technique in addition to adopting a time-based splitting strategy."

External validity relates to the potential of generalizing our study results. To maximize the ability to generalize our results, we include in our study more than 290,000 issue reports belonging to three popular open-source ecosystems, *i.e.,* Mozilla, Apache and Eclipse. Although our approach is evaluated on open source projects, it can be easily applied to other projects as long as the issue report information can be scraped. Similarly, the predictive model can be applied to other projects as long as the metrics are extracted and preprocessed, and the identification of influential metrics on a model can be easily performed. In addition, our results are limited to the types of software projects selected within each ecosystem. To eliminate bias in the software projects' characteristics, we collect issue reports of 490 different software projects belonging to the Apache, Mozilla and Eclipse ecosystems. The analyzed software projects belong to diverse topics such as web servers, browsers, mail clients, IDE, and database.

7 RELATED WORK

In this section, we present the plethora of available literature closely related to our research and summarize the contributions in each area: defect taxonomies and classifications, defect fixing effort prediction, defect localization, and change impact analysis.

7.1 Classifying software defects.

Generic defect taxonomies. In an attempt to understand possible defect types, researchers suggest taxonomies targeting general defects but focusing on one specific dimension or several dimensions, *e.g.*, impact, category, root cause and life cycle phase. Several studies propose *fine-grained taxonomies* [44, 63, 81] that are defined based on code patterns that represent defects, *e.g.*, conditional statements and initialization errors. Ni et al. [81] exploit defect fixes from source code to an AST level and categorize defects into fine-grained root cause categories, such as conditional test errors and data verification errors. Some researchers propose to map code-related fine-grained categories to coarse-level dimensions. For example, Li et al. [63] manually mapped fine-grained code-related root causes to three coarse-level categories, *i.e.*, memory, concurrency and semantic.

The aforementioned approaches differ from our study in two ways: (1) present a *fine-grained taxonomy* for defects that is code-related, which is out of the scope of our study, and (2) when providing coarse-level matching for the fine-grained code-related root causes, they are limited to only a few categories, *i.e.*, memory, concurrency and semantic.

Other researchers introduce *coarse-level defect taxonomies*. Catolino et al. [23] perform an empirical study to find coarse-level defect categories. The authors manually analyze 1,280 defects and build a taxonomy of 9 root cause categories: configuration, GUI, Performance, Program anomaly, Test-code, Database, Network, Permission and Security issues. Catolino et al. then model an automated supervised classifier to label defects. Although the classifier achieves 64% as F-measure in general across all the defect categories, it fails to predict the defects for the configuration and network categories. Ahmed et al. [5] also present a framework, CapBug, which classifies defects according to the coarse-level categories. CapBug is similar to Catolino et al.'s approach in two ways. CapBug follows a supervised technique to categorize issue reports. It relies on manually predefined categories. Unlike Catolino's work, CapBug studies six defect categories, *i.e.,* Program anomaly, GUI, Network or Security, Configuration, Performance, and Test code. CapBug achieves 88% accuracy in predicting the category of the defects by using the Random Forest model and SMOTE technique to deal with class imbalance.

The aforementioned classification approaches provide automated techniques to classify the defects based on the *coarse-level categories*. However, these approaches suffer from the following disadvantages: (1) it requires an expensive amount of human effort to categorize the issue reports manually, and (2) the output and quality of the classifiers depend on the labelled training data and the predefined set of defect categories. In contrast, our proposed approach does not require any labelled data.

Specific defect taxonomies. A rich collection of research work focuses on a specific programming language [3, 28, 39] or defects discovered in a specific system [49, 50, 83–85, 93, 105, 109, 130, 131] or even a specific class of defects[4, 33, 101, 106, 111, 125]. For example, Ciborowska et al. [28] shed light on the differences in the characteristics and localization of the defect across COBOL and non-COBOL software. To achieve this, Ciborowska et al. refine the taxonomy introduced by Catolino et al. [23] by considering the opinion of surveyed developers.

Some researchers have examined the types of defects related to specific systems, such as AI-based systems [49, 50, 105, 131]. Others amend existing taxonomies to classify AI defects. For example, Thung et al. [105] study the characteristics of 500 defects belonging to three machine learning systems (Apache Mahout, Apache Lucene, and Apache OpenNLP). Thung et al. manually categorize defects leveraging the taxonomy designed by Carolyn et al. [93] and include an additional category, 'configuration'. Other researchers design new taxonomies. For example, Zhang et al. [131] propose a new taxonomy for TensorFlow defects by analyzing 175 issue reports.

Researchers also investigate specific classes of defects. For instance, the research community shed light on understanding the characteristics of performance defects [101, 125, 132] since they lead to user dissatisfaction. For example, Sanchez et al. [101] propose a taxonomy of three dimensions, *i.e.,* effects, causes and contexts of defects for real-world performance defects. In an effort to support practitioners build more secure software, researchers propose taxonomies for security defects [4, 106, 125].

The above proposed taxonomies represent only a subset of the defects that can occur in software systems, and therefore present limited capabilities in supporting developers in understanding other defect types. Our study includes all types of defects and is not limited to a specific defect class.

7.2 Defect fixing effort

Defect fixing time. Several studies leveraged the issue report information (*i.e.,* component, priority and severity) to predict the defect fixing effort in terms of time required for fixing defects [9, 10, 37, 46].

Giger et al. [37] carry out an empirical study to investigate the possibility of predicting the defect fixing time from the issue report attributes. In their study, Giger et al. aim to classify the issue reports into two classes: requiring *Slow* or *Fast* fixing effort. The authors build a decision trees model that achieves a precision of 0.654. Giger et al. find that assignee, reporter and monthOpened are the top 3 issue report attributes influencing the predicting fixing effort time. Similarly, Ardimento and Mele [10] also treat the fixing time prediction as a binary classification problem (*i.e.,* Slow and Fast classes). The authors treat the problem as a supervised text categorization task but used BERT [30] with a classifier to predict the fixing time. In their approach, the authors propose a new set of features, including the description of the issue and developers' comments on which they perform transfer learning.

In contrast to the above, some researchers treat the fixing effort prediction as a regression problem. Yuan et al. [124] propose BuFTNN, a novel neural network model that leverages several features (*i.e.,* developers' activities, developers' sentiments, the semantics of bugs, and efforts caused by understanding and analyzing source code) to predict a defect's fixing time. The proposed model is validated on four real-life projects, including Eclipse, and outperforms the state-of-the-art, DeepLSTMPred [94], effort prediction model by 7.3% in F1-score, on average.

Another line of work handles the fixing effort prediction problem as information retrieval. For instance, the time required to fix a defect is anticipated by querying textually similar issue reports marked as fixed. The fixing time of the new issue report is then estimated by relying on the fixing time of similar retrieved reports. Weiss et al. [112] rely on the k-Nearest Neighbors algorithm combined with textual similarity to identify similar defects. Zhang et al. [128] follow a similar approach to Weiss et al.; however, they focus on commercial projects.

In the above studies, the fixing time is calculated as the time elapsed between the issue being reported/assigned and resolved. However, the estimated time might not indicate the actual fixing time. In some cases, developers might be working on several tasks or working part-time. In other instances, the issue report status might not be updated on time. Different from the above studies that predict the fixing effort in terms of time, we measure the change impact, i.e., code churn size and the number of files changed, as a proxy for the fixing effort. Our study complements the above approaches.

Code churn size. Measuring lines of code has been considered a standard way of estimating the developer effort [18]. Nevertheless, the possibility of predicting the defect fixing effort in terms of code churn (*i.e.,* number of lines of code modified) remains scarcely explored. To the best of our knowledge, Thung's study [104] is the only existing work that predicts defect fixing effort by considering the code churn size. Thung designs a supervised machine learning approach to classify issue reports into *low* and *high* categories. The author fed the summary and description of the issue report as input to a Support Vector Machine model. Thung evaluates the model on 1,029 bugs from *hadoop-common* and *struts2* and achieves an AUC of 0.61. Although both Thung's work and ours aim to predict the fixing effort in terms of code churn size, our study differs in several dimensions. First, we utilize a different set of features as independent variables. For instance, we use 25 metrics, of which 10 are extracted from the issue report, and 15 are synthesized based on the identified issue topic. Second, we implement eight different machine learning models and use a large dataset of 298,548 issue reports as opposed to 1,029 in the case of Thung's work. In addition, we also experiment with several thresholds for binning the issue reports into *small* and *large* change impact categories, whereas, in Thung's work, the chosen classification threshold is not justified.

7.3 Defect localization

Defect localization is one of the crucial yet most costly and time-consuming steps of software maintenance [32]. There are three widely used families of defect localization techniques: Spectrum-based (SBBL), mutation-based (MBBL) and Information retrieval-based (IRBL). SBBL techniques [21, 113, 129] employ a series of test cases to identify the buggy code spectra. SBBL is considered a fine-grained defect localization as it is capable of pointing the buggy statement. MBBL techniques [24, 45, 118] employ test results after mutating the program to determine the buggy code. IRBL techniques employ information from the issue report to localize the defect. IRBL techniques focus on the textual similarity between the issue report description and the source code [115]. In IRBL techniques, program entities of different granularity might be identified as faulty components. Some research work [122] points to the files containing the defect. For instance, Chen et al. [25] propose Pathidea, an information retrieval defect localization approach that relies on both log snippets and stack traces in issue reports to localize the defect file. Other research work [14, 77, 114, 115] focuses on identifying the commit that caused the defect. One team at Facebook proposes Bug2Commit [77], an IRBL approach that leverages unsupervised techniques to find commit-level defects. Both IRBL localization approaches and our work leverage the issue reports information to support the developers with the software maintenance task. Our research work complements the defect localization line of work. The defect localization approaches seek to identify the elements of a program causing the failure with different granularity, i.e., statement, method and file. However, our work focuses on predicting the change impact, i.e., the amount of change, in terms of lines of code and the number of files changed.

7.4 Change impact analysis

Software Change Impact Analysis (CIA) represents a collection of techniques that help developers identify the effects of a change or to estimate the amount of change needed [19] during the software maintenance and evolution phase. Depending on the technique and its application, CIA can be helpful before implementing the change (*e.g.,* predicting the change impact) or after implementing it (*e.g.,* performing change propagation) [61]. Researchers define metrics and prediction methods to conduct CIA along four main CIA quantification parameters: (1) instability (*i.e.,* likelihood that various software artifacts change simultaneously), (2) amount of change (*i.e.,* the size of changes affecting a software artifact), (3) change proneness (*i.e.,* likelihood that a software artifact will change due to bug fixes or requirement changes), (4) and changeability (*i.e.,* the level of ease related to the affecting the changes to a software artifact) [57]. The existing research related to the amount of change is the closest to our work. Existing work quantifies the amount of change in terms of code churn [43, 75], the number of changed artifacts (*i.e.,* modules, operations, members, classes, or files) [97] and incremental changes [7]. Similar to our goal, some existing work estimates the change set, *i.e.,* the amount of change, using static analysis [91, 99], dynamic analysis [20, 47] or Mining Software Repositories (MSR) [36, 126]. MSR techniques leverage historical information from the defect artifacts and source code repository to predict the potential code change. Gethers et al. [36] leverage the issue report textual information and the source code to estimate the amount of change. The authors adopt information retrieval methods to couple the description of the issue report to the potential software entity, *e.g.,* method. Zanjani et al. [126] couple information retrieval, machine learning and source code analysis to build a corpus of source code entities queried when an issue report description is submitted. There are key differences between our work and the aforementioned change impact related research. First, the existing change impact research work estimates the amount of change after identifying the change set by considering the source code. In our work, we focus on a simple approach that predicts the amount of change by only leveraging information from issue reports and without having to identify first the actual change set in the source code. Second, we predict

the amount of change in terms of code churn and the number of files changed using a new set of features extracted from the issue reports instead of only relying on the textual description. In addition to extracting 10 metrics from the issue reports, we synthesize 15 new collective features, *i.e.,* the topics of issue reports, which we extract through topic modelling and include in the predictive model. Furthermore, we investigate the most influential metrics for predicting the change impact.

8 CONCLUSION

With the prevalence of large-scale and complex software systems, it is crucial to support developers in defect-fixing related activities as it is estimated that at least 50% of the developer's time is spent on debugging and fixing defects [1, 2]. Fixing a defect requires the developer to assign the defects, which entails manual work. In this paper, we aim to support the practitioners by helping them identify the topics of issue reports and leveraging the topics in predicting the change impact, *i.e.,* small or large amount of change, required to fix the defects.

Firstly, we automatically assign topics (*i.e.,* performance, UI, database) to issue reports using the state-of-the-art topic modelling technique ETM. We apply our approach to 298,548 issue reports from Mozilla, Apache and Eclipse. Our approach successfully identifies 15 topics of issue reports and achieves an accuracy of 79%, on average. Secondly, we train 8 models to predict the change impact in terms of code churn size and the number of files changed leveraging 25 metrics. XGBoost achieves an AUC of 73–84%. We find that the topics of issue reports improve the recall of the predictive model by up to 45% per software project and the AUC by 5% on average across the ecosystems. Finally, we measure the metrics' power in predicting the amount of change size. We find that the number of attachments and comments per developer is among the top 3 influential metrics across the three ecosystems. To make the most of the change impact predictive model, we suggest that 1) the developers attach relevant documents to the issue reports, 2) list the issues that are blocked by the defect in question and 3) indicate the accurate severity of the defect.

In the future, we aim to include more metrics in our predictive model. Furthermore, we aim to explore how we can incorporate the predicted amount of change into existing defect assignment tools.

ACKNOWLEDGMENTS

The first author would like to acknowledge the support of NSERC Vanier Canada Graduate Scholarships.

REFERENCES

[1] [n. d.]. *Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year.* Accessed: 2022-10-18.

[2] [n. d.]. *This is what your developers are doing 75% of the time, and this is the cost you pay.* Accessed: 2021-06-17.

[3] Nurul Haszeli Ahmad, Syed Ahmad Aljunid, and Jamalul-lail Ab Manan. 2011. Taxonomy of C Overflow Vulnerabilities Attack. In *Software Engineering and Computer Systems*, Jasni Mohamad Zain, Wan Maseri bt Wan Mohd, and Eyas El-Qawasmeh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 376–390.

[4] Nurul Haszeli Ahmad, Syed Ahmad Aljunid, and Jamalul-lail Ab Manan. 2011. Taxonomy of C Overflow Vulnerabilities Attack. In *Software Engineering and Computer Systems*, Jasni Mohamad Zain, Wan Maseri bt Wan Mohd, and Eyas El-Qawasmeh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 376–390.

[5] Hafiza Anisa Ahmed, Narmeen Zakaria Bawany, and Jawwad Ahmed Shamsi. 2021. CaPBug-A Framework for Automatic Bug Categorization and Prioritization Using NLP and Machine Learning Algorithms. *IEEE Access* 9 (2021), 50496–50512. https://doi.org/10.1109/ACCESS.2021.3069248

[6] Shirin Akbarinasaji. 2018. Prioritizing Lingering Bugs. *SIGSOFT Softw. Eng. Notes* 43, 1 (mar 2018), 1–6. https://doi.org/10.1145/3178315.3178326

[7] Theodoros Amanatidis and Alexander Chatzigeorgiou. 2016. Studying the Evolution of PHP Web Applications. *Inf. Softw. Technol.* 72, C (apr 2016), 48–67. https://doi.org/10.1016/j.infsof.2015.11.009

[8] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who Should Fix This Bug?. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 361–370. https://doi.org/10.1145/1134285.1134336

[9] Pasquale Ardimento and Andrea Dinapoli. 2017. Knowledge Extraction from On-Line Open Source Bug Tracking Systems to Predict Bug-Fixing Time. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics* (Amantea, Italy) *(WIMS '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 9 pages. https://doi.org/10.1145/3102254.3102275

[10] Pasquale Ardimento and Costantino Mele. 2020. Using BERT to Predict Bug-Fixing Time. In *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*. 1–7. https://doi.org/10.1109/EAIS48028.2020.9122781

[11] Maram Assi, Safwat Hassan, Yuan Tian, and Ying Zou. 2021. FeatCompare: Feature comparison for competing mobile apps leveraging user reviews. *Empirical Software Engineering* 26 (07 2021).

[12] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A Neural Probabilistic Language Model. *J. Mach. Learn. Res.* 3, null (mar 2003), 1137–1155.

[13] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13, null (feb 2012), 281–305.

[14] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in Large-Scale Services. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 493–509.

[15] Pamela Bhattacharya and Iulian Neamtiu. 2011. Bug-Fix Time Prediction Models: Can We Do Better?. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (Waikiki, Honolulu, HI, USA) *(MSR '11)*. Association for Computing Machinery, New York, NY, USA, 207–210. https://doi.org/10.1145/1985441.1985472

[16] Steven Bird and Edward Loper. 2004. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions* (Barcelona, Spain) *(ACLdemo '04)*. Association for Computational Linguistics, USA, 31–es. https://doi.org/10.3115/1219044.1219075

[17] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3, null (mar 2003), 993–1022.

[18] Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. 2009. *Software Cost Estimation with COCOMO II* (1st ed.). Prentice Hall Press, USA.

[19] Arnold Robert S Bohner, Shawn. 1996. *Software change impact analysis / Shawn A. Bohner, Robert S. Arnold.* Los Alamitos, Calif. : IEEE Computer Society Press.

[20] Haipeng Cai and Douglas Thain. 2016. DistIA: A Cost-Effective Dynamic Impact Analysis for Distributed Programs *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 344–355. https://doi.org/10.1145/2970276.2970352

[21] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 378–381. https://doi.org/10.1145/2351676.2351752

[22] Laura V. Galvis Carreño and Kristina Winbladh. 2013. Analysis of user comments: An approach for software requirements evolution. In *2013 35th International Conference on Software Engineering (ICSE)*. 582–591. https://doi.org/10.1109/ICSE.2013.6606604

[23] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181. https://doi.org/10.1016/j.jss.2019.03.002

[24] N. B. Chaleshtari and S. Parsa. 2020. SMBFL: slice-based cost reduction of mutation-based fault localization. *Empirical Software Engineering* 25 (2020), 4282–4314.

[25] An Ran Chen, Tse-Hsun Peter Chen, and Shaowei Wang. 2021. Pathidea: Improving Information Retrieval-Based Bug Localization by Re-Constructing Execution Paths Using Logs. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3071473

[26] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. 2016. A Survey on the Use of Topic Models When Mining Software Repositories. *Empirical Softw. Engg.* 21, 5 (oct 2016), 1843–1919. https://doi.org/10.1007/s10664-015-9402-8

[27] Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, and Ahmed E. Hassan. 2012. Explaining Software Defects Using Topic Models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (Zurich, Switzerland) *(MSR '12)*. IEEE Press, 189–198.

[28] Agnieszka Ciborowska, Aleksandar Chakarov, and Rahul Pandita. 2021. Contemporary COBOL: Developers' Perspectives on Defects and Defect Location. arXiv:2105.01830 [cs.SE]

[29] Maëlick Claes and Mika V. Mäntylä. 2020. 20-MAD: 20 Years of Issues and Commits of Mozilla and Apache Development. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 503–507. https://doi.org/10.1145/3379597.3387487

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[31] Adji Dieng, Francisco Ruiz, and David Blei. 2020. Topic Modeling in Embedding Spaces. *Transactions of the Association for Computational Linguistics* 8 (07 2020), 439–453. https://doi.org/10.1162/tacl_a_00325

[32] Nicholas DiGiuseppe and James Jones. 2014. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering* 20 (08 2014). https://doi.org/10.1007/s10664-014-9304-1

[33] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (2015), 82–101. https://doi.org/10.1016/j.jss.2015.04.066

[34] Osama Ehsan, Lillane Barbour, Foutse Khomh, and Ying Zou. 2021. *Is Late Propagation a Harmful Code Clone Evolutionary Pattern? An Empirical Study.* Springer Singapore, Singapore, 151–167.

[35] Omar El Zarif, Daniel Alencar Da Costa, Safwat Hassan, and Ying Zou. 2020. On the Relationship between User Churn and Software Issues *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 339–349. https://doi.org/10.1145/3379597.3387456

[36] Malcom Gethers, Huzefa Kagdi, Bogdan Dit, and Denys Poshyvanyk. 2011. An adaptive approach to impact analysis from change requests to source code. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, 540–543. https://doi.org/10.1109/ASE.2011.6100120

[37] Emanuel Giger, Martin Pinzger, and Harald Gall. 2010. Predicting the Fix Time of Bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering* (Cape Town, South Africa) *(RSSE '10)*. Association for Computing Machinery, New York, NY, USA, 52–56. https://doi.org/10.1145/1808920.1808933

[38] Florin Gorunescu. 2011. *Data Mining: Concepts, models and techniques*.

[39] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2021. BUGSJS: a benchmark and taxonomy of JavaScript bugs. *Software Testing, Verification and Reliability* 31, 4 (2021), e1751. e1751 stvr.1751.

[40] Ruidan He, Wee Sun Lee, Hwee Tou Ng, and Daniel Dahlmeier. 2017. An Unsupervised Neural Attention Model for Aspect Extraction. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 388–397. https://doi.org/10.18653/v1/P17-1036

[41] Tobias Hecking and Loet Leydesdorff. 2018. Topic Modelling of Empirical Text Corpora: Validity, Reliability, and Reproducibility in Comparison to Semantic Maps. *ArXiv* abs/1806.01045 (2018).

[42] I. Herraiz, G. Robles, J.M. Gonzalez-Barahona, A. Capiluppi, and J.F. Ramil. 2006. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *Conference on Software Maintenance and Reengineering (CSMR'06)*. 8 pp.–213. https://doi.org/10.1109/CSMR.2006.17

[43] Abram Hindle. 2015. Green Mining: A Methodology of Relating Software Change and Configuration to Power Consumption. *Empirical Softw. Engg.* 20, 2 (apr 2015), 374–409. https://doi.org/10.1007/s10664-013-9276-6

[44] Thomas Hirsch and Birgit Hofer. 2020. Root cause prediction based on bug reports. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*. IEEE, 171–176. https://doi.org/10.1109/ISSREW51248.2020.00067

[45] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-Based Fault Localization for Real-World Multilingual Programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 464–475. https://doi.org/10.1109/ASE.2015.14

[46] Pieter Hooimeijer and Westley Weimer. 2007. Modeling Bug Report Quality. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/1321631.1321639

[47] Lulu Huang and Yeong-Tae Song. 2007. Precise Dynamic Impact Analysis with Dependency Analysis for Object-oriented Programs. In *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*. 374–384. https://doi.org/10.1109/SERA.2007.109

[48] Yonghui Huang, Daniel Alencar Costa, Feng Zhang, and Ying Zou. 2019. An Empirical Study on the Issue Reports with Questions Raised during the Issue Resolving Process. *Empirical Softw. Engg.* 24, 2 (apr 2019), 718–750. https://doi.org/10.1007/s10664-018-9636-3

[49] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[50] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[51] Hadi Jahanshahi and Mucahit Cevik. 2022. S-DABT: Schedule and Dependency-aware Bug Triage in open-source bug tracking systems. *Information and Software Technology* 151 (2022), 107025. https://doi.org/10.1016/j.infsof.2022.107025

[52] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving Bug Triage with Bug Tossing Graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/1595696.1595715

[53] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E. Hassan. 2021. The Impact of Correlated Metrics on the Interpretation of Defect Models. *IEEE Transactions on Software Engineering* 47, 2 (2021), 320–331. https://doi.org/10.1109/TSE.2019.2891758

[54] Nilam Kaushik, Mehdi Amoui, Ladan Tahvildari, Weining Liu, and Shimin Li. 2013. Defect Prioritization in the Software Industry: Challenges and Opportunities. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 70–73. https://doi.org/10.1109/ICST.2013.40

[55] Guolin Ke, Jia Zhang, Zhenhui Xu, Jiang Bian, and Tie-Yan Liu. 2018. TabNN: A Universal Neural Network Solution for Tabular Data.

[56] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential Biases in Bug Localization: Do They Matter?. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 803–814. https://doi.org/10.1145/2642937.2642997

[57] Maria Kretsou, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Ignatios Deligiannis, and Vassilis C. Gerogiannis. 2021. Change impact analysis: A systematic mapping study. *Journal of Systems and Software* 174 (2021), 110892. https://doi.org/10.1016/j.jss.2020.110892

[58] William H. Kruskal and Wilson Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47 (1952), 583–621.

[59] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 579–590. https://doi.org/10.1145/2786805.2786880

[60] Stanislav Levin and Amiram Yehudai. 2017. The Co-evolution of Test Maintenance and Code Maintenance through the Lens of Fine-Grained Semantic Changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 35–46. https://doi.org/10.1109/ICSME.2017.9

[61] Bixin Li, Xiaobing Sun, Hareton K. N. Leung, and Sai Zhang. 2013. A survey of code-based change impact analysis techniques. *Software Testing* 23 (2013).

[62] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. 2018. Unsupervised Deep Bug Report Summarization. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 144–155.

[63] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, California) *(ASID '06)*. Association for Computing Machinery, New York, NY, USA, 25–33. https://doi.org/10.1145/1181309.1181314

[64] Bo Lili, Zhu Xuanrui, Sun Xiaobing, Ni Zhen, and Li Bin. 2021. Are Similar Bugs Fixed with Similar Change Operations? An Empirical Study. *Chinese Journal of Electronics* 30 (01 2021), 55–63. https://doi.org/10.1049/cje.2020.10.010

[65] Charles X. Ling, Jin Huang, and Harry Zhang. 2003. AUC: A Better Measure than Accuracy in Comparing Learning Algorithms. In *Advances in Artificial Intelligence*, Yang Xiang and Brahim Chaib-draa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–341.

[66] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 11–19. https://doi.org/10.1109/ESEM.2017.8

[67] Yasitha Liyanage, Mengfan Yao, Christopher Yong, Daphney-Stavroula Zois, and Charalampos Chelmis. 2018. WHAT MATTERS THE MOST? OPTIMAL QUICK CLASSIFICATION OF URBAN ISSUE REPORTS BY IMPORTANCE. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 106–110. https://doi.org/10.1109/GlobalSIP.2018.8646639

[68] Ruchika Malhotra and Madhukar Cherukuri. 2020. Software Defect Categorization based on Maintenance Effort and Change Impact using Multinomial Naïve Bayes Algorithm. In *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 1068–1073. https://doi.org/10.1109/ICRITO48877.2020.9198037

[69] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press. https://doi.org/10.1017/CBO9780511809071

[70] Mary McHugh. 2012. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB* 22 (10 2012), 276–82. https://doi.org/10.11613/BM.2012.031

[71] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Softw. Engg.* 21, 5 (oct 2016), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

[72] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., Red Hook, NY, USA, 3111–3119.

[73] David Mimno, Hanna M. Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. 2011. Optimizing Semantic Coherence in Topic Models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (Edinburgh, United Kingdom) *(EMNLP '11)*. Association for Computational Linguistics, USA, 262–272.

[74] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. 2016. Studying high impact fix-inducing changes. *Empirical Software Engineering* 21, 2 (2016), 605–641. https://doi.org/10.1007/s10664-015-9370-z An erratum to this article can be found at http://dx.doi.org/10.1007/s10664-016-9455-3 ..

[75] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. 2016. Studying high impact fix-inducing changes. *Empirical Software Engineering* 21, 2 (2016), 605–641. https://doi.org/10.1007/s10664-015-9370-z An erratum to this article can be found at http://dx.doi.org/10.1007/s10664-016-9455-3 ..

[76] Shuji Morisaki, Akito Monden, Tomoko Matsumura, Haruaki Tamada, and Ken-ichi Matsumoto. 2007. Defect Data Analysis Based on Extended Association Rule Mining. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 3–3. https://doi.org/10.1109/MSR.2007.5

[77] V. Murali, Lee Gross, R. Qian, and S. Chandra. 2021. Industry-Scale IR-Based Bug Localization: A Perspective from Facebook. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), 188–197.

[78] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 284–292. https://doi.org/10.1109/ICSE.2005.1553571

[79] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. 2013. Bug Report Assignee Recommendation Using Activity Profiles *(MSR '13)*. IEEE Press, 22–30.

[80] Jaechang Nam and Sunghun Kim. 2015. CLAMI: Defect Prediction on Unlabeled Datasets (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 452–463. https://doi.org/10.1109/ASE.2015.56

[81] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. 2020. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software* 163 (2020), 110538. https://doi.org/10.1016/j.jss.2020.110538

[82] Lucas D. Panjer. 2007. Predicting Eclipse Bug Lifetimes. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 29–29. https://doi.org/10.1109/MSR.2007.25

[83] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. How Is Video Game Development Different from Software Development in Open Source?. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 392–402.

[84] A. Rahman and Effat Farhana. 2020. An Exploratory Characterization of Bugs in COVID-19 Software Projects. *ArXiv* abs/2006.00586 (2020).

[85] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3377811.3380409

[86] Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo Ansaldi, Yasutaka Kamei, and Ahmed E. Hassan. 2021. The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3056941

[87] Christian Robottom Reis, Renata Pontin de Mattos Fortes, Renata Pontin, and Mattos Fortes. 2002. An Overview of the Software Engineering Process and Tools in the Mozilla Project.

[88] Ripon K. Saha, Sarfraz Khurshid, and Dewayne E. Perry. 2015. Understanding the triaging and fixing processes of long lived bugs. *Information and Software Technology* 65 (2015), 114–128. https://doi.org/10.1016/j.infsof.2015.03.002

[89] Munish Saini and Kuljit Kaur Chahal. 2018. Change Profile Analysis of Open-Source Software Systems to Understand Their Evolutionary Behavior. *Front. Comput. Sci.* 12, 6 (dec 2018), 1105–1124. https://doi.org/10.1007/s11704-016-6301-0

[90] Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage.

[91] Lajos Schrettner, Judit Jász, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. 2014. Impact analysis in the presence of dependence clusters using Static Execute After in WebKit. *Journal of Software: Evolution and Process* 26, 6 (2014), 569–588. https://doi.org/10.1002/smr.1614 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1614

[92] C.B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[93] Carolyn Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect Categorization: Making Use of a Decade of Widely Varying Historical Data. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 149–157. https://doi.org/10.1145/1414004.1414030

[94] Reza Sepahvand, Reza Akbari, and Sattar Hashemi. 2020. Predicting the bug fixing time using word embedding and deep long short term memories. *IET Software* 14, 3 (2020), 203–212. https://doi.org/10.1049/iet-sen.2019.0260

[95] Leif Singer and Kurt Schneider. 2012. It was a bit of a race: Gamification of version control. In *2012 Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques (GAS)*. 5–8. https://doi.org/10.1109/GAS.2012.6225927

[96] Kalyanasundaram Somasundaram and Gail C. Murphy. 2012. Automatic Categorization of Bug Reports Using Latent Dirichlet Allocation. In *Proceedings of the 5th India Software Engineering Conference* (Kanpur, India) *(ISEC '12)*. Association for Computing Machinery, New York, NY, USA, 125–130. https://doi.org/10.1145/2134254.2134276

[97] Srdjan Stevanetic and Uwe Zdun. 2018. Supporting the Analyzability of Architectural Component Models - Empirical Findings and Tool Support. *Empirical Softw. Engg.* 23, 6 (dec 2018), 3578–3625. https://doi.org/10.1007/s10664-017-9583-4

[98] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 45–54. https://doi.org/10.1145/1806799.1806811

[99] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. 2010. Change Impact Analysis Based on a Taxonomy of Change Types. In *2010 IEEE 34th Annual Computer Software and Applications Conference*. 373–382. https://doi.org/10.1109/COMPSAC.2010.45

[100] Shaheen Syed and Marco Spruit. 2017. Full-Text or Abstract? Examining Topic Coherence Scores Using Latent Dirichlet Allocation. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 165–174. https://doi.org/10.1109/DSAA.2017.61

[101] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2020. TANDEM: A Taxonomy and a Dataset of Real-World Performance Bugs. *IEEE Access* 8 (2020), 107214–107228. https://doi.org/10.1109/ACCESS.2020.3000928

[102] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. 1 (2017).

[103] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2018. The Impact of Automated Parameter Optimization for Defect Prediction Models. (2018).

[104] Ferdian Thung. 2016. Automatic Prediction of Bug Fixing Effort Measured by Code Churn Size. In *Proceedings of the 5th International Workshop on Software Mining* (Singapore, Singapore) *(SoftwareMining 2016)*. Association for Computing Machinery, New York, NY, USA, 18–23. https://doi.org/10.1145/2975961.2975964

[105] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (11 2012). https://doi.org/10.1109/ISSRE.2012.22

[106] K. Tsipenyuk, B. Chess, and G. McGraw. 2005. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy* 3, 6 (2005), 81–84. https://doi.org/10.1109/MSP.2005.159

[107] Harold Valdivia-Garcia, Emad Shihab, and Meiyappan Nagappan. 2018. Characterizing and predicting blocking bugs in open source projects. *Journal of Systems and Software* 143 (2018), 44–58. https://doi.org/10.1016/j.jss.2018.03.053

[108] Renan Vieira, Antônio da Silva, Lincoln Rocha, and João Paulo Gomes. 2019. From Reports to Bug-Fix Commits: A 10 Years Dataset of Bug-Fixing Activity from 55 Apache's Open Source Projects *(PROMISE'19)*. Association for Computing Machinery, New York, NY, USA, 80–89. https://doi.org/10.1145/3345629.3345639

[109] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 413–424. https://doi.org/10.1109/MSR.2017.59

[110] Jing Wang, Patrick Shih, Yu Wu, and John Carroll. 2015. Comparative case studies of open source software peer review practices. *Information and Software Technology* 67 (11 2015), 1–12. https://doi.org/10.1016/j.infsof.2015.06.002

[111] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. 2020. An Empirical Study on Regular Expression Bugs. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 103–113. https://doi.org/10.1145/3379597.3387464

[112] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How Long Will It Take to Fix This Bug?. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 1–1. https://doi.org/10.1109/MSR.2007.13

[113] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and S.C. Cheung. 2019. Historical Spectrum based Fault Localization. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2948158

[114] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 262–273.

[115] Rongxin wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: locate crash-inducing changes based on crash reports. *ICSE '18: Proceedings of the 40th International Conference on Software Engineering*, 536–536. https://doi.org/10.1145/3180155.3182516

[116] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. 2017. Improving Automated Bug Triaging with Specialized Topic Model. *IEEE Transactions on Software Engineering* 43, 3 (2017), 272–297. https://doi.org/10.1109/TSE.2016.2576454

[117] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. 2012. DRETOM: Developer Recommendation Based on Topic Models for Bug Resolution *(PROMISE '12)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/2365324.2365329

[118] Tongtong Xu. 2019. Improving Automated Program Repair with Retrospective Fault Localization. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 159–161. https://doi.org/10.1109/ICSE-Companion.2019.00066

[119] Jifeng Xuan, He Jiang, Zhilei Ren, Jun Yan, and Zhongxuan Luo. 2017. Automatic Bug Triage using Semi-Supervised Text Classification. *CoRR* abs/1704.04769 (2017). arXiv:1704.04769 http://arxiv.org/abs/1704.04769

[120] Meng Yan, Xin Xia, D. Lo, A. Hassan, and Shanping Li. 2019. Characterizing and identifying reverted commits. *Empirical Software Engineering* (2019), 1–38.

[121] Geunseok Yang, Tao Zhang, and Byungjeong Lee. 2014. Towards Semi-automatic Bug Triage and Severity Prediction Based on Topic Model and Multi-feature of Bug Reports. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. 97–106. https://doi.org/10.1109/COMPSAC.2014.16

[122] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 689–699. https://doi.org/10.1145/2635868.2635874

[123] Soner Yigit and Mehmet Mendes. 2018. Which Effect Size Measure is Appropriate for One-Way and Two-Way ANOVA Models? : A Monte Carlo Simulation Study. *REVSTAT-Statistical Journal* 16, 3 (Jul. 2018), 295–313. https://doi.org/10.57805/revstat.v16i3.244

[124] Wei Yuan, Yuan Xiong, Hailong Sun, and Xudong Liu. 2021. Incorporating Multiple Features to Predict Bug Fixing Time with Neural Networks. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 93–103. https://doi.org/10.1109/ICSME52107.2021.00015

[125] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (Zurich, Switzerland) *(MSR '12)*. IEEE Press, 199–208.

[126] Motahareh Bahrami Zanjani, George Swartzendruber, and Huzefa Kagdi. 2014. Impact Analysis of Change Requests on Source Code Based on Interaction and Commit Histories. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 162–171. https://doi.org/10.1145/2597073.2597096

[127] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2012. An Empirical Study on Factors Impacting Bug Fixing Time. In *2012 19th Working Conference on Reverse Engineering*. 225–234. https://doi.org/10.1109/WCRE.2012.32

[128] Hongyu Zhang, Liang Gong, and Steve Versteeg. 2013. Predicting bug-fixing time: An empirical study of commercial software projects. In *2013 35th International Conference on Software Engineering (ICSE)*. 1042–1051. https://doi.org/10.1109/ICSE.2013.6606654

[129] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting Spectrum-Based Fault Localization Using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/3092703.3092731

[130] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2020. A Framework and DataSet for Bugs in Ethereum Smart Contracts. 139–150. https://doi.org/10.1109/ICSME46990.2020.00023

[131] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[132] Guoliang Zhao, Safwat Hassan, Ying Zou, Derek Truong, and Toby Corbin. 2021. Predicting Performance Anomalies in Software Systems at Run-Time. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 33 (apr 2021), 33 pages. https://doi.org/10.1145/3440757

[133] Minhaz F. Zibran. 2016. On the Effectiveness of Labeled Latent Dirichlet Allocation in Automatic Bug-Report Categorization. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 713–715. https://doi.org/10.1145/2889160.2892646

[134] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643. https://doi.org/10.1109/TSE.2010.63

[135] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2020. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Transactions on Software Engineering* 46, 8 (2020), 836–862. https://doi.org/10.1109/TSE.2018.2870414