IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 10 April 2024, accepted 5 May 2024, date of publication 8 May 2024, date of current version 21 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3399101

RESEARCH ARTICLE

Enhancing Software Co-Change Prediction: Leveraging Hybrid Approaches for Improved Accuracy

MOHAMMED ZAGANE[1] AND MAMDOUH ALENEZI[2], (Member, IEEE)
[1]Department of Computer Science, Mustapha Stambouli University, Mascara 29000, Algeria
[2]Department of Software Engineering, Prince Sultan University, Riyadh 12435, Saudi Arabia

Corresponding author: Mohammed Zagane (m_zagane@univ-mascara.dz)

This work was supported by the College of Computer and Information Sciences, Prince Sultan University, Riyadh, Saudi Arabia.

ABSTRACT Accurate prediction of co-changes in software systems is crucial for efficient development and maintenance, especially as systems grow in complexity. While deep learning-based approaches have shown promise, they often struggle with diverse and complex data. In this paper, we present a novel hybrid approach that combines traditional software engineering methods with deep learning techniques to improve co-change prediction accuracy. Our approach leverages software metrics and deep learning models, incorporating the unique characteristics induced by naming conventions, such as PascalCase and camelCase, used by developers for naming consistency. By utilizing char n-gram embedding and sub-token context, we enrich the vector representations of source file names, capturing relationships and dependencies between files. We comprehensively evaluate our hybrid approach using three open-source software projects. The findings of this study have significant implications for the development of more effective software co-change prediction tools and techniques, enabling better decision-making in software development and maintenance processes. Our approach outperforms traditional software engineering methods and deep learning-based approaches, demonstrating its potential to significantly improve software development and maintenance efficiency.

INDEX TERMS Co-change prediction, change impact analysis, change prediction, code metrics, neural embedding model, char n-gram embedding, recommendation systems.

I. INTRODUCTION

In the realm of software development and maintenance, accurately predicting co-changes has become increasingly crucial as software systems grow in complexity [1]. Co-change prediction involves identifying the modules or components that are likely to change together in the future [2]. Precise co-change prediction can assist developers in anticipating potential changes, prioritizing testing efforts, and reducing the overall time and cost required for software maintenance [3].

While traditional software engineering methods, such as statistical analysis and modelling, have been widely employed for co-change prediction [1], they have limitations when it comes to handling diverse and intricate data such as changelogs data. Deep learning-based approaches have shown promise in addressing these limitations by leveraging extensive data to learn complex patterns and relationships. However, these approaches often face challenges in interpreting the learned patterns and require substantial amounts of labelled data, which can be difficult to acquire.

To overcome these limitations, this paper introduces a novel hybrid approach that combines the strengths of traditional software engineering methods with the power of deep learning techniques. Our approach incorporates a combination of code metrics and a char n-gram embedding model to significantly enhance co-change prediction accuracy. By integrating the interpretability of traditional methods with the learning capabilities of neural-based embedding models, our approach effectively leverages co-change patterns induced by both the source code and the changelogs data which provides a more effective and efficient way to predict co-changes.

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

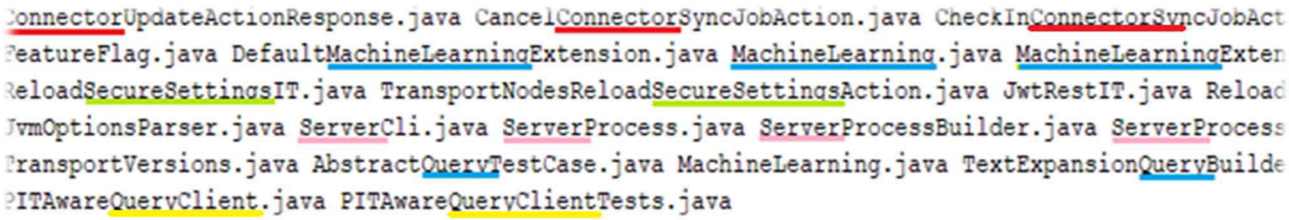

FIGURE 1. Co-changed source files often share similar naming patterns and convention.

Previous studies that utilized changelogs data in co-change prediction have demonstrated that the context provided by co-change instances, as captured by dependency networks [4] or word embedding techniques [5], offers valuable insights into source files that tend to change together. Word embedding techniques enable the creation of low-dimensional representations of sets of tokens or words (source file names), placing co-occurring tokens closer together in the resulting vector space [5]. However, prior studies overlook an important characteristic induced by naming conventions, such as PascalCase and camelCase, which developers adopt to maintain naming consistency throughout projects. Source file naming often signifies relationships, particularly when files are interconnected or dependent (Figure 1). For example, consider three Java files: 'BookRepository.java' implies a class handling database operation specific to books, 'BookService.java' indicates a higher level of business logic related to books, likely utilizing the repository, and 'BookController.java' denotes a controller managing book-related requests, likely interacting with 'BookService' for operations. Here, the names themselves suggest relationships and dependencies. In our analysis of the Spring Frame-work and Elasticsearch projects used in the experimental evaluation, we observed that more than 28.80% in Spring Frame-work and 35% in Elasticsearch of co-change instances contain identical parts in all file names, with no instances found where file names lack identical parts. To leverage this characteristic, our approach incorporates char n-gram embedding [7], which relies on sub-token context. In this technique, the vector representation of each token (source file name) is enriched by the vector representations of its sub-tokens which is better, in the context of our study, than the hole-token embedding used in related work.

The contribution of this study is two-fold:

- Proposing a novel hybrid approach that combines the strengths of code metrics in quantifying co-change attributes from the source code and the power of char n-gram embedding to effectively learn co-change patterns from the changelogs data,
- Proposing and making publicly available a hand-curated dataset of co-change instances and code metrics extracted from three well-known open-source projects, we also propose pre-trained models. These materials can be used to replicate our study and also, they can be used by other researchers in future works.

The rest of this paper is organized as follows: in section II, we present the most relevant related works, in section III, we describe the proposed approach and the methodology to establish and evaluate it, in section IV, we present the experimental setups, in section V, we present and discuss the obtained results and highlight the limitations of the work and in section VI, we summarize the work done in this study and indicate perspectives for future works.

II. RELATED WORK

Given the crucial role of change management in the software maintenance task, huge research efforts have been devoted to developing effective approaches to software change impact analysis (SCIA) and co-change prediction (CCP). For the sake of brevity, we will focus on presenting the most relevant research works that leverage code metrics and changelogs data. For more details about techniques and methods employed in the field, please refer to the systematic literature reviews done such as [8] and [9].

Code metrics were widely leveraged to solve challenging problems in the field of software engineering such as defect prediction [10], [11], [12], [13], [14], [15], [16] and vulnerability prediction [17], [18], [19], [20], [21], [22], [23]. This is motivated by the fact that code metrics are known for their ability to quantify software attributes such as size, complexity and coupling which are proven in practice to be correlated with defects and vulnerabilities [24]. Motivated by similar intuition, several research works in the field of SCIA and CCP have used different code metrics types to propose approaches and methods. Especially, the metrics that directly quantify dependency and coupling between software modules gained more researchers' interest. In [25], researchers have investigated the relationship between object-oriented coupling and ripple effects. They used metrics such as Coupling Between Object (COB) and Data Abstraction Coupling (DAC) to propose a SCIA model. Researchers have reported that these metrics can indicate class pairs with higher ripple effect probability. They have also recommended focusing on all sorts of requirements and design documentation that provide additional information for coupling aspects that cannot be captured by source code only. Code metrics that indirectly quantify coupling and dependencies were also investigated. Researchers in [26] have investigated the ripple effects in non-object-oriented software

projects. They used four variations of McCabe's complexity metric and an approximated algorithm to assess the SCIA for C programs. Researchers in [27] have investigated the correlation of the cohesion metrics with the impact of write interactions between class members. They have proposed revisions to the existing cohesion metrics to consider the impact of the write interactions. Code metrics were also used as features with machine learning techniques to build prediction models of SCIA. Abdi et al. in [28] proposed prediction models built using object-oriented coupling metrics and several classification algorithms.

To address the lack of code metrics in particular and all source code-based techniques in general, in capturing some aspects of dependencies and coupling between the software modules, researchers have leveraged external sources of information in SCIA and CCP. Significant parts of these works have investigated the usefulness of using changelogs provided by version-controlling systems (VCS) to capture co-change patterns between software modules and also combine them with information extracted from the source code to propose hybrid approaches that improve accuracy. Researchers in [29] found that contextual information, such as commit metadata and developer communication, can significantly enhance prediction models and reduce false recommendations. Similarly, researchers in [30] reviewed hybrid techniques for SCIA, highlighting their potential to enhance accuracy. These studies and other such [31], [32] that leveraged hybrid approaches to change-prone prediction collectively underscore the potential of hybrid approaches in improving the accuracy of software co-change prediction.

The success of deep learning in the field of natural language processing especially neural model-based word embedding techniques such as word2vec [33], FastText [7] and BERT [34] that provide text vector representations; has motivated researchers in the field of software engineering to explore similar techniques for source code analysis. Inspired by word2vec, researchers in [35] have proposed code2vec a source code representation model that is based on the notion of AST paths and the attention mechanism. This model aims to transform source code into numerical vectors that preserve semantic and syntactic characteristics of the code, enabling deep learning models to effectively analyze and classify code for various software engineering tasks such as semantic labelling of code snippets, captioning a block of code, generating code to complete a missing piece of a larger program and defect prediction. In the field of co-change prediction, a promising model, FCP2Vec, inspired of Code2vec is proposed in [5]. FCP2Vec (File-level Change Propagation to Vector) aims to represent file names in co-change instances extracted from the changelogs of a software project to vector representations that preserve co-change patterns. Researchers proposed a recommendation system based on unsupervised nearest neighbors that can suggest for developers the source files that may need modification based on the file being presently worked on.

The drawbacks of this approach lie in two key aspects: firstly, it is based on word2vec embeddings, which cannot leverage the rich information offered by the naming conventions employed in software projects. It only captures semantic information induced by the context of each file in its co-change context. Secondly, the proposed recommendation system is limited only to changelogs data and does not consider the source code of the software entities (packages, files, or classes), missing very important information related to software attributes (complexity and coupling) that can give insight into the likelihood of co-change occurrences. In this paper, we proposed a hybrid approach that aims to improve co-change prediction by addressing these drawbacks.

III. METHODOLOGY

In this section, we present a hybrid approach to predict software co-changes, which combines the strengths of both machine learning and software engineering techniques. Our proposed approach aims to address the research gap identified in the previous section, by providing a more accurate and effective way of predicting co-changes in software systems.

A. APPROACH OVERVIEW

In this work, we propose a novel approach to software co-change prediction that leverages both changelogs and source code information to enhance prediction accuracy at the file level. Our approach is designed to be implemented as a plugin in an Integrated Development Environment (IDE) or as a standalone tool that can help in change management tasks such as change impact and propagation analysis.

Our approach builds upon the previous work of Lee and Hong [4], [5], who treated the co-change prediction problem as a recommendation system problem. In this context, the goal is to recommend the top K elements (source files) that are likely to co-change with the currently edited source file (the query element). The key contribution of our work lies in the use of both changelogs and source code information to improve co-change prediction performances. We extract co-change instances from changelogs, which provide valuable insights into source files that tend to change together. We then harness this information using a char n-gram embedding technique that captures relationships induced by the context within co-change instances, as well as insightful information related to files' sub-names induced by naming conventions. This technique avoids the problem of Out-of-Vocabulary (OOV) and enables the model to capture nuanced relationships between files. In addition to leveraging changelogs, our approach integrates code metrics to improve prediction accuracy. Code attributes such as coupling and complexity have been proven to be correlated with co-change, and integrating them into the prediction task can provide valuable insights into the relationships between software modules. By incorporating code metrics into our model, we can quantify these attributes and improve the accuracy of our predictions.

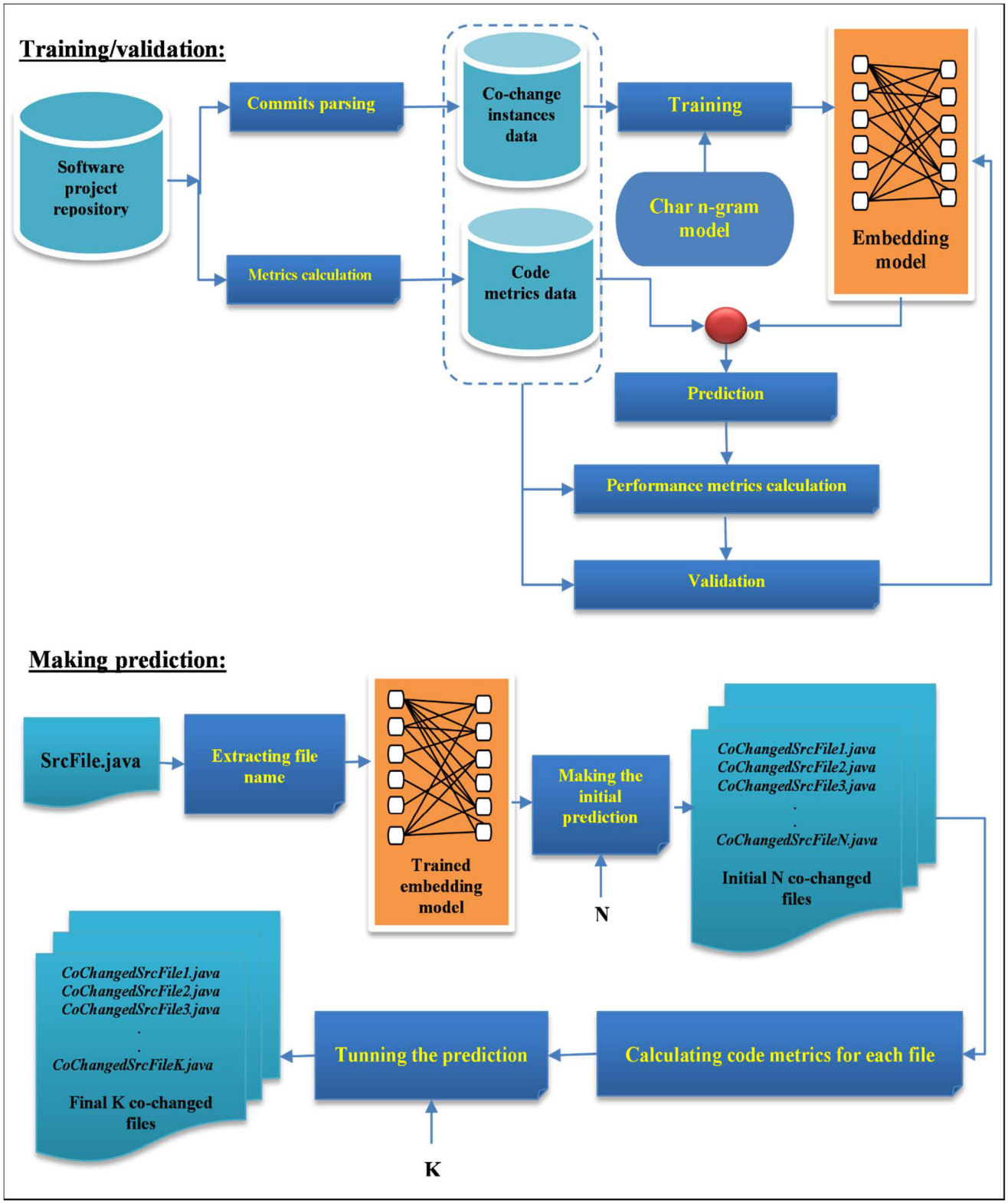

FIGURE 2. Overview of our proposed approach for co-change prediction in software systems. (N is the number of filenames to be selected in the first step, and K is the number of final filenames recommended to developers for co-change).

Figure 2 shows how the approach leverages both changelogs and source code information to generate recommendations for co-changed files. The input to the system is a source file under edition, and the output is a list of recommended files that are likely to co-change with the edited file. Overall, our approach offers a novel and effective solution to

the software co-change prediction problem, leveraging both changelogs and source code information to improve prediction accuracy at the file level. By integrating code metrics into the model, we can provide valuable insights into the relationships between software modules and enhance the accuracy of our predictions. The approach consists of several steps, which are described in detail in the following sub-sections.

B. PREDICTION

In our approach, the process of co-change prediction involves two steps: initial prediction and prediction refinement.

During the first step, we utilize a char n-gram embedding model to generate the initial N (where N > K) co-change file names (a list of N file names) based on the query file name. This capability stems from the model's capacity to suggest file names whose vector representations closely match that of the query file name or the vectors of its sub-tokens. In [5], the authors utilized the K ranking score using unsupervised nearest neighbor (UNN) [36]. They adopted a uniform interface for three different neighbor algorithms: ball tree, KDT tree, and brute force. In our approach, we opted for a simple ranking based on cosine similarity [37], which avoids additional computational load without compromising performance.

This is achieved by following these sub-steps:

1. **Vector representation:** retrieve the vector representation for the query file's name from the model's learned embeddings,
2. **Cosine similarity calculation:** calculate the cosine similarity between the vector representation of the query file name and all other file names in the model vocabulary,
3. **Sorting:** sort the file names based on their cosine similarity scores in descending order,
4. **Returning Top-N closely file names:** returns the top-N file names (co-change file names) with the highest cosine scores as the query file name.

In the second step, we refine the first prediction and generate the final K co-change file names. This is achieved by incorporating code metrics calculated from the first N source files selected in the first step. This integration involves recalculating the similarity scores and reordering the initial N files predicted by the embedding model, to select the top K co-change files (Algorithm 1).

C. DATA PREPARATION

As with any data-driven approach, to train and validate our framework, we need to collect and prepare data from the repositories of the studied software projects.

The final used data consists of two parts, changelogs data and code metrics data. The changelogs data is used to train the char n-gram embedding model which makes the first predictions and the metrics data is used to refine the first predictions to generate the final predictions. In the following subsections, we describe each data and the processes of collecting and preparing them.

Algorithm 1 Prediction;

```
Inputs:
    QueryFileName // the file under edition
    EmbeddingModelPath
    N,K
Outputs:
    FinalPrediction // list of file names with hybrid similarity scores
Begin
  //First step
  Model = LoadModel ( EmbeddingModelPath );
  FirstPrediction = Model.GetTopNSimilar(QueryFileName, N );
  //Second Step
  Foreach FileName, SimilarityScore In FirstPrediction Do
    Metrics = CalculateMetrics(FileName);
    HybridSimilarityScores [ FileName] = Mean ( Metrics) +
    SimilarityScore;
  End Foreach;
  Sort ( HybridSimilarityScores );
  For i =0 To K Do// select the first k file names
    FinalPrediction.Add ( HybridSimilarityScores [i]);
  End For;
  Return FinalPrediction ;
End.
```

1) CHANGELOGS DATA

Each instance in the changelog data consists of the file names of the co-changed source files in a commit. Algorithm 2, shows the steps followed to prepare this data.

Algorithm 2 ChangeLogDataPreparation;

```
Inputs:
    RepositoryURL
Outputs:
    CoChangeInstancesList
Begin
  Repo = CloneRepository (RepositoryURL);
  CommitsList = ExtractAllCommit(Repo);
  Foreach Commit In CommitsList Do
    ChangedFiles = getCommitChangedFiles(Commit);
    If ( 50 < Count (ChangedFiles) < 2) Then
      Continue; // ignore commit
    Else
      Clear(CoChangeInstance);
      Foreach ChangedFile In ChangedFiles Do
        FileName = ExtractFileName (ChangedFile);
        CoChangeInstance + = FileName + ' ';
      End Foreach;
    CoChangeInstancesList.Append (CoChangeInstance);
  End Foreach;
  Return CoChangeInstancesList;
End.
```

After cloning the repository, we parse it to extract the list of all commits done on the project and filter them, as recommended in [5] by removing commits that involve less than two source files and commits that involve more than 50 source files. After that, we form the co-change instances by extracting file names of the co-changed files from each commit.

```
1  Generated.java GeneratedClass.java GeneratedClassTests.java
2  GeneratedClassesTests.java InMemoryGeneratedFilesTests.java
3  LazyConnectionDataSourceProxy.java DataSourceTransactionManagerTests.java
4  DefaultUriBuilderFactory.java UriBuilder.java UriComponentsBuilder.java DefaultUriBuilderFactoryTests.java DefaultWebClient.java WebCl
5  AbstractHandlerExceptionResolver.java ExceptionHandlerExceptionResolverTests.java
6  ServletCookieValueMethodArgumentResolver.java ServletCookieValueMethodArgumentResolverTests.java
7  ApplicationContextExpressionBenchmark.java ContextNamespaceHandlerTests.java ApplicationContextExpressionTests.java ResourceEditorTest
8  EvaluationTests.java SpelDocumentationTests.java Inventor.java
9  ReactorNetty2ServerHttpRequest.java ReactorServerHttpRequest.java CookieIntegrationTests.java
10 ResourceArrayPropertyEditor.java ResourceArrayPropertyEditorTests.java
11 ResourceArrayPropertyEditor.java ResourceArrayPropertyEditorTests.java
12 Jackson2Tokenizer.java Jackson2TokenizerTests.java
13 MultipartParser.java DefaultPartHttpMessageReaderTests.java MultipartRouterFunctionIntegrationTests.java
14 AnnotatedTypeMetadata.java AnnotationMetadataTests.java
15 AnnotationConfigUtils.java ConfigurationClassParser.java ComponentScanAnnotationIntegrationTests.java AnnotatedTypeMetadata.java
16 ComposablePointcutTests.java MethodMatchersTests.java AbstractCacheAnnotationTests.java BeanDefinitionTests.java CollectionsWithDefaul
17 ScopedProxyAutowireTests.java LazyInitTargetSourceTests.java AbstractPropertyAccessorTests.java BeanWrapperEnumTests.java BeanWrapperC
18 BeanFactoryUtilsTests.java InjectAnnotationBeanPostProcessorTests.java CustomScopeConfigurerTests.java PropertyPathFactoryBeanTests.ja
19 BeanWrapperGenericsTests.java AotServicesTests.java BeanDefinitionPropertiesCodeGeneratorTests.java PropertyPathFactoryBeanTests.java
```

FIGURE 3. Example of the final changelogs data.

A portion of 19 co-change instances from the final changelogs data used in the experimental evaluation is shown in the Figure 3.

2) CODE METRICS DATA

Each instance in the code metrics data consists of a source file name and the values of the used code metrics. Any file-level, class-level or method-level code metrics can be used. Since we deal with co-changes in file level, the code metrics for a file can be calculated as follows: the method-level or class-level metrics for all the methods or classes in a file are aggregated to compose the final metrics of the file. In this study we evaluated the metrics described in the Experiments section.

3) DATA PREPROCESSING

When feeding changelog data to the char n-gram embedding model for training, a simple data preprocessing step is required. This step consists of removing the file extensions (e.g., .java) from file names. We remove extensions because they can be considered stop words, similar to words like 'of,' 'the,' and 'and' in natural text classification problems. Additionally, since the code metrics data are combined with the similarity scores returned by the char n-gram embedding model (which range from 0 to 1) to calculate the hybrid similarity score (as shown in Algorithm 1), a normalization preprocessing step is applied to the metrics data.

D. PERFORMANCE EVALUATION

To assess the effectiveness of our co-change prediction approach, we employ two well-established metrics commonly used for evaluating top-K recommendation systems: Hit Ratio (HR) and Normalized Discounted Cumulative Gain (NDCG). These metrics are particularly relevant in this context as our approach aims to recommend the most likely co-changing source files (relevant items) at the top of a ranked list for a given query file (currently edited file). [38], [39]:

- Hit Ratio (HR@K): Measures the proportion of queries where the truly co-changing file appears within the top K recommendations. A higher HR@K indicates that our approach successfully identifies relevant co-changing files within the top K positions of the recommendation list. The HR@K can be calculated by the following formula,
- Normalized Discounted Cumulative Gain (NDCG@K): This metric not only considers whether the relevant co-changing file is present in the top K recommendations but also takes into account its ranking position. Relevant files ranked higher in the list contribute more to the overall score, emphasizing the importance of prioritizing the most likely co-changing files. A higher NDCG@K reflects our approach's ability to not only identify relevant files but also position them prominently within the recommendation list for developers.

HT@K is calculated by the formula (1) where NH@K (number of hits @ K) represents the number of source file names present in each top-K list and TNI (total number of test items) represents the total number of source file names in each test list (test instances in the dataset).

$$HR@K = \frac{NH@K}{TNI} \tag{1}$$

NDCG@k is calculated by the formula (2) where rs_i represents the relevance score which can be binary (1 for truly co-changing files, 0 for others). The logarithmic term ($\log2(i + 1)$) in the formula applies a discounting factor that diminishes the contribution of lower-ranked items, further emphasizing the importance of placing highly relevant

co-changing files at the top of the recommendation list [39].

$$NDCG@k = \sum_{i=1}^{k} \left(\frac{2^{rs_i} - 1}{\log_2(i+1)}\right) \quad (2)$$

IV. EXPERIMENTS

To carry out the experimental evaluation of our approach, we prepared a dataset following the data preparation described in the previous section. We used three well-known and open source Java projects available on GitHub. Two of them are already used in related work. Spring Framework is a comprehensive framework for building enterprise Java applications, offering features such as dependency injection and aspect-oriented programming to streamline development processes. Elasticsearch, an open-source distributed search engine, provides powerful indexing and search capabilities, making it ideal for large-scale data analysis and search applications. Apache Cassandra, a distributed NoSQL database management system, excels in handling vast amounts of data across multiple nodes with high availability and fault tolerance. These projects represent diverse domains within the Java ecosystem, offering valuable insights into the effectiveness and applicability of the proposed co-change prediction approach across different software development contexts. Table 1 shows some statistics about the dataset used in the experimental evaluation.

TABLE 1. Statistics of the dataset (size of software projects in terms of total number of commits and the extracted co-change instances).

Project	# total commits	# extracted co-change instances
Spring Framework	28732	13148
Elasticsearch	74173	31581
Apache Cassandra	29663	5041

A. EVALUATION PROTOCOL

To accurately evaluate the performances of our approach, in our experiments, we adopted a cross-validation protocol. For each co-change instance in the dataset, we select one file name as the query item, and the remaining file names are used as recommendations. This process is repeated so that all file names in each co-change are used as the query item and also as part of the recommendation. The final performance results are then calculated by averaging the results of all the iterations [40].

B. SETUP AND PARAMETER TUNING

To implement all operations related to the experimental evaluation, we used the following Python libraries:

- Char n-gram embedding: we utilized the Facebook FastText [7] implementation provided in Gensim version 4.3.2,
- Evaluation metrics: we employed the implementation of the HR and the NDCG provided in scikit-learn version 1.4.1,

For code metrics calculation, we used the CK open-source tool [41]. CK allows the calculation of 35 class-level and method-level code metrics in Java projects (refer to the CK reference for the complete list). As mentioned before, since we work at the file-granularity level, a metric for a file is calculated by summing the values of that metric across all the file's methods/classes. To select the metrics relevant to our task, we implemented a custom selection method and experimentally tuned the list of metrics to use. The final metrics evaluated in our experiments are shown in Table 2.

TABLE 2. Code metrics used in the experimental evaluation.

Metric	Description
FAN-IN	Counts the number of input dependencies for a class
FAN-OUT	Counts the number of output dependencies for a class
LOC (Lines of code)	Counts the total lines of code (ignoring empty lines and lines of comments)
WMC (Weight Method Class)	Counts the number of branch instructions in a class (McCabe's complexity)
Quantity of comparisons	Quantify the comparison by counting comparison operators such as == and !=
Quantity of parenthesized expressions	Counts the number of expressions inside parenthesis
RFC (Response for a Class)	Counts the number of unique method invocations in a class

When training the embedding models and to mitigate extensive computational loads and time required for automatic parameter tuning techniques such as grid search, random search, and Bayesian optimization, we experimentally tuned our hyperparameters. We initiated this process with initial values recommended in related work specifically in [5]. For hyperparameters specific to char n-gram embedding, such as the minimum size (min_n) and the maximum size (max_n) of sub-tokens, we commenced with default values. The final values used in our experiments are the following: vector size = 37, window size = 12, min_n = 3, max_n = 20, and epochs = 1000. All the experiments and related operations are caried out on a laptop with the following characteristics: i7-4500U CPU, 8.00 GB of RAM and Windows 10 OS.

V. RESULTS AND DISCUSSION

This section presents and discusses the results obtained from the experimental evaluation designed to validate our co-change prediction approach. We focus on the impact of two key adjustable parameters (N and K) on the effectiveness of our approach, as measured by Hit Ratio at K (HR@K) and Normalized Discounted Cumulative Gain at K (NDCG@K), where higher values indicate better performance. As a reminder, our approach can be implemented to operate as an IDE plugin and predicts co-changing source files (recommendations) based on the currently edited file

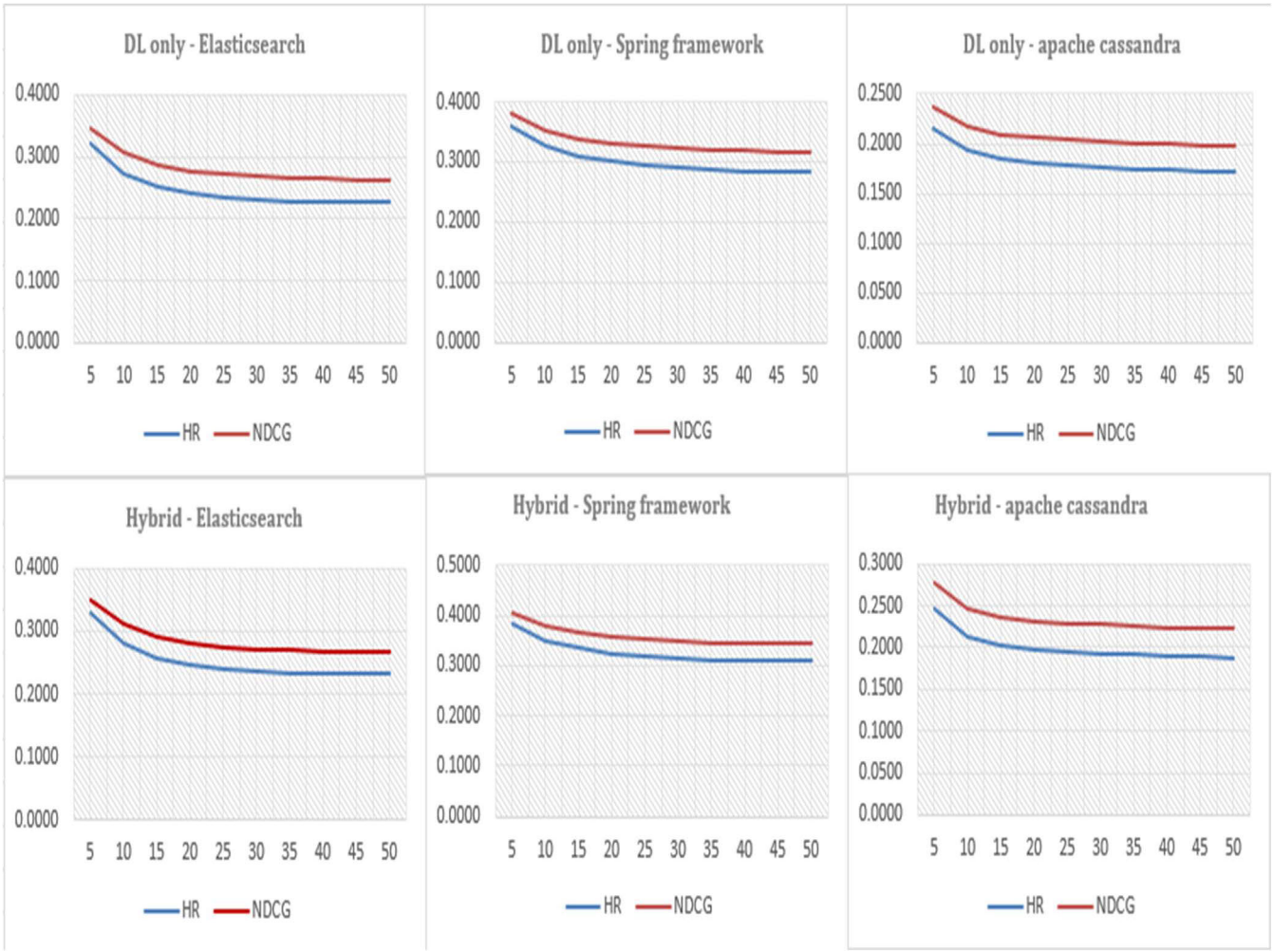

FIGURE 4. Performance in terms of HR@K and NDCG@K vs. length of final prediction (K filenames).

(query element). The prediction process involves two steps: an initial selection of N source file names based on the similarity score calculated from the vector representations obtained by the DL model, followed by refinement using code metrics to generate the final K recommendations from the initial pool of N candidate source files.

A. INFLUENCE OF K (NUMBER OF CO-CHANGE FILE NAMES TO CONSIDER)

In the evaluation of recommendation systems, the selection of K (the number of recommendations to consider) is influenced by various factors, including the specific characteristics of the dataset and the objectives of the recommendation system. In many research studies, the value of 10 is commonly chosen, with results reported in terms of HR@10 and NDCG@10. This practical choice reflects the typical number of recommendations presented to users in real-world scenarios.

To understand how performance varies with K, we conducted evaluations of our approach using different values of K. In this series of experiments, the parameter N is set to 80, a value that yielded promising results in our preliminary investigations. The results are shown in Figure 4. As can be seen, the best performance in both HR@K and NDCG@K indicators was observed when K was set to 5 and 10. Notably, performance consistently declined with increasing values of K across all three studied projects: Spring Framework, Elasticsearch, and Apache Cassandra. Additionally, it is evident that the hybrid approach, incorporating code metrics, consistently outperforms the DL-only approach in terms of both HR@K and NDCG@K across all scenarios.

As expected, we observed a notable correlation between the performance of our approach and the size of training data (number of co-change instances. Specifically, we found that the performance tended to improve with the size of the project. Indeed, performances in the cases of Spring Framework and Elasticsearch were better than performances in the case of Apache Cassandra which have the smallest number of co-change instances. This can be interpreted by the fact that char n-gram embedding, in particular, and all neural and DL-based approaches, in general, tend to perform better with cases that involve more training data than cases with less training data.

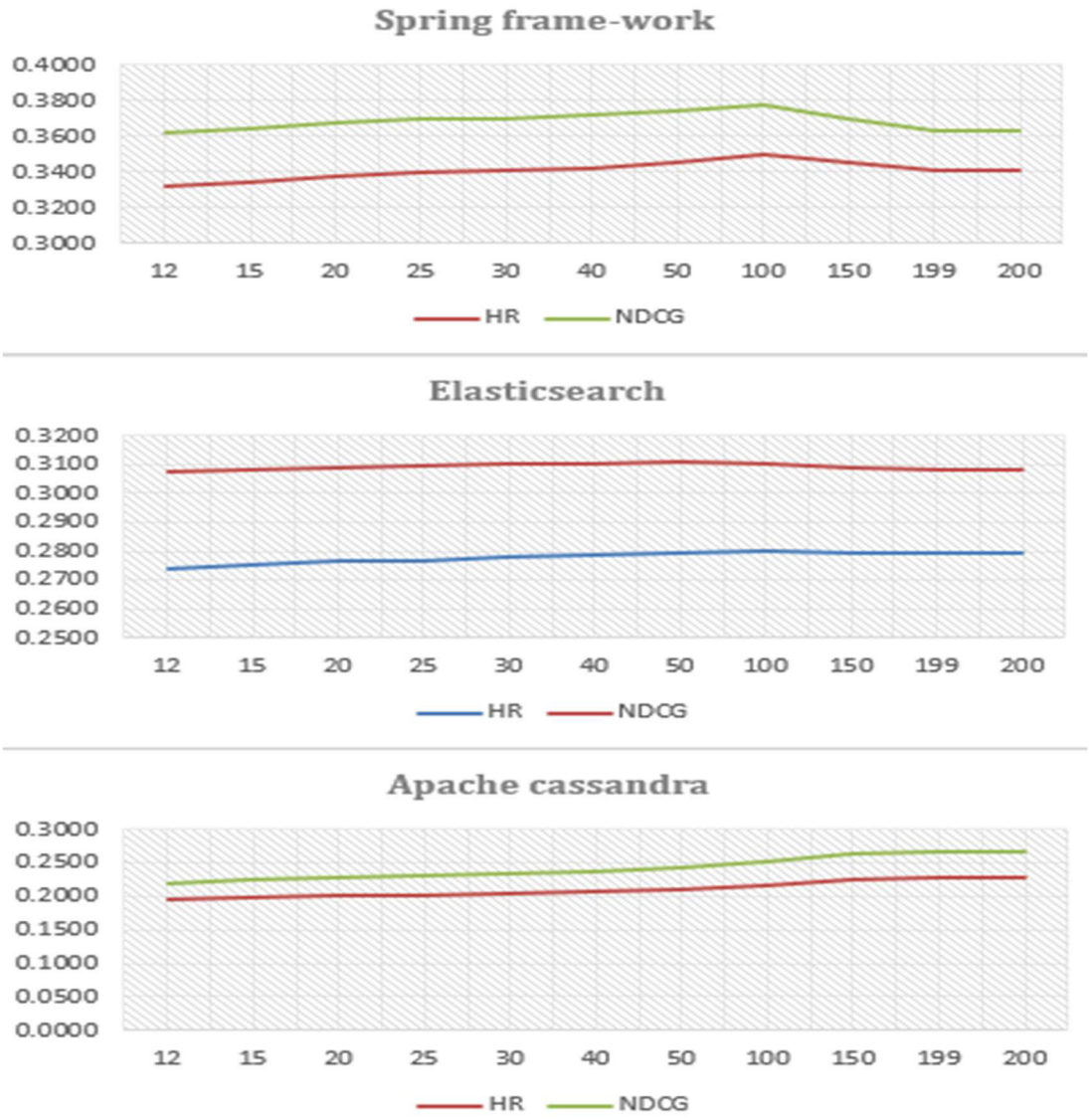

FIGURE 5. Performance in terms of HR@10 and NDCG@10 vs. length of first prediction (N filenames).

B. INFLUENCE OF N (LENGTH OF THE FIRST PREDICTION)

To investigate the influence of the initial prediction length (N) on the final prediction accuracy, we conducted experiments using various N values. We fixed K at 10, which is a commonly used value in related work. The results in terms of HR@10 and NDCG@10 are summarized in Figure 5.

As shown in Figure 5, we observe a consistent improvement in both HR@10 and NDCG@10 for the Spring framework and Elasticsearch projects as the initial prediction length (N) increases from 12 to 100. This indicates that considering a larger pool of candidate files in the initial prediction stage leads to more relevant and better-ranked recommendations. However, there seems to be a diminishing return beyond N=100, with both HR@10 and NDCG@10 showing a slight decline. This suggests an optimal range for N where the approach achieves its best performance. In our case, the range between 80 and 120 appears to be critical, as both HR@10 and NDCG@10 peak within this range. An interesting observation is the behaviour for the Apache Cassandra project (small-scale). Unlike the other two projects, the performance metrics for Apache Cassandra continue to improve even beyond N=120. This suggests that the Apache Cassandra project, with less training data, might benefit from a larger initial prediction length to achieve best performances. This highlights the potential impact of the training data size on the effectiveness of the chosen initial prediction length and we can conclude that when the training data is small a higher value of N is suggested.

C. COMPARISON WITH RELATED WORK

As mentioned earlier (see approach overview for more details), this study addresses co-change prediction as a

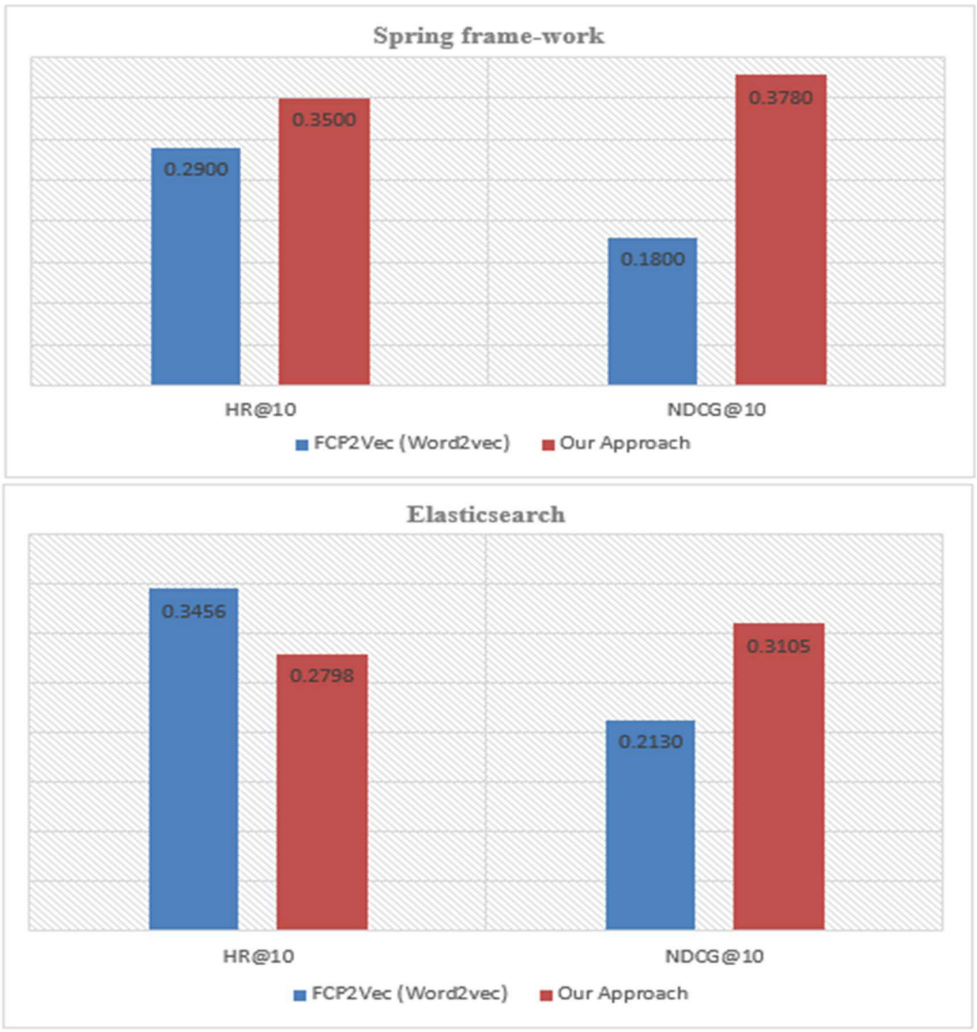

FIGURE 6. Comparison with baseline method.

recommendation system problem. Our hybrid approach draws inspiration from the work presented in [5]. However, our work offers key advancements by integrating code metrics and leveraging co-change data through a char n-gram embedding technique. For a more comprehensive understanding of the improvements introduced by our approach, we compared it directly with [5]. We ensured a fair comparison by using the same software projects (Spring framework and Elasticsearch), performance indicators (HR@10 and NDCG@10), and granularity level (file-level) as in the baseline method [5]. As shown in Figure 6, our approach outperforms FCP2vec in terms of NDCG for both projects: Spring framework and Elasticsearch. Specifically, our approach achieves an NDCG improvement of 19% for the Spring framework and 10% for Elasticsearch. A higher NDCG indicates that our approach prioritizes the most relevant co-changing files at the top of the recommendation list, potentially due to the combined influence of code metrics that quantify characteristics of the source code that might influence co-change behaviour and the char n-gram embedding technique which captures file name semantics by considering character n-grams and potential naming conventions that can reveal relationships between co-changing source files. Interestingly, while our approach achieves a significant improvement in NDCG for both projects, the Hit Ratio (HR) results show a different trend. In the case of Spring framework project, our approach surpasses the baseline method in HR as well, indicating a higher likelihood of recommending at least one truly co-changing file at the top of the recommendation list. However, in the case of the Elasticsearch project, the baseline method outperforms our approach in HR. This contrasting outcome for HR between project sizes is in line with what we found earlier about the performance tended to improve with the size of the project. It's important to remember that HR focuses solely on whether the top recommendation is a true co-change, while NDCG considers the ranking quality of all recommendations. Our approach's consistent improvement in NDCG suggests it prioritizes the most relevant co-changing files throughout the recommendation list, even if the very top recommendation might occasionally

differ between the approaches in the Elasticsearch project. Given the importance of NDCG in capturing the overall quality of recommendations, our approach demonstrates a more robust performance across all scenarios.

D. LIMITATIONS AND OPEN RESEARCH PROBLEMS

We are aware that our work may have the following limitations that represent an interesting open research problem to be addressed in future works:

- Programing Language Applicability: While our evaluation focused on Java projects, a potential limitation of the current implementation is its restriction to a single programming language. However, it's important to note that our approach is built upon language-agnostic aspects. Changelogs data, which tracks file modifications, and code metrics, which can be calculated using established tools, are not inherently tied to specific programming languages. This suggests that the core principles of our approach could potentially be extended to support co-change prediction in projects written in different languages. Future work could explore the generalizability of the approach by evaluating its effectiveness on software projects developed with various programming languages beyond Java,
- Advanced DL Models Exploration: Char n-gram embedding model, which can be considered a shallow neural network architecture. While effective in our approach (no need to huge computing power and training data), some researchers argue that shallow architectures may not be considered as full DL models. To potentially enhance even more the effectiveness of co-change prediction, future work could investigate the integration of more advanced DL models such as: RNNs which are well-suited for sequential data and Transformers for their ability to model long-range dependencies and relationships between elements which could be particularly beneficial for capturing intricate patterns within changelogs data.

VI. CONCLUSION

In this work, we tackled the complex challenge of co-change prediction in software engineering, proposing a novel hybrid approach that combines the strengths of both changelogs data and code metrics. Our two-step prediction process leverages Char n-gram embedding to generate initial predictions, which are then refined using code metrics to provide accurate co-change recommendations.

Our experimental evaluation demonstrated the effectiveness of our approach, achieving significant improvements in co-change prediction accuracy compared to a baseline method. The integration of code metrics into the prediction refinement stage proved to be a key factor in enhancing the accuracy of our co-change predictions, providing valuable insights into the relationships between software modules.

The final K co-change file names generated by our approach have the potential to support various software engineering tasks, such as change impact analysis and code review, and can be used to improve the efficiency and effectiveness of software development processes.

While our approach has shown promising results, there are still some limitations and open research problems that need to be addressed. Future research works could focus on exploring new techniques to improve the accuracy of co-change predictions, such as incorporating additional data sources or developing more sophisticated machine learning models. Moreover, investigating the applicability of our approach to different software systems and evaluating its impact on software maintenance and evolution would be interesting avenues for further research.

Our work represents a significant step forward in the field of co-change prediction, demonstrating the potential of hybrid approaches that combine the strengths of different data sources and techniques. We believe that our findings will inspire and support further research in this area, ultimately leading to the development of more effective and efficient software engineering tools and techniques.

ACKNOWLEDGMENT

The authors would like to acknowledge the support of Prince Sultan University for paying the article processing charges (APC) of this publication. They also thank the anonymous reviewers for their insightful comments, which have helped them to improve the quality of this work.

REFERENCES

[1] A. Agrawal and R. K. Singh, "Predicting co-change probability in software applications using historical metadata," *IET Softw.*, vol. 14, no. 7, pp. 739–747, Dec. 2020, doi: 10.1049/iet-sen.2019.0368.

[2] W. Jin, D. Zhong, Y. Cai, R. Kazman, and T. Liu, "Evaluating the impact of possible dependencies on architecture-level maintainability," *IEEE Trans. Softw. Eng.*, vol. 49, no. 3, pp. 1064–1085, Mar. 2023, doi: 10.1109/TSE.2022.3171288.

[3] Y. Huang, Z. Tang, X. Chen, and X. Zhou, "Towards automatically identifying the co-change of production and test code," *Softw. Test., Verification Rel.*, vol. 34, no. 3, Jan. 2024, doi: 10.1002/stvr.1870.

[4] J. Lee and Y. S. Hong, "Data-driven prediction of change propagation using dependency network," *Eng. Appl. Artif. Intell.*, vol. 70, pp. 149–158, Apr. 2018, doi: 10.1016/j.engappai.2018.02.001.

[5] H. A. Ahmed and J. Lee, "FCP2 Vec: Deep learning-based approach to software change prediction by learning co-changing patterns from changelogs," *Appl. Sci.*, vol. 13, no. 11, p. 6453, May 2023, doi: 10.3390/app13116453.

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. 1st Int. Conf. Learn. Represent. ICLR Workshop Track*, 2013.

[7] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, Dec. 2017, doi: 10.1162/tacl_a_00051.

[8] S. Lehnert. (2011). *A Review of Software Change Impact Analysis*. [Online]. Available: https://api.semanticscholar.org/CorpusID

[9] M. Kretsou, E.-M. Arvanitou, A. Ampatzoglou, I. Deligiannis, and V. C. Gerogiannis, "Change impact analysis: A systematic mapping study," *J. Syst. Softw.*, vol. 174, Apr. 2021, Art. no. 110892, doi: 10.1016/j.jss.2020.110892.

[10] V. Singh, V. Bhattacherjee, and S. Bhattacharjee, "An analysis of dependency of coupling on software defects," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, no. 1, pp. 1–6, Jan. 2012, doi: 10.1145/2088883.2088899.

[11] B. Turhan, A. Bener, and T. Menzies, "Nearest neighbor sampling for cross company defect predictors," in *Proc. Workshop Defects Large Softw. Syst.*, Jul. 2008, p. 26, doi: 10.1145/1390817.1390824.
[12] E. Sreedevi, Y. Prasanth, E. Sreedevi, and Y. Prasanth, "A novel multi-ensemble based feature selection and defect prediction model on software defect projects," *Int. J. Control Theory Appl.*vol. 9, no. 40, pp. 827–836, 2016.
[13] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–14, Dec. 2007, doi: 10.1109/TSE.2007.10.
[14] C. Tantithamthavorn, "A review of process metrics in defect prediction studies," *Methods Appl. Comput. Sci.*, vol. 1, no. 5, pp. 133–145, 2015, doi: 10.1007/978-3-319-26285-7.
[15] B. Turhan and A. Bener, "A multivariate analysis of static code attributes for defect prediction," in *Proc. 7th Int. Conf. Quality Softw. (QSIC)*, 2007, pp. 231–237, doi: 10.1109/qsic.2007.4385500.
[16] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Comput.*, vol. 22, no. S4, pp. 9847–9863, Jul. 2019, doi: 10.1007/s10586-018-1696-z.
[17] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proc. Symp. Bootcamp Sci. Secur.*, Apr. 2015, pp. 1–9, doi: 10.1145/2746194.2746198.
[18] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "LEOPARD: Identifying vulnerable code for vulnerability assessment through program metrics," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 60–71, doi: 10.1109/ICSE.2019.00024.
[19] I. Kalouptsoglou, M. Siavvas, D. Tsoukalas, and D. Kehagias, "Cross-project vulnerability prediction based on software metrics and deep learning," in *Proc. Comput. Sci. Appl. (ICCSA)*, vol. 12252, 2020, pp. 877–893, doi: 10.1007/978-3-030-58811-3_62.
[20] K. Z. Sultana, B. J. Williams, and A. Bosu, "A comparison of nano-patterns vs. software metrics in vulnerability prediction," in *Proc. 25th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2018, pp. 355–364, doi: 10.1109/APSEC.2018.00050.
[21] S. Moshtari and A. Sami, "Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction," in *Proc. 31st Annu. ACM Symp. Appl. Comput.*, Apr. 2016, pp. 1415–1421, doi: 10.1145/2851613.2851777.
[22] M. Zagane, M. K. Abdi, and M. Alenezi, "A new approach to locate software vulnerabilities using code metrics," *Int. J. Softw. Innov.*, vol. 8, no. 3, pp. 82–95, Jul. 2020, doi: 10.4018/ijsi.2020070106.
[23] M. Zagane, M. K. Abdi, and M. Alenezi, "Deep learning for software vulnerabilities detection using code metrics," *IEEE Access*, vol. 8, pp. 74562–74570, 2020, doi: 10.1109/ACCESS.2020.2988557.
[24] M. Zagane and M. K. Abdi, "Evaluating and comparing size, complexity and coupling metrics as web applications vulnerabilities predictors," *Int. J. Inf. Technol. Comput. Sci.*, vol. 11, no. 7, pp. 35–42, Jul. 2019, doi: 10.5815/ijitcs.2019.07.05.
[25] L. C. Briand, J. Wust, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proc. Int. Conf. Softw. Maintenance (ICSM) Softw. Maintenance Bus. Change*, Jun. 1999, pp. 475–482, doi: 10.1109/ICSM.1999.792645.
[26] S. Black, "Computing ripple effect for software maintenance," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 13, no. 4, pp. 263–279, Jul. 2001, doi: 10.1002/smr.233.
[27] G. Woo, H. S. Chae, J. F. Cui, and J.-H. Ji, "Revising cohesion measures by considering the impact of write interactions between class members," *Inf. Softw. Technol.*, vol. 51, no. 2, pp. 405–417, Feb. 2009, doi: 10.1016/j.infsof.2008.05.014.
[28] M. K. Abdi, H. Lounis, and H. Sahraoui, "Predicting change impact in object-oriented applications with Bayesian networks," in *Proc. 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf.*, Jul. 2009, pp. 234–239, doi: 10.1109/COMPSAC.2009.38.
[29] I. S. Wiese, R. Ré, I. Steinmacher, R. T. Kuroda, G. A. Oliva, C. Treude, and M. A. Gerosa, "Using contextual information to predict co-changes," *J. Syst. Softw.*, vol. 128, pp. 220–235, Jun. 2017, doi: 10.1016/j.jss.2016.07.016.
[30] Y. Shakirat, A. Bajeh, T. O. Aro, and K. Adewole, "Improving the accuracy of static source code based software change impact analysis through hybrid techniques: A review," *Int. J. Softw. Eng. Comput. Syst.*, vol. 7, no. 1, pp. 57–66, Feb. 2021, doi: 10.15282/ijsecs.7.1.2021.6.0082.
[31] X. Zhu, Y. He, L. Cheng, X. Jia, and L. Zhu, "Software change-proneness prediction through combination of bagging and resampling methods," *J. Softw., Evol. Process*, vol. 30, no. 12, Dec. 2018, Art. no. e2111, doi: 10.1002/smr.2111.
[32] R. Malhotra and M. Khanna, "An exploratory study for software change prediction in object-oriented systems using hybridized techniques," *Automated Softw. Eng.*, vol. 24, no. 3, pp. 673–717, Sep. 2017, doi: 10.1007/s10515-016-0203-0.
[33] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013, *arXiv:1310.4546*.
[34] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*
[35] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–29, Jan. 2019, doi: 10.1145/3290353.
[36] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 1, pp. 21–27, Jan. 1967, doi: 10.1109/TIT.1967.1053964.
[37] G. Sidorov, A. Gelbukh, H. Gómez-Adorno, and D. Pinto, "Soft similarity and soft cosine measure: Similarity of features in vector space model," *Computación Y Sistemas*, vol. 18, no. 3, pp. 491–504, Sep. 2014, doi: 10.13053/cys-18-3-2043.
[38] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, nos. 1–7, pp. 107–117, Apr. 1998, doi: 10.1016/s0169-7552(98)00110-x.
[39] M. Li, K. Tei, and Y. Fukazawa, "An efficient adaptive attention neural network for social recommendation," *IEEE Access*, vol. 8, pp. 63595–63606, 2020, doi: 10.1109/ACCESS.2020.2984340.
[40] M. Zagane, M. Alenezi, and M. K. Abdi, "Hybrid representation to locate vulnerable lines of code," *Int. J. Softw. Innov.*, vol. 10, no. 1, pp. 1–19, Jan. 2022, doi: 10.4018/ijsi.292020.
[41] M. Aniche, "Java code metrics calculator (CK)," Tech. Rep., 2015.

MOHAMMED ZAGANE received the engineer degree in computer science from the University of Mascara, in 2007, the magister degree in computer science from the Higher School of Computer Science, Algiers, in 2010, and the Ph.D. degree in computer science from Université Oran 1, Oran, Algeria, in October 2020. From 2009 to 2012, he was a Computer Engineer in the administration services of the state of Mascara. He is currently an Associate Professor with the Department of Computer Science, Mustapha Stambouli University, Mascara, Algeria. His research interests include applying machine learning and deep learning techniques to solve challenging problems in software engineering.

MAMDOUH ALENEZI (Member, IEEE) received the master's degree in software engineering from DePaul University and the Ph.D. degree in software engineering from North Dakota State University. He is currently a Distinguished Software Engineering Expert and the Dean of the Quality Assurance and Development, Prince Sultan University. He is also a Full Professor and has made significant contributions to the academic and scientific community. He has also served in various roles, including the Chair of the Computer Science Department, the Chief Information Technology Officer, and the Dean of Educational Services. With a strong focus on software engineering and security. He has authored more than 100 research articles. His research interests include software engineering, digital transformation, higher education, and intelligent transportation. He is a respected figure in the field, known for his exceptional research and professional experience.