Structured Information Retrieval Strategies for Localising Software Changes

Qiuchi Li*, Yijun Yu†, Dawei Song† and Bashar Nuseibeh†‡
* Department of Information Engineering, University of Padua, Padua, Italy
† School of Computing and Communications, The Open University, Milton Keynes, UK
‡ Lero - The Irish Software Research Centre, University of Limerick, Limerick, Ireland

Abstract—During software maintenance and evolution, developers need to deal with a large number of change requests by modifying existing code or adding code into the system. An efficient tackling of change request calls for an accurate localising of software changes, i.e. identifying which code are problematic and where new files should be added for any type of change request at hand, such as a bug report or a feature request. Existing automatic techniques for this change localisation problem are limited in two aspects: on the one hand, they are only limited to tackle a specific type of change request; on the other hand, they are focused on finding files that should be modified for a change request, yet barely capable of recommending what files or packages might be newly created. To address the limitations, we are inspired to propose a generalised change localisation approach to identify the to-be-modified files (mostly for bugs), and at the same time point out where new files or packages should be created (mostly for new feature requests) for an arbitrary type of change request. In order to tackle the key challenge of predicting to-be-created program elements, our proposed SeekChanges approach leverages the hierarchical package structure for Java projects, and model the change localisation problem as a structured information retrieval (IR) task. A systematic investigation of three structured IR strategies is carried out for scoring and ranking both the files that should be modified and the software packages in which the new files should be created to address change requests. Extensive experiments on four open source Java projects from the Apache Software Foundation demonstrate that structured IR strategies have a good performance on recommending newly created files, while the overall performance of localising change requests is equally satisfactory.

I. INTRODUCTION

Software changes are common place in incremental and evolutionary development. During maintenance and evolution, stakeholders often request different types of changes (e.g., fixing a bug and adding a feature), while developers aim to find solutions to accommodate such requests [17]. For large and active software projects, change requests appear frequently. In order to increase efficiency and productivity, developers need to process a change request quickly and accurately. However, even the most experienced developers will often find it difficult to locate the target code and apply changes onto it within a short period of time. As a result, it is becoming increasingly important for automatic approaches to identify where to apply changes to the software in response to these change requests. If the location of changes could be pinpointed quickly and accurately, it will largely save maintenance efforts and enhance productivity.

There are basically two types of changes: the modification (including deletion) of existing files, and the addition of new files to the software project. Locating the two types of changes is different: at the time of request, the files that needs to be modified are existent, so the recommendation of to-be-modified files only requires returning their names as the output. However, the files (or some higher level program elements) that should be added to resolve a change request do not exist in the project. Therefore, the best recommendation possible is to find out the higher level program element and say “some code should be added under this program element”. For tackling a real change request, it is often difficult to know the specific type the changes will take on in advance, so the recommendation of both types of changes should be treated as equally important.

This paper targets at Java projects. Therefore, we can define the recommendation on the two types of changes as File Recommendation and Package Recommendation respectively, since new Java files (and packages) are always created under certain packages. The task of recommending both two types of changes is then defined as Change Localisation as follows:

Definition 1 (Change Localisation): Change Localisation is the task of recommending what source code files should be modified and where new source code files should be created for a general change request.

In literature, automated approaches have been proposed to reduce developers’ maintenance effort in tackling different types of change requests, either for locating a feature (i.e. feature location) or for locating a bug (i.e. bug localisation). However, these approaches are insufficient to deal with a common case, where a bug report or a new feature request calls for both modifying existing files and adding new files or packages to certain packages. Our analysis of four open-source Java projects from the Apache Software Foundation reveals that 30% to 50% changed files are actually added files (Section IV). This illustrates the significance and need for an integrated approach to give package recommendation and file recommendation on a general change request simultaneously.

Therefore, change localisation presents a new challenge beyond the well-studied areas of bug localisation or feature location. In order to give file and package recommendation at the same time, one needs to investigate a uniform scoring strategy, which produces a relevance score for each file as well as each package with the change request, and conduct a
fair ranking of those program elements at different hierarchy. There is an abundance of studies on scoring and ranking source code files [10, 28, 31], but it remains a challenge to assign a reasonable score for each package based on the obtained scores of source code files within the package.

A motivation example from a real-world software project, Pig1, is shown in Figure 1 which lists part of the hierarchical structure for packages and files.

![Hierarchically structured Java packages and source code files](image)

The resolution of a change request Pig-1794, involves creating a new subpackage `js` under the package `org.apache.pig.scripting`, and two Java files, namely `JsFunction.java` and `JsScriptEngine.java` are created under the new package. Since this package did not exist when the change request was submitted, existing methods are unable to give a hint to the developer that new packages or files might be created under `org.apache.pig.scripting`. Change localisation is potentially capable of returning `org.apache.pig.scripting` as a relevant program element, suggesting that “some code” should be created under the package `org.apache.pig.scripting`.

Inspired by information-retrieval (IR)-based bug localisation and feature location techniques, we adopt an information retrieval framework for the change localisation problem. However, even though IR-based approaches [2, 9] have been proposed to localise a general change request, they are still unable to recommend on newly added files.

On the other hand, the hierarchical structures of Java files and packages have been shown to play an important role in localising a bug or a new feature. For example, for bug localisation, Saha et al. [20] have extracted structural features within bug reports and Java source files using their structural information retrieval techniques. Ye et al. [28] have learned heuristics from API documents of Java sources to improve the performance of localisation. However, none of them have incorporated the hierarchical structure of packages and Java code files into a unified scoring and ranking strategy for both source code files and packages.

In order to investigate whether such structural information could be leveraged to support improved information retrieval strategies, we propose a novel IR-based change localisation framework called SeekChanges. For a bug report or a feature request, three structured IR strategies are introduced for localising both modified files and packages where code should be newly created at the immediate time upon request. The experimental results on four collected Java Software projects demonstrate that our structured IR strategies can effectively conduct file recommendation and package recommendation at the same time.

The major contributions of the paper are listed as follows:

- Our SeekChanges framework is the first to be capable of handling a general change request at the request time;
- For the first time, both modified files and locations of newly added code are retrieved by SeekChanges;
- For the first time, structured IR strategies are investigated to capture Java package structure information for localising software changes;
- New datasets that contain both bugs and new feature requests are collected.

The remainder of the paper is organised as follows. Section 2 introduces the datasets collected for demonstrating the performance of SeekChanges. Section 3 describes the proposed approach, including two unstructured IR strategies and three structured IR strategies for this problem. Section 4 introduces the experimental setup. Section 5 shows the experimental results and presents an in-depth discussion of the results. Section 6 discusses the potential threats to validity. The related works are introduced in Section 7, and Section 8 concludes the paper.

II. PROBLEM DATASETS

Many open-source projects use issue-tracking systems to record change requests and the subsequent maintenance activities. As a result, the issue-tracking systems provide a rich source of data for evaluating a change localisation approach. The widely used issue-tracking systems include Jira, BugZilla, Github and SourceForge. In this research, we select four Java projects from Apache Software Foundation which uses Jira as the issue-tracking system, because Jira has a relatively simple workflow, making it easy to extract truly changed files for change requests.

As is shown in Figure 2, a typical Jira issue report has a unique issue ID, denoting the name of the software project and the unique ID within the project. The title of the report is a brief Summary of the issue, while in the Description a more detailed account of the issue is presented. Other details such as the type, priority, affected/fixed versions, status, and relevant commits/patches are also available.

The information considered in our study is shown inside red frames. The collection of the dataset involves selecting the fix version, choosing “Bug”, “Improvement” and “New Feature” as the issue types, and “Resolved” and “Closed” as the status types. By doing this all resolved change requests between two adjacent releases are collected. For each change request, the truly modified and added files are extracted from

1pig.apache.org

2https://issues.apache.org/jira/
3https://www.bugzilla.org/
4https://github.com/
5https://sourceforge.net/
Fig. 2. A typical Jira Issue Report, where the information considered for our structured IR approach are framed.

III. THE SEEKCHANGES FRAMEWORK

The SeekChanges framework aims at recommending both the packages where new Java files should be created and files that should be modified for a general change request. In Java projects, packages and files can be organised into a hierarchical structure, in which all elements are potential retrieval candidates in our approach. In this way, change localisation can be viewed as a typical structured IR task [1], which makes structured IR strategies applicable. Figure 3 presents an overview of the process of the proposed SeekChanges approach.

The overall process of SeekChanges consists of five steps. The first three steps, namely Preprocessing, Indexing and Classical IR Modeling, forms a general IR-based localisation procedure for file recommendation. For an input change request, a relevance score with the change request is computed for each source code file after these traditional steps. Steps 4 and 5 are newly introduced to localise the packages in which new files or packages should be created. Based on the scores for source code files, they adopt structured IR strategies to compute scores for packages by incorporating the hierarchical structures. Each step is further introduced as below:

1) Preprocessing. In this step, a query is formulated by extracting information from an issue report, either from bug summary [20], description [20], or the combination of them [30]. In our study, the concatenation of bug description and bug summary is applied for query formulation. Then stop-words are removed, and each term is stemmed and transformed to lowercase. Source code files contain many composite tokens such as getData or deleteFiles. In preprocessing, they are split into individual terms. Then, we stem the terms, remove the stop words and conduct lowercase transformation in the same way as we deal with the bug data. After this step, the textual contents of both query and source code files are produced.

2) Indexing. In the indexing step, different features of a source code file can be extracted and indexed, such as the whole document itself [30], methods, variables or comments [20]. In our approach, each source code file on the whole is indexed.

3) Classical IR Modeling. In this step, many classical IR models can be applied in order to obtain a relevance score between the query and a source code file, such as Unigram Language Model (UM) [18], Vector Space Model (VSM) [13, 30], Latent Semantic Indexing (LSI) [16], Okapi BM25 [2, 20], Learning-to-Rank [28] or even deep learning [10]. Additional information, such as the stack traces information [13, 27], similar issue reports in history [30], historical co-change files [24] or relevant commits [26], can also be incorporated and combined with the original relevance score generated by an IR model.

4) Organising Packages into Hierarchical Structures. In this step, Java packages and files are organised into a uniform structure, as shown in Figure 1. The structure is similar to the XML document representation in a typical structured information retrieval scenario. Essentially, we represent the root package as the root node for the XML document and all other packages and files as sub-document elements at different levels. Using such
representation, the change localisation problem can be naturally viewed as an XML retrieval problem, which is a well-studied structured IR topic.

5) **Structured IR Modeling.** This step applies structured IR strategies to compute scores for packages on the basis of the scores of the contained files. In our study three structured IR strategies are investigated, namely ElementScoring, Propagation and Aggregation (detailed in Section III-B).

Finally, a ranked list of files and packages is generated using the relevance scores calculated as above. Each program element in the list corresponds to a unique recommendation: when a file is returned, it means it should be modified; when a package is returned, we claim new files should be created under the package. The top returned results may contain both a package and a child file of this package. It means that the file needs to be modified, and some new files or packages should be created under the package as well.

Next, we present the mathematical models behind the two classical non-structural IR strategies and the three structural IR strategies we propose to investigate on top of them. Following the terminologies in the IR field, we use a query $q$ and a document $d$ to represent a bug and a source code file respectively. An element $e$ is used to represent a program element in the hierarchical program package structure, and a leaf element $e_l$ is simply a Java file.

### A. Classical non-structured IR strategies

In our study, Vector Space Model (VSM) and Unigram Language Model (UM) are implemented as the baseline retrieval models. VSM is selected because it is the basis of mainstream approaches on IR-based bug localisation. Since the three structured IR strategies are originally based on UM [1], it is necessary to compare them with original UM as well. Latent topic models such as LSI or LDA are extensively used for relevant tasks [5], [15], but research has found that they do not outperform VSM or UM [18], [31]. Therefore, they are not implemented for comparison in our study.

1) **Vector Space Model (VSM):** VSM is an algebraic model for representing documents as vectors of terms. In the IR context, VSM represents documents and queries as vectors in the same space: $\vec{q} = (W_{q1}, W_{q2}, ..., W_{qn})$, and $\vec{d} = (W_{d1}, W_{d2}, ..., W_{dn})$, where $n$ is the corpus’ vocabulary size, and each $W_{qi}$, $W_{di}$ correspond to the weights for term $i$ in query and document respectively. The relevance score between a document $d$ and a query $q$ is determined by the cosine similarity of their vector representations, i.e.

$$score(d, q) = \cos(\vec{q}, \vec{d}) = \frac{\sum_{i=1}^{n} W_{qi}W_{di}}{\sqrt{\sum_{j=1}^{n} W^{2}_{qi}} \sqrt{\sum_{k=1}^{n} W^{2}_{dk}}}$$

The strategy to assign weights of terms in $\vec{q}$ and $\vec{d}$ determines the performance of VSM. Different weighting schemes have been developed by combining the Term Frequency (TF) and Document Frequency (DF). In our study, the commonly used Term Frequency and Inverse Document Frequency
(TF*IDF) weights are adopted. For the $k^{th}$ term in the corpus vocabulary, denoted by $v(k)$, its term frequency in a document $d$ is defined by $\#(v(k), d)$, the number of times it occurs in $d$:

$$TF(v(k), d) = \#(v(k), d)$$  \hspace{1cm} (2)

Its inverse document frequency is computed by conducting the logarithm of dividing the number of documents in the corpus by the number of documents containing $v(k)$, i.e.

$$IDF(v(k)) = \log \frac{|C|}{0.5 + |d \in C : v(k) \in d|}$$  \hspace{1cm} (3)

where $|C|$ is the overall number of documents, $|d \in C : v(k) \in d|$ is the number of documents containing $v(k)$. The small offset 0.5 is used to avoid zero denominator. The corresponding TF*IDF weight for term $v(k)$ in document $d$, denoted by $W_d^k$, is thus computed as:

$$W_d^k = TF(v(k), d) \times IDF(v(k))$$  \hspace{1cm} (4)

For either a document or a query, the TF-IDF weight for each different term can be computed by Equation (4), and the similarity value between a query and a document is computed using Equation (1).

2) Unigram Language Model (UM): In UM, the language model for each document $d$ is a simple multinomial distribution $\theta_d(t)$ over terms $t$ where the probability for each term $p_{\theta_d}(t)$ is its term frequency in the document $\#(t, d)$ normalised by the total number of terms $|d|$ in the document:

$$p_{\theta_d}(t) = \frac{\#(t, d)}{|d|}$$  \hspace{1cm} (5)

The original UM often needs to be smoothed with the corpus’ unigram model in order to assign a background probability for the terms that are in the corpus but not in the document. Our study utilised the Dirichlet Smoothing technique [29], which is a linear interpolation between the original document’s unigram model and the corpus’ unigram model:

$$p_{\hat{\theta}_d}(t) = \frac{|d|}{|d| + \mu} p_{\theta_d}(t) + \frac{\mu}{|d| + \mu} p_{\theta_e}(t)$$  \hspace{1cm} (6)

where $p_{\theta_e}(t)$ refers to the corpus’ unigram model, computed by concatenating all documents in the collection, and following Equation (5). The Dirichlet smoothing parameter $\mu$ determines the weights for document language model and corpus language model. A larger value of $\mu$ means the corpus’ unigram model has a greater influence on the smoothed language model. $\mu$ is often manually specified for each document collection. The relevance score between $d$ and a query $q$ is regarded as the query likelihood (QL), or the probability of $q$ under the pre-calculated multinomial distribution:

$$score(d, q) = P(q|M_e) = \prod_{i=1}^{m} p_{\hat{\theta}_d}(q_i)$$  \hspace{1cm} (7)

where $m$ is the number of terms in the query. By using Equation (7), we have assumed that the query terms are conditionally independent given the document $d$.

B. Our Structured IR strategies

Our study is aimed at localising a change request to both possibly modified files, and packages containing possibly added code for Java projects. We hypothesise that if a change request is textually similar to a package, new files or packages might be created under the package. The higher the similarity is, the more likely new code will be added.

Java projects have a hierarchical structure for packages and source code files. A root package containing contents of the whole project is present; a package can have child packages and child source code files. Therefore, the scoring for each package is largely dependent on the scores of its child files.

Existing models have been built for ranking source code files exclusively. For the change request Pig-1794 in the motivating scenario (see Figure 1) where the actual result involves adding many files under the scripting package, these methods will fail to give the relevant recommendation. In our study, however, every element is a retrieval candidate, and the scripting package can be retrieved in an ideal case. In fact, our proposed strategy does successfully rank org.apache.pig.scripting as the most relevant result.

Inspired by IR strategies applied to XML retrieval [1], we systematically investigate three structured IR strategies, namely ElementScoring, Aggregation and Propagation, to score the packages on the basis of the scores for source code files (Step 5 in Figure 3). Even though UM is the state-of-the-art model for structured IR strategies and provide a well-principled probability framework to incorporate structured information into scoring and ranking [1], in this paper, the structured IR strategies are investigated on both UM and VSM.

In the rest of this section, each of the three strategies is described in detail.

1) ElementScoring: ElementScoring is built on the assumption that the relevance score for an element $e$ to the given query $q$ (in other words the conditional probability of $e$ given $q$) depends on both the prior probability of relevance for $e$ and the probability of $q$ generated from the element’s language model $M_e$:

$$P(e|q) \propto P(e)P(q|M_e)$$  \hspace{1cm} (8)

where $P(q|M_e)$ is either the query likelihood function computed by UM in Section III-A2, or the similarity score computed by VSM in Section III-A1. The selection of element prior probability $P(e)$ is determined by the number of terms length($e$) in $e$. Non-leaf elements (packages) are formed by concatenating all its children nodes, or all source code files within the package. $P(e)$ is therefore calculated as

$$P(e) = \frac{\text{length}(e)}{\sum_{e'} \text{length}(e')}$$  \hspace{1cm} (9)

where the summing notation $\sum_{e'}$ represents the summation of all elements in the hierarchical structure.

2) Propagation: The propagation method starts with scores for leaf elements (source code files), and propagates the scores in a bottom-up manner to compute scores for non-leaf
elements. The score for each non-leaf node is obtained by adding up the scores of its child elements, multiplied by a propagation factor $\alpha$:

$$score(e, q) = \alpha \times \sum_{e_c} score(e_c, q)$$

(10)

where $e_c$ is a child element of $e$, and $score(e_c, q)$ denotes the relevance score for $e_c$. $\alpha \in [0, 1]$ is a real-valued parameter that controls the impact of child elements to the parent element. Replacing Equation (10) with a non-iterative expression and smoothing it with the collection score, we obtain the following ranking strategy:

$$score(e, q) = \rho \times n \times \sum_{e_1} \alpha^{d(e,e_1)-1} \times score(e_1, q)$$

$$+ (1 - \rho) \times score(root, q)$$

(11)

where $score(root, q)$ is the score obtained using the root package, which includes the contents of the whole corpus, $\rho \in [0, 1]$ is the smoothing parameter determining the extent to which the root package affects the final score and $n$ is the number of leaf elements for $e$. $d(e, e_1)$ is the distance between $e$ and a leaf element $e_1$, represented by the length of the path from $e_1$ to $e$ in the hierarchy. For example, when $e_1$ is a child of $e$ then $d(e, e_1) = 1$; when $e_1$ is a grandchild of $e$ then $d(e, e_1) = 2$, and so forth. $score(e_1, q)$ is the relevance score for a leaf element $e_1$, which could be computed either by UM or VSM in the same way ElementScoring technique. Equation (11) is used as the scoring function adopted in a state of the art XML retrieval system [22, 23].

3) Aggregation: The aggregation method assumes that the probability distribution of a parent element can be viewed as an aggregation of its own content probability distribution and the distribution of its child elements. The aggregation process starts from computing the probability distribution for source code files (leaf nodes), and the probabilities are aggregated to form the probability distribution for packages in higher hierarchies. The probability distributions for leaf elements can be computed either using UM according to Section III-A2 or VSM according to Section III-A1. For the UM case, the probability a non-leaf element $e$ assigns to a term $k_i$ is computed as

$$P(k_i|M_e) = \sum_{e_j} w_j P(k_i|M_{e_j})$$

(12)

where $P(k_i|M_{e_j})$ is the probability of $k_i$ in each child element $e_j$, and $w_j$ reflects the contribution of child element $e_j$ to the element $e$. Then the query likelihood function in Equation (7) is used to compute the relevance score. For VSM, we adapt Equation (12) to update the document vector of a non-leaf element $e$:

$$\vec{e} = \sum_{e_j} w_j \vec{e}_j$$

(13)

where $\vec{e}$ and $\vec{e}_j$ corresponds the vector representation of $e$ and $e_j$ respectively. By applying Equation (11) the similarity score for each element with the query is obtained. In this paper, the assignment of weights $w_j$ are performed with equal weights, i.e.

$$w_j = \frac{1}{|child(e)|}, \forall e_j \in child(e)$$

(14)

Note that aggregation is inherently different from propagation, because the scores are propagated from element leaves all the way up to packages for the propagation strategy, whilst the aggregation strategy actually sums up term probabilities or weights.

IV. EXPERIMENTAL SETUP

The experiment is composed of two steps. First, the three proposed structural IR strategies in SeekChanges are evaluated and compared with baseline models VSM and UM in terms of the overall change localisation performance as well as the package recommendation effectiveness. The result of overall change localisation reflects the effectiveness of recommending for a real change request in practice, while the evaluation result for package recommendation signals the performance of recommending to-be-added locations for an algorithm, which is the research focus of this study. The best-performed structured IR strategy is identified from the comparison result. Second, we combine the best-performed structured IR strategy with a state-of-the-art bug localisation approach to recommend packages and files simultaneously. In order to examine if the package recommendation is effective, the performances of package recommendation the overall change localisation are compared with that of file recommendation under the bug localisation approach. The experiments are conducted on four open-source projects in Apache Software Foundation that have used JIRA as issue-tracking system. The effectiveness is evaluated with the most commonly used metrics for IR-based localisation approaches, including MAP, MRR and TopK.

A. Datasets

We have proposed three structured IR strategies to localise change requests and give package recommendations on the basis of file relevance scores. To validate the adaptability of the proposed methods onto different types of change requests, we select software projects from Apache Software Foundation with Jira Issue-tracking systems. We did not choose the iBUGs dataset [3] or other datasets that have been commonly used in IR-based bug localisation papers [13, 27, 30], because they only contain bug reports, and the newly added files are excluded from truly changed file list. Instead, we select four active Java projects from JIRA Issue tracking system: Myfaces Core[6], Pig, Wicket[7] and Zookeeper[8]. Each project is targeted at a unique application domain. Myfaces Core is focused on JavaServer technologies. Pig is a platform for analyzing large datasets. Wicket is the open source Java Web framework. Zookeeper is a centralised service for maintaining configuration information.

[6] myfaces.apache.org
[7] wicket.apache.org
[8] https://zookeeper.apache.org
For each project, we manually select the two adjacent releases and collect data as introduced in Section III. The obtained datasets contain different types of change requests, and both modified and added files are obtained for each change request. Table I presents basic statistics of the datasets.

| Project | Request release | Fix release | #Issues | #Files |
|---------|-----------------|-------------|---------|--------|
| Myfaces | 2.00            | 230.1       | 15      | 994    |
| Pig     | 0.81            | 0.90        | 79      | 1412   |
| Wicket  | 6.00-0.9 beta3  | 6.10        | 28      | 2811   |
| Zookeeper| 3.36           | 3.40        | 60      | 499    |

We also computed the components of the change files for each dataset. Table II presents the number of actually changed files (#ΔFiles), added files (#(+Files)), and ratio of added files in the total number of changed files (%(+Files)) for all the four projects. It shows that the overall ratio of newly created files takes up from 30.72% (Zookeeper) to 47.37% (Myfaces) of all changed files. Only capable of finding modified files, previous study fail to recover this many oracles from the beginning. Therefore, developers would benefit from package recommendation simultaneously with file recommendation, as our structured IR strategies aim to tackle.

| CR Type   | #ΔFiles | #(+Files) | %(+Files) |
|-----------|---------|-----------|-----------|
| Myfaces   | 95      | 45        | 47.37     |
| Zookeeper | 765     | 235       | 30.72     |
| Wicket    | 84      | 37        | 44.05     |
| Pig       | 659     | 235       | 35.66     |

B. Experimental Settings

The first experiment evaluates the effectiveness of the three proposed methods, i.e. ElementScoring, Propagation and Aggregation, relying VSM and UM for computing file scores respectively. They are also contrasted with the VSM and UM models applied alone each. For the second experiment, the best-performed structured IR strategy is applied to perform package recommendation based on the scores for code files computed by BugLocator [30], which is a state-of-the-art approach for bug localisation. The parameter settings for the models are shown below:

- **ElementScoring.** The ElementScoring method described in Section III-B1 is implemented.

- **Propagation.** The Propagation method is implemented according to Section III-B2. In the first experiment, the values for ρ and α are assigned with 0.9 and 0.8 respectively according to simple manual inspection around the best parameter combination according to [22], [23].

- **Aggregation.** The Aggregation method has been introduced in Section III-B3.

- **UM.** The UM introduced in III-A2 is implemented as a baseline system. Each package is treated as a document, and its content is comprised of the contents of all source code files in the package. The Dirichlet Smoothing parameter μ = 1000 is used.

- **VSM.** The VSM introduced in Section III-A1 is implemented as a baseline system. Similar to UM, each package is treated as a document, and we merge the contents of all source code files in the package as its content representation.

- **BugLocator.** BugLocator [30] computes the relevance score between a bug report and a code file by combining two aspects of similarity: I) their revised Vector Space Model (rVSM) similarity; II) the similarity between the bug report and historical bug reports that have modified the code file. A linear combination parameter α ∈ [0, 1] is used to controls the contribution of the two aspects in the final relevance score. In our experiment, the suggested parameter α = 0.2 in [30] is used.

All the experiments are run on a server with Intel Core i5 3.1GHz processor with 4 cores and 8GB RAM. The operating system is 64-bit Mac. All the source code is written in Java and executed in Eclipse 4.4.1.

C. Evaluation Metrics

The three most commonly used IR-based bug localisation metrics, Top K, MRR and MAP are adopted as the effective measures for the change localisation problem.

- **Top K.** It measures the proportion of the queries containing at least one positive instance in the top K ranking result in all queries, so it is a real value in the range of [0,1]. The value of the Top K measure depends on the choice of K. The larger the value of K is, the larger the Top K value will be. A larger value of Top K represents a better performance. Without loss of generality, in our study K is set to 10, i.e. the Top 10 metric is used.

- **Mean Average Precision (MAP).** It measures the mean value of Average Precision (AP) over all queries. For a query q, the average precision is obtained by averaging the precision at all retrieval ranks, i.e.

  \[
  AP(q) = \frac{\sum_{i=1}^{\text{pos}} \frac{\text{rank}(\text{pos}, i)}{|\text{pos}|}}{\text{pos}}
  \]

  where |pos| is the number of positive instances for query q, and rank(pos, i) is the ith best rank of the positive instances in the returned list. Since rank(pos, i) is always equal or larger than i then AP(q) falls into the range [0, 1]. A larger value of AP(q) means the rank is closer to the ground truth, i.e. rank(pos, i) = i, thus indicating a better performance. MAP, as the mean value of AP, is then obtained through the following equation:

  \[
  MAP = \frac{\sum_{q=1}^{Q} \text{AP}(Q(q))}{|Q|}
  \]

  MAP is a value between [0, 1]. A higher MAP value represents a better overall performance.
• **Mean Reciprocal Rank (MRR).** This measure takes into account the first positive instance. For each query, the reciprocal rank of the first correct returned result is calculated, and MRR is the average of the obtained reciprocal ranks over the whole set of queries:

\[
MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}(q_i, 1)}
\]

(17)

where \( \text{rank}(q, 1) \) denotes the rank for the first correct answer for a query \( q \).

To evaluate the efficiency of proposed approach, the average execution time per query \( (ET/Q) \) is selected. It is computed by dividing the overall execution time by the number of queries, i.e. change requests. It has been used to evaluate the run-time overhead for bug localisation methods [21].

V. EXPERIMENTAL RESULTS

This section presents the experimental results and conducts an in-depth analysis, focusing on the effect of the structured IR strategies for change localisation, compared to non-structured classical IR models and state-of-the-art IR strategies.

A. Investigation of best-performed structured IR strategies

In order to obtain the structured IR strategy for change localisation with the best performance, we evaluate and compare the performances of 8 models: the three structured IR strategies, namely ElementScoring, Aggregation and Propagation, are applied on the basis of VSM or UM for obtaining file relevance scores respectively. VSM and UM on their own are also built as baseline models.

The effectiveness evaluation results on four Java open source projects are shown in Table III and IV. Table III shows the general performance of change localisation, while Table IV exhibits the performance of package recommendation. In both tables, each value corresponds to one specific evaluation metric (MRR, MAP or Top10), one specific file scoring strategy (VSM, UM), and one specific structured IR strategy (None refers to the baseline VSM or UM model, ElementScoring, Propagation or Aggregation) on one Project (Myfaces, Pig, Wicket, Zookeeper). The values in bold are the highest among the structured IR strategies above one basic IR model on one evaluation metric for a specific project.

The efficiency results are displayed in Table V. It shows that the highest average execution time per query is 3.69s, indicating a negligible added cost in terms of computational overhead. In fact, it is claimed in [20] that a value of less than 6s for \( ET/Q \) can already be regarded as computationally efficient. Therefore, all experimented strategies are considered efficient.

In Table III and IV, Propagation and Aggregation occupy 90% of all best values, making them most suitable candidate strategies. Between them, Propagation is selected as the best-performed IR strategy for three reasons. First, Propagation performs better in VSM-based file scores, while Aggregation is competent in extending UM-based file scores for package recommendation. Since VSM is much more widely applied compared to UM, Aggregation performs better in VSM-based file scores, while Propagation performs better in UM-based file scores.
that the introduction of structured IR strategies does bring benefits to the change localisation performances for classical IR models.

B. Adapting the best-performed structured IR strategy to state-of-the-art bug localisation approach

From the first experiment, Propagation has been identified as the best-performed structured IR strategy. However, it is worth noting that the absolute performances of Propagation are still quite low. Since the structured IR strategy is applied on the basis of scores for Java files, its performance on package recommendation largely relies on the accuracy of file relevance scores. In the first experiment, the file scores generated by the classical VSM or UM models are of poor qualities, so it is reasonable for Propagation to have a low package recommendation performance as well. In order to demonstrate its capability of tackling real change requests, we refer to a state-of-the-art tool BugLocator [30] for computing relevance scores for source code files, and apply the Propagation strategy to compute scores for all packages in the project on the basis of much more accurate file scores. Similar to the previous experiment, we also evaluate the performance of change localisation and package recommendation. Rather than using the default parameter combination, we conduct a grid search over the parameter space $\rho \in \{0, 0.1, ..., 1\}$ and $\alpha \in \{0, 0.1, ..., 1\}$ to obtain the best performed results on each of the four datasets.

The results for this experiment is shown in Table VI. The best parameter combinations are $\alpha = 0.8, \rho = 0.75$ for Myfaces, $\alpha = 0.5, \rho = 0.2$ for Pig, $\alpha = 0.3, \rho = 0.4$ for Wicket and $\alpha = 0.3, \rho = 0.5$ for Zookeeper respectively. Compared to the performance of BugLocator on file recommendation, we obtained insignificantly different, if not as good, performances for package recommendation (Propagation, add) and the overall change localisation (Propagation, all) also has similar performances.

The positive results on package recommendation demonstrates that the scores for source code files are properly leveraged by the Propagation strategy to generate predictions on possible locations for newly created files.

On the other hand, putting all packages into ranking brings a potential risk of lowering the ranks of to-be-modified files and adversely affecting the overall performance of change localisation, but the experiment results eliminate such concern.

Therefore, SeekChanges actually provides a viable framework to efficiently and effectively generalising any existing IR-based change localisation approach so that they could also infer to-be-added packages. It does not alter the way the file relevance scores are computed, but rather preserve the high quality of file scores, and pass it along the hierarchical package structure to achieve accurate package recommendation. Any existing file recommendation approach can be combined with Propagation to have the ability of tackling a general change request, and performing file recommendation and package recommendation simultaneously with decent performances.

| Project   | Method          | MRR  | MAP   | Top10  |
|-----------|----------------|------|-------|-------|
| Myfaces   | BugLocator      | 0.354 | 0.3280| 0.6000|
|           | Propagation, all| 0.2802| 0.2780| 0.6000|
|           | Propagation, add| 0.3138| 0.3082| 0.5000|
| Pig       | BugLocator      | 0.2932| 0.2134| 0.4375|
|           | Propagation, all| 0.3336| 0.2194| 0.5625|
|           | Propagation, add| 0.3822| 0.3806| 0.6667|
| Wicket    | BugLocator      | 0.3588| 0.2916| 0.6153|
|           | Propagation, all| 0.3822| 0.3806| 0.6667|
|           | Propagation, add| 0.3336| 0.2194| 0.5625|
| Zookeeper | BugLocator      | 0.4608| 0.3334| 0.4583|
|           | Propagation, all| 0.4776| 0.3746| 0.8000|
|           | Propagation, add| 0.4356| 0.3896| 0.7091|
| SWT       | BugLocator      | 0.2361| 0.2122| 0.6645|
|           | Propagation, all| 0.4706| 0.4706| 0.9411|
| AspectJ   | BugLocator      | 0.3125| 0.2715| 0.5123|
|           | Propagation, all| 0.4608| 0.3334| 0.4583|
|           | Propagation, add| 0.4776| 0.3746| 0.8000|
| Tomcat    | BugLocator      | 0.4356| 0.3896| 0.7091|
|           | Propagation, all| 0.4608| 0.3334| 0.4583|
|           | Propagation, add| 0.4776| 0.3746| 0.8000|

and generally more effective than UM in the field of change localisation, Propagation has a larger practical value. Second, Propagation generally performs better than Aggregation in recommend the mixed ranking (with 10 best values over 8), so it is more likely to be effective on real change localisation tasks. Lastly, Propagation has two free parameters, making it more adaptable to different datasets. In all, the experiment result illustrates that Propagation is the best-performed structured IR strategy.

Furthermore, the tables present a high stability among the structured IR strategies. Applied on top of baseline models (VSM and UM), structured IR strategies exhibit consistently superior performances in overall change localisation performances, and the best structured IR strategies also significantly outperform classical IR models on all four datasets. It reveals
C. Summary

In the experiments, we identified Propagation as the best-performed structured IR strategy. By combining the Propagation strategy with a state-of-the-art IR-based bug localisation model, i.e., BugLocator, it succeeded in obtaining good performance on recommending the locations for newly created files, whilst the overall performance of retrieving to-be-modified files and to-be-added locations is comparable to BugLocator.

VI. Threats to Validity

The validity of our case studies could be affected by the following potential threats, from external, construct and internal perspectives as classified by Perry et al. [14].

External validity: The datasets for our study are collected from four open-source Java projects with Jira Issue-tracking system from Apache Software Foundation. For other commonly used Issue-tracking systems such as Bugzilla or Github, however, data is not collected and experimented, so it is possible that the proposed methods do not apply well to those projects because of the datasets selection. For more concrete conclusions, our methods should also be applied to commercial projects whose data is considered to be cleaner.

Construct validity: Our methods are rested on the underlying assumption that new files should be created under the package that is textually similar to a change request, and the textual content of a package is represented by combining all source code files under the package. However, merging all file contents is not necessarily the best representation of package contents. If an API level description is combined it might make a better representation of the package compared to simply concatenating all the contents.

Internal validity: The performances of the proposed strategies are only evaluated on IR models that only captures textual similarity and historical similar bugs. As a result, the absolute performances are relatively low in comparison to the approaches exploiting additional heuristics such as stack traces [27], contextual relationships [4], etc. To control the compounding factors and isolate the effect of structured IR strategies in the study, we intentionally exclude these heuristics. In our further studies the effectiveness of structured IR strategies could be evaluated under these complicated factors for a more concrete conclusion.

VII. Related Work

Feature location via IR: To address the overhead issue of program analysis approaches, information retrieval (IR) approaches [5–2], [12], [13], [16], [19] are generally simple, computationally efficient while preserving decent performance. Treating source code as textual documents and feature descriptions as natural language queries, a direct application of classic IR models (e.g., vector space model [12] and Okapi BM25 models [4]) has achieved a decent accuracy performance. The leverage of traditional IR models itself such as LSI [12] and FCA [15] has achieved comparable performance to program analysis approaches. IR approaches can also be combined with static [6], [7] and dynamic analysis [16], [19] information to further boost the performance.

While feature location mainly aims to locate source code implementations of a requested feature, existing bug localisation models [13], [20], [24], [26], [27], [30] identify retrospectively which files have been modified for a fixed bug. However, feature location aims at locating existing features where the source code implementation is always available. For feature requests calling for creating new files, the existing approach will erroneously map them to existing irrelevant files.

Bug localisation via IR: In recent years, IR-based approaches have been proposed as more practical and efficient solutions for bug localisation, especially since the comparative study conducted by Rao and Kak [11] who claim that simple IR models such as VSM and UM are the best performed models on bug localisation problems. Later work focused mostly on extending VSM as the fundamental model with additional heuristics. Zhou et al. [25], [30] proposed BugLocator, which put the document length and similar bug reports in the history into the similarity calculation, resulting in a significant improvement in performance. Subsequent research [13], [20], [24], [26–28] further refined BugLocator, either by replacing VSM with a better IR model [20], [28], by putting stack trace into consideration [13], [27], or by incorporating more historical information into the framework [24], [26]. Deep learning models [8], [10] are also built to tackle bug localisation. Unfortunately, these approaches are unable to give recommendations on newly created source code files for bug reports, and it seems the experiments are conducted at post-fix time, which does not represent real maintenance scenarios.

To sum up, neither feature location nor bug localisation approaches are able to take care of the change request in which files might be created to fulfill the task. Therefore, we propose a uniform IR model and process to retrieve bugs as well as new features and give suggestions on modified and added files for a change request.

VIII. Conclusions and Future Work

The process of creating change requests and seeking for ways to address them have become critical in evolutionary software development and maintenance. Researchers are motivated to investigate different automatic methods reduce maintenance time spent on addressing different types of change requests. Among them many approaches for localising either bugs or features are considered, but few of them actually target localising the package containing newly created files upon a general change request, and thus a non-negligible ratio of newly created files cannot be retrieved.

To address these limitations, we have proposed a uniform change localisation framework to recommend both modified files and packages with added files for a general change request for Java projects. Three structured IR strategies, namely ElementScoring, Propagation, and Aggregation, have
been proposed and compared with non-structured IR strategies, namely VSM and UM. The best-performed strategy, Propagation, have been combined with a state-of-the-art IR-based bug localisation approach, resulting in a generalised framework capable of efficiently recommending modified files and package containing added files at the same time, with good performances for any type of change request.

Our work could be further improved in at least two areas. First, further work could expand the scope to map a change request to program elements at lower levels of abstraction such as methods or variables. Second, the file relevance scores could be obtained by advanced IR models [10], [28].

**REFERENCES**

[1] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 9 pp.–29, Sept. 2005.

[3] V. Dallmeier and T. Zimmermann. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE ’07, pages 433–436, New York, NY, USA, 2007. ACM.

[4] T. Dilshener. Improving information retrieval-based concept location using contextual relationships. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE ’12, pages 1499–1502, Piscataway, NJ, USA, 2012. IEEE Press.

[5] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Guhneuc. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In *The 16th IEEE International Conference on Program Comprehension*. ICPC 2008, pages 53–62, June 2008.

[6] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE ’07, pages 14–23, New York, NY, USA, 2007. ACM.

[7] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *IEEE 31st International Conference on Software Engineering*, ICSE 2009, pages 232–242, May 2009.

[8] X. Huo, M. Li, and Z. Zhou. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. volume 16, pages 230, New York, NY, USA, July 2016.

[9] H. Kagdi and D. Poshyvanyk. Who can help me with this change request? In *IEEE 17th International Conference on Program Comprehension, 2009. ICPC ’09*, pages 273–277, May 2009.

[10] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481, Nov. 2015.

[11] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug Localization Using Latent Dirichlet Allocation. *Inf. Syst. Technol.*, 52(9):972–990, Sept. 2010.

[12] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering, 2004. Proceedings*, pages 214–223, Nov. 2004.

[13] L. Moreno, J. Treadway, A. Marcus, and W. Shen. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSM E)*, pages 151–160, Sept. 2014.

[14] D. E. Perry, S. E. Sim, and S. M. Easterbrook. Case studies for software engineers. In A. Finkelstein, J. Estublier, and D. S. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004)*, 23–28 May 2004, Edinburgh, United Kingdom, pages 736–738. IEEE Computer Society, 2004.

[15] D. Poshyvanyk and A. Marcus. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *15th IEEE International Conference on Program Comprehension, 2007. ICPC ’07*, pages 37–48, June 2007.

[16] D. Poshyvanyk, A. Marcus, V. Rajlich, Y. G. Guhneuc, and G. Antoniol. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. In *14th IEEE International Conference on Program Comprehension, 2006. ICPC 2006*, pages 137–148, 2006.

[17] V. Rajlich. Software Evolution and Maintenance. In *Proceedings of the on Future of Software Engineering, FOSE 2014*, pages 133–144, New York, NY, USA, 2014. ACM.

[18] S. Rao and A. Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR ’11*, pages 43–52, New York, NY, USA, 2011. ACM.

[19] M. Reveille, B. Dit, and D. Poshyvanyk. Using Data Fusion and Web Mining to Support Feature Location in Software. In *2010 IEEE 18th International Conference on Program Comprehension (ICPC)*, pages 14–23, June 2010.

[20] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 345–355, Nov. 2013.

[21] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 161–170, Sept. 2014.

[22] K. Sauvagnat and M. Boughanem. Using a Relevance Propagation Method for Adhoc and Heterogeneous Tracks at INEX 2004. In *Proceedings of the Third International Conference on Initiative for the Evaluation of XML Retrieval, INEX’04*, pages 337–348, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] K. Sauvagnat, L. Hliaoua, and M. Boughanem. XFIRM at INEX 2005: Ad-hoc and Relevance Feedback Tracks. In *Proceedings of the 4th International Conference on Initiative for the Evaluation of XML Retrieval, INEX’05*, pages 88–103, Berlin, Heidelberg, 2006. Springer-Verlag.

[24] C. Tanitthamthavorn, A. Ihara, and K.-I. Matsumoto. Using Co-change Histories to Improve Bug Localization Performance. In *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 543–548, July 2013.

[25] F. Thung, T.-D. B. Le, P. S. Kochhar, and D. Lo. BugLocator: Integrated Tool Support for Bug Localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 767–770, New York, NY, USA, 2014. ACM.

[26] S. Wang and D. Lo. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*, pages 53–63, New York, NY, USA, 2014. ACM.

[27] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 181–190, Sept. 2014.

[28] X. Ye, R. Bunescu, and C. Liu. 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, FSE 2014*, pages 689–699, New York, NY, USA, 2014. ACM.

[29] C. Zhai. Statistical Language Models for Information Retrieval A Critical Review. *Found. Trends Inf. Retr.*, 2(3):137–213, Mar. 2008.

[30] J. Zhou, H. Zhang, and D. Lo. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports. In *Proceedings of the 34th International Conference on Software Engineering, ICSE ’12*, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.

[31] Y. Zhou, Y. Tong, R. Gu, and H. Gall. Combining Text Mining and Data Mining for Bug Report Classification. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–320, Sept. 2014.