applied sciences

Article

FCP2Vec: Deep Learning-Based Approach to Software Change Prediction by Learning Co-Changing Patterns from Changelogs

Hamdi Abdurhman Ahmed and Jihwan Lee *

Major in Industrial Data Science & Engineering, Department of Industrial and Data Engineering, Pukyong National University, Busan 48513, Republic of Korea; hamdi@pukyong.ac.kr
* Correspondence: jihwan@pknu.ac.kr

Abstract: As software systems evolve, they become more complex and larger, creating challenges in predicting change propagation while maintaining system stability and functionality. Existing studies have explored extracting co-change patterns from changelog data using data-driven methods such as dependency networks; however, these approaches suffer from scalability issues and limited focus on high-level abstraction (package level). This article addresses these research gaps by proposing a file-level change propagation to vector (FCP2Vec) approach. FCP2Vec is a recommendation system designed to aid developers by suggesting files that may undergo change propagation subsequently, based on the file being presently worked on. We carried out a case study utilizing three publicly available datasets: Vuze, Spring Framework, and Elasticsearch. These datasets, which consist of open-source Java-based software development changelogs, were extracted from version control systems. Our technique learns the historical development sequence of transactional software changelog data using a skip-gram method with negative sampling and unsupervised nearest neighbors. We validate our approach by analyzing historical data from the software development changelog for more than ten years. Using multiple metrics, such as the normalized discounted cumulative gain at K (NDCG@K) and the hit ratio at K (HR@K), we achieved an average HR@K of 0.34 at the file level and an average HR@K of 0.49 at the package level across the three datasets. These results confirm the effectiveness of the FCP2Vec method in predicting the next change propagation from historical changelog data, addressing the identified research gap, and show a 21% better accuracy than in the previous study at the package level.

Keywords: change propagation; change management; change prediction; neural language model; recommendation

Citation: Ahmed, H.A.; Lee, J. FCP2Vec: Deep Learning-Based Approach to Software Change Prediction by Learning Co-Changing Patterns from Changelogs. *Appl. Sci.* **2023**, *13*, 6453. https://doi.org/10.3390/app13116453

Academic Editor: Vito Conforti

Received: 5 April 2023
Revised: 14 May 2023
Accepted: 23 May 2023
Published: 25 May 2023

1. Introduction

In recent years, software development has become increasingly complex and collaborative, resulting in a growing dependency among elements. These dependencies are interconnected, with one component providing services necessary for another. For example, elements X, Y, and Z may have an interdependence, where Z provides services necessary for Y, and Y provides services necessary for X. Modifying a component can introduce errors and create incompatibilities with other parts of the software. To maintain compatibility, the programmer must fix these inconsistencies, but these fixes can also create new issues. This ripple effect of changes in software elements is often referred to as change propagation (CP) [1,2]. The cost of change propagation in the software industry has been reported in the literature. Unforeseen and uncorrected inconsistencies can lead to software errors [3,4] and, according to estimates, software maintenance costs account for more than 60% of the total cost of a product's life cycle. In addition, ripple effect analysis is responsible for approximately 40% of software maintenance costs [5], so it is important to use tools and analysis techniques to improve the quality of the change propagation process. Therefore, the ability to predict these change propagation effects is crucial for ensuring the stability and reliability of software systems.

Scholars have used data from version control systems (VCSs) to examine the changes that software systems have undergone, which has facilitated the prediction of their characteristics [6,7]. This can aid in identifying hidden dependencies or interactions in the system [8]. Several approaches have been proposed to analyze CP at different abstraction levels using various techniques. For example, a heuristic approach [9] was used to predict a change in one source code element propagating to other elements. Zimmermann et al. [10] developed a stochastic model known as K3B, which predicts how far a set of changes propagates throughout the system [11]. Siavash et al. [12] used the Bayesian belief network as a probabilistic tool to predict possible models of the affected system, given a change in the system at the package level. Oliva et al. [6] analyzed change coupling at the file level. Pan et al. [5] proposed a class coupling network (CCN) model of the structure of software at the class level to evaluate the stability of the software.

Previously, Lee and Hong [13] proposed a method to predict the change probability of software elements given the set of current edited files. To estimate the probability, they utilized a graphical model called a dependency network (DN). A DN provides an approximation method for estimating the joint distribution among random variables. It first learns the local distribution of each software element and combines them using Gibbs sampling for probabilistic inference. Although their approach provides a well-defined model for predicting change propagation, the DN-based approach has two major limitations: scalability issues when handling a large number of files and difficulties in interpreting complex and large models. As a result, the approach is limited to the package level, and its predictions are not specific enough to be useful at the granularity level where changes are implemented.

To address this limitation, we introduce FCP2Vec, which is inspired by the Word2vec algorithm [14,15]. Word2vec is a shallow neural network that utilizes unsupervised learning to learn relationships between words within a sentence. Although not considered as a deep learning technique in the strictest sense, Word2vec shares some similarities with deep learning models, such as its neural network foundation. The neural network learns from a collection of sentences to predict the surrounding words for each word in the sentence. Then, this process results in a set of vector representations for each word in the corpus, where each vector captures the meaning and context of the word, which can be used to perform various natural language processing tasks such as text classification, similarity and analogy tasks, and information retrieval. For example, if two words have similar vector representations, it means that they have similar meanings or contexts. Similarly, our neural network model trains on a collection of changelogs to predict co-changing elements for each element, resulting in a vector representation of software elements that can be used to determine the degree of changeability between software elements. Thus, if a developer modifies some set of elements, then the recommendation system can use this vector representation to suggest other elements that could be changed together.

To validate our study, we used three publicly available datasets: Vuze (formerly Azureus) [16], Spring Framework [17], and Elasticsearch [18]. These datasets, which consist of open-source Java-based software development changelogs, were extracted from two types of VCS. Vuze was extracted from the concurrent version system (CVS), whereas Spring Framework and Elasticsearch were extracted from Git. Vuze is an open-source BitTorrent client that uses the BitTorrent protocol to transfer files. It is written in Java and has over a decade of development history in its changelog data. The Spring Framework is an open-source Java framework that facilitates the development of enterprise-level applications by providing features such as dependency injection, aspect-oriented programming, and support for web applications [17]. Similarly, Elasticsearch is an open-source Java-based distributed search and analytics engine [18]. We have trained the FCP2Vec model using these datasets' changelogs and have shown that FCP2Vec predicts potential changes, providing recommendations based on the current open file and detecting hidden dependencies that cannot be identified through program analysis. Comparative results also indicate that our model is highly scalable and can support a much deeper level of granularity, such as the file level, which was not possible with the dependency network [13]. This approach can help programmers predict the number of possible change propagations and the K files related to their current task, where K is a user-defined constant number.

The remainder of this paper is organized as follows. Section 2 reviews the literature on change propagation related to the software domain review. Section 3 motivates our approach by providing an overview of the problem definition and the formal definition of Word2vec and skip-gram. Section 4 describes our proposed approach. Section 5 presents an empirical case study of Vuze, Spring Framework, and Elasticsearch. Finally, we conclude and anticipate future work and limitations of our approach in Section 6.

2. Materials and Methods

In this section, we review some notable research articles related to change propagation and a general overview of machine learning and deep learning architectures.

Learning methodologies, such as machine and deep learning, have been successfully implemented in diverse fields, including, but not limited to, image recognition, sematic image segmentation, and natural language processing. This success can be attributed to several factors: the growing power of hardware, the expanding amount of data available for training, and recent developments of data in machine learning algorithms. These progressive strides have equipped deep learning methods with the capacity to effectively employ complex, compositional non-linear functions. This enables the automatic learning of distributed and hierarchical features, utilizing both labeled and unlabeled data to their fullest potential [19]. Table 1 offers a comprehensive overview of the latest approaches in this domain and their relative counterparts.

Table 1. Comprehensive overview of machine learning and deep learning in different domains.

Author(s)	Year	Summary
Menghani [20]	2023	Presents a survey of model efficiency in deep learning and the problems therein, moving from modeling methods and infrastructure to hardware, and how hardware directly or indirectly affects the efficiency.
Menghani [20]	2023	Covers a broad range of fairness-enhancing mechanisms and their use to compensate for bias in machine learning.
Salem et al. [21]	2022	Proposes XAI-DL, an interpretable deep learning model for solar-driven distillation systems that provides explanations to help users trust the system.
Lewowski et al. [22]	2022	Proposes a review of machine learning and artificial intelligence methods in automated code smell detection.
Cabrera Lozoya et al. [23]	2021	Proposes and evaluates a transfer learning method to learn how to classify security relevant commits.
Pan et al.	2019	Characterization of software stability via change propagation simulation.
Alon et al. [24]	2019	A neural model for representing code snippets as continuous distributed vectors called "code embeddings". The model can predict semantic properties of the snippet and can predict method names from the vector representation of its body.
Alon et al. [25]	2018	Proposes CODE2SEQ, an alternative seq2seq model for source code that uses attention to focus on the relevant program path in the source code AST to create better source code encodings.

2.1. Leveraging Changelog Data in Change Propagation

Version control is a crucial tool for tracking changes made to a software development project over time. One of the most widely used systems is the concurrent version system (CVS). It creates and maintains a local repository for each project and can push changes to remote or other local repositories as needed. This allows teams to easily undo any changes that may have caused problems. The CVS also allows developers to track changes to files and see who made them. To identify and quantify change coupling, it is necessary to first recover the change history for relevant files. This is often carried out by parsing and analyzing the log of the version control system (VCS). In most cases, it is assumed that if two elements are committed together, they are change coupled. However, older systems such as the CVS that do not support atomic commits may require additional processing to reconstruct change transactions [6]. Git, a renowned distributed version control system, diligently records changes to any assortment of computer files. It is commonly employed to synchronize the endeavors of programmers collectively developing source code for software. Among its objectives are data integrity, speed, and endorsement for distributed, non-linear workflows [26].

Due to the importance of change propagation, research has been carried out to understand, predict, and control how change propagates. Early findings on CP such as in [3] present a model of change propagation during software maintenance and evolution. Impact analysis and change propagation are among the major issues in software change management. Han [27] introduced an approach to provide impact analysis and change propagation support as an integral part of software engineering environments, so that they can be applied during both software development and maintenance. The prediction of such a change provides a significant challenge in the management of the redesign and customization of complex products, where many change propagation paths may be possible. Aryani et al. [28] introduced a method that uses the domain-level behavioral model of a system to analyze change propagation.

Researchers have found that change couplings can be identified and evaluated from the log of a VCS using the following three different methods. First, raw counting. Some researchers use a raw counting method to identify change coupling. This method looks at the relationships between changes made to different parts of a system and typically uses a symmetry matrix or co-change matrix data structure. A co-change matrix is a matrix that shows the relationships between changes to different parts of a system, and change couplings are the degree to which changes to one part of a system are connected to or dependent on changes to another part of the system. There are several ways that co-change matrices can be inferred or constructed. One way is to analyze the version history of a system, such as by examining commit logs or other records of changes made to the system over time. By examining which files were modified in each commit, it is possible to construct a co-change matrix that shows which parts of the system were changed together [29–31].

Second, associated rule. Some researchers, for instance, [10,32–34], used association rule mining. It is a process of finding relationships between items in large datasets. It looks at the frequency of item pairs and then tries to identify patterns that occur frequently. These patterns can be used to make predictions about future events.

Third, time series analysis. Some researchers, for instance, [35–37], applied time series analysis. It is a type of data analysis that studies the patterns and trends of a set of data points over a period of time. Time series analysis is used to identify changes in data points over time and to understand the underlying drivers of such changes. It can be used to forecast future values of data points and analyze the impact of certain events on a set of data points. Time series analysis is a promising approach for dealing with some of the issues associated with rule-based and raw counting algorithms [38].

There is a great deal of research on using historical changes to identify code dependencies in software systems. Various data mining techniques have been employed to predict the change propagation of changes by extracting historical changelogs. Gall et al. [39] uses the history of release of a large software system together with change data to identify potential code restructuring opportunities. Mockus et al. [40] incorporates information on development, historical changes, and other metadata to predict the risk of failure in new development changes. Hassan et al. [9] propose a heuristic to predict the propagation of change at the element level. Zimmermann et al. [10] annotated software versions to extract association rules to guide programmers on further changes to the code. Ying et al. [34] proposed an approach to assist developers with modification tasks by augmenting analyses based on source code and making them more predictable, based on historical change patterns. Finlay et al. [41] used the Hoeffding tree classifier to predict whether a software build will succeed or fail. Sun et al. [42] proposed a framework to analyze the ripple effects of changes proposed to the software and provided a way to evaluate these to predict the impact of the changes.

The study by [4] explores the application of data mining techniques to predict change propagation, applicable to open-source software systems, by focusing on Linux, FreeBSD, and Apache. Due to the complex semantics of change propagation and the indirect nature of previously proposed language techniques, it remains difficult to reason about the efficiency of self-adjusting programs and change propagation.

In some instances, researchers have applied a hybrid approach to address specific challenges. For example, Canfora et al. [38] found that while association rules are often effective, they can fail to capture logical relationships between artifacts that are modified in subsequent change sets. Consequently, the authors proposed a hybrid recommender that combines time series (Granger causality) modeling with association rules to enhance the accuracy of functional change discovery. Kagdi et al. [43] introduced a method for analyzing the impact of changes in software code by integrating techniques from information retrieval and software evolution. Similarly, Ref. [44] developed a relational topic-based coupling approach for classes, using relational topic models to assess the closeness of two elements and demonstrating how to refine object-oriented coupling metrics.

Some of the trends in recent articles to predict change propagation have been to apply a probabilistic approach and other methods at a different level of abstraction. Ferreira et al. [11] introduced K_3B, a probabilistic model to estimate the impact of change propagation, which can be used to estimate the effort required for software maintenance tasks. Siavash et al. [12] used the Bayesian belief network as a probabilistic tool to predict possible models of the affected system, given a change in the system at the package level. Oliva et al. [6] analyzed change coupling at the file level. The authors explored the use of Bayesian networks (BNs) for modeling and analyzing change propagation in software systems to outperform traditional change propagation prediction methods. Similarly, Ref. [13] proposes a dependency network to estimate the probability of change propagation at the package level, which is more effective than a BN in modeling complex systems. Pan et al. [5] proposed a class coupling network (CCN) model of the structure of software at the class level to evaluate the stability of the software.

2.2. Neural Language Model

Data-driven decisions are becoming increasingly important, and natural language processing (NLP) is playing an increasingly vital role. NLP is the process of extracting meaningful information from text. Its application to tasks such as sentiment analysis, information retrieval, language detection, text summarization, and many others has made it an invaluable tool for making decisions based on large amounts of unstructured data. The specific objectives of many NLP applications can all be generalized to calculating the probability of a given sequence of words [45,46].

In NLP, words are the basic unit of meaning and word vectors are used to represent them. The process of mapping words to real vectors is called word embedding. One-hot encoding is a common approach for language modeling, but it can be problematic as it results in a loss of meaning for words [47]. Representing words as vectors, using Word2vec, captures the semantic relationship between words and the context in which they are used. This allows models to better understand the meaning of words and to consider the context in which they are used. Additionally, representing words as vectors also helps reduce the number of features needed in a model [14,15].

Over the past few years, distributed representation has become a popular tool for representing words and phrases in NLP applications. Most of the work in the Word2vec area is based on the distributional hypothesis [48,49], which holds that words that appear in the same context have similar meanings. This assumes that contexts are influential in shaping a word's meaning. However, research has recently extended the use of distributed representation beyond word representation to other types of data. For example, distributed representation can be used to represent images and audio clips [50]. Table 2 highlights some of the versatility of word embedding usage in various domains.

Table 2. Exploring the versatility of word embeddings: some of the usage in various domains.

Categories	Publication
Audio and music analysis	[50,51]
Information extraction	[52]
Recommender systems	[53–56]
Sentiment analysis, text classification, and clustering	[57–60]

From a recommendation perspective, item representation plays a crucial role in the effectiveness of a recommendation system. Systems employing more precise and diverse representations of items are likely to yield more accurate recommendations. Numerous techniques have been proposed for representing items in a manner that takes into account user transactions and item content information, including review text and images. Some notable studies in this domain [61] include the use of co-purchase data to target and personalize ads in the Yahoo Mail advertising system; Vasile et al. [62] proposed a method to leverage the existing metadata of items to make better recommendations; and [55] introduces an approach for constructing product embeddings that uses several modality-specific features, such as text description and images.

In the context of constructing a machine learning model, the capacity to enhance its performance is key. The calibration of the model's hyperparameters can wield a considerable influence on its precision. Default hyperparameters are employed to streamline a model for a particular set of problems or data, frequently being selected by the machine learning algorithm or adjusted by the model developer. However, when the problem domain extends beyond the initial scope, the circumstances may deviate. For example, Caselles-Dupré et al. [63] implement the skip-gram with a negative sampling technique, a variant of Word2Vec, to generate item embeddings for recommendation systems. The authors scrutinize the significance of hyperparameters in a recommendation context and ascertain that the optimization of previously disregarded hyperparameters can markedly enhance performance. Additionally, they discover that the optimal hyperparameters for natural language processing and recommendation tasks differ. This methodology has been employed to produce item embeddings that have proven successful in the recommendation domain, yielding substantially improved performance and elevated recommendation accuracy. In this study, we have also adopted a similar conjecture to fine-tune the change propagation setting.

3. Background

3.1. Problem Definition

This paper aims to address the problem of software change propagation, which pertains to the process of identifying and monitoring the influence of software changes on other components of a software system. When a software system undergoes modifications, it is crucial to recognize and comprehend how those changes affect other parts of the system where changes in one part of the can have significant impacts on other parts, potentially causing bugs or errors. For example, as shown in Figure 1, there is a list of files that have changed together in the past grouped as a single *transaction* or co-change. Let us consider that the developer ("*Dev1*") changes files "a.java" and "b.java" together, so they are called a co-change or transaction or session, and we use them to train in predicting the probabilities of CP.

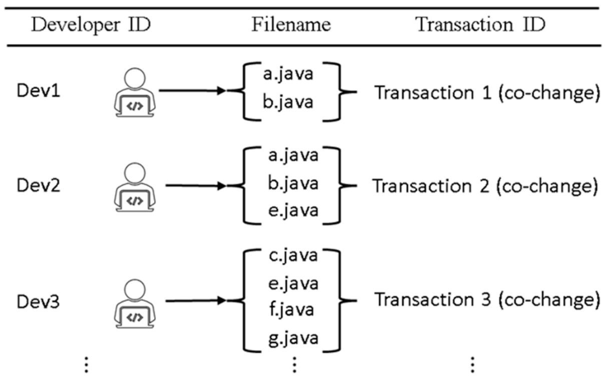


Figure 1. An example of a historical changelog.

To refer to a module of the system at any given level of abstraction, throughout this work, we will use the term "element". The scope of an element within the system is determined by the level of granularity. An element can be defined as a conceptual component at different levels, such as a package (directory), subpacket (subdirectory), files such as classes, interfaces, and so on. The Java package is a collection of similar subpackages, interfaces, and classes. A subpackage is a package that is defined inside another package. This is carried out to make the structure of the packages more generic and organized. For example, if one has a package named "University 1" and a subpackage "department 1", all the supporting Java files will be defined inside it.

Although co-change is not the same as CP, it can be helpful in predicting it because if elements have changed together in the past, it is more likely that they will change together in the future, and changes are likely to propagate among those elements. In this case study, we opt to define the element at the file level as a file name. To predict future CP and use it as an element recommendation, we will use each transaction pattern in the changelog obtained from the CVS as input for the proposed model and predict the next element that most likely needs to be changed located nearby in the vector space.

Speaking more formally, given a set S of element changelogs obtained from the CVS, where different operations (create, read, update, and delete) are applied by N developers, where the developer's changelog s = $(t_1, t_2, \ldots, t_m) \in S$ is defined as continuous sequence of M transactions and each transaction element changelog $t_m = (f_{m1}, f_{m2}, \ldots, f_{mB_m})$ consists of B_m that is operationally modified, our objective is to find a d-dimensional real-valued representation $v_f \in \mathbb{R}^d$ of each element f that is chosen in such a way that co-changed similar elements are found nearby in the vector space.

This probabilistic view of the change propagation that uses the concept of distributional models to learn the vector representation of elements is based on a skip-gram of Word2vec [15]. Word2vec was initially designed for NLP purposes. Here, we investigate and use them as tools in the CP domain. Taking this perspective motivates the development of a probabilistic method to solve the problem.

3.2. Word2vec

Word2vec is one of the most popular word embedding models introduced in [14,15]. It is a technique that maps each word to a fixed-length vector. As illustrated in Figure 2, Word2vec contains two models: the skip-gram (SG) (Figure 2b) and the continuous bag of words (CBOW) (Figure 2a). These models rely on conditional probabilities to generate semantically meaningful representations. In other words, rather than requesting a model to predict the next word, we can ask it to predict the next word based on the surrounding words in a context.

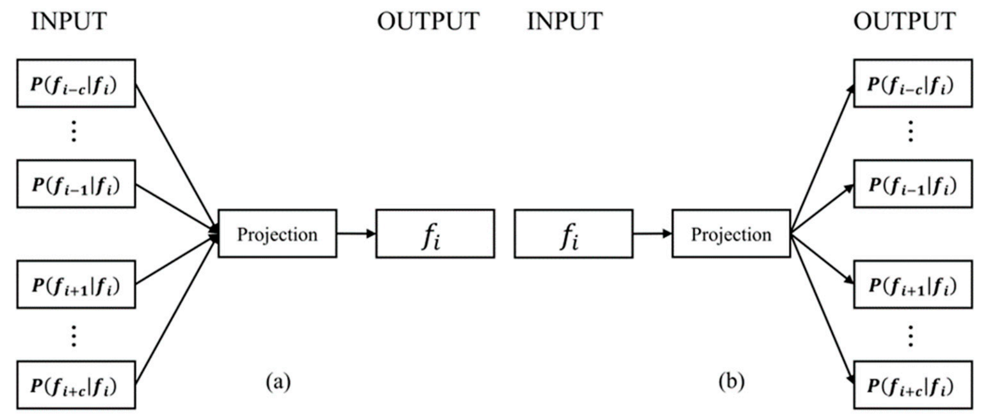


Figure 2. Architecture of the Word2vec training model. (**a**) CBOW and (**b**) SG model architectures.

CBOW is an approach that uses context words to predict a target word. It works by taking the average of the vectors that represent each context word and then passing this vector through a softmax layer to make a prediction. In contrast, the SG model predicts context words from a given target word. This model uses an embedding layer to map each target word into a vector space and then passes this through a softmax layer to make a prediction about the surrounding words.

The CBOW and SG models share some similarities, including the use of training samples consisting of pairs of words chosen based on their proximity to one another, as well as the use of a softmax function as the final layer in the network.

Both the skip-gram and CBOW models in the Word2vec tool are considered as self-supervised learning (SSL) models. SSL [64] is a machine learning process in which the model is trained to learn from input data without needing external labels. This process is also known as predictive learning or pretext learning. Word2vec is similar to an autoencoder in that it encodes each word in a vector. However, rather than training against the input words through reconstruction, as a restricted Boltzmann machine does, Word2vec trains words against other words that neighbor them in the input corpus.

In SSL, the unsupervised problem is transformed into a supervised problem by auto-generating the labels. This makes it possible to use a large amount of unlabeled data. However, it is crucial to set the right learning objectives to obtain supervision from the data themselves. The process of the SSL method is to identify any hidden patterns in the input data. For example, in NLP, if we have a few words, using SSL, we can complete the rest of the sentence. This approach uses the structure of the data to use a variety of supervisory signals across large datasets, without needing labels.

Skip-Gram

The skip-gram model assumes that a word can be used to predict the words that come before and after it in a text sequence. For example, consider the text sequence “we”, “rise”, “by”, “lifting”, “others”. As shown in Figure 3, if we choose “by” as the center word and set the size of the context window to 2, then the SG model considers the conditional probability of generating the context words from the left side of the center word “we” and “rise” and, from the right side, “lifting” and “others”; $P(\text{“}we\text{”}, \text{“}rise\text{”}, \text{“}lifting\text{”}, \text{“}others\text{”} \,|\text{“}by\text{”})$, The “window size” parameter is used to limit the number of words in each context (see Table 3). Suppose that we represent the input of those example words; the output of the model will be a vector containing the probability that a randomly selected nearby word is each word in our vocabulary.

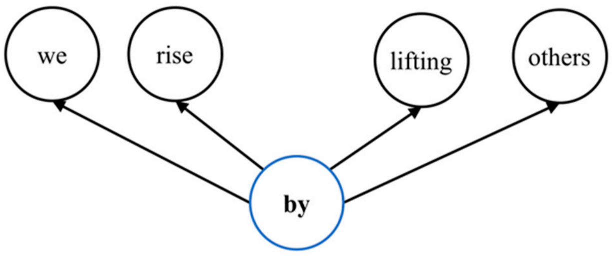


Figure 3. The SG model considers the probability of predicting the surrounding context words given a center word.

Table 3. An example of a set of SG pairs of (target word, context word) where the context word appears in the neighboring context of the target word.

Window Size	Text					Training Samples
2	we	rise	by	lifting	others	(by, we) (by, rise) (by, lifting) (by, others)

In the SG model training context, for example, if input target word $f(i)$ = "by" is given to the model, there will be four target context words, i.e., our window size is $(\max context\ location\ C = 2)$ (Context window size K $= 4$), with four activation functions, four predictions, and four errors summed to obtain the total error. The network will tell us the probability that every word in our vocabulary V is the nearby word we choose for the window size. The vocabulary here is built from the sentence. For example, if we have 5 unique words and use "by" as a hot encoding vector, the vector will have 5 components (one for each word in our vocabulary) and place "1" in a position corresponding to the word "by" which is a unique spot and "0" in all other positions.

The dot product in the hidden layer (no activation function at this point) passes to the output layer. The output layer computes the dot product between the hidden layer's output vectors and weighs the output layer matrix. Then, the softmax function computes the probability of a word appearing in the context of $f(i)$ at a given context location, which helps us distribute the probability throughout the output nodes, as we want to convert out the last layer output in terms of probability.

More formally, the SG model learns the representation of words by maximizing the objective function over the entire set of $S\ sentences$, defined as follows.

$$\mathcal{L} = \sum_{s \in S} \sum_{f_i \in s} \sum_{-c \le j \le c, j \ne 0} log\ p(f_{i+j} | f_i), \tag{1}$$

where c is the training context window size (that may depend on f_i) and words from the same sentences are ordered arbitrarily. Probability $p(f_{i+j}|f_i)$ of observing a neighboring word f_{i+j} given the current word f_i is defined using the well-known softmax function,

$$p(f_{i+j}|f_i) = \frac{exp\left(u_{f_i}^T v_{f_{i+j}}\right)}{\sum_{f=1}^{F} exp\left(u_{f_i}^T v_f\right)}, \tag{2}$$

where u_f and v_f are the vector representations of word f as input and output, respectively, F is the number of unique words in the entire vocabulary.

According to Equations (1) and (2), the SG model captures the context of the word sequence, so that words with similar contexts will have similar vector representations. The numerator in Equation (2) demonstrates this by giving a larger value for similar words through the dot product of the two vectors. If the words do not occur in each other's context, the representations will be different, and thus the numerator will be a small value.

However, the main problem arises when we want to calculate the denominator. The denominator is a normalizing factor that must be calculated over the entire vocabulary. Using Equation (2) is intractable due to the computational complexity of $\nabla p(f_{i+j}|f_i)$, which is linear function of vocabulary size $|F|$ that could easily reach millions of words. To solve this problem, a negative sampling method was used as proposed in [15].

Negative sampling is a method that considers the context of words by maximizing the similarity between words that occur in the same context and minimizing the similarity between words that occur in different contexts. To do this, instead of comparing all words in a vocabulary, the negative sampling randomly selects a small number of words $(2 \leq k \leq 20)$ to optimize the objective. The k value depends on the dataset. The value recommended by [15] is a smaller k for larger datasets and vice versa.

One of the motivations for using the skip-gram with negative sampling model as the FCP2Vec algorithm for predicting change propagation as a recommendation system is its scalability. This is because it can approximate the original softmax loss for all possible words, using negative sampling [15].

4. Proposed Approach

This section provides an overall procedure for modeling and predicting change propagation. The design is illustrated in Figure 4. In the following subsection, each step is described in more detail.

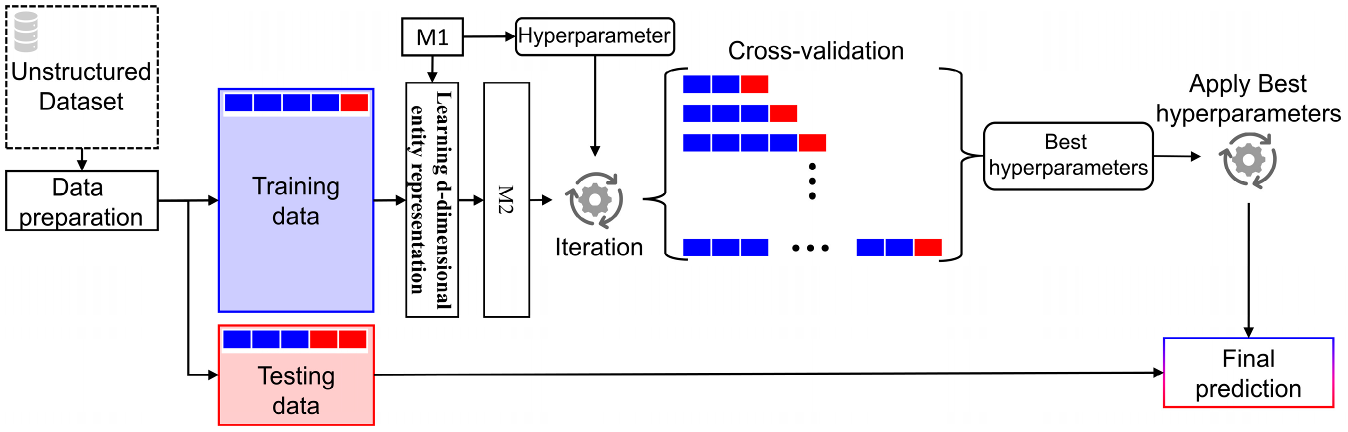
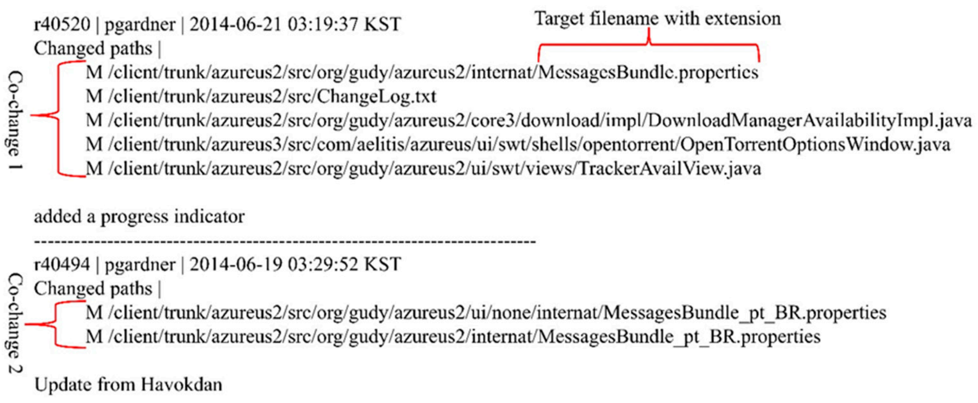


Figure 4. Overall framework for predicting CP using the architecture of FCP2Vec model: Word2vec skip-gram with negative sampling model (M1) and unsupervised nearest neighbor model (M2).

4.1. Data Preparation

In this subsection, we begin by extracting the relevant information from the changelogs of the software repository. To prepare the raw data (see Figure 5) extracted from the CVS, we applied the helper algorithm that takes the raw changelog data as input and outputs next-step process-ready data. To use the historical data contained therein for the proposed system approach, we extract them in the form of event batches or co-changes.

There are two issues related to co-changes that were filtered out. The first issue is that, in some cases, numerous elements are involved in a single co-change, which can happen because of trivial matters or more complicated issues such as "branching and merging" [10]. For example, any co-changes containing less than two elements or more than 50 elements were discarded because they do not provide any insight into the change propagation phenomenon.



```
r40520 | pgardner | 2014-06-21 03:19:37 KST
Changed paths |
        M /client/trunk/azureus2/src/org/gudy/azureus2/internat/MessagesBundle.properties
        M /client/trunk/azureus2/src/ChangeLog.txt
        M /client/trunk/azureus2/src/org/gudy/azureus2/core3/download/impl/DownloadManagerAvailabilityImpl.java
        M /client/trunk/azureus3/src/com/aelitis/azureus/ui/swt/shells/opentorrent/OpenTorrentOptionsWindow.java
        M /client/trunk/azureus2/src/org/gudy/azureus2/ui/swt/views/TrackerAvailView.java

added a progress indicator
------------------------------------------------------------------------
r40494 | pgardner | 2014-06-19 03:29:52 KST
Changed paths |
        M /client/trunk/azureus2/src/org/gudy/azureus2/ui/none/internat/MessagesBundle_pt_BR.properties
        M /client/trunk/azureus2/src/org/gudy/azureus2/internat/MessagesBundle_pt_BR.properties

Update from Havokdan
```

Figure 5. Sample of data from the Vuze raw historical development changelog.

The second issue is that we are interested in files with continuity status in the system and removing the deleted state. In computer programming, the four basic operations of persistent storage are create, read, update, and delete (CRUD) [65]. This operation has been applied to the historical changelog dataset, and two (see Table 4) of them are shown in the form when they are initially created (Append), when they are updated (Modified), and when they are deleted (D).

Table 4. Status of files before and after processing data.

Dataset	Operation	File		Package	
		Append	Modified	Append	Modified
Vuze	Before processing data	6036	51,697	6036	51,697
	After processing data	4927	36,890	2647	20,532
Spring Framework	Before processing data	14,814	143,721	14,456	137,920
	After processing data	8421	69,766	3806	29,338
Elasticsearch	Before processing data	46,761	426,896	45,652	416,932
	After processing data	33,492	242,053	16,206	132,901

4.2. Data Splitting

To ensure that the model can accurately and consistently predict future outcomes, we split the dataset into three distinct sets: training set, validation set, and test set. The training set is used to train the model. The validation set is used to evaluate the performance of the model and adjust its hyperparameters, while the test set is used to evaluate the ability of the model to generalize.

We analyzed developers' interactions with elements in terms of the order in which they occur over time. Each first co-change element $(t-1)$ was used as a training set and fit to the M1 model. Then, we used performance on a randomly sampled $((t-1)^{th}, t^{th})$ validation set of pairs of elements to benchmark the hyperparameters. The final result is obtained by making predictions on a random sample $((t-1)^{th}, t^{th})$ of element pairs (query element, next element) from the test set that does not overlap with the validation set. For our experiments, we selected the following sample size splitting ratios: 90:5:5, 85:7.5:7.5, 80:10:10, and 70:15:15.

4.3. Learning D-Dimensional Element Representation

During the development stage or maintenance, a change occurs in a single file that could propagate into other related files. If we can present each file as a vector, then we create a low-dimensional representation of a set of components such that the components that co-occur in the developer's co-changes are close in the resulting vector space. If a developer starts editing the file, the proposed approach shows what could be changed next based on the current editing session before committing a file to the repository; to do so, we can easily extract the filename associated with the file, as shown in Figure 5.

In this paper, we use the architecture of the SG (see Figure 2b) with the negative sampling method approach to the problem of maximization by minimizing the log-likelihood of a sampled negative instance. As depicted in Figure 5, we use the transaction sequence as a "sentence" and element (file name, for example, [file name.extension]) within the sequence as "words". Once we have a co-changelog history of the developer software as a sequence, we can apply an FCP2Vec algorithm to create vectors for these change commit files and then compare them to calculate the similarity to the k ranking score using unsupervised nearest neighbor (UNN) [66]. We employ the implementation of the scikit-learn [67] function as a uniform interface for three different neighbor algorithms: ball tree, KDT tree, and brute force.

There is often a great deal of variability in optimal hyperparameter values, depending on the data and the task being executed. As Word2vec was initially designed for language models, we investigated the default hyperparameters for the proposed approach employing Bayesian optimization over hyperparameters. Once the data were trained with the Word2vec model as the first model (M1), we selected the optimal hyperparameters and retrained the model. The output vector representation of the M1 model was used as input for the second model (M2), which was UNN. Then, to predict the final estimate, we used the last element in the training sequence as the query element and predicted the K closest elements to the query element using a UNN algorithm.

4.4. Evaluation Metrics

The feature vector is considered "good" enough to obtain acceptable classification accuracy. To answer how good element embedding algorithms are, we evaluated the embedded element using the next event prediction task (NEP), a common way to evaluate the quality of the embedded elements for recommendation [63,68,69]. We also adopted similar evaluation metrics and they are defined as follows:

- Hit ratio at K (HR@K). It is equal to $1/K$ if the test element appears in the top K list of predicted elements [70].
- Normalized discounted cumulative gain (NDCG@K). It considers both the order and relevance of the results. NDC@K favors higher ranks in the ordered list of predicted elements [71]. Since we consider that the ideal NDCG@K for a list retrieved from the UNN is 1, $\frac{1}{(log_2(1+i))}$, where i is the position of the relevant element in the retrieved list. The higher the position in the retrieved list, the more favorable it is.

HR and NDCG are 1 if the next element and the query element are the same, i.e., the prediction is correct. If the next element and query element are different, we look up the K UNN for the next element in the top K list of the retrieved elements and compute HR@K and NDCG@K. The main difference between NDCG and HR@K is that the latter does not have a decay factor, meaning that all results within the top K are weighed equally. In this paper, given that each user has only one ground truth item, HR@K becomes equivalent to Recall@K and exhibits a proportional relationship with Precision@K [72]. In our report, we present the HR and NDCG values with K set to 10. With both HR and NDCG metrics, superior performance is indicated by higher values.

4.5. Hyperparameters

Word2vec accepts several hyperparameters that affect both the speed and the quality of the training. We have tuned the following hyperparameters and used the most optimal for our problem. For the first model, we mainly tuned the following hyperparameters: *alpha*, *epoch*, *negative*, *ns_exponent*, *sample*, and *window*. For the vector *size* in Word2vec, we have used the rule of thumb suggested by [73] as *embedding size = min (50, (number of categories/2))*.

The early implementation of the Word2vec algorithm in the Gensim library [74] required some code modification to tune the hyperparameters, such as the "*ns_exponent*".

The updated version (4.2.0) that we used in the experiment in this paper does not require modifying the source code to tune the hyperparameters. We employed Bayesian optimization (BO) to tune these hyperparameters, which is an implementation of the scikit-learn library [67]. We used BO because it can work with non-derivable parameters that cannot be tuned using gradient descent, for example, the *learning rate*. BO can find better values for these parameters by sampling from a probability distribution over the parameter space and evaluating the objective function at the sample points. This can be more efficient than other methods, such as grid search or random search, because it uses a probabilistic model to guide the search process, taking into account the past performance of the objective function and making informed decisions about which points to evaluate next [75].

Table 5 presents a summary of the optimal hyperparameters for the Word2vec and UNN models, tailored to optimize HR@10. This table exclusively reports the finest hyperparameter value for the best data split at both the file and package levels. Upon evaluation, these parameters were determined to be particularly fitting for our use case. For the Word2vec and UNN models, all the other hyperparameters are set to their default values, except for those specified in Table 5.

Table 5. Optimal hyperparameters for each dataset at both file and package levels. The search method employed encompasses uniform and log-uniform. In the uniform method, integers are sampled uniformly between the lower and upper limits. In the log-uniform method, integers are uniformly sampled between "log (lower, base)" and "log (upper, base)", where the logarithm has a base of "base". By default, the base is set to 10.

Hyperparameters	Description and Default Value	Searched	Optimal Values					
			Vuze		Spring Framework		Elasticsearch	
			File	Package	File	Package	File	Package
alpha	The initial learning rates. The default value is set to 0.025.	The range between 1×10^{-2} and 1×10^{-1}. Prior as uniform.	0.1	0.1	0.06	0.09	0.1	0.1
epochs	The number of iterations over the corpus. The default value is set to 5.	The range is between 100 and 500. Prior as log-uniform.	500	500	500	291	500	500
negative	Negative sampling distribution. The default value is set to 5.	The range between 1 and 10. Prior as uniform.	10	10	10	10	10	10
sample	The threshold for configuring higher frequency words is randomly downsampled. The default value is set to 0.001.	The range is between 0 and 1×10^{-5}. Prior as uniform.	1×10^{-5}	0.0	8×10^{-6}	0.0	8×10^{-6}	1×10^{-5}
ns_exponent	Used to shape the negative sampling distribution. A value of 1.0 samples exactly in proportion to the frequency, 0.0 samples all words equally, while a negative value samples low-frequency words more than high-frequency words. The default value is set to 0.75.	The range between -1 and 1. Prior as uniform.	1.0	0.07	1.0	-0.4	1.0	0.4
window	Maximum distance between the current and the predicted file within a session. The default value is set to 5.	The range is between 2 and 10. Prior as uniform.	2	2	2	2	2	2
sg	Training algorithm. The default value is set to 0.	Fixed: 1: skip-gram algorithm.	1	1	1	1	1	1

Table 5. *Cont.*

Hyperparameters	Description and Default Value	Searched	Optimal Values					
			Vuze		Spring Framework		Elasticsearch	
			File	Package	File	Package	File	Package
vector_size	The number of dimensions in the vector space in which the word embeddings are represented. The default value is set to 100.	Fixed number calculated from the rule of thumb (37) or 200 as random choice.	37	200	37	37	37	37
n_neighbors	Number of neighbors to be used for K neighbor queries. Default value is set to 5.	Fixed number 10.	10	10	10	10	10	10

5. Empirical Studies

The selection of the three datasets for the case study was based on their size, complexity, and historical change data availability. Vuze, Spring Framework, and Elasticsearch offer a wide range of historical change data due to their long-standing usage in the industry and open-source nature (see Figure 6).

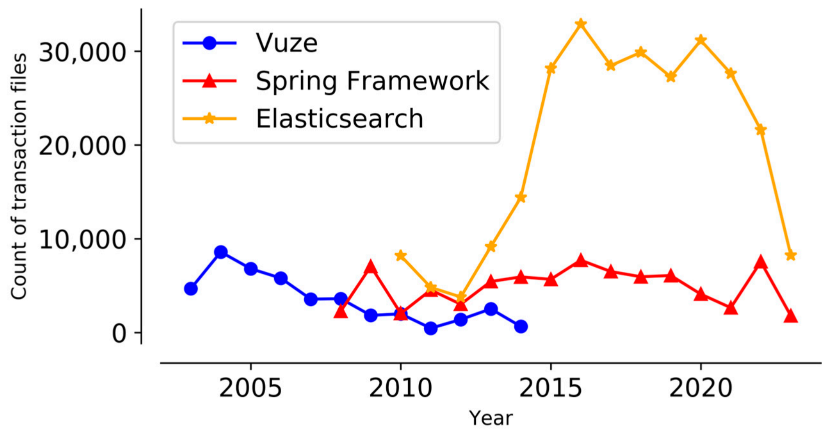


Figure 6. Ascending order of transactional changelogs spanning Vuze (10 July 2003~2 August 2014), Spring Framework (11 July 2008~3 May 2023), and Elasticsearch (8 February 2010~5 May 2023).

Vuze, known for its moderate size and architectural complexity, was chosen as a representative case study for moderately complex software systems. Historical change data for Vuze show a peak in development activity from 2003 to 2006, followed by a gradual decline until 2011. This extensive history provides a rich dataset for analysis, offering insights into how changes propagate over the lifespan of the software system. The Spring Framework, a widely used Java framework that facilitates the development of enterprise-level applications, represents large-scale software systems in our study. Its change history shows a unique pattern with a peak in activity from 2008 to 2009, followed by a decline and then a resurgence from 2012 to 2016. This pattern provides an opportunity to analyze change propagation in large-scale software systems undergoing periods of intense development and relative stability. Elasticsearch, a Java-based distributed search and analytics engine, offers another perspective on large-scale software systems. Its change history shows a different pattern compared to the other two datasets. There was a gradual decrease in activity from 2010 to 2012, followed by a period of high activity until 2020, and then a gradual decline. This pattern allows us to explore how change propagation evolves during periods of varying development activity.

5.1. Transaction Data

We conducted our experiments on Vuze [16], Spring Framework [17], and Elasticsearch [18] datasets that provide a robust and diverse set of software development changelogs to test the proposed FCP2Vec method. Each dataset offers unique features in terms of scale and complexity, with Vuze having 5883, Spring Framework having 158,553, and Elasticsearch having 473,657 indexed cases (see Table 6), with other utility detail information that shows before and after processing the data. All three datasets span over a decade (Vuze, 10 July 2003~2 August 2014; Spring Framework, 11 July 2008~3 May 2023; and Elasticsearch, 8 February 2010~5 May 2023), providing a rich history of software changes for the proposed approach. Their diversity in terms of size, complexity, and historical change data help us to ensure that our method is robust and adaptable to different software systems.

Table 6. Utility information about development changelog datasets. This table provides the unique counts before (UCB) and after (UCA) processing data for each dataset.

Utility Information	**Vuze**		**Spring Framework**		**Elasticsearch**	
	File UCB ∣ UCA	**Package UCB ∣ UCA**	**File UCB ∣ UCA**	**Package UCB ∣ UCA**	**File UCB ∣ UCA**	**Package UCB ∣ UCA**
Package	-	386 ∣ 385	-	525 ∣ 512	-	1911 ∣ 1884
Files	4778 ∣ 4261	-	12,670 ∣ 10,786	-	41,778 ∣ 37,077	-
Total developers	41 ∣ 28	41 ∣ 25	912 ∣ 441	882 ∣ 277	2047 ∣ 897	1968 ∣ 735
Maximum session length	1173 ∣ 50	1173 ∣ 50	6624 ∣ 50	6623 ∣ 50	14,908 ∣ 50	14,881 ∣ 50
Minimum session length	1 ∣ 2	1 ∣ 2	1 ∣ 2	1 ∣ 2	1 ∣ 2	1 ∣ 2
Total sessions	19,124 ∣ 7298	19,124 ∣ 6094	24,467 ∣ 13,941	22,065 ∣ 8721	63,641 ∣ 36,039	60,027 ∣ 28,535

5.2. System Environments

The experiment was conducted on a system equipped with an Intel® Core^{TM} i7-6700HQ CPU and NVIDIA GeForce GTX 960M GPU, with 16.0 GB installed RAM. The operating system installed was Ubuntu 22.04. This setup provides sufficient computational power needed by the FCP2Vec model. For software dependencies, Python version (V) 3.9.12 was used as the primary programing language. A variety of Python libraries were also utilized, each of a specific version: Pandas V. 1.4.4, Gensim V. 4.2.0, Numpy V. 1.18.5, Matplotlib V. 3.5.2, scikit-learn V. 1.1.1, tqdm V. 4.64.1, using the Pythonic version of Word2vec from the Gensim [74] open-source library.

5.3. Results and Discussion

Evaluating the performance of unsupervised learning methods can be challenging as there is no standard way to confirm their accuracy on independent datasets. Therefore, the evaluation methods used in this study are based on the specific problem and dataset we used. Tables 7 and 8 show an average score of 10-fold with the 95% confidence intervals of our experiments with various sample sizes, to assess the model's sensitivity to changes in data size. The results of trial I in Table 7 were based on the default hyperparameters of the skip-gram with a negative sampling training algorithm, but this did not perform as well as the optimized setting in trial II of Table 7. As expected, the performance of a language model dataset can vary depending on the specific task and language it is used for, and the default hyperparameters may not be suitable for other datasets in different domains.

Table 7. Evaluation performance using normalized discount cumulative gain (NDCG) and hit ratio (HR) of the FCP2Vec model in multiple trial scenarios. Trial I results from the default parameter values for an estimator and Trial II results from Bayesian optimization over hyperparameters using the following split ratios: A = 90:5:5, B = 85:7.5:7.5, C = 80:10:10, and D = 70:15:15.

Datasets	**Granularity**	**Metrics**	**Trial I**			
			A	**B**	**C**	**D**
Vuze	File level	HR@10	14.35 ± 0.69	16.25 ± 0.42	16.31 ± 0.26	**16.70** ± 0.41
		NDCG@10	0.081 ± 0.002	0.095 ± 0.001	0.098 ± 0.002	0.10 ± 0.001
	Package level	HR@10	14.76 ± 0.341	15.69 ± 0.484	16.09 ± 0.379	**16.35 ± 0.301**
		NDCG@10	0.088 ± 0.001	0.096 ± 0.001	0.103 ± 0.001	0.10 ± 0.001
Spring Framework	File level	HR@10	11.49 ± 0.379	11.80 ± 0.543	**12.56 ± 0.253**	12.54 ± 0.221
		NDCG@10	0.079 ± 0.002	0.08 ± 0.003	0.087 ± 0.002	0.08 ± 0.001
	Package level	HR@10	22.50 ± 0.731	22.59 ± 0.723	**23.13 ± 1.096**	22.27 ± 0.829
		NDCG@10	0.16 ± 0.004	0.17 ± 0.006	0.177 ± 0.007	0.173 ±0.004
Elasticsearch	File level	HR@10	15.41 ± 0.257	**16.51 ± 0.186**	16.27 ± 0.251	15.97 ± 0.098
		NDCG@10	0.101 ± 0.001	0.112 ± 0.001	0.11 ± 0.001	0.107 ± 0.001
	Package level	HR@10	18.29 ± 0.259	**18.476 ± 0.276**	18.405 ± 0.195	17.82 ± 0.149
		NDCG@10	0.130 ± 0.001	0.130 ± 0.001	0.131 ± 0.000	0.127 ± 0.000
			Trial II			
Vuze	File level	HR@10	37.80 ± 1.06	**38.61 ± 0.72**	37.00 ± 0.55	37.51 ± 0.21
		NDCG@10	0.21 ± 0.005	0.22 ± 0.003	0.21 ± 0.002	0.58 ± 0.003
	Package level	HR@10	**57.76 ± 0.227**	55.20 ± 0.455	53.89 ± 0.326	20.67 ± 0.0
		NDCG@10	0.357 ± 0.001	0.325 ± 0.001	0.334 ± 0.001	0.12 ± 0.0
Spring Framework	File level	HR@10	26.62 ± 0.325	27.19 ± 0.424	**29.84 ± 0.455**	28.31 ± 0.267
		NDCG@10	0.171 ± 0.001	0.172 ± 0.002	0.189 ± 0.001	0.180 ± 0.002
	Package level	HR@10	51.76 ± 0.451	52.23 ± 0.231	**53.45 ± 0.202**	53.01 ± 0.000
		NDCG@10	0.362 ± 0.002	0.359 ± 0.002	0.363 ± 0.000	0.363 ± 0.000
Elasticsearch	File level	HR@10	33.86 ± 0.344	**34.56 ± 0.312**	34.47 ± 0.249	34.55 ± 0.181
		NDCG@10	0.213 ± 0.001	0.220 ± 0.001	0.220 ± 0.001	0.220 ± 0.001
	Package level	HR@10	35.395 ± 0.488	**35.72 ± 0.332**	35.32 ± 0.406	35.348 ± 0.208
		NDCG@10	0.225 ± 0.002	0.224 ± 0.001	0.222 ± 0.002	0.222 ± 0.001

We optimized the hyperparameters of the Word2vec model based on the characteristics of our changelog dataset. As seen in Table 7, for trial II, there is a significant difference in performance between the default and optimized hyperparameters for our dataset. We also conducted an analysis of the model's sensitivity to different sample sizes and found that the 85:15 split ratio yielded the best results for the Vuze dataset at the file level. Table 8 shows the optimized results for each year in the Vuze dataset. We trained the model on one year at a time, starting with 2003, and then added each subsequent year to the training set, until all years from 2003 to 2014 were included. This allowed us to learn the patterns of the data for each year individually. Table 5 summarizes the optimized hyperparameters shown in Table 7 for trial II.

Figure 7 shows the ability of the proposed method to visually depict the changes made to different elements in a vector space. The elements are placed close to each other if they have undergone similar changes. To create this visualization, a dimensionality reduction technique was used, which aims to keep as much information as possible while reducing the number of dimensions or features in the dataset. This technique can be useful for visualizing high-dimensional data, as it can be challenging to visualize data in more than three dimensions. By reducing the dimensionality of the data to two or three dimensions, it becomes much easier to plot and visualize the data, which can aid in understanding patterns and relationships within the data. A common method for dimensionality reduction is t-distributed stochastic neighbor embedding (t-SNE) [76], which is a non-linear method that is often used for visualizing high-dimensional data. As shown in Figures 7 and 8, we used t-SNE to reduce the vector size from thirty-seven dimensions to two dimensions. This means that the data were originally represented by thirty-seven different features, but were reduced to just two dimensions, which makes them easier to plot and visualize. The more densely packed a cluster of dots is, the more elements that are close together and were changed together in the past. This can be an indication of some sort of relationship or pattern within the data.

Table 8. Evaluation performance of each study period using the normalized cumulative discount gain (NDCG) and the hit ratio (HR) of the FCP2Vec model on Vuze dataset at file level. All trial II results from Bayesian optimization over hyperparameters. Based on the split ratios: A = 90:5:5, B = 85:7.5:7.5, C = 80:10:10, and D = 70:15:15.

Year	Metrics	A	B	C	D
2003	HR@10	52.77 ± 0.0	55.55 ± 0.0	59.72 ± 1.07	51.85 ± 5.35
	NDCG@10	0.244 ± 2.09	0.29 ± 0.0	0.32 ± 0.0	0.25 ± 4.18
2004	HR@10	39.27 ± 0.41	36.12 ± 0.22	38.09 ± 0.13	36.96 ± 0.32
	NDCG@10	0.23 ± 0.002	$0.18 \pm 6.7 \times 10^{-4}$	$0.20 \pm 3.5 \times 10^{-4}$	0.18 ± 0.001
2005	HR@10	41.59 ± 0.67	41.34 ± 0.19	35.61 ± 0.069	35.48 ± 0.74
	NDCG@10	0.26 ± 0.005	0.24 ± 0.001	$0.20 \pm 2.1 \times 10^{-4}$	0.21 ± 0.003
2006	HR@10	34.85 ±1.32	34.26 ± 0.74	35.56± 1.0	36.68 ± 0.46
	NDCG@10	0.20 ± 0.006	0.19 ± 0.005	0.211 ± 0.003	0.21 ± 0.004
2007	HR@10	35.89 ± 0.76	35.17 ± 0.79	38.86 ± 0.71	39.09 ± 0.42
	NDCG@10	0.19 ± 0.002	0.20 ± 0.004	0.22 ± 0.004	0.22 ± 0.004
2008	HR@10	40.03 ± 0.99	39.85 ± 0.45	39.35 ± 0.65	38.10 ± 0.22
	NDCG@10	0.24 ± 0.004	0.23 ± 0.004	0.23 ± 0.0058	0.31 ± 0.003
2009	HR@10	37.85 ± 0.94	36.81 ± 0.53	37.56 ± 0.70	37.01 ± 0.62
	NDCG@10	0.22 ± 0.003	0.22 ±0.004	0.22 ± 0.004	0.21 ± 0.003
2010	HR@10	36.40 ± 1.27	35.81 ± 0.82	37.18 ± 0.21	37.13 ± 0.57
	NDCG@10	0.20 ± 0.006	0.20 ± 0.004	0.77 ± 0.005	0.21 ± 0.002
2011	HR@10	36.20 ± 1.27	37.11 ± 0.60	39.66 ± 0.80	36.09 ± 0.56
	NDCG@10	0.20 ± 0.006	0.20 ± 0.003	0.22 ± 0.003	0.20 ± 0.002
2012	HR@10	36.14 ± 0.63	35.94 ± 0.89	34.01 ± 0.58	35.14 ± 0.36
	NDCG@10	0.20 ± 0.004	0.20 ±0.004	0.19 ± 0.003	0.20 ± 0.002
2013	HR@10	34.21 ± 1.03	36.62 ± 0.51	36.76 ± 0.93	35.50 ± 0.43
	NDCG@10	0.20 ± 0.007	0.21 ± 0.005	0.21 ± 0.002	0.20 ± 0.002
2014	HR@10	37.80 ± 1.06	38.61 ± 0.72	37.00 ± 0.55	37.51 ± 0.21
	NDCG@10	0.21 ± 0.005	0.22 ± 0.003	0.21 ± 0.002	0.58 ± 0.003

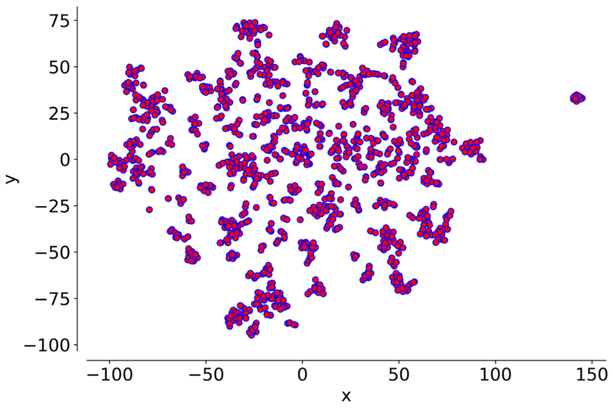


Figure 7. The two-dimensional vector representation of Vuze changelogs at file level in the vector space. The denser the dot cluster, the more files that are close together and the more frequently they were changed together in the past.

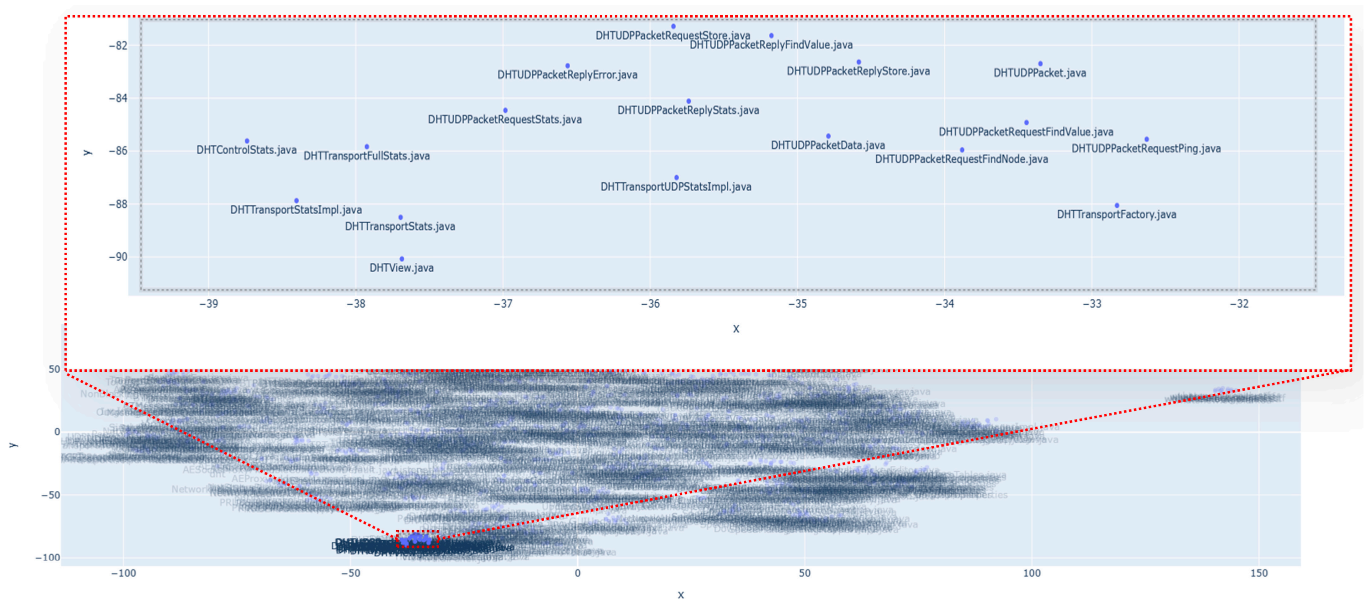


Figure 8. A randomly selected data point from the two-dimensional vector representation of Vuze changelogs at file level in the vector space. The denser the dot cluster is, the more files that are close together and the more frequently they were changed together in the past.

For instance, as shown in the zoomed-in part of Figure 8, the following [“DHTTransportUDP.java”, “DHTUDPPacketReply.java”, “DHTUDPPacketRequest.java”, “DHTUDPPacket.java”] elements that experience simultaneous co-changes were chosen randomly from the input data. These data were used as input to our model except the last element of the list, which is [“DHTUDPPacket.java”]. Then, using (query element, next element) we found the following interesting results shown in Table 9.

Table 9. A sample of random results from the Vuze dataset at the file level, employing single-input queries and group queries with the highest probability score as output. The outcomes are organized in descending order, accompanied by a probability score for each element.

Metrics	Results	
	Most Similar Top 10 Elements	**Probability Score**
Single input query “DHTUDPPacketRequest.java”	DHTUDPPacketRequestFindValue.java	0.885
	DHTUDPPacketRequestPing.java	0.870
	DHTUDPPacketRequestFindNode.java	0.869
	DHTUDPPacketReply.java	0.868
	DHTUDPPacket.java	0.867
	DHTUDPPacketData.java	0.862
	DHTTransportUDP.java	0.841
	DHTUDPPacketRequestStore.java	0.838
	DHTTransportStatsImpl.java	0.837
	DHTUDPPacketReplyStats.java	0.809
Group input query: [“DHTTransportUDP.java”, “DHTUDPPacketReply.java”, “DHTUDPPacketRequest.java”]	DHTUDPPacketRequestFindValue.java	0.914
	DHTUDPPacketData.java	0.910
	DHTUDPPacket.java	0.909
	DHTUDPPacketReplyFindNode.java	0.906
	DHTUDPPacketReplyPing.java	0.894
	DHTUDPPacketRequestFindNode.java	0.882
	DHTUDPPacketRequestStore.java	0.881
	DHTUDPUtils.java	0.880
	DHTUDPPacketRequestPing.java	0.876
	DHTUDPPacketReplyStats.java	0.871

The results presented in Table 9 demonstrate that elements that have undergone changes together in the past tend to be located close to each other in the vector space, as shown in Figure 8. Additionally, the probability scores for these elements are also higher within the searched range. This means that the higher the score, the more likely it is that changes will propagate to that particular element. This knowledge can assist in managing the complexity and risk associated with making changes to a software system.

Comparison between FCP2Vec and DN

In this study, a comparative analysis was undertaken to evaluate the performance of FCP2Vec and the DN with respect to two primary dimensions: change prediction performance and computation efficiency.

Change prediction performance: in our study, FCP2Vec change prediction performance was compared with that of the DN. For consistency, we used the same dataset (Vuze), granularity level, and top "K" (where K = 10). Figure 9 illustrates the comparative hit ratios of change prediction for the top 10 for both the DN and FCP2Vec. The result reveals a superior performance from FCP2Vec across all DN scenarios [13]. The HR@10 value for FCP2Vec stands at 0.58, which shows a significant increase compared to the DN scenarios. Specifically, FCP2Vec's HR@10 value is approximately 92% higher than DN scenario I, about 33% higher than DN scenario II, and approximately 21% higher than DN scenario III. This underscores FCP2Vec's superior performance in terms of change prediction.

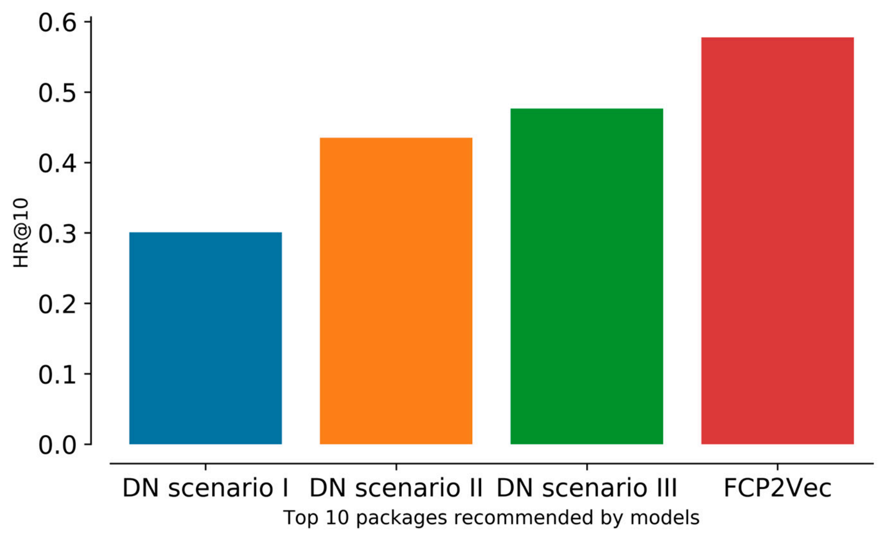


Figure 9. Comparative hit ratios of change prediction for dependency network (DN) and FCP2Vec in various scenarios at package level.

The performance outcome was made possible due to the model's capacity to learn from each context, thereby capturing relationships and similarities between elements. In terms of semantic similarity, FCP2Vec can discern sematic similarity between elements, for instance, elements frequently changed together may have closer vector representations. We leveraged this feature to recommend the next elements based on the developer's historical behavior. Furthermore, we applied the concept of transfer learning. Here, the embedding learned by Word2vec served as input features for other models. After Word2vec converted elements into high-dimensional vectors encapsulating the context co-change relationships, these feature vectors were used as input to the UNN algorithm. The underlying principle of the UNN was to create "K" nearest neighbors that could be utilized for ranking in our evaluation metrics. Consequently, this approach resulted in improved performance compared to the DN.

Computation efficiency: we evaluated the computational efficiency of our proposed model by conducting an experiment on the changelogs of Vuze at the file level. At the lowest granularity level, they contained more elements than at the package level and a previous study [13] also benchmarked computational efficiency at this level. The result was an escalation in computational complexity for the dependency network model in terms of learning and inference.

In our study, we compared the training and inference time of FCP2Vec with those of the dependency network. We excluded the elapsed optimization time from our comparison, as it was not considered in the previous study. Figure 10a shows the optimization time in relation to different split ratios and the count of transaction files annually. The findings indicate that the elapsed hyperparameter optimization time peaked with a split ratio of 80:10:10 and troughed with a ratio of 85:7.5:7.5, pointing to a rapid convergence. The prompt identification of optimal hyperparameters during optimization confers several benefits, including efficient resource utilization, accelerated experiment and model selection to reduce overfitting risk, and timely prediction. Figure 10b contrasts the elapsed training times using different split ratios and the annual count of transaction files. Our results highlighted that change propagation at the file level trained 30% quicker than the dependency network, a percentage that increased with increasing file counts. Furthermore, our model clocked an inference time of a mere 0.5 s, significantly outpacing the 17 s logged for the dependency network.

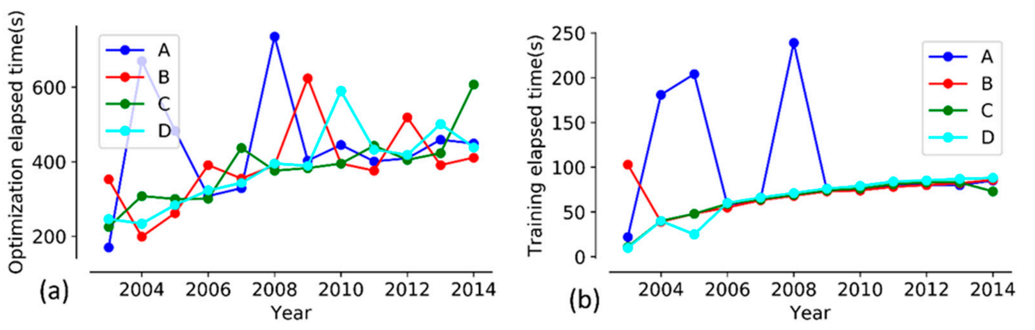


Figure 10. Computational time between different years and split ratio for Vuze datasets at file level: (**a**) elapsed optimization time and (**b**) elapsed training time based on different split ratios.

The result shows that utilizing file level change coupling can uncover concealed correlations that are absent from both the source code and associated documentation. For example, a class A may be modified in conjunction with another class B without having a structural dependence on it. Moreover, it does not require parsing code, making it more lightweight than structural analysis. Its language-independent nature makes it a highly adaptable and optimal choice for research involving a wide range of subject systems developed in different programming languages.

The significance of this research lies on its potential to improve the accuracy and scalability of change propagation in large software systems using the proposed approach. By addressing the current limitation of previous methods such as dependency networks, this research can contribute to the development of more reliable and efficient software systems, reducing the costs of maintenance and error correction. This research can also help software developers with a new set of tools and techniques to improve the quality change process and predict the effects of change propagation. Overall, applying change coupling in the software development process can help developers to better understand and manage the complexity and risk associated with making changes to a software system, leading to improved efficiency, reliability, and quality.

6. Conclusions

This article investigates how the evolutionary changelog dataset of software creates a low-dimensional representation of a set of components such that the components that co-occur in developer transactions are close in the resulting vector space. We used an FCP2Vec model to learn file vector representations that can be predicted from the K nearest top files. A dataset from software development changelogs was used to develop a change propagation model as a recommendation system. This system recommends the next similar file that might need to be changed based on the probability of propagation of changes at the file level. We have investigated a methodical way of suggesting files that are more likely to be changed due to change propagation, based on their rank score. To validate our approach, we conducted our experiments on Vuze, Spring Framework, and Elasticsearch datasets, open-source Java-based projects that have been developing changelog data for over a decade.

The use of changelog data offers a variety of benefits. One key advantage is that it allows for the discovery of historical relationships within a system. By analyzing the changes made to different elements, previously hidden connections may be uncovered, making this approach crucial for understanding complex systems. Additionally, utilizing changelog data can improve accuracy by enabling computers to recognize patterns that humans may overlook. It also allows for early detection of potential issues, leading to a faster and more effective resolution. This system is also beneficial for developers joining a project post-launch, as it provides insight into historical relationships. Additionally, it does not require code parsing, making it more efficient than structural analysis. Its language independence makes it a versatile and optimal choice for research across a variety of subject systems developed in different programming languages.

Despite demonstrating superior performance and adequately addressing scalability issues compared to previous methodologies, our current model nonetheless faces certain limitations. A prominent example is the inability to handle out-of-vocabulary elements. Specifically, if an element is not part of the original training corpus, Word2vec fails to recognize it. Consequently, this necessitates the model's retraining to incorporate the new element into the system. Although Word2vec is optimally designed for language models, its application across different domains demands substantial computational power, particularly when training on large elements. A potential solution to tackle these challenges may lie in the exploration of recent developments, such as transformer-based models and their variants.

As we look to the future, our aim is to further experiment using open-source software, in conjunction with diverse changelog data across various domains.

Author Contributions: Methodology, H.A.A.; software, H.A.A.; validation, H.A.A. and J.L.; formal analysis, H.A.A.; investigation, H.A.A.; writing—original draft preparation, H.A.A.; writing—review and editing, J.L; visualization, H.A.A.; supervision, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the BK21 FOUR of the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 5199990914451). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2022R1I1A3071200)

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bennett, K.H.; Rajlich, V.T.; Wilde, N. Software Evolution and the Staged Model of the Software Lifecycle. In *Advances in Computers*; Elsevier: Amsterdam, The Netherlands, 2002; pp. 1–54.
2. Yau, S.S.; Nicholl, R.A.; Tsai, J.J.-P.; Liu, S.-S. An Integrated Life-Cycle Model for Software Maintenance. *IEEE Trans. Softw. Eng.* **1988**, *14*, 1128–1144. [CrossRef]
3. Rajlich, V. A model for change propagation based on graph rewriting. In Proceedings of the 1997 Proceedings International Conference on Software Maintenance, Bari, Italy, 1–3 October 1997.

4. Yu, L.; Schach, S.R. Applying association mining to change propagation. *Int. J. Softw. Eng. Knowl. Eng.* **2008**, *18*, 1043–1061. [CrossRef]
5. Pan, W.; Jiang, H.; Ming, H.; Chai, C.; Chen, B.; Li, H. Characterizing Software Stability via Change Propagation Simulation. *Complexity* **2019**, *2019*, 9414162. [CrossRef]
6. Oliva, G.A.; Gerosa, M.A. Chapter 11—Change Coupling Between Software Artifacts: Learning from Past Changes. In *The Art and Science of Analyzing Software Data*; Morgan Kaufmann: Burlington, MA, USA, 2015; pp. 285–323.
7. Ball, T.; Kim, J.H.; Porter, A.; Siy, H. If Your Version Control System Could Talk. In Proceedings of the ICSE Workshop Process Modelling and Empirical Studies of Software Engineering, Boston, MA, USA, 18 May 1997.
8. Cataldo, M.; Herbsleb, J.D. Coordination breakdowns and their impact on development productivity and software failures. *IEEE Trans. Softw. Eng.* **2013**, *39*, 343–360. [CrossRef]
9. Hassan, A.; Holt, R. Predicting Change Propagation in Software Systems. In Proceedings of the 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11–14 September 2004.
10. Zimmermann, T.; Weißgerber, P.; Diehl, S.; Diehl, S. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.* **2005**, *31*, 429–445. [CrossRef]
11. Ferreira, K.A.M.; Bigonha, M.A.S.; Bigonha, R.S.; Lima, B.N.B.D.; Gomes, B.M.; Mendes, L.F.O. A model for estimating change propagation in software. *Softw. Qual. Control* **2018**, *26*, 217–248. [CrossRef]
12. Siavash, M.; Alaa, H.; Ladan, T. Using Bayesian Belief Networks to Predict Change Propagation in Software Systems. In Proceedings of the 15th IEEE International Conference on Program Comprehension, Banff, AB, Canada, 26–29 June 2007.
13. Lee, J.; Hong, Y.S. Data-driven prediction of change propagation using Dependency Network. *Eng. Appl. Artif. Intell.* **2018**, *70*, 149–158. [CrossRef]
14. Mikolov, T.; Chen, G.C.K.; Dean, J. Efficient estimation of word representations in vector space. In Proceedings of the Workshop at International Conference on Learning Representations, Scottsdale, AZ, USA, 2–4 May 2013.
15. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; Dean, J. Distributed Representations of Words and Phrases and their Compositionality. *arXiv* **2013**, arXiv:1310.4546.
16. Vuze-Azureus. Sourceforge. 2020. Available online: https://sourceforge.net/projects/azureus/ (accessed on 11 September 2020).
17. Spring Framework. 2023. Available online: https://github.com/spring-projects/spring-framework (accessed on 3 May 2023).
18. Elasticsearch. 2023. Available online: https://github.com/elastic/elasticsearch (accessed on 5 May 2023).
19. Khan, M.; Jan, B.; Farman, H.; Ahmad, J.; Farman, H.; Jan, Z. Deep Learning Methods and Applications. In *Deep Learning: Convergence to Big Data Analytics*; Springer: Singapore, 2019; pp. 31–42.
20. Menghani, G. Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better. *ACM Comput. Surv.* **2023**, *55*, 1–37. [CrossRef]
21. Salem, H.; El-Hasnony, I.M.; Kabeel, A.; El-Said, E.M.; Elzeki, O.M. Deep Learning model and Classification Explainability of Renewable energy-driven Membrane Desalination System using Evaporative Coole. *Alex. Eng. J.* **2022**, *61*, 10007–10024. [CrossRef]
22. Lewowski, T.; Madeyski, L. Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review. In *Developments in Information & Knowledge Management for Business Applications*; Kryvinska, N., Poniszewska-Marańda, A., Eds.; Springer International Publishing: Berlin/Heidelberg, Germany, 2022; Volume 3, pp. 285–319.
23. Lozoya, R.C.; Baumann, A.; Sabetta, A.; Bezzi, M. Commit2Vec: Learning Distributed Representations of Code Changes. *SN Comput. Sci.* **2021**, *2*, 150. [CrossRef]
24. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. In Proceedings of the ACM on Programming Languages, Phoenix, AZ, USA, 22–26 June 2019.
25. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating Sequences from Structured Representations of Code. *arXiv* **2018**, arXiv:1808.01400.
26. Loeliger, J.; McCullough, M. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2012.
27. Han, J. Supporting impact analysis and change propagation in software engineering environments. In Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering, London, UK, 14–18 July 1997.
28. Aryani, A.; Peake, I.D.; Hamilton, M.; Schmidt, H.; Winikoff, M. Change Propagation Analysis Using Domain Information. In Proceedings of the 2009 Australian Software Engineering Conference, Gold Coast, QLD, Australia, 14–17 April 2009.
29. Gall, H.; Jazayeri, M.; Krajewski, J. CVS Release History Data for Detecting Logical Couplings. In Proceedings of the IWPSE '03: 6th International Workshop on Principles of Software Evolution, Helsinki, Finland, 1–2 September 2003.
30. Zimmermann, T.; Diehl, S.; Zeller, A. How history justifies system architecture (or not). In Proceedings of the Sixth International Workshop on Principles of Software Evolution 2003, Proceedings, Helsinki, Finland, 1–2 September 2003.
31. Oliva, G.A.; Gerosa, M.A. On the Interplay between Structural and Logical Dependencies in Open-Source Software. In Proceedings of the 2011 25th Brazilian Symposium on Software Engineering, Sao Paulo, Brazil, 28–30 September 2011.
32. Bavota, G.; Dit, B.; Oliveto, R.; Penta, M.D.; Poshyvanyk, D.; Lucia, A.D. An empirical study on the developers' perception of software coupling. In Proceedings of the ICSE '13: 2013 International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013.

33. Wang, X.; Wang, H.; Liu, C. Predicting Co-Changed Software Entities in the Context of Software Evolution. In Proceedings of the 2009 International Conference on Information Engineering and Computer Science, Wuhan, China, 25–26 July 2009.
34. Ying, A.T.; Murphy, G.C.; Ng, R.; Chu-Carroll, M.C. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.* **2004**, *30*, 574–586. [CrossRef]
35. Antoniol, G.; Rollo, V.; Venturi, G. Detecting groups of co-changing files in CVS repositories. In Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05), Lisbon, Portugal, 5–6 September 2005.
36. Bouktif, S.; Gueheneuc, Y.-G.; Antoniol, G. Extracting Change-patterns from CVS Repositories. In Proceedings of the 2006 13th Working Conference on Reverse Engineering, Benevento, Italy, 23–27 October 2006.
37. Ceccarelli, M.; Cerulo, L.; Canfora, G.; Penta, M.D. An eclectic approach for change impact analysis. In Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 1–8 May 2010.
38. Canfora, G.; Ceccarelli, M.; Cerulo, L.; Penta, M.D. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In Proceedings of the 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, 12–18 September 2010.
39. Gall, H.; Hajek, K.; Jazayeri, M. Detection of logical coupling based on product. In Proceedings of the International Conference on Software Maintenance (Cat. No. 98CB36272), Bethesda, MD, USA, 16–19 March 1998.
40. Mockus, A.; Weiss, D.M. Predicting risk of software changes. *Bell Labs Tech. J.* **2000**, *5*, 169–180. [CrossRef]
41. Finlay, J.; Pears, R.; Connor, A.M. Data stream mining for predicting software build outcomes using source code metrics. *Inf. Softw. Technol.* **2014**, *56*, 183–198. [CrossRef]
42. Sun, X.; Li, B.; Zhang, Q. A Change Proposal Driven Approach for Changeability Assessment Using FCA-Based Impact Analysis. In Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference, Izmir, Turkey, 16–20 July 2012.
43. Kagdi, H.; Gethers, M.; Poshyvanyk, D. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In Proceedings of the 2010 17th Working Conference on Reverse Engineering, Beverly, MA, USA, 13–16 October 2010.
44. Gethers, M.; Poshyvanyk, D. Using Relational Topic Models to capture coupling among classes in object-oriented software systems. In Proceedings of the 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, 12–18 September 2010.
45. Chowdhary, K.R. Natural Language Processing. In *Fundamentals of Artificial Intelligence*; Springer: New Delhi, India, 2020; pp. 603–649.
46. Otter, D.W.; Medina, J.R.; Kalita, J.K. A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *32*, 604–624. [CrossRef]
47. Zhang, A.; Lipton, Z.C.; Li, M.; Smola, A.J. Dive into Deep Learning. *arXiv* **2021**, arXiv:2106.11342.
48. Harris, Z.S. Distributional Structure. *Word* **1954**, *10*, 146–162. [CrossRef]
49. Sahlgren, M. The distributional hypothesis. *Ital. J. Disabil. Stud.* **2008**, *20*, 33–53.
50. Liu, S.; Mallol-Ragolta, A.; Parada-Cabaleiro, E.; Qian, K.; Jing, X.; Kathan, A.; Hu, B.; Schuller, B.W. Audio self-supervised learning: A survey. *Patterns* **2022**, *3*, 100616. [CrossRef]
51. Chuan, C.-H.; Agres, K.; Herremans, D. From context to concept: Exploring semantic relationships in music with word2vec. *Neural Comput. Appl. Vol.* **2020**, *32*, 1023–1036. [CrossRef]
52. Kumar, A.; Starly, B. "FabNER": Information extraction from manufacturing process science domain literature using named entity recognition. *J. Intell. Manuf.* **2022**, *33*, 1572–8145. [CrossRef]
53. Capelleveen, G.V.; Amrit, C.; Zijm, H.; Yazan, D.M.; Abdi, A. Toward building recommender systems for the circular economy: Exploring the perils of the European Waste Catalogue. *J. Environ. Manag.* **2021**, *277*, 111430. [CrossRef] [PubMed]
54. Patra, B.G.; Maroufy, V.; Soltanalizadeh, B.; Deng, N.; Zheng, W.J.; Roberts, K.; Wu, H. A content-based literature recommendation system for datasets to improve data reusability—A case study on Gene Expression Omnibus (GEO) datasets. *J. Biomed. Inform.* **2020**, *104*, 103399. [CrossRef]
55. Nedelec, T.; Smirnova, E.; Vasile, F. Specializing Joint Representations for the task of Product Recommendation. In Proceedings of the DLRS 2017: 2nd Workshop on Deep Learning for Recommender Systems, Como, Italy, 27 August 2017.
56. Zheng, C.; Zhai, S.; Zhang, Z. A Deep Learning Approach for Expert Identification in Question Answering Communities. *arXiv* **2017**, arXiv:1711.05350.
57. Bravo-Marquez, F.; Tamblay, C. Words, Tweets, and Reviews: Leveraging Affective Knowledge between Multiple Domains. *Cogn. Comput.* **2022**, *14*, 388–406. [CrossRef]
58. Khatua, A.; Khatua, A.; Cambria, E. A tale of two epidemics: Contextual Word2Vec for classifying twitter streams during outbreaks. *Inf. Process. Manag.* **2019**, *56*, 247–257. [CrossRef]
59. Li, C.; Lu, Y.; Wu, J.; Zhang, Y.; Xia, Z.; Wang, T.; Yu, D.; Chen, X.; Liu, P.; Guo, J. LDA Meets Word2Vec: A Novel Model for Academic Abstract Clustering. In Proceedings of the WWW '18: Companion the Web Conference 2018, Geneva, Switzerland, 23–27 April 2018.
60. Jha, S.; Prashar, D.; Long, H.V.; Taniar, D. Recurrent neural network for detecting malware. *Comput. Secur.* **2020**, *99*, 102037. [CrossRef]

61. Grbovic, M.; Radosavljevic, V.; Djuric, N.; Bhamidipati, N.; Savla, J.; Bhagwan, V.; Sharp, D. E-commerce in Your Inbox: Product recommendations at scale. In Proceedings of the KDD '15: 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 10–13 August 2015.
62. Vasile, F.; Smirnova, E.; Conneau, A. Meta-Prod2Vec: Product Embeddings Using Side-Information for Recommendation. In Proceedings of the RecSys '16: 10th ACM Conference on Recommender Systems, Boston, MA, USA, 15–19 September 2016.
63. Caselles-Dupré, H.; Lesaint, F.; Royo-Letelier, J. Word2vec applied to recommendation: Hyperparameters matter. In Proceedings of the RecSys '18: 12th ACM Conference on Recommender Systems, Vancouver, BC, Canada, 2–7 October 2018.
64. Noroozi, M.; Vinjimoor, A.; Favaro, P.; Pirsiavash, H. Boosting Self-Supervised Learning via Knowledge Transfer. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018.
65. Martin, J. *Managing the Data-Base Environment*, 1st ed.; Prentice Hall: Englewood Cliffs, NJ, USA, 1983.
66. Cover, T.; Hart, P. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theory* **1967**, *13*, 21–27. [CrossRef]
67. Buitinck, L.; Louppe, G.; Blondel, M.; Pedregosa, F.; Mueller, A.; Grisel, O.; Niculae, V.; Prettenhofer, P.; Gramfort, A.; Grobler, J. API design for machine learning software: Experiences from the scikit-learn project. *arXiv* **2013**, arXiv:1309.0238.
68. Letham, B.; Letham, B.; Madigan, D. Sequential event prediction. *Mach. Learn.* **2013**, *93*, 357–380. [CrossRef]
69. Rendle, S.; Freudenthaler, C.; Schmidt-Thieme, L. Factorizing personalized Markov chains for next-basket recommendation. In Proceedings of the WWW '10: 19th International Conference on World Wide Web, Raleigh, NC, USA, 26–30 April 2010.
70. Le, Q.; Smola, A. Direct Optimization of Ranking Measures. *arXiv* **2007**, arXiv:0704.3359.
71. Järvelin, K.; Kekäläinen, J. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.* **2002**, *20*, 422–446. [CrossRef]
72. Sun, F.; Liu, J.; Wu, J.; Pei, C.; Lin, X.; Ou, W.; Jiang, P. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer. In Proceedings of the 28th ACM International Conference on Information and Knowledge Management, Beijing, China, 3–7 November 2019.
73. Howard, J.; Gugger, S. *Deep Learning for Coders with Fastai and PyTorch: AI Applications without a PhD*; O'Relly Media, Inc.: Sebastopol, CA, USA, 2020.
74. Řehůřek, R. *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*; ELRA: Valletta, Malta, 2010.
75. Snoek, J.; Larochelle, H.; Adams, R.P. Practical Bayesian Optimization of Machine Learning Algorithms. In Proceedings of the Advances in Neural Information Processing Systems 25 (NIPS 2012), Lake Tahoe, NV, USA, 3–6 December 2012.
76. Van der Maaten, L.; Hinton, G. Visualizing data using t-SNE. *J. Mach. Learn. Res.* **2008**, *9*, 2579–2605.