LEBANESE AMERICAN UNIVERSITY

Modeling Software System Interactions Using Temporal Graphs and Graph Neural Networks: a Focus on Change Propagation

By

Manuella Latif Germanos

A thesis
submitted in partial fulfilment of the requirements
for the degree of Master of Science
in Computer Science

School of Arts and Science
July 2022

الجامعة اللبنانية الأميركية
Lebanese American University

THESIS APPROVAL FORM

Student Name: Manuella Germanos I.D. #: 201503708

Thesis Title: Modeling Software System Interactions Using Temporal Graphs and Graph Neural Networks: a Focus on Change Propagation

Program: MS in Computer Science

Department: Computer Science and Math

School: Arts and Sciences

The undersigned certify that they have examined the final electronic copy of this thesis and approved it in Partial Fulfillment of the requirements for the degree of:

Master of Science in the major of Computer Science

Thesis Advisor's Name: Danielle Azar

Signature: [redacted] Date: 05 / August/ 2022
Day Month Year

Committee Member's Name: Nader El Khatib

Signature: [redacted] Date: 05 / August/ 2022
Day Month Year

Committee Member's Name: Eileen Marie Hanna

Signature: [redacted] Date: 05 / August/ 2020
Day Month Year

Committee Member's Name:

Signature: Date:1 / /
Day Month Year

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

THESIS COPYRIGHT RELEASE FORM

LEBANESE AMERICAN UNIVERSITY NON-EXCLUSIVE DISTRIBUTION LICENSE

Name: Manuella Germaonos

Signature: [redacted]

Date: 5 / 8 / 2022
Day Month Year

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

PLAGIARISM POLICY COMPLIANCE STATEMENT

I certify that:

1. I have read and understood LAU's Plagiarism Policy.
2. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.
3. This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Manuella Germanos

Signature [redacted]

Date: 5 / 8 / 2022
Day Month Year

This thesis is dedicated to my loving parents, Rafka and Latif.

I wouldn't be here without you (literally).

Acknowledgement

Words cannot express how grateful I am for the continuous support of the committee members throughout this thesis.
I am deeply indebted to Dr. Danielle Azar for, without her, I would not have been able to finish my Bachelor's degree, let alone this thesis. I would like to thank her for believing in me, and for guiding me throughout this journey.
I am extremely grateful to Dr. Eileen Marie Hanna as well. She was always ready to help me make sense of the confusion and offered great advice that sometimes flew over my head!
I am thankful to Dr. Nader El Khatib, he always lent me an ear whenever discussing the difficulties that I faced, and was supportive of my work at all times.
Special Thanks to Ralph Chahwan who supported me at all times and spent countless hours brainstorming with me.
I am also grateful for my family for pushing me to pursue my dreams. Their selflessness and unconditional love kept me standing even during the hardest days. I hope I can make them proud.
Last but not least, I would like to mention my friends and thank them for continuously supporting me and not giving up on me.

Modeling Software System Interactions Using Temporal Graphs and Graph Neural Networks: a Focus on Change Propagation

Manuella Latif Germanos

Abstract

The world is quickly adopting new technologies and evolving to rely on software systems for the simplest tasks. This prompts developers to expand their software systems by adding new product features. However, this expansion should be cautiously tackled in order to prevent the degradation of the quality of the software product. One challenge when modifying code - whether to patch a bug or add a feature- is being aware of which components will be affected by the change and amending possible misbehavior. In such cases, the study of change propagation or the impact of introducing a change is needed. By investigating how changing one component may impact the functionality of a dependency (another component), developers can prevent unexpected behavior and maintain the quality of their system. In this work, we tackle the change propagation problem by modeling the software system as a temporal graph where nodes represent system files and edges co-changeability i.e., the tendency of two files to change together. The graph representation is temporal so that nodes and edges can change with time reflecting addition of files in the system and changes in dependencies. We then employ a Temporal Graph Network and a Long Short-Term Memory model to predict which files will change when a modification is introduced to another file. We test our model on software systems of different functionality, size, and nature. Results show that our model significantly outperforms other recent published work.

Keywords: Change Impact Analysis, Change Propagation, Temporal Graphs, Graph Neural Network, Temporal Graph Network, Long Short-Term Memory, Deep Learning

Table of Contents

List of Figures

List of Tables

Chapter 1

Problem Statement

In 1822, Charles Babbage conceptualized and began developing the ”Difference Engine,” the first automatic computing machine to approximate polynomials [3]. More than a century later, in 1955, the Massachusetts Institute of Technology revolutionized computers and introduced the Whirlwind machine, the first digital computer with magnetic core RAM and real-time graphics [4]. With this technology in hand, it only took humanity 14 years to land on the moon [5].

Modern computers have only existed for a flicker of time compared to the existence of humanity. However, they managed to weave their way into every aspect of daily life. From tracking the grocery list to sending spaceships into outer space and back, humans have become dependent on technology, and developers have had to keep up with this constant growth. Developers must regularly update their software to introduce new features and patch any incorrect code. As their system grows, they must ensure that their program will be reliable and easy to modify in the future when needed. In this aspect, we speak of the maintainability of a software system.

Maintainability is a software structural quality[1]. It comprises multiple sub-characteristics such as modularity, reusability, modifiability, and testability[6]. The IEEE Standard Glossary of Software Engineering Terminology defines maintainability as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment[7]. Changing one component in a system can have a significant impact on other components and hence the maintainability

[1]A software quality attribute that relies heavily on the structure of the software.

of the entire system. This can be easily understood in the context of dependencies that multiple components can have. It is thus very important to be aware of and able to assess the impact of changing a particular component during the development phase [8]. As developers build their programs, dependencies will emerge between different components. These dependencies might be evident such as in the case of inheritance in Object-Oriented Programming, or obscure such as two components reading and writing to the same file. The obscure dependencies between the components will become even more concealed as the system ages and becomes more complex. As the system grows, modifications are applied to the code. Developers must ensure that they propagate the change of a component to all its affected dependencies to keep the components up-to-date and functional. An untrained developer might miss these obscure dependencies and might not account for the need to propagate a change correctly. In [9], the authors warned of such a scenario, calling it an ”Ignorant Surgery.” They argued that modifications done to the source code by developers who are not sufficiently knowledgeable are harmful because the developers will not correctly identify the components that need to be changed when introducing new features. In [10], the authors forewarn that these ignorant surgeries will introduce hard-to-find and hard-to-correct bugs, causing the quality of the software program to degrade. This makes understanding how changes propagate among the software components crucial to maintaining most, if not all, system quality attributes. This problem is often called ”software change propagation,” where the effort lies in predicting the set of components affected by a given modification. This set is often referred to as a ”change set.”

The change impact or change propagation problem can be studied at different granularities, such as the detection of change sets at the variable, method, class, and file levels [10]. Manual prediction of these sets is an uphill task requiring software engineering expertise and knowledge of the system. Therefore, researchers attempt to automate this job to obtain better results with less effort. To tackle this problem, researchers turned to artificial intelligence methods such as statistical analysis [11], heuristics [8], and machine learning models [12]. In this work, we propose a new approach to tackle this problem. We use a temporal graph to model the files of a system and their co-changeability metric. This metric reflects the proportion of times two files were changed together. Then, we apply a variant of the Graph Neural Network called Temporal Graph network to learn the pattern of change in the system and predict which files will change when we modify one. The chapters of this work are divided as follows: in Chapter 2, we review some of the most popular methods used to predict software change propagation. In Chapter 3, we visit key concepts and techniques used in this work. In Chapter 4, we discuss the proposed method. In Chapter 5 we detail the implementation and results of the method, we discuss the performance of the model, and possible threats to validity. Finally, we conclude in Chapter 6.

Chapter 2

Literature Review

In this chapter, we cover the literature review of change propagation. We also view some applications of Graph Neural Networks (GNNs) in different fields. Finally, we look into a variant of GNNs, called Temporal Graph Networks (TGNs), and list some of the work that employed it.

2.1 Change Propagation

To introduce new features to the software or patch existing bugs, developers must modify code fragments. These modifications might raise the need to patch other code fragments to prevent bugs from appearing. The need to change a code fragment due to a previous revision might chain and propagate across multiple files in the program. This incident is referred to as *change propagation*, and developers need to follow the changes needed in the software to prevent introducing new bugs. The bug or change propagation problem is a well-known problem tackled using multiple approaches. Some work attempted to estimate the components that change by studying their dependencies, while others used the historical changes in the software.

Among the work that attempted to estimate the change propagation patterns by understanding the dependencies of the software components are [13], where the authors attempted to locate risk containers in software programs to reduce the space that developers have to search through to find errors. The authors used three types of containers: design rule containers, resource containers, and use case containers. Design rule containers were based on the class diagram of the software, whereas resource containers grouped the files that share the same resources, such as files and databases. Finally, use case containers grouped files according to a sequence of use cases. The authors evaluated how well each container type isolated the change sets by computing the Containers in Common Co-change Probability (CCCP), which reflects the probability that two files changing together belong to the same container, and No Containers in Common Co-change Probability (NCCCP) which reflects the probability that two files changing together do not have any container in common. The authors then noted that design rule containers were the most effective when isolating change propagation as they achieved the highest CCCP (in the range [0.33-0.57]) and the lowest NCCP (in the range [0.25-0.39]) across four software systems.

In [8], the authors developed a wave propagation algorithm to predict which methods are affected when a change is introduced. Their strategy defined a core set and an impact set containing the method either directly affected by the change or indirectly. This method was tested on Ant and Jmeter and managed to better capture the propagation of change by predicting smaller change sets than the transitive closure, thus increasing the precision with a slight sacrifice in recall. The proposed method scored an average precision of 0.39 and an average recall of 0.8, compared to the transitive closure method, which scored an average precision of 0.3, and an average recall of 0.85.

In [14], the authors noted that change propagates in a single direction in the software program. Hence, they introduced the concept of propagation scope to evaluate the ability of a change to propagate. The proposed metric, called Edge Instability (EI), is dependent on the in-degree and out-degree of a component in the call graph. When computing the correlation of the proposed metrics against the actual change scope, EI exhibited a range of correlation [0.29-0.94], whereas previously used metrics, such as the clustering coefficient (CC), had a correlation range [-0.75,0.97]. Although CC managed to give a better correlation on some tested software systems, EI was consistently correlated with the change scope.

In [15], the authors proposed new source code dependencies and investigated their added value when predicting the change impact set. These dependencies are the "Include dependencies" extracted by analyzing the preprocessor directives of each source code file. The "Symbol dependencies" were extracted by analyzing the function calls which were going outside the file. Furthermore, the "Temporal dependencies" were computed by analyzing the sequence of execution of different methods present in the system. When these dependencies were added to already existing ones, they increased the precision and recall on various software programs by 17% to 89%.

The methods presented so far are convenient, especially when the changes in the software are not recorded. However, some fragments have hidden dependencies that are not easy to detect, and studying the historical changes of software can reveal them. We discuss next some work that used the history of the programs to understand their change propagation patterns.

In [16], the authors proposed a new data mining approach called Change Propagation Path (CPP). This approach employed the frequent pattern algorithm ECLAT to analyze the historical data within software repositories. The authors extracted the datasets from Github and used the ECLAT algorithm to generate patterns from the processed dataset. The authors then integrated their model into a web-based application to help developers predict the entity that must change and the entities they are working on. After timing the speed of the developers working with their tool and the developers working without their tool, the former group was able to maintain their program 50% faster.

In [10], the authors developed a prediction heuristic that takes as input a set of classes that were changed and outputs a group of classes that might also need change. The authors grouped their files using four methods: Developer change, where two files were considered related if they were changed by the same developer. Historical co-change, where two entities were considered related if they changed together previously. Code structure, where two entities were considered related if there exists some dependency between them (call, use, or define relationship). Finally, code layout, where if two entities exist in the same file or subsystem, then the entities were related. The authors tested these methods on five systems developed in C, and they noted that the historical co-change and code structure grouping achieved the best recall (0.87 and 0.83, respectively).

The researchers in [17] noted that change impact analysis tools that used the dependencies of the software were limited to homogeneous systems, i.e., systems that were developed using a single programming language. Hence, they introduced a new algorithm for mining coupling in heterogeneous systems i.e., systems that were developed using multiple programming languages. Their rule-based algorithm- Targeted Association Rule Mining for All Queries (TARMAQ)- studied the history of the program, and when given an input query, outputs a set of files that are most probable to be affected by the change. The authors compared the average precision of their proposed algorithm to that of ROSE and SVD, two well-known algorithms for extracting the impact sets of a change. Using the Friedman test with a post-hoc Wilcoxon test, the authors concluded that the average precision of TARMAQ was higher than that of ROSE and SVD.

In [18], the authors used association rule mining to find hidden dependencies in the code at different granularity. The authors also proposed using hyper-rule, where they combine different conventional association rules when determining the impact of a change. In other words, if different association rules can apply to a given query, the authors merge them to gain knowledge from all of them instead of the best ones. These hyper-rules improved the model performance, especially when the level of granularity was fine.

In [19], the authors proposed seven families of association rule mining techniques inspired by the previously mentioned algorithm TARMAQ and presented them in a tool called Adaptive Targeted Association Rule Mining (ATARI). Their approach uses adaptive rule mining to learn from a batch of changes instead of the entire software history. The rule mining process selected which transactions are important to learn the rules from and discards the remaining transactions. The goal of this work was to provide rules as effective as baseline methods while using less historical data. To do so, the authors compared all their proposed techniques to TARMAQ, and although TARMAQ managed to outperform the majority of their techniques in terms of Mean Average Precision (mAP), Tukey's HSD test showed that this difference is not significant. Therefore, the proposed family of techniques was able to extract powerful rules while a smaller history than TARMAQ.

The authors in [11] argued that, when studying the history of changes in a software program, newer changes should be given a higher weight when predicting future change propagation. They proposed a new co-changeability metric that lowers the weight of older commits while giving newer ones higher importance. The authors computed the changeability of a file by averaging its co-changeability across all the classes in the software program. When predicting a class change impact, all classes with a co-change probability higher than a specific cutoff were predicted as changed. This changeability measurement showed a positive Pearson correlation with Coupling between objects (CBO), indicating that it can reflect the dependencies of the files. Additionally, when the study compared the regular co-changeability metric with the one proposed in their work, the proposed metric showed a higher AUC value (in the range [0.63-0.84]) than the older one.

Network Science has also been used to address the issue of software changeability. The work in [20] used graph metrics to study the correlation between centrality measures and the scope of change propagation in an unweighted directed graph. The authors also proposed their metric, CIRank, which measures the co-changeability of two classes. The authors discovered that a small fraction of focus classes were responsible for most of the changes. Therefore, they probed for possible centrality measures that were indicative of a focus class. Their search concluded that CIRank was better fit to detect focus classes than the degree, closeness, betweenness, and PageRank centralities as it reached the maximum number of correct predictions in terms of top 5, top 10, top 20, and top 50 accuracy.

In [21], the authors used the change history to predict change propagation. They relied on the Concurrent Versions System (CVS) dataset to track the changes in the code. They only studied recent software changes that impacted 3 to 30 files as recommended by [22]. Their work then used an agglomerative clustering technique that relied on the weighted average of the Jaccard distance between the files in the change sets and the Jaccard distance between the sets of words associated with the change set. A membership matrix was then created for the clusters and source files. Moreover, if a file was changed, the authors referred to its membership matrix to select the cluster most susceptible to propagating this change. As the developer included changes in the code, the algorithm selected each pair of files, computed the change propagation of every file, and merged the predicted set of every file to get the final prediction. The authors tested their method on the JDT core and JDT UI systems and reported a higher f-measure than their competitors (when given one file, the f-measure was 0.3 on the JDT core system and 0.37 on the JDT UI system).

In [12], the authors used a Bayesian Belief Network (BBN) to predict the change propagation in the Azureus2 Java system. Their technique used both the dependency between the components of the file and the history of change to extract co-dependencies between the system components and predict change sets using the BBN model. The authors performed an ablation study by training their model on the dependency dataset alone (the BDM model), the historical change alone (the BHM model), and the dependency and history of changes together (the BDHM model). The authors also compared the performance to a control model where the software entities were assigned random probabilities of change. Their experiment showed that the best performing model was the BDHM model, which scored an overall accuracy of 0.513, whereas the other models scored an accuracy in the range of [0.26-0.47].

2.2 Graph Neural Networks

GNNs were first proposed in [23] as a variant of neural networks. The objective of GNN is to understand the underlying relationships in the data. This approach can be used in several areas of science and engineering. The GNN model quickly gained popularity, and many researchers started adopting it to analyze their data better. In this section, we describe some of the work in order to shed light on the diversity of the fields where GNNs have been successfully used.

Given the rich relational property of many biological systems, GNNs were frequently used to tackle biological problems. In [24], the authors tackled the problem of predicting possible molecule geometries using GNN. They proposed a deep generative graph neural network that learns the energy function of different molecules and attempts to predict the geometry of new molecules by minimizing their energy function and using structures seen in the train data. They used a complete undirected graph to represent a molecule. They fed their GNN molecules from three molecule datasets (QM9, COD, CSD) that possess distinct properties to allow the network to learn a plethora of geometries. When evaluating their model against other traditional methods, the authors used the root-mean-square deviation (RMSD) between generated and reference geometries. They found that their approach generated geometries that were closer to the molecule actual one than other methods and showed an RMSD range of [0.39-1.5] across the data sets.

In [25], the authors built a multimodal graph that included protein-protein interactions, drug-protein target interactions, and drug-drug interactions. The authors then employed Decagon, a Graph Convolutional Network (GCN), for multi-relational link prediction in multimodal networks. Their approach accurately predicted polypharmacy side effects and outperformed other baseline methods scoring an Area under the ROC curve value of 0.872, an area under the precision-recall value of 0.832, and an average precision at 50 of 0.8, outperforming all baseline models on all these metrics.

The work in [26] sought to overcome the scarcity of electronic medical records by building a classifier that was able to diagnose patients with common and rare diseases according to their symptoms. To do so, the authors created two heterogeneous graphs. The first was a medical concept graph linking diseases with their related symptoms. The second was a patient record graph that linked the patients to their observed symptoms from their electronic medical records. The authors then employed a GNN that learned from these graphs. Experiments were performed on a real-world electronic medical records dataset and the performance of the model was compared against baseline models. The Random Forest model managed to outperform the proposed technique when $K = 1$ in the top K predicted diseases. However, for all the remaining tests where $K > 1$, the proposed model outperformed all the remaining techniques in terms of recall (0.547-0.759), precision (0.224-0.393), and F-measure (0.346-0.457).

GNNs were also successfully used to model the spread of epidemics [27]. In this work, the authors did not only focus on people interactions with their environment but also on their closeness to clusters of infections. They used the SEIRD epidemic network, where a person could be Susceptible, Exposed, Infected/infectious, Recovered, or Deceased (SEIRD). The authors then built a GNN and tested it on a homogeneous network, in which any node had the same probability of being connected with all the others and a heterogeneous one that modeled the City of Boston and Cambridge, Massachusetts, in the United States. Surprisingly, the model performed better in the second case, where the accuracy reached 0.80, whereas, on the homogeneous graph, the accuracy was around 0.70. The researchers theorized that the non-trivial topology of the Boston scenario offered more information to learn.

In chemistry, GNNs have shown success in predicting structure-property relationships. In [28], the authors represented molecules as a graph by mapping the atoms to nodes and their bonds to edges. They aimed to develop GNN models that predicted three fuel ignition quality indicators: the derived cetane number (DCN), the research octane number (RON), and the motor octane number (MON) of oxygenated and nonoxygenated hydrocarbons. Due to the scarcity of data, the researchers employed ensemble learning. The authors trained multiple GNNs with randomly selected train and validation sets and combined them to build a more powerful regressor model. Compared to the state-of-the-art methods, this approach did not need molecular descriptors or structural group selection. It also exhibited a low MAE score (4.2-4.5), making it a reliable method.

The authors in [29] used GCNs to predict chemical reactivity. They trained the model on hundreds of thousands of reaction precedents covering a broad range of reaction types to help the model learn reactive sites most likely to change connectivity. Their model outperformed other state-of-the-art methods based on the United States Patent and Trademark Office (USPTO) dataset 410k/30k/40k reactions as it scored a top 1% accuracy of 0.856, a top 2% accuracy of 0.905, a top 3% accuracy of 0.928, and a top 5% accuracy of 0.934.

In [30], the authors employed GNNs for the task of drug discovery and molecular generation. Their proposed model, MG^2N^2, used multiple GNN models to build a graph representing a molecule sequentially. The experiments performed on the QM9 and Zinc datasets showed that this approach was able to generalize molecular patterns seen during the training phase without overfitting and outperformed most baseline models in terms of the validity of the generated molecules (0.753).

Additionally, GNNs have seen applications in Physics. For example, in [31], the authors relied on GNNs to perform particle reconstruction for their ability to overcome the irregular data in the high-energy physics field. The goal was to identify the nature of an incoming particle and estimate its energy from the energy deposition patterns in a simulated imaging calorimeter while minimizing latency and resource utilization. To do so, the authors modeled the system as a graph where vertices are entities, and the links represent the interactions of these entities. Each of the vertices, links, and the entire graph, possess some attributes. The authors considered Graph Networks a particular type of GNNs that consist of repeatable graph-to-graph mapping blocks. The model showed high reliability (the highest AUC was 0.98), low latency (155 cycles), and an acceptable resource utilization rate (56% of the digital signal processing units, 2.9% of the flip-flips, and 2.3% of the block RAM) when tested on simulation data. Although the proposed model still needed improvements before employing it in real life, this study showed that GNNs can perform fast inference and potentially be used in the future in real-time collider trigger systems.

GNNs were also successfully applied in the field of computer vision. In [32], the authors tackled the person re-identification task using Similarity-Guided Graph Neural Network (SGGNN). Their approach constructed a pairwise relationship between gallery-probe pairs, which helped understand the relationship information between different gallery images instead of only focusing on the relationship of every gallery image to the probe one. To validate their approach, SGGNN was tested against state-of-the-art methods on three datasets and managed to outperform the baseline method on every dataset by having a mAP range of [0.682-0.943], a top 1 accuracy range of [0.911-0.953], a top 5 accuracy of [0.88.4-0.991], and a top 10 accuracy of [0.912-0.996].

GNNs were also used on the image classification problem. In [33], the authors classified aerial images collected by satellite sensors and aerial cameras using an end-to-end aerial image classification model built on a Multiple Label Concept Graph (ML-CG). The model was tested on two datasets and showed high efficiency even when classifying images with many labels. The model outperformed state-of-the-art methods in terms of example-based recall (0.8994), F-scores (F1: 0.8542 and F2: 0.88), and the average label-based recall (0.8927), and F-scores (F1:0.999 and F2:0.8912), but it failed to do so in terms of example-based precision (0.8134 whereas the best model got 0.8212), and label-based precision (0.8853, whereas the best baseline model got 0.8878).

In [34], the authors used GNNs to detect objects from a Light Detection and Ranging (LiDAR) point cloud. Their proposed GNN, called Point-GNN, helps break the conventional grid encoding of the point cloud and uses graph encoding instead, which helps reduce information loss. This approach was tested on the KITTI data set, which contains three labels: car, pedestrian, and cyclist, and a model was trained to predict every label. These models reported an average precision in the range of [0.437-0.883] when predicting 3D images and an average precision range of [0.446-0.931] when predicting images taken at Bird's eye view.

In [35], the authors used GCNs to estimate the pose in 6D objects from RGB-Depth (RGB-D) images. Object 6D pose estimation helps predict 3D orientation and translation of rigid objects, an essential task for robot decision-making systems. The authors used a CNN to segment the input image and then transform it into a set of point clouds. A graph was then constructed based on the set of point clouds and fed to the GCN that predicts the dense pixel-wise correspondences that reflected the underlying relations between points. The GCN offered the advantage of exploiting known topological information of objects. This approach outperformed other state-of-the-art models when tested on the LINEMOD-OCCLUSION datasets = and had an average accuracy of 0.653.

GNNS received widespread attention in the field of Natural Language Processing (NLP). For example, in [36], the authors employed GNNs for text classification. Their work used a single text graph for a corpus based on word co-occurrence and document word relations. The graph is heterogeneous and contained word nodes and document nodes that learned words and document embedding jointly using a Graph Convolutional Network (GCN). When compared against baseline models on the datasets 20NG, R8, R52, and Oshumed, the proposed model achieved the highest accuracy values, which were in the range [0.68-0.97]. The model failed, however, to achieve the highest accuracy on the MR dataset, where it got an average accuracy of 0.76, whereas the best base model achieved an accuracy of 0.77. The authors theorized that their model failed to achieve high accuracy on the MR dataset because the documents are too short, so the graphs built are sparse and do not allow heavy message passing.

The work in [37] studied social media platforms to detect occurring events, such as the spread of the flu virus. In their work, the authors employed a new Event Detection model based on GNN -EDGNN- and extracted events on social media. The model employed the Conditional Random Field regularized Topic Model (CRFTM) to extract the topic of information from short texts, which is then used to build text-level graphs to overcome the sparsity of the graph generated from the short text problem. Their model was evaluated on a food-borne disease event dataset and outperformed existing models in precision (0.86), recall (0.84), and F1-measure (0.85).

The work in [38] used Graph Convolutional Networks (CGN) to build a syntax-aware Neural machine translation. The authors used GCNs on top of CNNS or Bi-directional RNNs to translate two challenging language pairs: English-German and English-Czech. The authors evaluated the performance of their model using multi-bleu ($Bleu_1$ and $Bleu_4$) and Kendal τ reordering scores. Results showed that, when paired with the GCN, the performance of CNN and Bi-RNN improved, especially in the $Bleu_4$ metric, to achieve a range of [23.3-23.9] on English-German translation and a range of [8.9-9.6] on the English-Czech translation.

In [39], the authors proposed RECON, a tool that used GNNs to identify relations in a sentence. Their model consisted of three elements, an RNN, a graph attention mechanism, and a GNN. Their approach effectively learned Knowledge Graph context and outperformed baseline models on the wikidata dataset in terms of micro precision (0.872), micro recall (0.8723), micro F-measure (0.872), macro recall (0.339), and macro F-measure (0.442). However, their model did not achieve the best macro precision (0.6359), whereby a baseline model achieved a higher value of 0.6921. The model was also evaluated on the NYT Freebase dataset, where the model achieved the highest top 10% precision (0.875) and top 30% precision (0.741).

Multiple recommender system problems were tackled using GNNs. In [40], the authors proposed GraphRec+, a GNN framework for social recommendations. The authors used three graphs: the user-user graph, the user-item graph, and the item-item graph. For each graph, the model needed to discover latent factors, i.e., hidden behaviors and characteristics, that might improve the recommendations made. Therefore, GraphRec+ had three components: the first component models users in an attempt to understand their latent factors. The second component models items in an attempt to understand their latent factors. The third aims to rate the predictions made. The authors tested their system on three datasets: Ciao, Epinions, and Flixster, and for every dataset, they performed two experiments: once training on 60% of the data, the other on 80% of the data. Across all data sets and for both experiments, the proposed model scored the lowest MAE [0.73-0.84] and RMSE [0.97-1].

In [41], the authors used a personal interest attention graph neural network (PIA-GNN) to predict the user's next click based on the user's current and historical sessions. In their approach, the succession of items clicked is modeled as a directed weighted graph whereby the weight of the link represents the number of times the user moved from the first item to the next. Using the cross-entropy loss function, the authors then trained a GNN with self-attention layers on the graph. The model was evaluated on two datasets: Yoochoose and Diginetica, and achieved the highest top 20 recall (0.7146 and 0.5262 respectively) and highest top 20 Mean Reverse Ranking (MRR) (0.3127 and 0.1839 respectively) against all baseline models.

GNNs were also used to tackle many combinatorial problems, such as the decision variant of the Traveling Salesman Problem [42, 43, 44], the graph coloring problem [45, 46], maximum constraints satisfaction problems such as Maximum Cut and Maximum Independent Set [47, 48], among many others.

2.3 Graph Neural Networks on Temporal Graphs

GNNs proved to be potent and versatile deep learning tools that were used to tackle problems from varying fields. However, the previously mentioned work focused on static graphs. Although the bulk of recent research papers used static graphs to model their system, more and more researchers are applying GNN, or variants of it, on temporal graphs. We review below some of this work.

In [49], the authors developed the Long Short-Term Memory R-GCN (LRGCN) model to tackle the path classification problem. The proposed method, called LRGCN-SAPE, used node correlations as an intra-time feature and studied the dependencies between two consequent graph snapshots as inter-time features. Their model has an LSTM component that learned the trends of edge failures and predicts future ones. The authors also proposed a new self-attention-based method for edge embedding called Self-Attentive Path Embedding (SAPE) that helped decide which features are the most important to learn from. The authors applied this approach to two tasks: predicting path failure in telecommunication networks and predicting path congestion in traffic networks. They compared their approach to many baseline models and found that their model reached significantly better levels of F-measure, where it scored 0.6189 when predicting path failure in telecommunication networks and 0.8675 when predicting path congestion.

The work in [50] attempted to predict cellular traffic by region as a first step to allow the demand-aware resource allocation of cellular data. The authors proposed Multi-View Spatio-Temporal Graph Network (MVSTGN) for cellular traffic prediction, which leveraged the Spatio-temporal characteristics of cellular data demand. Their model used the attention mechanism to aggregate important local information extracted from the node features and their relationships and important global information by aggregating information from past graph slices [1]. The authors divided the city of Milan, Italy, into an $N \times N$ matrix where each cell is a geographical square of $235m \times 235m$. After recording the cellular data traffic of this region for almost one year, the authors applied MVSTGN on the temporal graph generated and recorded the RMSE, MAE, and R^2 of their model on different types of cellular transactions (SMS, call, Internet). Their proposed model recorded an RMSE range of [30.94 - 165.04], an MAE range of [14.68-88.69], and an R^2 range of [0.88-0.95] and outperformed all baseline models.

In [51], the authors attempted to estimate travel time by leveraging spatial information as well as temporal ones. The authors mapped the roads as a directed graph where the vertices are road segments and connect with an edge the road segments that can reach each

[1]The temporal graph is a sequence of static graphs. Each of these static graphs is called a slice of the temporal one.

other. The authors used their model to predict how long a vehicle needs to traverse an edge by learning from multiple features, most importantly the traffic speed, which varies over time. To do so, the authors proposed their own Spatial-Temporal Graph Convolutional Networks (ST-GCN) architecture which learns from edge embeddings using multiple components such as the ST-GCN cell, which contained graph convolution layers that aggregate the information of the neighboring edges and a transformer layer to emphasize the information of the node itself. The information generated by these two components was fed to a transformer layer and then to a network of fully connected layers to make the predictions. The authors tested their approach on multiple data sets and reported that the RMSE range of their model was [121.14-52.35], their MAE range was [39.25-63.38], and their MAPE range was[0.14-0.25]. Their model outperformed other baseline methods.

In [52], the authors proposed an Attention-based interaction-Aware Spatio-Temporal GNN (AST-GNN) to predict pedestrian trajectories. Their approach was composed of two components: a Spatial GNN (SGNN) and a Temporal GNN (TGNN). The SGNN focused on capturing important interactions among pedestrians at a given time step, whereas the TGNN selects the important time steps to build the motion for the features of each pedestrian. The authors reported the average displacement error (ADE) and final displacement error (FDE) of their model on multiple datasets. The range of their reported ADE is [0.28-0.66], whereas their reported FDEs fell in the range of [0.45-1.02]. When compared to other state-of-the-art methods, their model managed to outperform the majority of them in a faster time.

In [53], the authors proposed a Spectral Temporal Graph Neural Network (SpecT-GNN) to predict the trajectory of agents. Their approach consisted of two units that share the same structure but are each applied to a different topology. One of the blocks is tasked with extracting the information from the agent graph, the other from the environment graphs. The agent graph contained information about the agents and their relationship with each other, whereas the environment graph contained information about the environment extracted from images. Their model employed a CNN to learn from the aggregated information of the two blocks and predict the trajectory of the agent. The authors compared their proposed approach to state-of-the-art methods on two datasets: the SDD dataset and the nuScenes dataset. Their model outperformed all its opponents in terms of $minADE_{20}$ and $minFDE_{20}$ scoring a $minADE_{20}$ of 8.21 and a $minFDE_{20}$ of

12.42 on the SDD dataset and on the nuScenes dataset, the model reported $minFDE_{20}$ values in the range of [0.28-1.87].

In [54], the authors tackled the task of wind forecasting using Spatio-temporal graphs. In their approach, they used an undirected graph where the wind farm regions are the nodes. The latter carried as features the wind speed and wind direction at every time step. If two farms were found to have a high feature correlation, they were connected by an edge that carried the distance between these two farms. The authors then used an LSTM to learn the temporal features of each node in the graph and extract these features at every time step. These features were then fed to a module of graph convolutional layers in order to make predictions. The authors tested their proposed approach on the Eastern Wind Integration dataset by predicting the wind speed at different intervals, starting at the next 10 minutes up to the next 3 hours. The work reported an RMSE range of [0.43-0.8] and an MAE of [0.3-0.78], which shows that the model outperformed state-of-the-art methods

In [55], the authors paired Recurrent Graph Networks GCNs) with Long Short-Term memory to propose a Temporal Graph Convolutional Neural Network (T-GCN)to forecast crimes of varying severity. The model first used a GCN to extract the embeddings of the regions in the graph. These embeddings were fed to an LSTM to learn the patterns of crime and predict future ones. They finally employed a CNN to decode the predictions of the LSTM and output the crimes that will occur in the graph regions. The authors evaluated their model using the Root Mean Square Error (RMSE), Mean Absolute Percentage Error (MAPE), and JessenShannon Divergence (JS) and compared their performance to Random Forest, XGBoost, and Conv-LSTM. The results showed that the proposed model had an RMSE range of [1.89-6.6], a MAPE range of [0.34-0.4], and a JS range of [0.06-0.11] on the four tested crime types. Compared to the other methods, T-GCN outperformed its counterparts in terms of RMSE and MAPE on three out of the four crime types.

In [56], the authors predicted the movements of a person by feeding a TGN a temporal graph representing the movement of their skeleton. Their ST-GCN architecture outperformed state-of-the-art methods on two datasets: Kinetics and NTU-RGB+D. The model recorded 0.307 accuracy on the Kinetics dataset and 0.815 and 0.884 accuracy on the NTU-RGB+D dataset in terms of X-sub (cross-subject, where they trained on clips from one set of actors and tested on clips from different actors) and X-View (cross view, where they trained on the angle of two cameras and tested on the angle of a third one).

In [57], the authors attempted to learn the functional connectivity between regions of the brain to better understand the brain connectome and used it to perform certain classification tasks of phenotypic characteristics. In this work, they focused on the task of gender classification. To do so, the authors modeled the functional connectivity of the brain as a temporal network where the regions of the brain are the nodes, and the edges link regions that showed activity correlation higher than a preset threshold. The authors then proposed a model called Spatio-Temporal Attention Graph Isomorphism Network (STAGIN). They evaluated their proposed approach on the Human Connectome Project (HCP) dataset and divided it into two groups: HCP-Rest, which showed the HCP-Rest which groups scans of the patient's brain while resting, and HCP-Task, which grouped the scans of the patient's brains while performing an activity. Their model outperformed state-of-the-art ones in terms of accuracy and AUC, scoring an accuracy value of 0.8701 and 0.882, as well as AUC values of 0.91 and 0.92, on the HCP-Rest and HCP-Task datasets, respectively.

Chapter 3

Background

This chapter explains the relevant concepts that help grasp the proposed approach. We start by explaining software quality. Then, we go over temporal graphs, the representation we use to model the system, and Temporal Graph Networks, the deep learning model we use to predict the impact of a change.

3.1 Software Quality

Software engineering focuses on many problems that arise when developing a software system, such as completing user requirements, meeting deadlines, managing hardware limitations, and maintaining the developed software system[58].

The practices of software engineering can be traced back to the early 1950s. However, research on software engineering emerged later, during the 1960s. Software engineering was considered a part of the computer science field and it is only in the 1980s that journals specific to the topic started emerging[58]. As the world became more reliant on software systems and access to programs became widespread, developers needed to produce complex programs that were usable, dependable, and safe. These attributes are commonly known as software quality attributes.

The ISO/IEC 9126 standard[59] summarizes the quality of a program according to six characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. Each of these characteristics is further dissected into subordinate ones that further detail the qualities of the system[60]. The most important of these qualities is, arguably, maintainability which is the effort and cost required to perform specific modifications to the software [61]. Nowadays, systems are constantly evolving to meet market requirements; hence, maintainability has become one of the most critical software quality attributes.

During the maintainability phase of the software system, developers have to introduce changes to the components of the system to either patch misbehavior or introduce new features. These changes usually impact more than one dependent system component. Therefore, developers must know which components will be affected once they introduce a change. This can be very difficult when conducting changes on legacy code or systems that the developers have not developed themselves, which is very common in the field [62]. This lack of knowledge of the software dependencies makes tracking how components might be affected by a change cumbersome and error-prone. Thus, instead of ameliorating the overall system quality, developers may unknowingly introduce further, more complicated bugs[10]. That being the case, assessing the impact of a change is essential to maintaining the system quality.

In this work, we attempt to answer the following question: knowing that the developer will change file f_i, which other files will also need to change? The cause of this change propagation might be a modification of a code section on which the other component is dependent. For example, in the Java-based system presented in Figure 3.1, class B is a child of class A. Class B would use some of the methods and variables that appear in class A. If a change is introduced to class A, for example, removing a variable such as *var1*, the developer would need to account for this in class B.

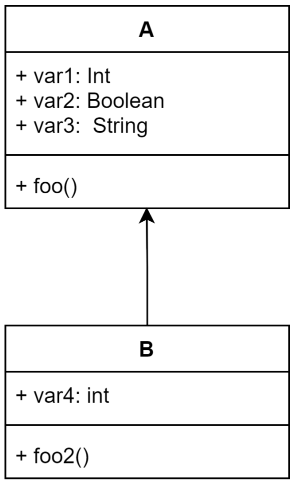


Figure 3.1: Class diagram of Java classes A and B where B is a child of A.

In some cases, the dependencies are hidden, for example, two components reading from and writing to the same file. Figure 3.2 represents how a change introduced to a file might propagate to one of its dependent files and might further propagate to other dependencies of that file.

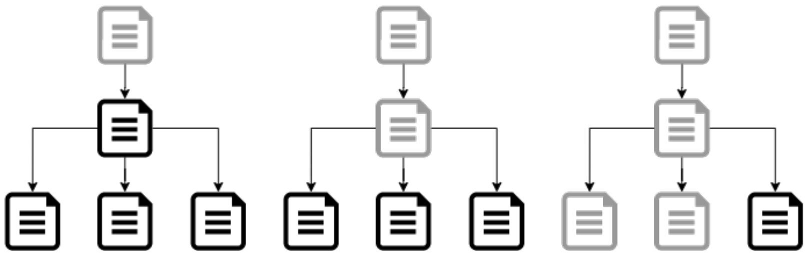


Figure 3.2: Change propagating to different dependencies. Items in grey are the ones that have changed.

Change impact analysis estimates how an introduced change will ripple across the components of a system and which of them will need to be changed as well[63]. This analysis can be done at different levels, such as methods, classes, or files. In our work, we focus on the file level granularity and cover both homogeneous systems which were developed using a single programming language, as well as heterogeneous ones that were developed using multiple languages. We also focus on different programming paradigms: object-oriented and other. Therefore, in order to build a general approach, we perform impact analysis and predict how a change propagates at the file level of the system.

3.2 Temporal Graph

Graph theory is a branch of mathematics that studies the impact of graphs on system properties[64]. Graphs can be used to represent relationships between different components within a complex system. In such graphs, system components are represented as vertices and relationships between them as edges connecting the vertices. More formally, a graph $G = (V, E)$ has a set of vertices V that contains the components of a system and a set of edges E that summarizes the relationships between the vertices of V. An edge (v_i, v_j) represents the relationship between vertices v_i and $v_j \in V$.

If the relationship between the vertices is symmetric, the graph is undirected i.e. edge (v_i, v_j) is equivalent to edge (v_j, v_i). Otherwise, the graph is directed and edge (v_i, v_j) is different from edge (v_j, v_i). The graph is said to be weighted if an additional attribute is

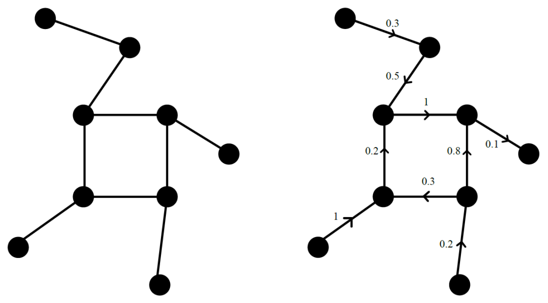


Figure 3.3: A undirected, unweighted graph(left) and a directed weighted graph(right)

added to the edge to quantify the relationship between the two vertices. In such a case, each edge (v_i, v_j) has a weight attribute $w_{i,j}$. Figure 3.3 shows an example of an undirected unweighted graph and a weighted directed one.

In this work, we resort to directed weighted graphs to model our problem. Given a software system, each file in the system is represented as a vertex, and edges link two files that changed together. The edge that connects A to B reflects the proportion of B was involved in a change that modified A as well.

To model the system as a temporal graph, we start with the first recorded change and add all the modified files to the graph as nodes and link them together. This would make the graph of the first time step a complete one. We then iteratively build the temporal graph by adding a time step for every recorded change and updating the nodes, edges, and their co-changeability value by referring to graph of the previous time step and the modified files of the current change set. The weight of the added edges is assigned as a co-changeability metric. The value of co-changeability between two files A and B is the answer to the following question: from the previous change set, how many times was file A changed when file B was also changed? We discuss in detail how to compute co-changeability in Chapter 4.

Usually, graphs are used to represent or model static systems. In such situations, neither vertices nor edges change throughout the model lifetime. This is not the case for many real-life applications, such as the growth of the World Wide Web[64] and traffic flow in computer networks[65]. Similarly, in this work, the graph that we use to represent one system does not remain static but evolves with the changes introduced to the system. It is often the case that the degree of vertices changes as well when new vertices are introduced to the graph as new files are added to the underlying software system. Such graphs are named *temporal*, *dynamic*, or *evolving* graphs [66].

Temporal graphs are represented as an ordered sequence of static graphs capturing, at each timestamp, the system characteristics at that time. Figure 3.4 shows an example of a growing temporal graph where nodes and edges are added from one timestamp to the next.

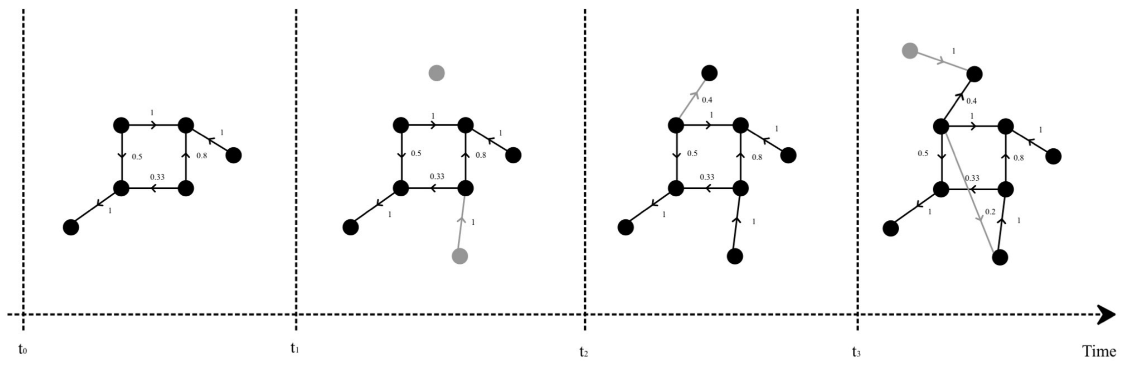


Figure 3.4: A temporal graph through four timestamps. Vertices (files) and edges(co-changeability of files) are added through time

3.3 Artificial Neural Networks and Deep Learning

Machine learning is a keystone in technological development and is frequently used to tackle many complex problems. Conventional machine learning techniques include probabilistic models, decision trees, and support vector machines, among others. In many real-life applications, these conventional techniques have trouble finding patterns in raw data. This requires experts in the field to clean and process the data[67]. Representation learning is a branch of machine learning that develops models that can learn and find patterns from raw data. The methods of this branch can differentiate important features from irrelevant ones when solving a problem. This makes the models more readily usable, as they require little to no data pre-processing [68]. Deep learning methods are representation methods obtained by stacking simple modules that transform the data, layer by layer, into representations at a higher, more abstract level. With the use of sufficient transformations, very complex functions can be learned [67]. Deep learning solved previously intricate problems for the machine learning community as it turned out to be very effective in finding complex patterns in high-dimensional systems. Such problems include finding image patterns[69, 70], speech recognition[71, 72], text translation[73, 74], microarray data analysis[75, 76], and reconstructing brain circuits[77, 78].

These problems required the application of deep learning methods on images, text, signals, and data frames. Several deep learning methods were developed for these types of data such as Multilayer Perceptrons (MLPs)[79], Convolutional Neural Networks (CNN)[80], Long Short Term Memory Networks (LSTMs)[81], Recurrent Neural Networks (RNNs)[82], Generative Adversarial Networks (GANs)[83], and Restricted Boltzmann Machines(RBMs) [84]. These complex deep learning techniques are built using a simple unit called *perceptron*, and the mechanism we use to connect different perceptrons helps construct many powerful models. In the next section, we go over the details of the basic architecture of an artificial neural network as deep learning is nothing more than a neural network with many layers.

3.3.1 The Perceptron and Artificial Neural Networks

A vanilla perceptron is a simple unit that takes a vector of real numbers as input, performs some simple calculation, and then outputs either -1 or 1 (Figure 3.5). The calculations performed are a linear combination of the input vector with a weight vector that the perceptron learns. If the sum of this combination is positive, the perceptron outputs 1; otherwise, it outputs -1. Formula 3.1 depicts the output o of a perceptron when given an input vector $< x_1, x_3, ..., x_n >$ and w_i is real-valued weight that controls the impact of input feature x_i on the output. In this formula, $-w_0$ is a threshold that the linear combination must surpass for the perceptron to output 1.

$$o(x_1, x_2, ...x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + +w_2x_2 + ... + w_nx_n > 0 \\ \text{-1} & \text{otherwise} \end{cases} \tag{3.1}$$

In order to simplifiy Formula 3.1, a constant input $x_0 = 1$ is added so that the inequality can be written as $\sum_{i=0}^{n} w_ix_i > 0$ which can also be formulated as a vector $\overrightarrow{w}.\overrightarrow{x} > 0$ and Formula 3.1 can thus be simplified as shown in Equation 3.2.

$$o(\overrightarrow{x}) = sgn(\overrightarrow{w}.\overrightarrow{x}) \tag{3.2}$$

Where

$$sgn(y) = \begin{cases} 1 & \text{if } y > 0 \\ \text{-1} & \text{otherwise} \end{cases} \tag{3.3}$$

The *sgn* function is called *activation function* as it dictates how the weighted sum of the inputs will be transformed into the output. As we will see later on in this section and Section 3.3.3, many possible activation functions can replace the *sgn*. Figure 3.5 represents this perceptron.

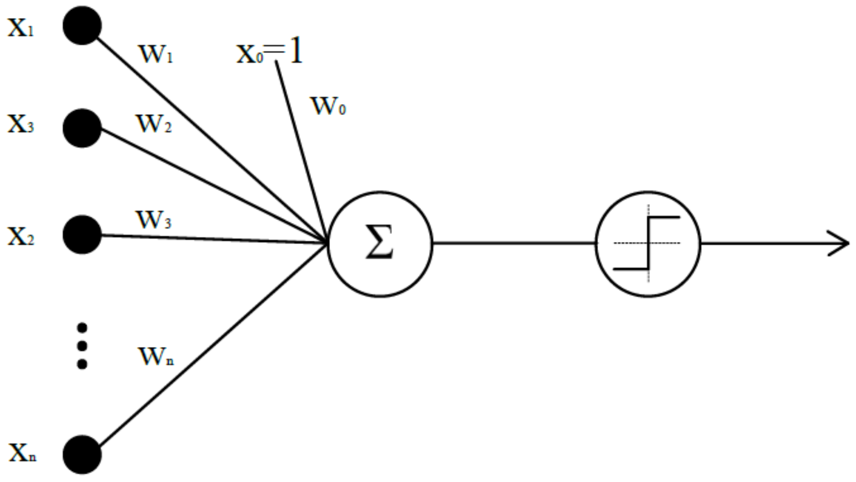


Figure 3.5: A perceptron

Looking back at the equations, we can see that a perceptron represents a hyperplane decision surface in the n-dimensional space of instance, where n is the number of inputs the perceptron receives. All the instances that lie on one side of the hyperplane are labeled with one class, whereas the points lying on the second side are labeled with the other as shown in Figure 3.6.

This perceptron is powerful enough to correctly classify data points of simple binary operators such as the AND, OR, $NAND$, and NOR. This is done by changing the output function and changing w_0, w_1, and w_2. For example, to correctly classify data points of the AND function, the perceptron could use the ReLU as its activation function [85]. The ReLU function is computed using Formula 3.4 and presented in Figure 3.7.

$$ReLU(x) = max(x, 0) \tag{3.4}$$

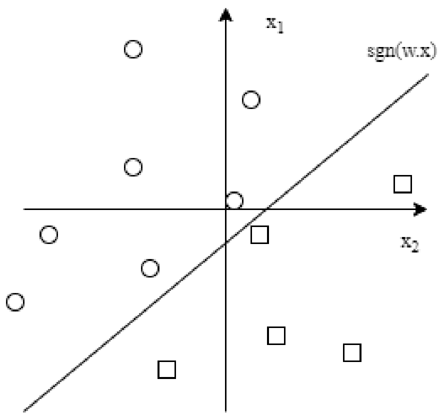


Figure 3.6: The space of inputs x_1 and x_2 with the hyperplane $o(x)$ separating the labels of the data points

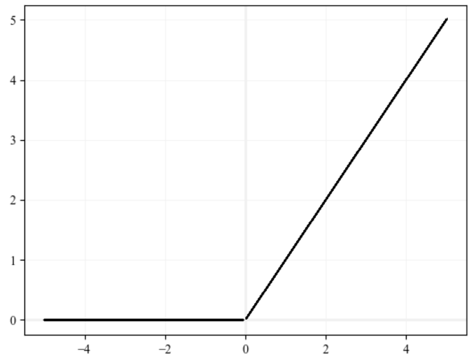


Figure 3.7: The ReLU function

The perceptron could then set $w_0 = -1$, and w_1, and w_2 both equal to one. This way, both x_1 and x_2 should be equal to 1 for the perceptron to evaluate to 1. The equation of such perceptron is shown in Formula 3.5.

$$o = ReLU(x_1 + x_2 - 1) \tag{3.5}$$

Similarly, the perceptron can learn to label the OR data points by setting $w_0 = -1$, and w_1 and w_2 both equal to 2. This ensures that as long as either x_1 or x_2 is 1, the perceptron evaluates to 1. The equation of this perceptron is shown in Equation 3.6.

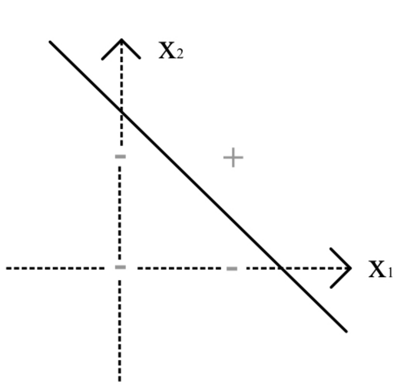
$$o = ReLU(2x_1 + 2x_2 - 1) \tag{3.6}$$

However, a simple perceptron cannot represent the binary operator XOR (Table 3.1) and this is because the data points are not linearly separable.

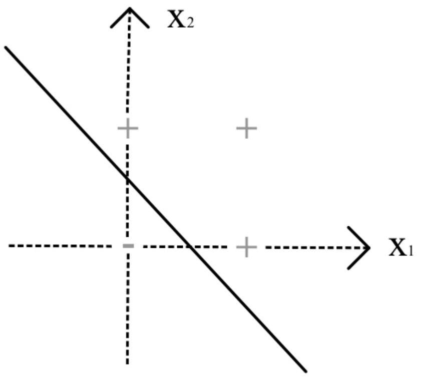
Table 3.1: the XOR operator

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

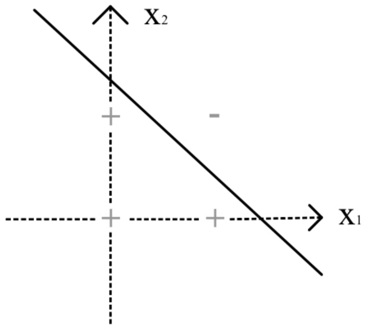
Figure 3.8 visualizes the distribution of data points for the AND, OR, $NAND$, and XOR operators. The figure shows that the data points of AND, OR, and $NAND$ are linearly separable since the plotted lines can separate the positive data points from the negative ones. Figure 3.8d shows that no single line can separate the data points of the XOR operator. Therefore, a single perceptron cannot learn this function.



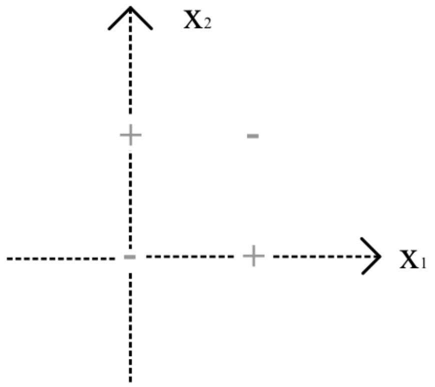
(a) Data points of *AND* operator



(b) Data points of *OR* operator



(c) Data points of *NAND* operator



(d) Data points of *XOR* operator

Figure 3.8: Data points plot of different binary operators

The challenge of classifying non-linearly separable data points is tackled by stacking the perceptrons one after the other to build an Artificial Neural Network (ANN). Figure 3.9 shows an example of an ANN where the first layer of perceptrons is called the *input layer*, the last layer of perceptrons is called the *output layer*, and everything in between is referred to as *hidden layers*. This type of network is called a *feed-forward network*, as every perceptron, now called a *neuron*, in one layer will send its output to be input to neurons in the next layer. This architecture can have multiple hidden layers, each containing a varying number of neurons. By stacking these layers, the ANN can learn more complicated functions.

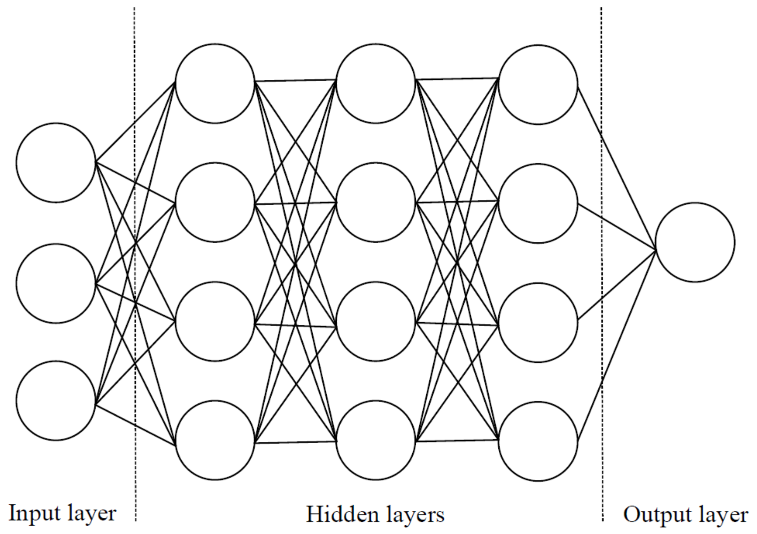


Figure 3.9: An ANN

One way to tackle the task of classifying the XOR operator is to break down its operations into a series of linear ones. The XOR equation could be re-written as: $XOR(x_1, x_2) = AND(NOT(AND(x_1, x_2)), OR(x_1, x_2))$. This series of simple binary operators can now be represented using a series of neurons. These neurons use an activation function θ defined in Formula 3.7.

$$\theta = \begin{cases} 1 & \text{if } \sum_{i=0}^{2} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

An ANN with two hidden layers represented in Figure 3.10 can then learn to classify the XOR data points.

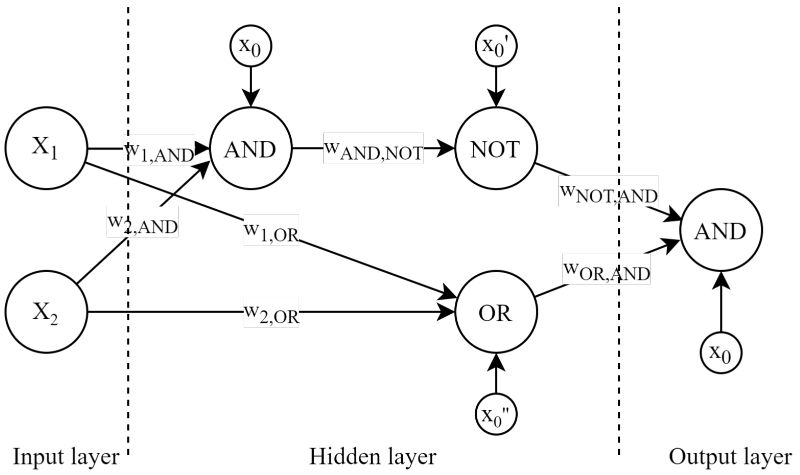


Figure 3.10: A possible ANN used to represent the XOR operator. Neurons are labeled with the operator that they perform.

In order for the neurons to perform the binary operators assigned to them, the weight can be set to the following values: $w_{1,AND} = w_{1,OR} = w_{2,AND} = w_{2,OR} = 1, w_{AND,NOT} = -1, w_{NOT,AND} = 1, W_{OR,AND} = 1$. Furthermore, the biases of the x_0, x_0', and x_0'' neurons should be -2, -1, and 0.5, respectively.

3.3.2 The Backpropagation Algorithm

Throughout the years, researchers developed many techniques to assign appropriate weights to the network such as the backpropagation algorithm [86], linear programming [87], and evolutionary algorithms [88], the most widely used one being backpropagation.

The backpropagation algorithm tries to find the best combination of weights for the network in order to minimize some error function. This is referred to as the *training process*. The algorithm starts by randomly initializing the weights of the network. The data points are then fed to the input layer of the network and then forwarded all the way, through the different hidden layers, to the output layer. After the network makes its prediction (classifies an instance), an error function is computed. This function measures the difference between the predicted label of the data point and the actual one. One frequently used error function is the one shown in Formula 3.8.

$$E(\overrightarrow{w}) = \frac{1}{2}(t_d - o_d)^2 \tag{3.8}$$

Where t_d is the target value for the data point d, and o_d is the output of the model for this data point.

The algorithm then computes the gradient of the error, $\nabla E(\overrightarrow{w})$, which shows the way of the steepest ascent. The gradient $\nabla E(\overrightarrow{w})$ is a vector whose components are the partial derivatives of the error function with respect to every weight in the network. The reasoning behind this is that the weights of a neuron in layer L_i impact the output that is sent to the neurons in layer L_{i+1}. This is then used to compute the output of the nodes in Layer L_{i+1} which, in turn, is sent to nodes in Layer L_{i+2} and so on until the final output layer. Therefore, all the weights in the network will eventually impact the final output. Backpropagation attempts to quantify the error caused by a weight w_i in the network by taking the partial derivative of the error function with respect to w_i(Formula 3.9).

$$\nabla E(\overrightarrow{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_n}\right] \tag{3.9}$$

where $\frac{\partial E}{\partial w_i}$ is the partial derivative of E with respect to the weight w_i, and $\nabla E(\overrightarrow{w})$ is the gradient of E with respect to $\overrightarrow{w}$ Since $\nabla E(\overrightarrow{w})$ shows the direction of the steepest ascent on the error function, $-\nabla E(\overrightarrow{w})$ shows the direction that leads to the fastest decrease in the error. Therefore, the weights are updated using the $-\nabla E(\overrightarrow{w})$ vector following Formula 3.10.

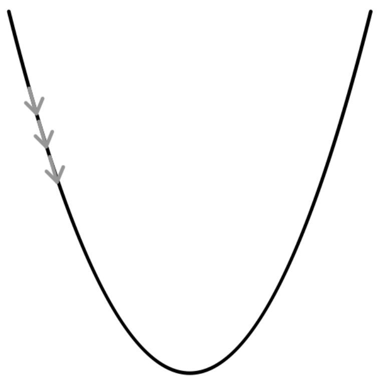
$$\overrightarrow{w} = \overrightarrow{w} - \eta \nabla E(\overrightarrow{w}) \tag{3.10}$$

Where η is the learning rate that controls how much the data points affect the weights update.

Or, in component form, the weights are updated using Formula 3.11.

$$w_i = w_i - \eta \nabla \frac{\partial E}{\partial w_i} \tag{3.11}$$

The network would be too slow to learn if this rate is too low. However, the network might overshoot and miss the minima if the rate is too high. Figure 3.11 visualizes this.


(a) low η, the network is slow to learn

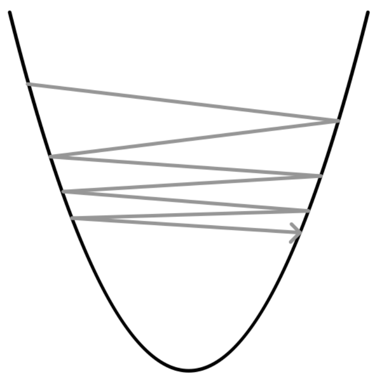

(b) large η, the network is missing the minima

Figure 3.11: Impact of η on the learning process

Although backpropagation tends to reach local minima instead of global ones, it can still find acceptable weight assignments to the network. Its pseudo-code is presented in Algorithm 1. Some stopping criteria that can be used are the model reaching a plateau in the error function, or repeating the training process a certain number of epochs, i.e. iterations.

Algorithm 1 Backpropagation

```
while stopping criteria not met do
    Feed input to the data
    Propagate the information forward from one neuron to the next
    Make prediction
    Compute the error function
    Compute the derivatives of the error with respect to the network weights
    Adjust the weights to minimize the error
end while
```

By stacking simple perceptrons, this ANN is able to learn a function that a perceptron alone cannot. Many complex architectures were built to tackle complex problems. However, one fallback of vanilla ANNs is that they do not consider possible correlations between the data points. As a matter of fact, the network learns from every data point without establishing a connection with the one before it. Therefore, this architecture suffers when fed sequential data such as videos, sound recordings, or temporal data. To efficiently learn from these types of data, the notion of memory was added to the network in more advanced architectures where each perceptron, now called a *unit*, has its memory - called *state*- that stores some information that was previously fed to it. The first architecture with such a characteristic was the recurrent neural network (RNN) which had only one hidden state. This model suffered from long-term memory loss and could not save values for many previous iterations. This problem was termed the *vanishing gradient problem* and it occurs in deep models that have many hidden layers [89]. This happens when computing the gradient of the earlier layers as the model needs to multiply many derivatives to know the $\nabla \frac{\partial E}{\partial w_i}$ of a weight that belongs to the first few hidden layers. And if these derivatives are smaller than one, chaining these multiplications will lead to a negligible number, so the weight will not be updated by a significant factor and these layers become stale. For example, if the network is using the *sigmoid* function as an activation function, its derivative is much smaller than one as seen in Figure 3.12. If the network contains n hidden layers, the network would need to multiply n derivatives to correct the weight vector of the first hidden layer, thus decreasing the gradient exponentially n times. As the number of hidden layer increases, the gradient of the early hidden layers would quickly decrease until reaching values asymptotic to zero, thus the weight vectors would not be corrected adequately.

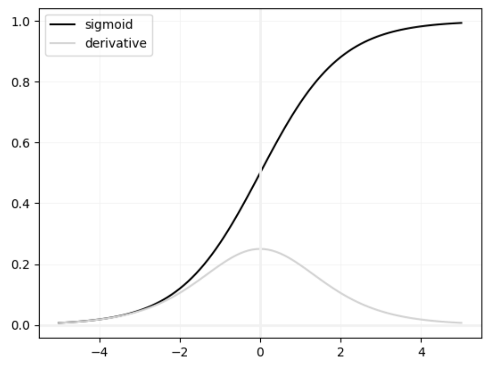


Figure 3.12: The sigmoid function and its derivative

To overcome this, a new architecture was proposed: The Long Short Term memory model that we describe in the next section.

3.3.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) is a deep learning model capable of learning order dependencies[1] in sequence prediction problems. LSTM is a trainable model that can store information and detect patterns while being immune to noise in the data (mislabeled data points or outliers). The model receives data, transforms it into information, and saves this data in its memory. Data is fed to this model during cycles; as more cycles pass, the LSTM should not lose the information captured from earlier cycles.

LSTMs build over RNN by attaching to each neuron in the network two different memories and multiple gates that control the flow of information into these memories. LSTMs have been shown to save information for longer cycles than their RNN counterparts [90] since they do not suffer from the vanishing gradient problem[89]. Therefore, this work will employ LSTMs to learn the co-changeability pattern of the files in a system.

The LSTM model is composed of multiple LSTM units. A standard LSTM unit comprises a *cell state*, a *hidden state*, and three *gates*: an *input gate*, an *output gate*, and a *forget gate* (Figure 3.13). The cell and hidden state are tasked with saving the information at arbitrary time intervals. The cell state holds the information of many past timestamps while the hidden state carries the information of the last seen timestamp. For this reason, the hidden state is known as the unit's short-term memory while the cell state is the unit's long-term memory. The cell state offers an advantage to LSTM that is missing in vanilla RNN. The gates are tasked with regulating the flow of information into and out of the cell.

[1]where the input i is not completely independent from the input $i-1$. Such as the case of videos or time series.

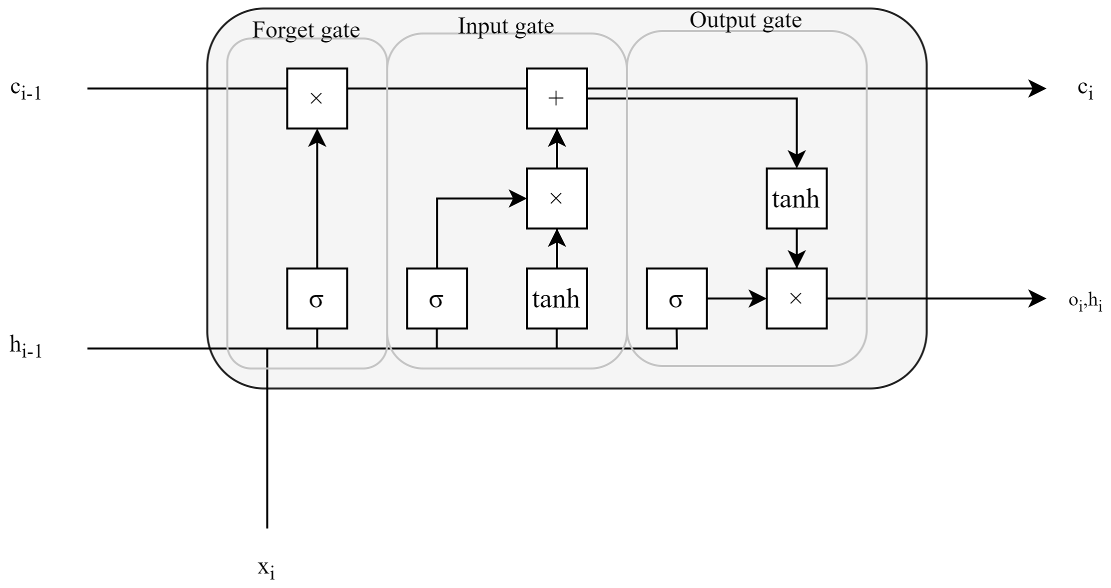


Figure 3.13: The gates of an LSTM unit [1]

The LSTM unit uses different activation functions in its computations. We list below the two main ones[91]:

1. The *sigmoid function* (σ)

   All the gates usually employ the non-linear *sigmoid* function to determine how much of the information is passed to the next stage. The *sigmoid* function has an output value in the range [0-1] and is computed using Formula 3.12.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.12}$$

   Figure 3.14 shows the plot of the *sigmoid* function.

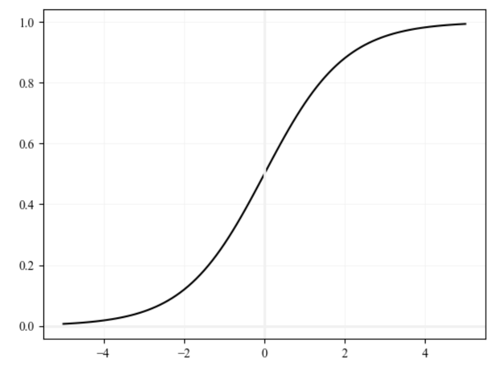


Figure 3.14: The *sigmoid* function

2. The *hyperbolic tangent function (tanh)*

   The gates usually employ the non-linear *tanh* function when updating the cell state and the hidden state. This function outputs values in the range [-1,1] which allows for increases or decreases in the cell state and hidden state. The *tanh* function is computed using Formula 3.13.

$$\alpha(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.13}$$

   Figure 3.15 shows the plot of the *tanh* function.

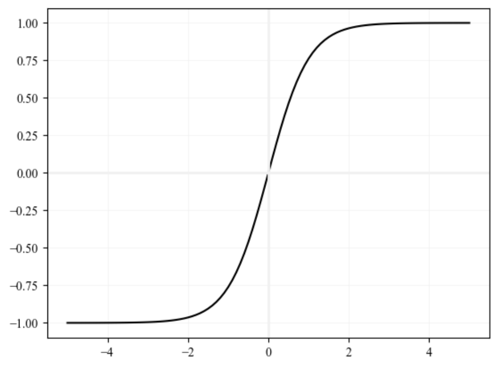


Figure 3.15: The tanh function

Each of the three gates manages the passage of information at a given stage in the process [92]. The cycle of updating the cell state and making a prediction at a given time step is the following:

Step 1: The forget gate chooses which information previously saved in the cell should be removed. This gate attempts to remove data points that might be outliers or noisy. The forget gate manages the passage of the information gained from the previous cycles (timestamps) using Formula 3.14.

$$f_t = \sigma(x_t \times U_f + H_{t-1} \times W_f) \tag{3.14}$$

where:

x_t: the input at the current timestamp t.

U_f: weight associated with the input.

H_{t-1}: the hidden state of the previous timestamp $(t-1)$.

W_f: weight matrix of the hidden state [2].

The *sigmoid* function helps restrict f_t to the range [0-1] which reflects the impact the output of this gate will have on upcoming operations. After computing f_t, it is multiplied by the cell state at time t (C_t). If $f_t \approx 0$, then the cell state C_t is discarded for this timestamp; the unit forgets everything. If $f_t \approx 1$, the cell state is passed as is, i.e. the unit will never forget anything. The values between 0 and 1 dictate how strong the output of this gate will affect the value of the cell state later on.

Step 2: The input gate deals with the information passed to the unit at the current timestamp. It attempts to learn the importance of the information at the current timestamp and whether it should be saved into the cell state or not. The equation of this gate is shown in Formula 3.15.

$$i_t = \sigma(x_t \times U_i + H_{t-1} \times W_i) \tag{3.15}$$

where:

x_t: input at the current timestamp

U_i: weight associated with the input.

[2]The f in U_f and W_f stands for "forget".

H_{t-1}: hidden state of the previous timestamp $(t-1)$

W_i: weight matrix of the hidden state[3].

The *sigmoid* function also limits i_t to the range [0-1]. The computed value of i_t is also used to update the cell state, and like the forget gate, the range [0-1] of the *sigmoid* function helps control the effect of the input when updating the cell state.

Step 3: To update the cell state, the unit must first compute the new information using Formula 3.16.

$$N_t = \alpha(x_t \times U_c + H_{t-1} \times W_c) \tag{3.16}$$

where:

x_t: the input at the current timestamp t.

U_c: weight associated with the cell state.

H_{t-1}: the hidden state of the previous timestamp $(t-1)$.

W_c: weight matrix of the cell state.

The cell state is then updated using Formula 3.17 where f_t is used to regulate the importance of the previous cell state.

$$c_t = f_t \times c_{t-1} + i_t \times N_t \tag{3.17}$$

As mentioned earlier, a low f_t makes the unit quickly forget older information, whereas a higher f_t helps it store older information. Also, i_t helps regulate the influence of the input of the current timestamp, x_t, on the cell state. The Tanh function is used instead of the *sigmoid* function in Equation 3.16 to allow for negative values of N_t ehich would cause values to be subtracted from the cell state,i.e. forgotten.

Step 4: The output gate extracts the valuable information from the now updated cell memory and generates the output of the LSTM unit. The information that the output gate finds useful is stored in the hidden state of the next time step. Formula 3.18 is that

[3]The i in W_i and U_i stands for input.

of the output gate.

$$o_t = \sigma(x_t \times U_o + H_{t-1} \times W_o) \tag{3.18}$$

where:

x_t: the input at the current timestamp t.

U_o: weight associated with the output.

H_{t-i}: the hidden state of the previous timestamp $(t - i)$.

W_o: weight matrix of the output.

To update the hidden state, the unit uses formula 3.19.

$$H_t = o_t \times tanh(C_t) \tag{3.19}$$

whereby H_t is the hidden state at timestamp t.

And finally, the output is computed using Formula 3.20.

$$Output = \sigma(H_t) \tag{3.20}$$

In our work, we used the *sigmoid* function to compute the output, as it is the go-to function when performing binary classification using deep learning models. The *sigmoid* function will restrict the output values to the range [0-1]. Therefore, if the output value is greater than 0.5, it is predicted to belong to one class; otherwise, it is classified as belonging to the other (Algorithm 2). The threshold value 0.5 is used in this case as it separates the cases when a model predicts a data point as belonging to one class more than the other. For example, if the model outputs the value 0.7, this number is close to 1 than 0, so the model is predicting that the input data point most probably belongs to label 1 more than it belongs to label 0. The output 0.2 however, reflects that the input data point most probably belongs to label 0 instead of 1.

Just like with perceptron, these units can be stacked one after the other to create a deeper LSTM model. In a stacked LSTM model, the output of one LSTM unit is the input to the

Algorithm 2 Assigning a a class class to the LSTM prediction

```
$output$ ←[0-1]
**if** $output > 0.5$ **then**
    $predict$ ← "affected by change"
**else**
    $predict$ ← "not affected by change"
**end if**
```

next. After creating and connecting the units, the model uses *BackPropagation Through Time* (BPTT) [93] to find the best weights to perform the task at hand.

The BPTT is derived from the Backpropagation algorithm used to train the weights of a regular ANN. The challenge of applying BPTT derives from the fact that in an LSTM, the output of the unit depends on the cell state and hidden state of the previous timestamp, which in turn depends on the cell state and the hidden state of the timestamp before that so on until the very first timestamps or the first data point that the network receives. Hence, to adjust the weights of a unit, the algorithm should go back through all timestamps and "unroll" the LSTM unit i.e. revisit all previous outputs since they influence the current output so that it can correctly compute the error function. Figure 3.16 shows how an LSTM unit can be unrolled to revisit old cell states and hidden states.

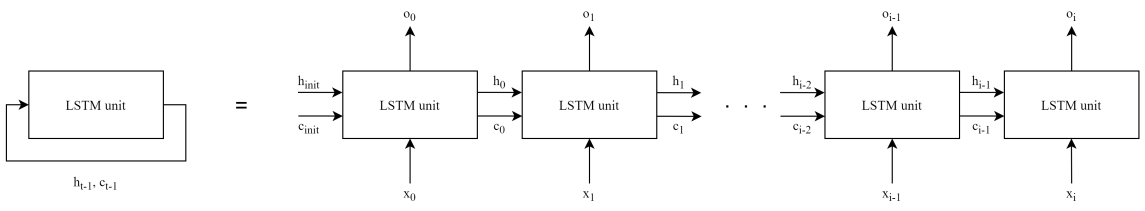


Figure 3.16: An unrolled LSTM [2]

Once the LSTM unit is unrolled, the weight of passing information from time step t_i to time step t_{i+1}, i.e. the weight associated with the cell state, is the same for all t_i. So the network BPTT can find the gradient with respect to all the weights of the gates and apply the weight correction method seen in Formula 3.10. The network trains by repeatedly visiting all data points and learning from them. After updating the internal weights of the units and the weights connecting the units with each other, the LSTM is ready to classify new, unseen data points. Algorithm 3 shows this process.

One additional complex data forms that deep learning methods began exploring are graphs which can readily be used to model interactions between components of various

Algorithm 3 The BPTT algorithm

```
$X \leftarrow$ training features
$Y \leftarrow$ training labels
Initialize weights of LSTM
for iteration in epoch do
    Forward propagate all the training points
    Record the predictions of the model
    Unroll the model
    Compute the error function
    Compute the gradient vector
    Correct every weight in the model using $w_i = w_i - \eta\nabla\frac{\partial E}{\partial w_i}$
end for
```

systems. To learn from this type of data, a new deep learning model has been created: the Graph Neural Network (GNN)[94], with its variant, the Temporal Graph Network (TGN)[95].

3.3.4 Temporal Graph Networks

Temporal Graph Networks (TGN) are one variant of Graph Neural Networks. The latter are a more recent class of deep learning methods that learns from the graph data structure [94]. Their use in solving various problems is making them more and more popular [95]. In GNNs, the nodes iteratively update their states by receiving messages from neighboring nodes. Through message-passing, a GNN updates every node's features by aggregating features from the adjacent nodes. In this work, we focus on Temporal Graph Network - the variant that was developed with temporal graphs specifically in mind. Although variants of this architecture were previously presented to tackle many problems, the work in [95] presents the general TGN architecture and explains the functionality of its components.

The general GNN architecture for temporal graphs is defined as an encoder-decoder pair. The encoder maps the temporal graph into node embeddings, whereas the decoder converts these node embeddings to make a prediction. To build the encoder, five modules are defined [95]. We list them below:

1. **Memory (or state)**: at a given time t, every node i that the model has seen so far has its own memory vector $s_i(t)$. This vector summarizes the history of the node and is updated after every interaction (i.e. the file being impacted by a change), even after training. Memory helps TGN keep track of long-term dependencies. In

our work, the memory of a node stores whether the node was changed at time t_i or not.

2. **Message Function**: after every event involving node i, a message is computed to update the node memory. When an interaction involves two nodes i and j, a message function is computed for each of them. The general formula for the message function is shown in Equation 3.21.

$$m_i = msg(s_i(t^-), s_j(t^-), t, e_{ij}(t)) \tag{3.21}$$

where:

m_i:	is the message computed for Node i at the current time step.
$s_i(t^-)$:	is the memory of node i at the previous time step $(t-1)$.
$s_j(t^-)$:	is the memory of node j at the previous time step $(t-1)$.
t:	is the current interaction
$e_{ij}(t)$:	is the edge feature at the current time step (t).
msg:	is a learnable function

3. **Memory Updater**: After every event in the graph, after every commit in our case, the memory of the affected nodes is updated. A learnable function class, such as LSTM or GRU, can be used as a memory updater. The memory is updated using Formula 3.22 where *mem* is the learnable function class.

$$s_i(t) = mem(m_i(t), s_i(t^-)) \tag{3.22}$$

4. **Embedding**: The embedding module is used to generate temporal embedding of Node i at time t. The embeddings are vectors representing the information of a node. It was observed that when a node has been inactive for a long time, its information will go out of date and the node becomes stale. This behavior is known as the *staleness problem* of nodes. Node embedding can overcome this problem by aggregating the information from the adjacent nodes and updating the memory of the node using this information [96]. Multiple methods can be used to embed a node,

such as embedding using the identity (the memory itself), time projection (uses an RNN or one of its variants to project how the embeddings would look like in the future)[97], and Temporal Graph Attention (which aggregates information from the neighbors of the node)[95]. In our work, we choose to keep the memory of the node intact and update the edge features, which contain the co-changeability of the two nodes, instead. This process is explained in the methodology chapter 4.

In [95], the authors propose the TGN architecture for edge prediction on the Wikipedia, Reddit, and Twitter datasets. In the first two datasets, the graphs are bipartite and the task is that of predicting if one user will show interest in a certain topic (either by following or clicking). The graph of the Twitter dataset is not bipartite and the task is to predict if a user will follow another one. An ablation study was performed and a variety of message aggregators, memory updates, embedding functions, and training methods were tested.

When trained on different data sets the trained model significantly outperformed other GNNs. Therefore, we took inspiration from this approach and present a modified TGN architecture to tackle the problem of change propagation in software systems.

Chapter 4

Methodology

Multiple repositories help developers share their code when implementing a program. An example of such repositories is GitHub. GitHub provides developers access control and several collaboration features, such as bug tracking, feature requests, and task management. These features make GitHub a valuable resource that allows the study of the evolution of software systems across different patches, where small changes are introduced to the software, and across different versions, where major changes are introduced.

When developers upload a patch to GitHub, the repository saves metadata, such as a list of the modified files and the upload time. The files that are included in the patch form a *commit*. Our proposed approach is a system that takes as input the GitHub repository and a file that was changed by the developer. It then outputs the files that are expected to change as well. In our approach, we consider the multiple commits of a software system on the GitHub Repository. We represent each commit as a change set that introduces a slice to the temporal graph. We build the slice of commit time c_i by duplicating that of slice c_{i-1}. Then, we loop through the files affected by the commit. If a given file was introduced to the system during the current commit, we add a representative node of this file to slice c_i. All the files that were modified in this commit will have their nodes connected to each other through edges. The edges established after this commit will persist throughout the entire software history graph, albeit, their weights will change with time to reflect the fluctuation of their co-changeability value. Figure 4.1 illustrated this step.

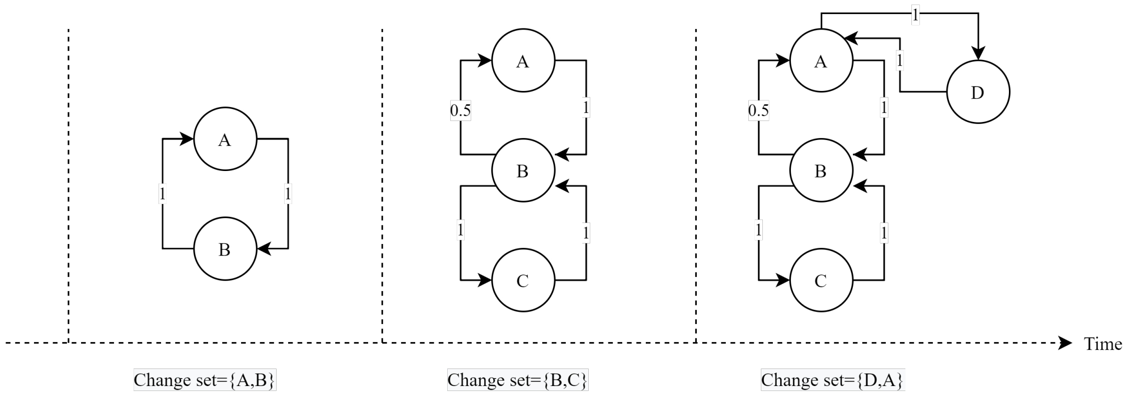


Figure 4.1: Example of a temporal graph built to model the change sets at every time step

We then apply a TGN to the temporal graph to learn which files change together. This will be used later on to predict future change sets. We define a *change set* as the set of files that change together at a particular commit. Figure 4.2 shows the workflow of this process.

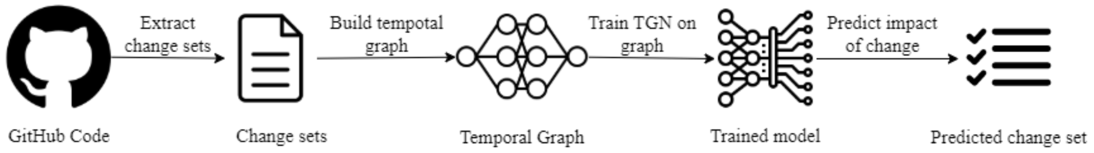


Figure 4.2: Workflow of the proposed approach

4.1 Data Extraction and System Modeling

We used the Python library pyDriller to extract change propagation datasets from GitHub [98]. This Python package can traverse the history of a system from its Github repository and mine the timestamp and the modified files of each commit. To employ this tool, we cloned the repository of each program and traversed its commits. We extracted all the recorded changes along with the modified files and data of every change. This allowed us to extract a log of each file system where each line holds the timestamp of the commit and the list of files modified. For example, consider a scenario where the developer changed File A and File B, pushed the commit to Github; then she changed File B and File C, and then pushed the commit to Github. And finally she added File D to the system and pushed the commit. Using pyDriller, we can read these commits in order and know the exact time step during which each file was modified. In this case, the output of pyDriller would be similar to the following:

```
7/12/2022-21:51, File A, File B
7/12/2022-23:12, File A
7/13/2022-11:11, File B, File C
7/13/2022-12:14, File A, File B, File C, File
D, File E
7/13/2022-13:10, File F, File A
```

We would know that File F was **added** to the system at the last commit since it was the first time it appeared in a commit.

We assume that every commit represents a change set. Therefore, we discard all the commits that modified only one file as they do not offer any information about the interactions among files in the system. Additionally, we remove the larger commits, which are usually indicative of a version update. These large commits have multiple files changed together, not because they influence each other but because of the significant modifications done to the system[99]. The result of this step is the change sets of a given software system in a temporal order. Therefore the only commits kept from the output of pyDiller would be:

```
7/12/2022-21:51, File A, File B
7/13/2022-11:11, File B, File C
7/13/2022-13:10, File F, File A
```

4.2 Graph Representation

We use a weighted directed temporal graph to model the change propagation problem. In our approach, the nodes of the graph are the files of the software in one commit. Two nodes are linked with an edge whenever the two files that they represent are changed at the same time (in the same commit). Once two files are connected with an edge, they will remain adjacent for the remaining slices of the temporal graph. The weight on an edge represents co-changeability between the two files. *Co-changeability* is a measure that reflects how likely it is for these two files to change in the same commit. As the system evolves and new changes are introduced, the co-changeability between the files is recomputed. A basic way to measure co-changeability is by counting the proportion of times these two files changed together in past commits and divide this number by the total times these two files were changed in general. If a given file A has been changed at times $S_A = t_i, t_j, ..., t_k$, and file B has been changed at times $S_B = t_m, t_n, ..., t_v$, a simple way to compute the co-changeability of A and B $Coch_{A,B}$ is using Formula 4.1.

$$Coch_{A,B} = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} \tag{4.1}$$

However, this formula does not take into consideration that some files may depend on one file only while others may be central ones that depend on and change with many other files. To visualize this better, we assume that we are learning the co-change pattern of a website. This website contains multiple HTML files, but one JavaScript file. The JavaScript file acts as a bridge between the HTML files and the API and modifies all the HTML files to add the information of the database to them. This means that if the developer modified an HTML file, the change will most likely only propagate to the JavaScript file. Whereas, if the developer modifies the JavaScript file, the change may propagate to any of the HTML files.

Our co-changeability value must reflect this. For example, assume that the system is made up of three HTML files: Home.html, ContactUs.html, and Account.html, and one JavaScript file called Backend.js. The commits of the file are in Table 4.1

Table 4.1: History of commits for website example

Commit	**Files affected**
c_0	Backend.js, home.html
c_1	Backend.js, ContactUs.html
c_2	Backend.js, Account.html

In this example, the JavaScript file is central in the system, and changes with all the other files, whereas each HTML file changes only with the JavaScript one. Using Formula 4.1, the last slice of the graph would look like the one in Figure 4.3 where the Javascript file is represented by a square and the HTML files are represented by circles. In that figure, all the edge weights are equal to 0.33.

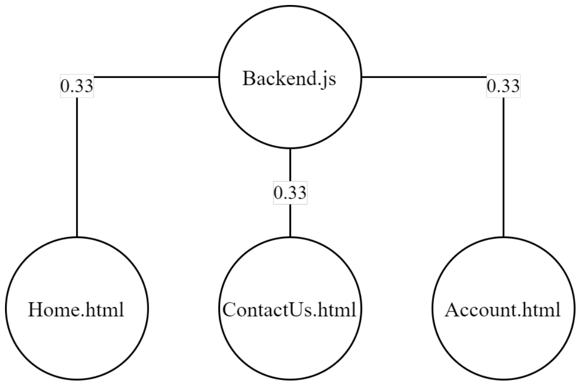


Figure 4.3: Undirected graph representing the website formed of three html files and one Javascript file.

However, a better way to read the history of the commits is that every time home.html (or any other HTML file) was changed, Backend.js was also changed. The graph in Figure 4.3 does not reflect the fact that every time an hmtl file changed, Backend.js also changed. After all, Equation 4.1 answers the question: What is the probability of File A and File B changing together? Instead, we would like to answer the following question: Knowing that File A changed, what is the probability that File B also changes? This question is better answered using Formula 4.2.

$$Coch*_{A,B} = \frac{|S_A \cap S_B|}{|S_A|} \tag{4.2}$$

$Coch*_{i,j}$ may be different than $coch*_{j,i}$. Therefore, a directed graph would be best suited to model this behavior. When using a directed graph to represent the *coch∗* values of the previous example, the weight of the edges on the last slice of the graph would be the ones in Figure 4.4.

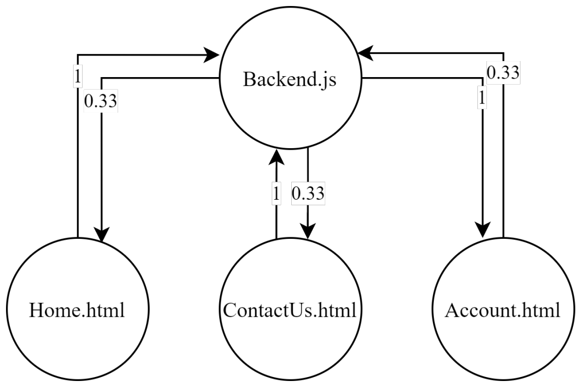


Figure 4.4: A directed graph to represent the website example

The graph of Figure 4.4 informs us that every time Home.html was changed, Backend.js also changed, but only in 33% of the cases when Backend.js was changed, did Home.html change as well. This better reflects the relationship between the files in the system.

These equations are biased against files added later in the development and maintenance process. As a matter of fact, older files would would score high co-changeability than newer ones because they tend to be involved in many more change sets. However, newly introduced files may have many strong dependencies. To correct this, when computing

$coch*_{A,B}$, we only consider the change events after introducing both files A and B to the system. For example, assume that we want to compute the co-changeability of a system file with the commits listed in Table 4.2.

Table 4.2: History of commits of example system

Commit	**Files affected**
c_0	A,B
c_1	A,C
c_2	B,D
c_3	A,B,C

At commit c_0, both files A and B were created. Therefore, their co-changeability at c_0 is 1, since, up to this point, every time A was changed, B was changed as well (in this case, the only change they went through is their creation). At commit c_1, a new file C is introduced to the system, and file A is modified simultaneously. After commit c_1, the $coch*_{A,B}$ will drop, since A was in a commit that B was not involved in (c_1). At time of commit c_1, File A was involved in commits $S_A = \{c_0, c_1\}$, and File B was involved in commits $S_B = \{c_1\}$. This makes $coch{*}_{A,B} = \frac{1}{2} = 0.5$. At the same time, file C was only involved in commit c_1, so the $coch{*}_{A,C} = 1$. As for $coch_{B,C}$, these two files were not involved in any of the same commits since the creation of the newer file (C), so $coch_{B,C} = 0$, and no edge is added between their representative nodes. We compute the co-changeability of all of these files in Table 4.3.

Table 4.3: Co-changeability of the files in the example system, the column value shows the value of the formula for these files

Commit no.	f_i	f_j	$coch{*}_{f_i,f_j}$	value
0	A	B	$\{c_0\}/\{c_0\}$	1
	B	A	$\{c_0\}/\{c_0\}$	1
1	A	B	$\{c_0\}/\{c_0, c_1\}$	0.5
	A	C	$\{c_1\}/\{c_1\}$	1
	B	A	$\{c_0\}/\{c_0\}$	1
	C	A	$\{c_1\}/\{c_1\}$	1
2	A	B	$\{c_0\}/\{c_0 c_1\}$	0.5
	A	C	$\{c_1\}/\{c_1\}$	1
	B	A	$\{c_0\}/\{c_0, c_2\}$	0.5
	B	D	$\{c_2\}/\{c_2\}$	1
	C	A	$\{c_1\}/\{c_1\}$	1
	D	B	$\{c_2\}/\{c_2\}$	1
3	A	B	$\{c_0 c_3\}/\{c_0 c_1 c_3\}$	0.66
	A	C	$\{c_1, c_3\}/\{c_1, c_3\}$	1
	B	A	$\{c_0, c_3\}/\{c_0, c_2, c_3\}$	0.66
	B	C	$\{c_3\}/\{c_2, c_3\}$	0.5
	B	D	$\{c_2\}/\{c_2, c_3\}$	0.5
	C	A	$\{c_1, c_3\}/\{c_1, c_3\}$	1
	C	B	$\{c_3\}/\{c_1, c_3\}$	0.5
	D	B	$\{c_2\}/\{c_2\}$	1

We iteratively build the temporal graph by adding nodes and updating the edges from commit time c_{i-1} to commit time c_i.

1. Create graph of commit time c_i, $G_i = (V_i, E_i)$, by copying all the elements from the previous commit time graph $G_i = (V_{i-1}, E_{i-1})$.

2. Add to V_i all the files that were not in V_{i-1} but that were introduced to the system at commit time c_i.

3. Add to E_i all the edges that connect two files modified in commit c_i and that did not already exist. If the edge did not exist previously, this means that these two files were never modified in the same commit previously.

4. Update the status of every node of V_i by indicating if it was changed in commit c_i

5. Recompute the weight of all the edges originating from a modified file at commit time c_i using Formula 4.2.

Using the commits of Table 4.2, the co-changeability values of Table 4.3, and the previously mentioned steps, we build the temporal graph of Figure 4.5.

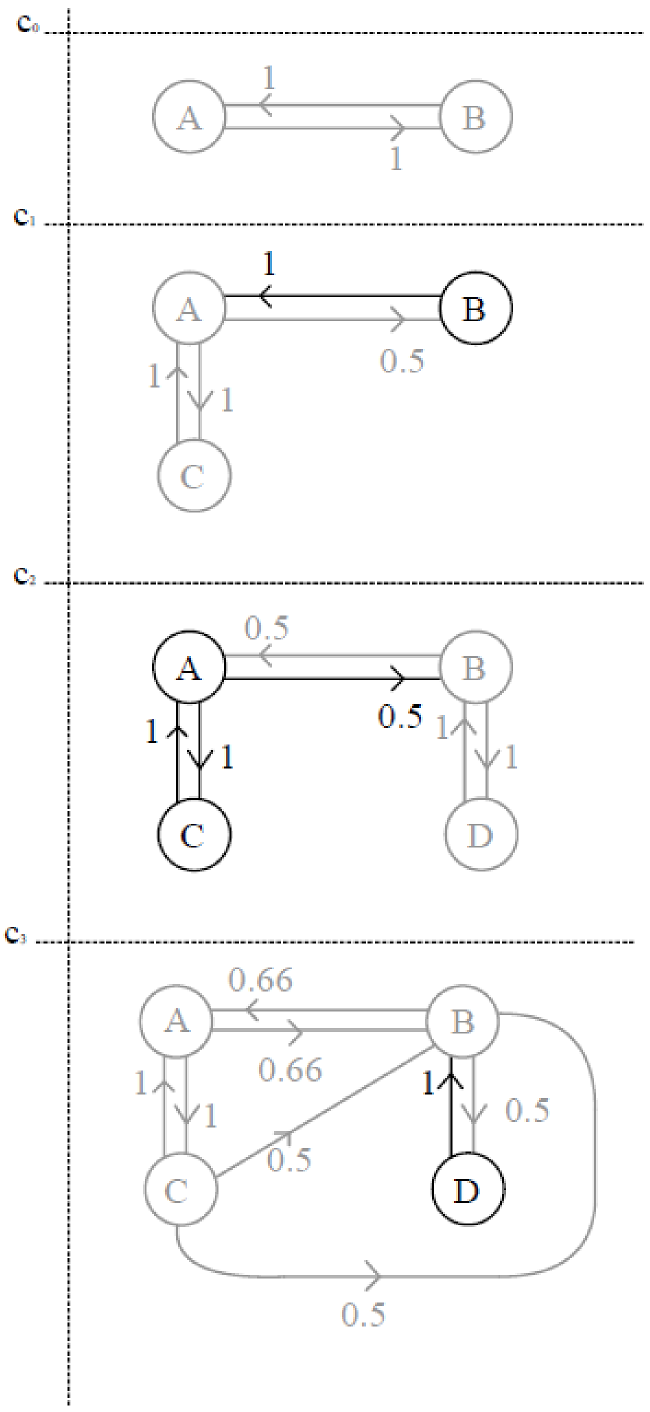


Figure 4.5: Temporal graph representation of the commits in Table 4.2

In Figure 4.5, at commit time c_0, we build the slice of graph G_0 by copying the graph of the previous commit, (which was empty) and adding all the vertices that were modified in c_0. This creates $V_0 = \{A, B\}$. We then connect all the nodes that were modified at c_0, which are A and B, and compute their co-changeability.

To build G_1, we create V_1 by adding V_0 to the node corresponding to C which was introduced during this commit. We add any missing edges to connect all the files modified at this commit time. These are edges (A, C) and (C, A). We then update the weights of all the edges originating from the modified files i.e. all the edges that have nodes A or C as the source. We do not link B and C since they did not change in the same commit so far.

To build G_2, we create V_2 which is the union of V_1 and $\{D\}$ and E_2 which is the union of E_1 and $\{(B, D), (D, B)\}$ since these files changed at this commit time. Finally, we update the weights of all edges originating from B or D.

Lastly, to build G_3, we create $V_3 = V_2$ since no files were added at this time step. We create E_3 which is the union of E_2 and the set $\{(B, C), (C, B)\}$ since this is the first time files B and C are changed in the same commit. And then we update the weights of all the edges originating from A, B, or C.

4.3 Temporal Graph Network

The temporal graph that we built in the previous step allows us to quickly review the history of a node and the evolution of its co-changeability with its adjacent nodes. Hence, when the developer flags a file as changed, we can review the temporal graph and study how this file historically evolved with its neighbors to determine which files would also be impacted by this change.

To study the impact of changing File f_i at commit time c_i, we search for the neighbor, f_j, with the highest co-changeability at commit time c_{i-1}. We review the history of f_i and f_j starting from the first time step they became adjacent, i.e. the first commit that modified both of them, and record $coch*_{f_i,f_j}$ and the status of f_j (whether it was changed or not) at every time step where f_i was changed. One thing to note, however, is that if one file is modified at commit time c_i, the co-changeability of this file with its neighbors at this time reflects whether the second file also changed or not. In other words, in the previous example (Table 4.2), we focus only on the state of A and B (changed or not) as well as the co-changeability value $Coch*_{B,A}$ at that time as shown in Figure 4.6.

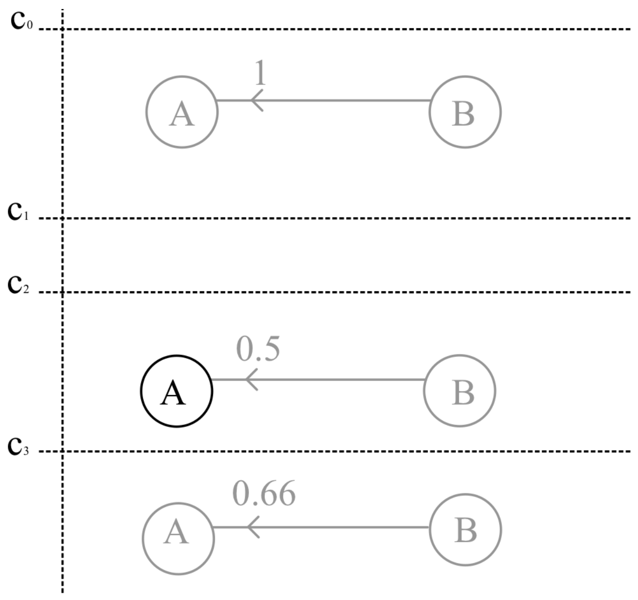


Figure 4.6: The state of A and B at every time step where B was changed.

At commit time c_1, File A was changed but File B was not. This was reflected in their co-changeability since it dropped at that time. When passing this information to the LSTM model, we cannot pass the co-changeability of A and B at time c_1 for it to predict whether B will change at that time, because the answer is already present in the drop of co-changeability of the two files. This creates a leakage in our train set. Instead, when predicting if B will change at c_1, we should pass the co-changeability of A and B at c_0 since it holds no information on the status of the files at c_1. For example, if after commit c_3 in the previous example, the developer flags that file B changed, the model revisits the neighbor with the highest co-changeability at the most recent graph slice, here G_3. This neighbor is A since $coch*_{B,A} = 0.66$ in G_4. And then it builds a data frame that records at each row $coch*_{B,A}$ at time $ci - 1$, and the status of A at commit time c_i for every commit c_i where B was changed.

If the only possible data frame to build in this case is the one in Table 4.4. In the first row, the table holds $coch*_{B,A}$ at c_1 and the status of A at c_2 since B was modified at commit time c_2. In the second row, it holds the $coch*_{B,A}$ at c_2 and the status of A at c_3 since B was modified at commit time c_2.

Table 4.4: Data frame created from the history of A and B

previous $coch*_{B,A}$	state of A
1	not changed
0.5	changed

After building the data frame, the LSTM is trained on the co-changeability values to predict whether the neighbor is affected by the change or not. Then, the model performs the final prediction on the most recent co-changeability value to predict if the neighboring file is affected by the change that was recently flagged by the developer. In our example, this means that the LSTM predicts if A will change with an input co-changeability of 0.66 since it is the value of $coch*_{A,B}$ at c_3.

If the LSTM predicts the file to be impacted by the change, we add the latter to a queue and mark the file as modified. Then, we move to the next highest co-changeability neighbor of the node. In the example, that would be either node C or node D since they both have a co-changeability value of 0.5 with B at commit time c_4.

After visiting the neighbors of file F_i, we dequeue a file and then visit the neighbors of the dequeued file to check for the possibility of the change propagating to them. To limit the computational time of our method, we do not visit neighbors with co-changeability lower than a pre-set cutoff value μ. Additionally, we do not allow our change set to grow larger than a threshold ρ. The values ofμ and ρ are parameters of our network that we tune during our parameter tuning experiment detailed in Section 5.4.

After completing this iteration and allowing the model to perform its predictions, we assess the performance of the model by comparing the predicted change set to the actual one. A new slice is then added to the temporal graph containing nodes following the steps previously stated. This is done to ensure the data in the temporal graph does not go out of date. Algorithm 4 shows the pseudo-code of this TGN.

Going back to the components of a TGN defined in [95], we list below our implementation of the TGN components.

- **The memory:** In our implementation, the memory of a node a time t is a vector of length t that indicates at every time step whether the node was changed or not. For example, at time of commit c_3 in the example of Table 4.2, File A was changed

Algorithm 4 Proposed TGN pseudo-code

```
$Q \leftarrow Queue()$
Q.add(source_of_change_file)
$change\_set \leftarrow \phi$
while Q not empty and size of $change\_set < \rho$ do
    $current\_node$=$Q$.dequeue()
    for neighbor in E[current_node] do
        if $coch*_{current\_node,neighbor} > \mu$ and neighbor not in change_set then
            Retrieve history of current node and neighbor
            Train LSTM on the set
            if LSTM predicts neighbor to change then
                Add neighbor to change_set
                Add neighbor to Q
            end if
        end if
    end for
end while
return change_set
```

at commit times c_0, c_1, and c_3 but not at c_2. The vector memory of the node that represents A is hence $s_a = [1, 1, 0, 1]$.

- **The message function:** The message passed from B to A is their co-changeability values across the time steps at which B changed. We use this when we know that B has changed, and we predict whether B is changed as well.

- **The memory updater:** The memory updater is the LSTM architecture that learns from the messages passed to a node and states whether this node will be affected by the change or not. If the LSTM is attempting to learn whether File A will be impacted by a change made to File B, we revisit all the time steps of the graph starting from the time step where A and B first became adjacent. We then record the weight of edge (B, A) and whether A was changed at every time step where B was changed. The LSTM then predicts whether File A will be impacted by the change at this time step and the state of the file is updated accordingly.

- **The embedding:** To combat the staleness problem, we update Edge (A, B) whenever A is changed. This ensures that the relationship between the nodes that represents the files is kept up to date as long as any one of these files is changed.

Chapter 5

Experiments and Results

In this chapter, we describe the experiments that we conducted to validate our approach. We start by describing the data set that we formed out of 15 different projects. We then introduce the metrics that we use in the assessment of our model performance. Then, we describe the experimental setup and the obtained results. We discuss these results focusing on each software system separately, first, then we focus on the results across all 15 projects. All our claims derived from the results are supported by statistical tests.

5.1 Datasets

We validate our model on 15 projects (Table 5.1). These differ in their size (number of files), number of commits, number of extracted change sets, and the programming languages they were developed with. The programming languages used to implement the tested programs are Batchfile, Blade, C++, C#, CMake, CoffeeScript, CSS, Dart, Dockerfile, GAP, Go, Handlebars, HiveQL, HTML, Java, JavaScript, Jupyter Notebook, Lex, Objective-C, Perl, PHP, Python, R, Ruby, Scala, Shell, Starlark, Swift, Thrift, TypeScript, and XSLT.

Table 5.2 summarizes the information pertaining to the number of commits of every system. Looking at the numbers, one can see that, for most of the systems, there is a significant difference between the median and the mean indicating the existence of outliers overshooting the value of the mean. Outliers are recorded commits that modified a large proportion of the system. Normally, these map back to complete version updates and hence affect a vast majority of the files regardless of whether the dependencies exist between these

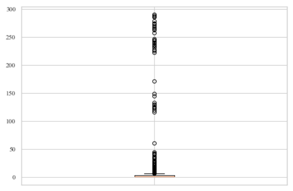
Table 5.1: Software systems used in this study

Project name	Application	Languages used
Alamofire	HTTP networking library	Swift(100%)
Ant	Java tool	Java(77.7%) HTML(17.8%) XSLT(3.6%) GAP(0.3%) Shell(0.3%) Batchfile(0.2%) Other(0.1%)
Cassandra Cassandra	Database management system	Java(97.0%) Python(1.6%) HTML(0.8%) Shell(0.3%) GAP(0.3%) Lex(0.0%)
Cassandra Website	Website Tool	CSS(43.4%) JavaScript(18.6%) Handlebars(16.8%) Shell(16.0%) Dockerfile(2.9%) Python(2.3%)
Flutter	Application Framework	Dart(99.1%) Objective-C(0.2%) Java(0.2%) C++(0.1%) Shell(0.1%) CMake(0.1%) Other(0.2%)
Gephi	visualization platform	Java(99.4%) Other(0.6%)
Hbase	Database management system	Java(95.8%) Ruby(1.9%) Perl(0.9%) Shell(0.8%) Python(0.3%) Thrift(0.1%) Other(0.2%)
Laravel	web framework	PHP(81.2%) Blade(17.5%) Other(1.3%)
Lucene	Text search engine library	Java(97.7%) HTML(1.1%) Python(0.7%) Lex(0.3%) Perl(0.1%) Shell(0.1%)
Monitor Control	Apple Application	Swift(98.6%) Objective-C(1.4%)
pyDriller	Python framework	Python(100%)
React	App framework	JavaScript(95.7%) HTML(1.9%) CSS(1.0%) C++(0.7%) TypeScript(0.3%) CoffeeScript(0.3%) Other(0.1%)
Rocketmq clients	Clients for Apache RocketMQ	C++(42.6%) Java(41.8%) C#(8.0%) Go(4.0%) Starlark(2.8%) C(0.6%) Other(0.2%)
Spark	Analytics tool	Scala(67.5%) Python(11.9%) Java(7.3%) Jupyter Notebook(6.7%) HiveQL(2.9%) R(2.0%) Other(1.7%)
WWW site	website	HTML(74.8%) CSS(16.8%) JavaScript(5.3%) XSLT(2.0%) Other(1.1%)

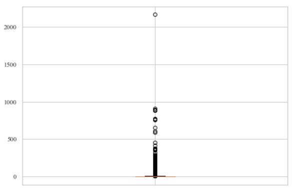
Table 5.2: Distribution of the size of commits before processing

Software system	No. of commits	Min.	Median	Mean	Q90	Max.
Alamofire	1883	1	2	7.816251	9	290
Ant	20604	1	2	4.849495	7	2169
Cassandra	25393	1	3	4.866577	9	537
Cassandra website	997	1	2	33.75025	5	1983
Flutter	42421	1	2	5.410575	10	2571
Gephi	8818	1	3	9.285552	13	4716
Hbase	28936	1	2	5.813416	11	2052
Laravel	5994	1	1	2.255255	4	102
Lucene	52873	1	3	6.338207	11	5570
Monitor control	358	1	3	6.215084	17	72
pyDriller	858	1	2	2.719114	5	31
React	17978	1	2	4.676994	9	514
Rocketmq clients	1033	1	3	8.673766	15	294
Spark	50520	1	3	5.127534	9	11283
WWW site	976	1	1	7.547131	4	1681

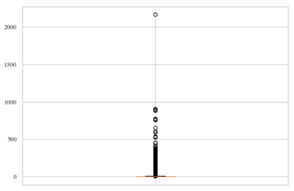
files or not. The significant number of outliers can be seen in in Figure 5.1 which shows a boxplot of the distribution of the change set sizes for every software system.



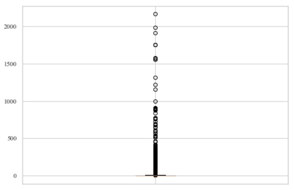
(a) Change sets size distribution of Alamofire



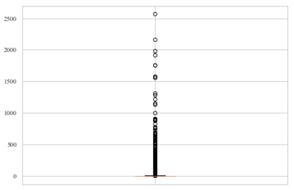
(b) Change sets size distribution of Ant



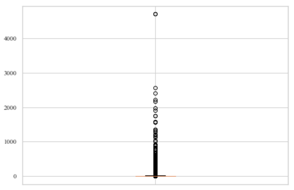
(c) Change sets size distribution of Cassandra



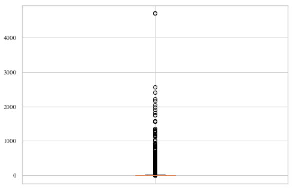
(d) Change sets size distribution of Cassandra website



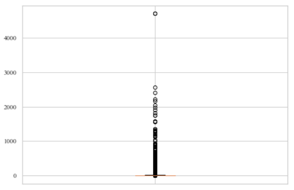
(e) Change sets size distribution of Flutter



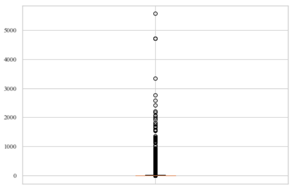
(f) Change sets size distribution of Gephi



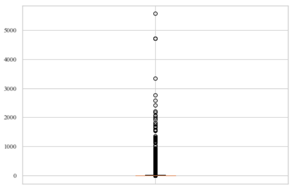
(g) Change sets size distribution of Hbase



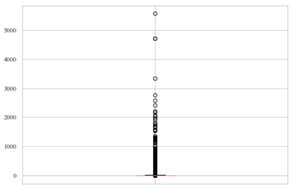
(h) Change sets size distribution of Laravel



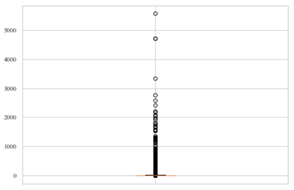
(i) Change sets size distribution of Lucene



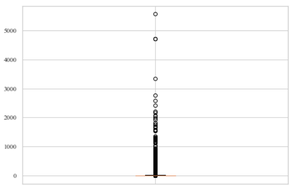
(j) Change sets size distribution of Monitor control



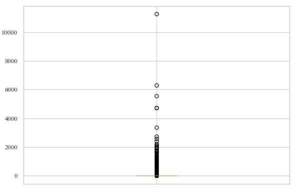
(k) Change sets size distribution of pyDriller



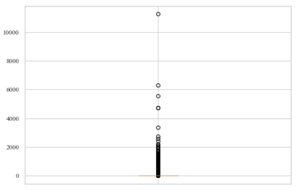
(l) Change sets size distribution of React



(m) Change sets size distribution of Rocketmq clients



(n) Change sets size distribution of Spark



(o) Change sets size distribution of WWW site

Figure 5.1: Boxplot showing the size distribution of the change sets

Commits that affect one file and large commits, both, introduce noise to our data. Therefore, similarly to most previous work on this problem[99, 11, 21, 22], we discard any commit that affects a single file only, and any commit that modified more files than a cut-off value. We set the cut-off value to be equal to the 90^{th} quartile in the distribution of the number of files modified by the commits for each system and use the commits that meet these size requirements as change sets of our system. Table 5.3 shows the distribution of the number of files modified by the commits after performing this data cleaning step. There is still a difference between the mean value and median value, however, this difference is now much smaller indicating fewer outliers in the data set. Due to the large number of commits of some of the systems, we only consider the first 1,000 commits in these experiments.

Table 5.3: Distribution of the size of commits after processing. The No. of files is the number of files at the last commit considered, and prop. affected files is the average proportion of files affected by the changes out of all the files of the system at the time of commit.

Software system	No. of files	No. of commits	Min.	Median	Mean	Max.	prop. affected files
Alamofire	168	486	2	3	5.049383	25	0.053
Ant	1065	6190	2	3	3.889822	13	0.017
Cassandra	623	9408	2	3	4.078444	13	0.019
Cassandra website	132	400	2	3	3.3225	43	0.163
Flutter	817	14729	2	3	4.289022	16	0.017
Gephi	1950	2828	2	4	6.035007	23	0.014
Hbase	550	10151	2	3	4.64969	17	0.02
Laravel	616	1017	2	2	3.105211	10	0.018
Lucene	1175	18057	2	3	4.840284	18	0.017
Monitor control	190	138	2	4	5.514493	24	0.082
pyDriller	110	256	2	3	3.09375	8	0.074
React	724	5957	2	3	4.322142	13	0.014
Rocketmq clients	574	379	2	4	6.174142	24	0.034
Spark	617	18852	2	3	4.115638	12	0.02
WWW site	310	122	2	3	4.204918	20	0.04

5.2 Performance metrics

Since we tackle the problem of predicting the impact of a change as a binary classification problem whereby a file is either impacted by a change or not, we use binary classification metrics to assess our developed method performance. However, the proportion of files impacted by a change is minuscule compared to the number of files unaffected. Additionally, we are more interested in correctly predicting the impacted files than the ones that have not been affected by the change since the former ones should be brought to the attention of the developer when they inject a change in a file. This also can be used by the project manager in assessing the effort and time incurred by a proposed change. Therefore, we use sensitivity, specificity, and g-mean to assess our method efficacy and we also report precision, F-measure, Matthew's correlation coefficient(MCC), and accuracy to obtain a general report on the performance of the model. These metrics can be extracted from the confusion matrix (Figure 5.2) which summarizes the performance of a model by showing the number of data points correctly or mistakenly classified to both classes in the binary classification task. In our problem, a positive instance is a file that is impacted by a change while a negative one is a file that is not impacted by a change.

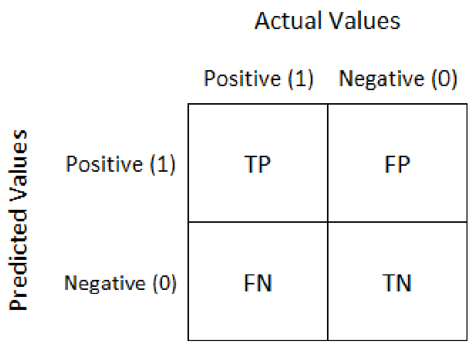


Figure 5.2: Confusion matrix. TP: True positives, TN: True negatives, FP: False positives, FN: False negatives.

In the confusion matrix, the following cells can be found:

- True positives(TP): number of files correctly classified as being impacted by the change.
- False positives(FP): number of files incorrectly classified as being impacted by the change while in reality, they are not.
- False negatives (FN): the number of files incorrectly classified as not being impacted by the change while in reality, they are.

- True negatives (TN): number of files correctly classified as not being impacted by the change.

5.2.1 Sensitivity or recall

Sensitivity or recall is an important metric to consider when the positive instances should be predicted as accurately as possible. This metric evaluates the ability of the model to correctly identify the true positives[100] and is computed using Equation 5.1.

$$Sensitivity = \frac{TP}{TP + FN} \tag{5.1}$$

In our work, sensitivity or recall represents the probability of correctly classifying an impacted file as affected by the change. This is very important in our study since the positive cases help the project managers estimate the time, effort, and budget a particular change would incur.

5.2.2 Specificity

This measures the ability of the model to correctly identify true negatives in a data set [100] and is computed using Equation 5.2.

$$Specificity = \frac{TN}{TN + FP} \tag{5.2}$$

In our work, specificity represents the ability of the model to correctly recognize the files that are not affected by a change.

5.2.3 Positive Predicted Value

Precision or Positive Predicted Value rate (PPV) measures the proportion of correctly classified files among those that were classified as positive [101]. Precision is computed using Formula 5.3

$$Precision = \frac{TP}{TP + FP} \tag{5.3}$$

5.2.4 F-measure

The F-measure is the harmonic mean of recall (or sensitivity) and precision [102] and is computed using Formula 5.4.

$$F - measure = 2 \times \frac{Recall * Precision}{Recall + Precision} \tag{5.4}$$

5.2.5 G-mean

The geometric mean (G-mean) is used to measure the accuracy of prediction for both classes while not favoring the majority class (Equation 5.5). This is an important metric in our study since our data set is significantly imbalanced [103].

$$G - mean = \sqrt{Sensitivity * Specificity} \tag{5.5}$$

5.2.6 Accuracy

Accuracy shows the proportion of values that were correctly classified. This metric, however, is extremely biased toward the majority class, therefore it is an inefficient metric for the problem of change propagation since the vast majority of the files will not be affected by a change [102]. Accuracy is computed using Formula 5.6.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.6}$$

5.2.7 Matthew's Correlation Coefficient

Matthew's Correlation Coefficient (MCC) represents the correlation between the predicted values of a model and the actual ones[104] and is a number between -1 and 1. Formula 5.7 is used to compute the MCC value.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5.7}$$

5.2.8 Area Under Curve

The Area Under Curve (AUC) reflects the probability of the model to correctly classify a positive instance as positive, and a negative instance as negative [105, 106]. AUC is computed using Formula 5.8.

$$AUC = \frac{1}{2}\left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP}\right) \tag{5.8}$$

5.3 Experimental Setup

We implemented the proposed approach in Python using the NumPy and TensorFlow libraries. We ran our experiments on an Intel(R) Core(TM) i7-9750H CPU with 32GB RAM and an Intel(R) UHD Graphics 630 graphics card. We used random search to perform parameter tuning as is detailed in Section 5.4.

We use the cross-validation on a rolling-origin-recalibration method to split the data into train-test sets, and assess the performance of the models [107]. This method is used on temporal data where the model is trained on the data points starting from time t_0 to t_i, and then tested on data point at time t_{i+1}. In the next iteration, the model is trained on the data points from time t_0 till t_{i+1} and then tested on the data point of t_{i+2}. This continues until the model is tested on the last data point. Figure 5.3 visualizes this technique.

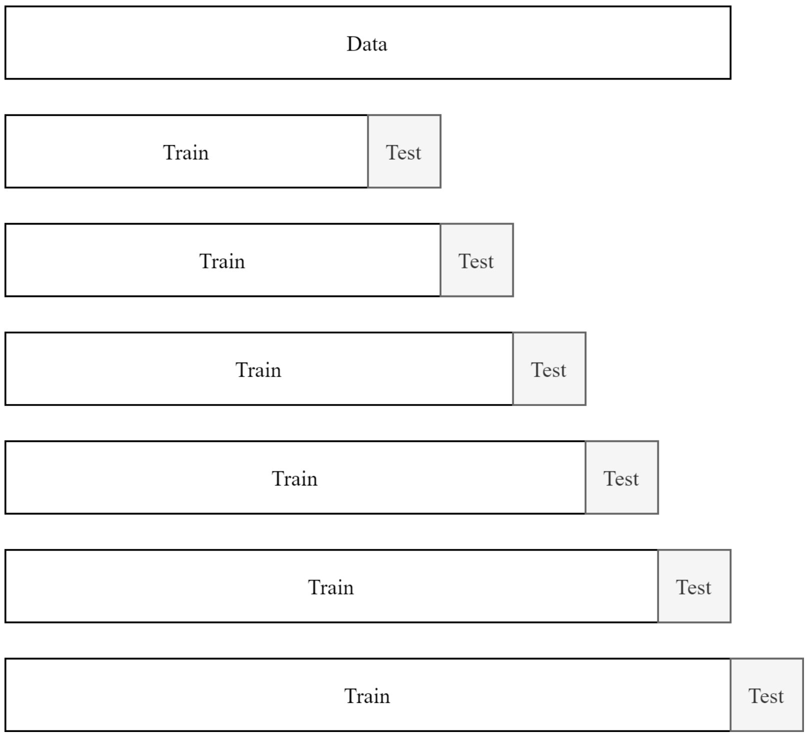


Figure 5.3: Cross-validation using the rolling-origin-recalibration method

To account for the element of randomness in the model, we repeat each experiment on every data set 5 times and report the average and standard deviation of the performance metrics.

5.4 Parameter Tuning

The performance of the proposed model depends on many parameters. It is known that the parameters have a significant effect on the model. Therefore, fine-tuning them is essential. The two widely used techniques for parameter tuning are Random Search and Grid Search. In our proposed work, we fine-tune the parameters of the model using Random Search. This technique is a stochastic one that generates random parameter configurations and tests the performance of the model when set to these different combinations. When compared to Grid Search, an exhaustive parameter tuning method that tests every possible parameter combination, Random Search was found to reach acceptable results in less time. The pseudo-code of Random Search can be found in Algorithm 5.

Algorithm 5 Pseudo-code of Random Search

```
while Stopping criteria not met do
    Generate a set of parameters
    Train and test the model
    if this configuration reaches better results then
        Save these parameters
    end if
end while
```

To limit the computational time of our method, we do not visit neighbors with co-changeability lower than a pre-set cutoff value μ. Additionally, we do not allow our change set to grow larger than a threshold ρ. We use Random Search to set the following parameters:

- μ: the co-changeability cut-off value.
- ρ: the predicted change set size threshold.
- L_1size: number of units in the first layer of the LSTM.
- L_2size: number of units in the second layer of the LSTM.
- epochs: number of complete passes over the train set the LSTM goes through while training.

- optimizer: the optimizer used to set the learning rate and weights of the neural network.

Table 5.4 records the set of best parameters found for each software system.

Table 5.4: Best found parameters for every software system

Software system	μ	ρ	$L_1 size$	$L_2 size$	Epochs	Optimizer
Alamofire	0.1	80	8	64	6	Adam
Ant	0.005	95	16	16	7	Nadam
Cassandra	0.005	70	16	16	9	Nadam
Cassandra Website	0.25	60	32	2	6	SGD
Flutter	0.1	85	32	4	4	SGD
Gephi	0.003	80	64	64	8	SGD
Hbase	0.1	60	64	64	6	Adam
Laravel	0.005	95	64	2	14	RMSProp
Lucene	0.1	95	64	4	11	Nadam
Monitor Control	0.005	95	4	4	5	Adam
pyDriller	0.1	75	4	4	6	Nadam
React	0.005	95	8	8	15	SGD
Rocketmq Clients	0.2	60	32	32	15	Adagrad
Spark	0.005	60	8	4	7	RMSProp
WWW Site	0.3	50	8	32	9	Adam

5.5 Results

We start by discussing the results obtained on each software system separately then we summarize the average performance of our model.

In our discussion, we compare between the performance of the proposed model and that presented in [11]. We chose this work as it is a recent publication that used the concept of aging in the history of software systems. In this work, the authors gives less importance to older commits when calculating the co-changeability between two files. This makes the work similar to ours in the sense that some temporal dimension is introduced to the data.

Table 5.5: Mean ± standard deviation of the proposed approach

	Alamofire		Gephi		pyDriller	
Sensitivity	0.561429	±0.007284	0.409034	±0.001958	0.542188	±0.023199
Specificity	0.995314	±0.000618	0.997089	±5.25E-05	0.986687	±0.001517
PPV	0.881429	±0.013418	0.561205	±0.00404	0.777604	±0.020027
F-measure	0.66581	±0.001633	0.357619	±0.003944	0.602664	±0.01272
G-Mean	0.741376	±0.004047	0.607028	±0.000824	0.719992	±0.013731
Accuracy	0.980349	±0.000395	0.994481	±4.74E-05	0.964622	±0.00085
MCC	0.684443	±0.002527	0.40942	±0.003676	0.614343	±0.012642
AUC	0.778371	±0.00334	0.703062	±0.001005	0.764437	±0.011079
	Ant		Hbase		React	
Sensitivity	0.48659	±0.003258	0.532073	±0.007499	0.60567	±0.006021
Specificity	0.998043	±0.000111	0.998808	±9.71E-05	0.998702	±3.33E-05
PPV	0.753196	±0.010873	0.878154	±0.008496	0.791111	±0.007057
F-measure	0.51892	±0.004983	0.612811	±0.00582	0.647404	±0.004903
G-Mean	0.671931	±0.002126	0.708647	±0.004677	0.761981	±0.003685
Accuracy	0.994825	±0.00012	0.994341	±0.000115	0.996629	±4.00E-05
MCC	0.562388	±0.00536	0.653203	±0.005447	0.670279	±0.005395
AUC	0.742317	±0.001649	0.765441	±0.003747	0.802186	±0.003008
	Cassandra		Laravel		Rocketmq clients	
Sensitivity	0.457016	±0.001899	0.524476	±0.003127	0.421413	±0.008877
Specificity	0.998158	±7.50E-05	0.998871	±5.94E-05	0.993952	±0.000117
PPV	0.772812	±0.005255	0.888345	±0.005961	0.662078	±0.009058
F-measure	0.519726	±0.00127	0.645082	±0.003265	0.437137	±0.007128
G-Mean	0.656791	±0.00177	0.721041	±0.001834	0.619653	±0.00567
Accuracy	0.993808	±4.83E-05	0.994368	±6.93E-05	0.982993	±0.000177
MCC	0.5607	±0.001392	0.67252	±0.00352	0.475385	±0.006963
AUC	0.727587	±0.000922	0.761673	±0.001569	0.707683	±0.004462
	Cassandra website		Lucene		Spark	
Sensitivity	0.781308	±0.019067	0.496301	±0.002596	0.485068	±0.003813
Specificity	0.950317	±0.002141	0.998508	±8.18E-05	0.997498	±6.32E-05
PPV	0.707788	±0.010278	0.82343	±0.006339	0.67604	±0.006793
F-measure	0.709586	±0.011756	0.554753	±0.00388	0.501159	±0.004931
G-Mean	0.849797	±0.010475	0.680138	±0.001746	0.673655	±0.002741
Accuracy	0.932384	±0.002691	0.995539	±7.79E-05	0.993414	±9.45E-05
MCC	0.693872	±0.013248	0.60041	±0.003769	0.534668	±0.005149
AUC	0.865813	±0.00944	0.747404	±0.001312	0.741283	±0.001938
	Flutter		Monitor control		WWW site	
Sensitivity	0.430457	±0.006165	0.463359	±0.005495	0.566667	±0.038188
Specificity	0.997046	±7.88E-05	0.962864	±0.003056	0.987971	±0.002507
PPV	0.639448	±0.007597	0.554475	±0.034973	0.820833	±0.033203
F-measure	0.442763	±0.00403	0.407382	±0.010212	0.633333	±0.018305
G-Mean	0.632492	±0.004765	0.646682	±0.00374	0.735005	±0.022725
Accuracy	0.993088	±6.18E-05	0.941351	±0.002901	0.966363	±0.001599
MCC	0.481157	±0.003179	0.42862	±0.015276	0.647561	±0.015617
AUC	0.713752	±0.003061	0.713111	±0.0037	0.777319	±0.018168

Also, the results in [11] showed higher AUC than the baseline models that did not age the commits.

We repeat each experiment on every software system 5 times, and each time we start the propagation of change for every commit from a different file. Furthermore, to ensure that the comparison is fair, for every commit in the experiment, we propagate the change from the same file when predicting using our model and that of [11].

To test the statistical significance of the obtained results, we perform a one-tailed Mann-Whitney test[108] with a significance level $\alpha = 0.05$ to verify if the performance of our model on each metric is significantly better than that of [11]. We formulate the null hypothesis H_0 as follows: *Given metric M, the performance of our proposed approach on data set D is not better than that of [11].* Rejecting H_0 ensures that the performance of our model is significantly better on metric M.

5.5.1 Alamofire

Alamofire is a homogeneous software system developed using Swift. The model of [11] outperforms the proposed model in terms of specificity and PPV. However, our proposed approach outperforms its counterpart in terms of sensitivity, F-measure, G-mean, accuracy, MCC and AUC (Table 5.6). This indicates that the proposed approach is better at identifying the files that will be impacted by the change than its counterpart on Alamofire.

Table 5.6: Comparative results on Alamofire

Our results			Results of [11]		
Sensitivity	**0.561429**	±0.007284	Sensitivity	0.375645	±0.004795
Specificity	0.995314	±0.000618	Specificity	**0.997807**	±0.000268
PPV	0.881429	±0.013418	PPV	**0.913881**	±0.010554
F-measure	**0.66581**	±0.001633	F-measure	0.48827	±0.003738
G-Mean	**0.741376**	±0.004047	G-Mean	0.588192	±0.004501
Accuracy	**0.980349**	±0.000395	Accuracy	0.974081	±0.00021
MCC	**0.684443**	±0.002527	MCC	0.548518	±0.002462
AUC	**0.778371**	±0.00334	AUC	0.686726	±0.002286

Table 5.7 shows the results of the Mann-Whitney test for the Alamofire software system.

The results reinforce that the proposed model can reach statistically significantly better results in terms of sensitivity, F-measure, G-mean, accuracy, MCC, and AUC.

Table 5.7: p-value and decision of Mann-Whitney on Alamofire

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.9999	Fail to reject H_0
PPV	0.996	Fail to reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.2 Ant

Ant is a heterogeneous tool developed mainly in Java. While the model of [11] outperforms the proposed model in terms of PPV, the proposed model still outperforms it in terms of sensitivity, specificity, F-measure, G-mean, accuracy, MCC, and AUC (Table 5.8). This indicates that the proposed model has a better tendency to identify which files will be affected by the change than its counterpart.

Table 5.8: Comparative results on Ant

Our results			Results of [11]		
Sensitivity	**0.48659**	±0.003258	Sensitivity	0.471105	±0.002197
Specificity	**0.998043**	±0.000111	Specificity	0.997696	±0.000122
PPV	0.753196	±0.010873	PPV	**0.765721**	±0.007375
F-measure	**0.51892**	±0.004983	F-measure	0.501946	0.±002341
G-Mean	**0.671931**	±0.002126	G-Mean	0.656615	±0.00172
Accuracy	**0.994825**	±0.00012	Accuracy	0.99447	±0.000112
MCC	**0.562388**	±0.00536	MCC	0.550932	±0.002456
AUC	**0.742317**	0.001649	AUC	0.734401	±0.001058

Table 5.9 shows the results of the Mann-Whitney test for the Ant software system. The results reinforce that the proposed model can reach statistically significantly better results in terms of sensitivity, specificity, F-measure, G-mean, accuracy, MCC, and AUC.

Table 5.9: p-value and decision of Mann-Whitney on Ant

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.003	Reject H_0
PPV	0.952	Fail to reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.007	Reject H_0
AUC	0.003	Reject H_0

5.5.3 Cassandra

Cassandra is a heterogeneous tool with the vast majority of its code developed in Java. Although the model of [11] outperforms the proposed model in terms of specificity and PPV, the latter one outperforms it in terms of sensitivity, F-measure, G-mean, accuracy, MCC, and AUC (Table 5.10). This indicates that the proposed model has a better tendency to identify the files that will be impacted by change.

Table 5.10: Comparative results on Cassandra

Our results			Results of [11]		
Sensitivity	**0.457016**	±0.0018	Sensitivity	0.337976	±0.001707
Specificity	0.998158	±7.50E-05	Specificity	**0.999144**	±8.04E-05
PPV	0.772812	±0.0052	PPV	**0.906948**	±0.00427
F-measure	**0.519726**	±0.0012	F-measure	0.453071	±0.001862
G-Mean	**0.656791**	±0.0017	G-Mean	0.560852	±0.001473
Accuracy	**0.993808**	±4.83E-05	Accuracy	0.992457	±6.87E-05
MCC	**0.5607**	±0.0013	MCC	0.523763	±0.001574
AUC	**0.727587**	±0.0009	AUC	0.66856	±0.000828

Table 5.11 shows the results of the Mann-Whitney test for the Cassandra software system. The results reinforce that the proposed model can reach statistically significantly better results in terms of sensitivity, F-measure, G-mean, accuracy, MCC, and AUC.

Table 5.11: p-value and decision of Mann-Whitney on Cassandra

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.999	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.4 Cassandra website

Cassandra website is a heterogeneous system that does not have a dominant development language. Although the language that is mostly used in its development is CSS, the percentage of files written using this language is 43.4%. The model of [11] outperforms the proposed model in terms of specificity, PPV, and accuracy. However, our proposed approach outperforms that of [11] in terms of sensitivity, F-measure, G-mean, MCC, and AUC (Table 5.12). This indicates that the proposed model is better at identifying files impacted by the change.

Table 5.12: Comparative results on Cassandra website

Our results			Results of [11]		
Sensitivity	**0.781308**	±0.019067	Sensitivity	0.615576	±0.005062
Specificity	0.950317	±0.002141	Specificity	**0.982623**	±0.001787
PPV	0.707788	±0.010278	PPV	**0.85919**	±0.012867
F-measure	**0.709586**	±0.011756	F-measure	0.68704	±0.008065
G-Mean	**0.849797**	±0.010475	G-Mean	0.765712	±0.003929
Accuracy	0.932384	±0.002691	Accuracy	**0.942318**	±0.00201
MCC	**0.693872**	±0.013248	MCC	0.685098	±0.008848
AUC	**0.865813**	±0.00944	AUC	0.7991	±0.002991

Table 5.13 shows the results of the Mann-Whitney test for the Cassandra website software system. The results reinforce that the proposed model reaches statistically significantly better results in terms of sensitivity, F-measure, G-mean, and AUC but that the difference in MCC is not statistically significant.

Table 5.13: p-value and decision of Mann-Whitney on Cassandra website

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.999	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.015	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.999	Fail to reject H_0
MCC	0.273	Fail to reject H_0
AUC	0.003	Reject H_0

5.5.5 Flutter

Flutter is a heterogeneous tool mostly developed using Draft. The model of [11] outperforms the proposed model in terms of specificity, PPV, F-measure, accuracy, and MCC. However, our proposed approach outperforms that of [11] in terms of sensitivity, G-mean, and AUC (Table 5.14). The high sensitivity value indicates that the proposed model had a better tendency to detect the files affected by the change. And although the model fails to outperform its opponent in the majority of the performance metrics, it was able to keep high G-mean and AUC values, which indicates that even though the model predicted false positives at a higher, it maintains a semblance of balance when predicting the labels and does not heavily favor one label over the other. We also note that the accuracy and specificity in [11] are already very high and our proposed approach reaches very close results.

Table 5.14: Comparative results on Flutter

Our results			Results of [11]		
Sensitivity	**0.430457**	±0.006165	Sensitivity	0.362858	±0.002897
Specificity	0.997046	±7.88E-05	Specificity	**0.998672**	±0.000111
PPV	0.639448	±0.007597	PPV	**0.835553**	±0.009336
F-measure	0.442763	±0.00403	F-measure	**0.444367**	±0.00413
G-Mean	**0.632492**	±0.004765	G-Mean	0.575371	±0.002321
Accuracy	0.993088	±6.18E-05	Accuracy	**0.993618**	±0.000128
MCC	0.481157	±0.003179	MCC	**0.508253**	±0.004653
AUC	**0.713752**	±0.003061	AUC	0.680765	±0.00147

Table 5.15 shows the results of the Mann-Whitney test for the Flutter software system. The results reinforce that the proposed model can reach statistically significantly better results in terms of sensitivity, G-mean, and AUC.

Table 5.15: p-value and decision of Mann-Whitney on Flutter

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.999	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.5	Fail to reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.999	Fail to reject H_0
MCC	0.999	Fail to reject H_0
AUC	0.003	Reject H_0

5.5.6 Gephi

Gephi is a heterogeneous tool but the vast majority of its files were developed using Java. The model of [11] outperforms the proposed model in terms of sensitivity, PPV, F-measure, and MCC. However, our proposed model outperforms its counterpart in terms of specificity, G-mean, Accuracy, and AUC (Table 5.16). Both models obtain very close results in terms of sensitivity, specificity, G-mean, and AUC. This indicates that they can predict how the change propagates at a similar success rate.

Table 5.16: Comparative results on Gephi

Our results			Results of [11]		
Sensitivity	0.409034	±0.001958	Sensitivity	**0.409873**	±0.00212
Specificity	**0.997089**	±5.25E-05	Specificity	0.993387	±0.000198
PPV	0.561205	±0.00404	PPV	**0.601158**	±0.00676
F-measure	0.357619	±0.003944	F-measure	**0.363059**	±0.002422
G-Mean	**0.607028**	±0.000824	G-Mean	0.604694	±0.001539
Accuracy	0.**994481**	±4.74E-05	Accuracy	0.990924	±0.000196
MCC	0.40942	±0.003676	MCC	**0.417472**	±0.002692
AUC	**0.703062**	±0.001005	AUC	0.70163	±0.001033

Table 5.17 shows the results of the Mann-Whitney test for the Gephi software system. The test shows that the proposed model statistically significantly outperforms its counterpart in terms of specificity, G-mean, and accuracy.

Table 5.17: p-value and decision of Mann-Whitney on Gephi

Metric	p-value	Conclusion
Sensitivity	0.654	Fail to reject H_0
Specificity	0.003	Reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.924	Fail to reject H_0
G-Mean	0.015	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.999	Fail to reject H_0
AUC	0.111	Fail to reject H_0

5.5.7 Hbase

Hbase is a heterogeneous tool that was mostly developed using Java. while the model of [11] outperforms the proposed model in terms of specificity, and PPV, the latter one outperforms it in terms of sensitivity, F-measure, G-mean, accuracy, MCC, and AUC (Table 5.18). This indicates that the proposed model has a better tendency to predict files that were affected by the change.

Table 5.18: Comparative results on Hbase

Our results			Results of [11]		
Sensitivity	**0.532073**	±0.007499	Sensitivity	0.359416	±0.001111
Specificity	0.998808	±9.71E-05	Specificity	**0.999328**	±5.27E-05
PPV	0.878154	±0.008496	PPV	**0.930827**	±0.003162
F-measure	**0.612811**	±0.00582	F-measure	0.487137	±0.001464
G-Mean	**0.708647**	±0.004677	G-Mean	0.581299	±0.001039
Accuracy	**0.994341**	±0.000115	Accuracy	0.992181	±4.83E-05
MCC	**0.653203**	±0.005447	MCC	0.553064	±0.001426
AUC	**0.765441**	±0.003747	AUC	0.679372	±0.00055

Table 5.19 shows the results of the Mann-Whitney test for the Hbase software system. The results reinforce that the proposed model attains statistically significantly better results in terms of sensitivity, F-measure, G-mean, accuracy, MCC, and AUC.

Table 5.19: p-value and decision of Mann-Whitney on Hbase

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.999	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.8 Laravel

Laravel is a heterogeneous tool mostly developed using PHP. The model of [11] outperforms the proposed model in terms of accuracy only while the proposed approach outperforms it in all other metrics (Table 5.20). Furthermore, given that Laravel is severely imbalanced, accuracy cannot be considered a good measure of performance. Obtained results indicate that the proposed model is more successful in identifying the change sets than its counterpart.

Table 5.20: Comparative results on Laravel

Our results			Results of [11]		
Sensitivity	**0.524476**	±0.003127	Sensitivity	0.497293	±0.000664
Specificity	**0.998871**	±5.94E-05	Specificity	0.998402	±0.000105
PPV	**0.888345**	±0.005961	PPV	0.852061	±0.00158
F-measure	**0.645082**	±0.003265	F-measure	0.628039	±0.00158
G-Mean	**0.721041**	±0.001834	G-Mean	0.68736	±0.000651
Accuracy	0.994368	±6.93E-05	Accuracy	**0.994435**	±0.00011
MCC	**0.67252**	±0.00352	MCC	0.619177	±0.001087
AUC	**0.761673**	±0.001569	AUC	0.747847	±0.000358

Table 5.21 shows the results of the Mann-Whitney test for the Laravel software system. The results reinforce that the proposed model can reach statistically significantly better results in terms of sensitivity, specificity, PPV, F-measure, G-mean, MCC, and AUC.

Table 5.21: p-value and decision of Mann-Whitney on Laravel

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.003	Reject H_0
PPV	0.003	Reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.888	Fail to reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.9 Lucene

Lucene is a heterogeneous system mostly developed using PHP. The model of [11] outperforms the proposed model in terms of sensitivity alone while our proposed approach outperforms that of [11] in terms of specificity, PPV, F-measure, G-mean, accuracy, MCC, and AUC (Table 5.22). The performance of the proposed model on this software is the opposite of what is seen on the majority of the systems so far as it managed to better classify the negative labels, i.e. files that were not impacted by the change.

Table 5.22: Comparative results on Lucene

Our results			Results of [11]		
Sensitivity	0.496301	±0.002596	Sensitivity	**0.500143**	±0.004749
Specificity	**0.998508**	±8.18E-05	Specificity	0.996227	±0.000172
PPV	**0.82343**	±0.006339	PPV	0.737058	±0.009245
F-measure	**0.554753**	±0.00388	F-measure	0.504545	±0.005681
G-Mean	**0.680138**	±0.001746	G-Mean	0.678574	±0.004184
Accuracy	**0.995539**	±7.79E-05	Accuracy	0.992746	±0.000154
MCC	**0.60041**	±0.003769	MCC	0.551392	±0.005847
AUC	**0.747404**	±0.001312	AUC	0.748185	±0.002355

Table 5.23 shows the results of the Mann-Whitney test for the Lucene software system. The results reinforce that the proposed model reaches statistically significantly better results in terms of specificity, PPV, F-measure, and MCC. The test however shows that the proposed model does not significantly outperform its counterpart on G-mean and AUC.

Table 5.23: p-value and decision of Mann-Whitney on Lucene

Metric	p-value	Conclusion
Sensitivity	0.888	Fail to reject H_0
Specificity	0.003	Reject H_0
PPV	0.003	Reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.273	Fail to reject H_0
Accuracy	0.003	Reject H_0
MCC	0.003	Reject H_0
AUC	0.789	Fail to reject H_0

5.5.10 Monitor Control

Monitor Control is a heterogeneous system that was mainly developed using Swift. The model of [11] outperforms the proposed model in terms of specificity, PPV, F-measure, accuracy, and MCC. However, our proposed model outperforms its counterpart in terms of sensitivity, G-mean, and AUC (Table 5.24). This shows that the proposed model is biased towards predicting labels as positives, i.e., files as changed. And although it fails to outperform its counterpart on the five out of eight of the performance metrics, it did showed a good senstivity value indicating that the model is able to detect the files that will be impacted by a change as well as good G-mean and AUC values which indicates that it keeps some semblance of balance in its predictions.

Table 5.24: Comparative results on Monitor control

Our results			Results of [11]		
Sensitivity	**0.463359**	±0.005495	Sensitivity	0.424124	±0.003549
Specificity	0.962864	±0.003056	Specificity	**0.978412**	±0.001216
PPV	0.554475	±0.034973	PPV	**0.669498**	±0.015681
F-measure	0.407382	±0.010212	F-measure	**0.413566**	±0.007902
G-Mean	**0.646682**	±0.00374	G-Mean	0.610572	±0.004582
Accuracy	0.941351	±0.002901	Accuracy	**0.951091**	±0.001437
MCC	0.42862	±0.015276	MCC	**0.453391**	±0.008975
AUC	**0.713111**	±0.0037	AUC	0.701268	±0.002259

Table 5.25 shows the results of the Mann-Whitney test for the Monitor control software system. The test shows that the proposed model statistically significantly outperforms its counterpart in terms of sensitivity, G-mean, and AUC.

Table 5.25: p-value and decision of Mann-Whitney on Monitor control

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.999	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.789	Fail to reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.999	Fail to reject H_0
MCC	0.992	Fail to reject H_0
AUC	0.003	Reject H_0

5.5.11 pyDriller

pyDriller is a homogeneous tool that was developed using Python. The model of [11] outperforms the proposed model in terms of PPV and accuracy. However, our proposed approach outperforms its counterpart in terms of sensitivity, F-measure, G-mean, MCC, and AUC (Table 5.26). This shows that the model is somewhat biased towards predicting files as changed, but that it can better predict the files affected by the change and that it is able to balance its performance better than on some of the previous software systems such as Monitor Control and Flutter.

Table 5.26: Comparative results on PyDriller

Our results			Results of [11]		
Sensitivity	**0.542188**	±0.023199	Sensitivity	0.405409	±0.003851
Specificity	0.986687	±0.001517	Specificity	**0.995753**	±0.000287
PPV	0.777604	±0.020027	PPV	**0.895789**	0.005219
F-measure	**0.602664**	±0.01272	F-measure	0.530504	±0.004593
G-Mean	**0.719992**	±0.013731	G-Mean	0.623401	±0.003181
Accuracy	0.964622	±0.00085	Accuracy	**0.971023**	±0.000391
MCC	**0.614343**	±0.012642	MCC	0.575329	±0.004706
AUC	**0.764437**	±0.011079	AUC	0.700581	±0.002009

Table 5.25 shows the results of the Mann-Whitney test for the pyDriller software system. The results reinforce that the proposed model is capable of reaching statistically significantly better results in terms of sensitivity, F-measure, G-mean, MCC, and AUC.

Table 5.27: p-value and decision of Mann-Whitney on pyDriller

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.999	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.999	Fail to reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.12 React

React is a heterogeneous tool that was mostly developed using JavaScript. The model of [11] outperforms the proposed model in terms of specificity, and PPV. However, our proposed approach outperforms its counterpart in terms of sensitivity, F-measure, G-mean, accuracy, MCC, and AUC (Table 5.28). This shows that the model is better at predicting the files affected by the change and can balance its performance better than some previous software systems such as the case of Monitor control and Flutter.

Table 5.28: Comparative results on React

Our results			Results of [11]		
Sensitivity	**0.60567**	±0.006021	Sensitivity	0.433088	±0.00207
Specificity	0.998702	±3.33E-05	Specificity	**0.99936**	±2.52E-05
PPV	0.791111	±0.007057	PPV	**0.891961**	±0.003319
F-measure	**0.647404**	±0.004903	F-measure	0.547719	±0.001547
G-Mean	**0.761981**	±0.003685	G-Mean	0.639943	±0.001599
Accuracy	**0.996629**	±4.00E-05	Accuracy	0.99583	±1.74E-05
MCC	**0.670279**	±0.005395	MCC	0.598329	±0.001276
AUC	**0.802186**	±0.003008	AUC	0.716224	±0.001028

Table 5.29 shows the results of the Mann-Whitney test for the React software system. The results reinforce that the proposed model can reach statistically significantly better results in terms of sensitivity, F-measure, G-mean, accuracy, MCC, and AUC.

Table 5.29: p-value and decision of Mann-Whitney on React

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.999	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.13 Rocketmq clients

Rocketmq clients is a heterogeneous tool that does not have a dominant programming language. The most used programming language during its development is C++, but it only accounts for 42.6% of its files. The model of [11] outperforms the proposed model in terms of PPV (table 5.30). However, our proposed approach outperforms its counterpart in terms of sensitivity, specificity, F-measure, G-mean, accuracy, MCC, and AUC.

This shows that the model somewhat achieved a balance as it is able to better identify the files affected by a change.

Table 5.30: Comparative results on Rocketmq clients

Our results			Results of [11]		
Sensitivity	**0.421413**	±0.008877	Sensitivity	0.338076	±0.003656
Specificity	**0.993952**	±0.000117	Specificity	0.965404	±0.003162
PPV	0.662078	±0.009058	PPV	**0.745738**	±0.007276
F-measure	**0.437137**	±0.007128	F-measure	0.354144	±0.004547
G-Mean	**0.619653**	±0.00567	G-Mean	0.540133	±0.002362
Accuracy	**0.982993**	±0.000177	Accuracy	0.953826	±0.003056
MCC	**0.475385**	±0.006963	MCC	0.416766	±0.004489
AUC	**0.707683**	±0.004462	AUC	0.65174	±0.001018

Table 5.29 shows the results of the Mann-Whitney test for the Rocketmq clients software system. The results reinforce that the proposed model can reach statistically significantly better results in terms of sensitivity, specificity, F-measure, G-mean, accuracy, MCC, and AUC.

Table 5.31: p-value and decision of Mann-Whitney on Rocketmq clients

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.003	Reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.14 Spark

Spark is a heterogeneous system that was mostly developed using Scala. The model of [11] outperforms the proposed model in terms of PPV and MCC. However, our proposed approach outperforms that of [11] in terms of sensitivity, specificity, F-measure, G-mean, accuracy, and AUC (Table 5.32). This shows that the model somewhat achieved a balance as it is able to better identify the files affected by a change than those that are not.

Table 5.32: Comparative results on Spark

Our results			Results of [11]		
Sensitivity	**0.485068**	± 0.003813	Sensitivity	0.446137	±0.001734
Specificity	**0.997498**	±6.32E-05	Specificity	0.997413	±6.44E-05
PPV	0.67604	±0.006793	PPV	**0.798251**	±0.00771
F-measure	**0.501159**	±0.004931	F-measure	0.493248	±0.003407
G-Mean	**0.673655**	±0.002741	G-Mean	0.640558	±0.001531
Accuracy	**0.993414**	±9.45E-05	Accuracy	0.992471	±6.33E-05
MCC	0.534668	±0.005149	MCC	**0.547153**	±0.003999
AUC	**0.741283**	±0.001938	AUC	0.721775	±0.000859

Table 5.33 shows the results of the Mann-Whitney test for the Spark software system. The test shows that the proposed approach significantly outperforms its counterpart in terms of sensitivity, F-measure, G-mean, Accuracy, and AUC. However, even though our model attained an average specificity higher than that of [11], the Mann-Whitney test showed that the difference is not statistically significant.

Table 5.33: p-value and decision of Mann-Whitney on Spark

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.111	Fail to reject H_0
PPV	0.999	Fail to reject H_0
F-measure	0.015	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.999	Fail to reject H_0
AUC	0.003	Reject H_0

5.5.15 WWW site

WWW site is a heterogeneous tool that was mostly developed using HTML. Our proposed approach achieved perfect performance on this system and outperforms its counterpart in every performance metric (Table 5.34). This shows that the model is able to learn the patterns of change propagation better than the model of [11].

Table 5.34: Comparative results on WWW site

Our results			Results of [11]		
Sensitivity	**0.566667**	±0.038188	Sensitivity	0.505462	±0.00382
Specificity	**0.987971**	±0.002507	Specificity	0.96429	±0.002417
PPV	**0.820833**	±0.033203	PPV	0.687451	±0.030368
F-measure	**0.633333**	±0.018305	F-measure	0.490801	±0.014032
G-Mean	**0.735005**	±0.022725	G-Mean	0.678843	±0.002378
Accuracy	**0.966363**	±0.001599	Accuracy	0.958802	±0.002351
MCC	**0.647561**	±0.015617	MCC	0.524046	±0.014924
AUC	**0.777319**	±0.018168	AUC	0.734876	±0.001625

Table 5.35 shows the results of the Mann-Whitney test for the WWW site software system. The results show that the proposed approach managed to statistically significantly outperform its counterpart on all metrics.

Table 5.35: p-value and decision of Mann-Whitney on WWW site

Metric	p-value	Conclusion
Sensitivity	0.003	Reject H_0
Specificity	0.003	Reject H_0
PPV	0.003	Reject H_0
F-measure	0.003	Reject H_0
G-Mean	0.003	Reject H_0
Accuracy	0.003	Reject H_0
MCC	0.003	Reject H_0
AUC	0.003	Reject H_0

5.5.16 Overview of Results

Below, we list the general observations we derived from all the experiments we conducted across all data sets:

- We performed a total of 120 comparisons across all software systems using 8 metrics, out of which our proposed model outperformed that of [11] on 85 i.e. in 70.8% of the comparisons. Eighty two out of these 85 comparisons were statistically significant. Therefore, the model statistically significantly outperformed its counterpart on 68.33% of the comparisons.

- In terms of sensitivity: We performed 15 comparisons. Our model statistically significantly outperformed its counterpart on 13 of them (86.66%).

- In terms of specificity: We performed 15 comparisons. Our model outperformed its counterpart on 8 of them(53.33%). Out of these 8 comparisons, 7 were statistically significant. Therefore, the model statistically significantly outperformed its counterpart in 46.66% cases.

- In terms of PPV: We performed 15 comparisons out of which our model significantly outperformed its counterpart on 3 of them (20%).

- In terms of F-measure: We performed 15 comparisons out of which our model significantly outperformed its counterpart on 12(80%).

- In terms of G-mean: We performed 15 comparisons in all of which our model outperformed its counterpart (100%). Out of these 15 comparisons, 14 were statistically significant. Therefore, the model statistically significantly outperformed its counterpart on 93.33% of these comparisons.

- In terms of accuracy: We performed 15 comparisons out of which our model statistically significantly outperformed its counterpart on 10 (66.66%).

- In terms of MCC: We performed 15 comparisons out of which our model outperformed its counterpart on 12 of them (80%). Out of these 12 comparisons, 10 of them were statistically significant. Therefore, the model statistically significantly outperformed its counterpart on 66.66% of these comparisons.

- In terms of AUC: We performed 15 comparisons out of which our model outperformed its counterpart on 14 of them (93.33%). Out of these 14 comparisons, 13 of them were statistically significant. Therefore, the model statistically significantly outperformed its counterpart on 86.66% of these comparisons.

The above indicates that the proposed system shows major improvement in terms of sensitivity, F-measure, G-mean, MCC, and AUC. It also outperforms its counterpart the majority of times in terms of specificity and accuracy. However, it scores lower values in terms of PPV. That is because the proposed model has a tendency to overestimate the number of files affected by a change. We encourage this behavior by setting high ρ values, which is the predicted change set size threshold (see Section 5.4). This is because we believe that a false negative, i.e. predicting a file as not affect by a change, when in actuality it is, is more harmful to the maintainability of the software than making a false positive prediction, i.e. predicting an unaffected file as affected.

5.5.17 Statistical Analysis Across Systems

To further validate our analysis and ensure that the average performance of the proposed model is statistically significantly higher than that of [11], we perform a one-tailed Wilcoxon Signed-Rank test [109] with a significance value $\alpha = 0.05$. The samples compared are the averages of each metric across the different systems. We group the values of a given performance metric of our model across all the software systems in one group, and the results of the counterpart on that same metric across all software systems in another group. And then, we compare these two groups. The null hypotheses H_0 is the following: *Given metric M, our model does not statistically significantly outperform the model in [11] on M.* Table 5.36 shows the obtained p-values.

Table 5.36: Results of Wilcoxon test at a confidence level $\alpha = 0.5$

Metric	p-value	Conclusion
Sensitivity	0.0017	Reject H_0
Specificity	0.6044	Fail to reject H_0
PPV	0.9681	Fail to reject H_0
F-measure	0.0285	Reject H_0
G-Mean	0.0011	Reject H_0
Accuracy	0.1392	Fail to reject H_0
MCC	0.0483	Reject H_0
AUC	0.0008	Reject H_0

Results of the statistical test reinforce that the proposed model statistically significantly outperforms its counterpart in terms of sensitivity, F-measure, G-mean, MCC, and AUC. However, it failed to do so on the specificity, PPV, and accuracy metrics.

5.6 Discussion

Looking at all the results, we note that our model outperforms that of [11] in metrics that reflect the rate of true positives, false negatives (sensitivity) as well those that reflect unbiasedness (G-mean, F-measure, MCC, and AUC), as seen in Figure 5.4. The proposed model also significantly outperforms its counterpart on the majority of the systems in terms of accuracy. However, since the accuracy metric is biased towards the majority label, and since the data sets used are extremely imbalanced where the vast majority of the files will not change in a commit, we do not rely on this metric to claim the supremacy of our model.

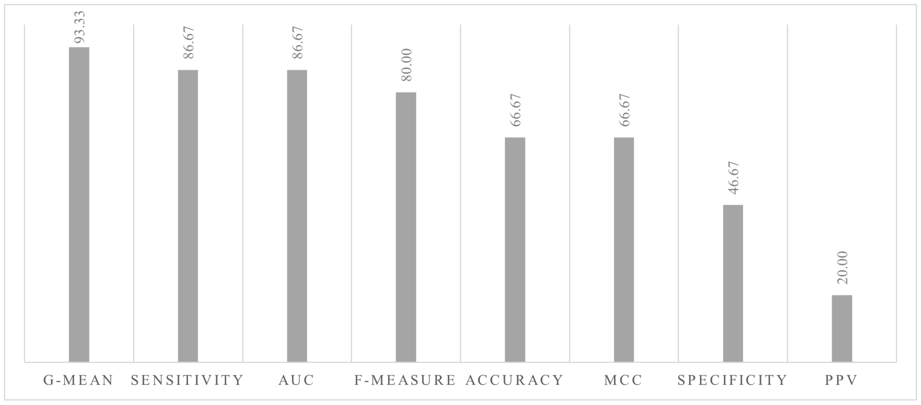


Figure 5.4: Percentage of comparisons where the proposed model significantly outperforms its counterpart

The proposed model is better at predicting files that have been affected by the change since it has high recall values, but it also has a tendency to overestimate their number since the performance on PPV is sometimes low.

On the performance metrics that take into consideration both labels in an imbalanced data set, namely F-measure, G-mean, MCC, and AUC, our model outperforms its counterpart on most of the data sets. The model does not outperform its counterpart in terms of specificity and PPV, though. The specificity metric focuses on the true negative and false positive values while PPV focuses on the proportion of files that were correctly predicted as changed. Therefore, our model wrongly predicts more than its counterpart, that some files were changed. Therefore, the model in [11] is better at detecting files that will not change. We acknowledge that having many false positives in the prediction set is bothersome to the developer as they would have to go through the files and check for non-existent dependencies. However, having many false negatives is way more damaging as it would mislead the developer and the project manager equally. As a matter of fact, false negatives results in developers leaving out some required changes and project managers underestimating the total time, effort and budget required for a particular change. In the best-case scenario, a model should be able to achieve low rates of false positives and false negatives, however, we believe that if this is not attainable, it is better to achieve a low level of false negatives, and an acceptable level of false positives to uphold the overall quality of the software system.

We investigate the cases where our model did not perform well in terms of sensitivity and balanced metrics. We specifically investigate the Gephi and Lucene systems where the model of [11] slightly outperforms the proposed model on sensitivity. Both of these systems are heterogeneous and are mostly built using Java. However, many other systems such as Cassandra, and Hbase were also mainly developed using Java. On the latter systems, the performance of the proposed model is superior, which negates the claim that the model failed to learn well on Gephi and Lucene due to the underlying programming languages used. Both of these systems also have different functionalities where one is a visualization tool (Gephi), and the other is a search engine library (Lucene). The distribution of the size of commits (min, median, mean, and max) in these systems conforms as well to those of the remaining systems (Table 5.3). Additionally, the number of commits on both of these systems is large, but so is that of many other systems tested (Ant, Casandra, Flutter, Hbase, Laravel, React, Spark), and the model performed well on these systems. We note however, that both Lucene and Gephi have the largest number of files (Table 5.3). The model therefore might have failed to learn from these systems due to their large size. However, the relationship between the performance of the models and the system size requires further investigation. We can therefore conclude that there is no direct link between the nature (homogeneous or heterogeneous), function, length of history (number of commits), or the programming languages used to develop these systems and the performance of the models but we do notice that the number of files might have some effect on the performance of the model, although the trend is not clear.

Furthermore, we investigate the characteristics of the temporal graphs built and analyze their average density. The *density* of a graph is the ratio of edges that are present in the graph to the maximum number of edges that this graph can have. The density of a directed graph with no self-loops[1] is computed using Formula 5.9 where $|E|$ is the number of edges in the graph and $|V|$ is the number of vertices.

$$Density_G = \frac{|E|}{|V||(V-1)|} \tag{5.9}$$

We compute the density of every slice in the temporal graph built and report the average for every system in Table 5.37. Results show that Gephi has the lowest density value. However, the density of Lucene does not stand out as there are many systems (Flutter, Laravel, Spark, WWW site) with a temporal graph that is less dense and on which the proposed model managed to perform well.

[1]A self-loop is an edge from one vertex to itself.

Table 5.37: Average density of the temporal graphs

Software system	Density
Alamofire	0.780765128
Ant	0.680531282
Cassandra	0.735817785
Cassandra website	0.680880658
Flutter	0.628611804
Gephi	0.621887397
Hbase	0.76857526
Laravel	0.6606467
Lucene	0.678689195
Monitor control	0.684564073
pyDriller	0.687118019
React	0.687249903
Rocketmq clients	0.682907611
Spark	0.674177019
WWW site	0.631825596

We conclude that the cause of degradation in the performance of the proposed model on Lucene and Gephi might be a combination of the above parameters. For example, the model might have failed to learn the patterns of change propagation on Gephi due to the large number of files in the graph and its low density. However, we need to investigate the performance of the model on many more software systems before we can reach a solid conclusion.

5.7 Generalizability

In our study, we reported the results on software systems of varying size, functionality, and nature. We tested the proposed approach on both homogeneous and heterogeneous systems developed using a variety of programming languages. Results show that the model performs reasonably well on the tested systems, which encourages the possibility that the proposed approach can learn from different types of software systems.

5.8 Realism and Limitations

When extracting the history of a software system from a versioning repository like Github, it is impossible to know which files were modified before which others. Therefore, in the testing phase, the model chose a random file as the source of the change and then predicted which files are affected by this change. To account for this, we report the average performance of the model on different runs for each software system selecting, at each time, a different source. Additionally, we assume that a commit is a complete change set i.e. it contains all the changes that the developer implemented when correcting a single bug, or implementing a single feature. However, this might not be the case as some commits may map to a developer fixing two or more bugs, or may not contain all the changes the developer implemented when adding a feature. In case the commit maps to two or more bug corrections or features introduced, the model would create dependencies between files that might not be dependent. And in case the commit does not contain all the files that the developer changed to fix a bug or introduce a new change, then the model will fail to detect some actual dependencies in the code. Although the notion of considering a commit as an entire change set has been used extensively in previous work, it is an oversimplification of the actual change sets.

Chapter 6

Conclusion and Future Work

In this work, we presented an innovative approach to track how change propagates in software systems. We first model the software system as a temporal directed graph where nodes represent system files and edges co-changeability of two files. We add the temporal dimension to the graph to better learn the patterns of change development in the system. To build the graph, we extract the change sets of the software system and connect the files that were changed together after a certain time step. The edges of the graph carry the co-changeability of the two files. We then employ a TGN that learns the pattern of co-changeability between the files of the system. To do so, when a file is marked as changed, the model visits all its adjacent nodes and learns using an LSTM component their pattern of co-changeability in the past. The LSTM model then predicts if the current change will impact this adjacent node as well.

We tested our approach on systems from the Github repository. These systems differ in their size (number of files), their functionality, as well as the programming languages used to implement them. Furthermore, some of the systems are heterogeneous (different programming paradigms and/or different programming languages) while others are homogeneous (one programming language is used in their development). We performed experiments on 15 different systems. Results show that overall, the proposed approach outperforms current methods in predicting how a change propagates in a software system in terms of sensitivity, F-measure, G-mean, MCC, and AUC. The model fails to outperform its counterpart on 2 out of the 15 systems but the difference is minimal and the performance is very comparable in some of the metrics. From the pool of systems that were tested, this failure cannot be attributed to a programming language or length of history. However, a faint pattern emerges when looking at the number of files in the system as the two systems our model did not perform well on have the largest number of files. To have a definitive conclusion on why the model did not outperform its counterpart on these systems, we would need to test it on numerous additional datasets.

This work is the first to explore the use of temporal networks and TGNs on the problem of predicting software change propagation. Although the results using the proposed components are superior, there are still many possible configurations to test and questions to be answered.

An interesting path to explore at the level of the TGN would be the application of new modules to the memory, message function, memory updater, and embeddings of the TGN. At the level of the system representation, it would be interesting to test the model on different granularities of the system where graph nodes can represent methods, or packages, etc. Finally, TGNs can help reformulate the problem from a binary classification one to an overlapping community detection one where the model can immediately identify the sets of files - rather than one file at a time - that are most likely to change together and group them into overlapping sets. We hope that the current research paves the way for these new venues.

Bibliography

[1] Changhui Jiang, Yuwei Chen, Shuai Chen, Yuming Bo, Wei Li, Wenxin Tian, and Jun Guo. A mixed deep recurrent neural network for mems gyroscope noise suppressing. *Electronics*, 8(2):181, 2019.

[2] Faisal Mohammad, Mohamed A Ahmed, and Young-Chon Kim. Efficient energy management based on convolutional long short-term memory network for smart power distribution system. *Energies*, 14(19):6161, 2021.

[3] Charles Babbage. On the application of machinery to the computation of astronomical and mathematical tables. 1824.

[4] Martin H Weik. *A survey of domestic electronic digital computing systems*. Number 971. Ballistic Research Laboratories, 1955.

[5] JL Nayler. Soviet space exploration—the first decade. w. shelton. arthur barker, london. 1969. 350 pp. illustrated. 45s. *The Aeronautical Journal*, 73(700):342–342, 1969.

[6] Dave Zubrow. Measuring software product quality: The iso 25000 series and cmmi. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2004.

[7] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[8] Bixin Li, Qiandong Zhang, Xiaobing Sun, and Hareton Leung. Using water wave propagation phenomenon to study software change impact analysis. *Advances in Engineering Software*, 58:45–53, 2013.

[9] D.L. Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287, 1994.

[10] Ahmed E Hassan and Richard C Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293. IEEE, 2004.

[11] Anushree Agrawal and Rakesh K Singh. Predicting co-change probability in software applications using historical metadata. *IET Software*, 14(7):739–747, 2020.

[12] Siavash Mirarab, Alaa Hassouna, and Ladan Tahvildari. Using bayesian belief networks to predict change propagation in software systems. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 177–188. IEEE, 2007.

[13] Andrew Leigh, Michel Wermelinger, and Andrea Zisman. Evaluating the effectiveness of risk containers to isolate change propagation. *Journal of Systems and Software*, 176:110947, 2021.

[14] Lei Wang, Han Li, and Xinchen Wang. The influences of edge instability on change propagation and connectivity in call graphs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 197–213. Springer, 2016.

[15] Mrinaal Malhotra and Jitender Kumar Chhabra. Improved computation of change impact analysis in software using all applicable dependencies. In *International Conference on Futuristic Trends in Network and Communication Technologies*, pages 367–381. Springer, 2018.

[16] Ali Ben Abdullah, Abdelsalam M. Maatuk, and Osama M. Ben Omran. Change propagation path: An approach for detecting co-changes among software entities. In *The 7th International Conference on Engineering & MIS 2021*, pages 1–6, 2021.

[17] Thomas Rolfsnes, Stefano Di Alesio, Razieh Behjati, Leon Moonen, and Dave W Binkley. Generalizing the analysis of evolutionary coupling for software change impact analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 201–212. IEEE, 2016.

[18] Thomas Rolfsnes, Leon Moonen, Stefano Di Alesio, Razieh Behjati, and Dave Binkley. Aggregating association rules to improve change recommendation. *Empirical Software Engineering*, 23(2):987–1035, 2018.

[19] Sydney Pugh, David Binkley, and Leon Moonen. The case for adaptive change recommendation. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 129–138. IEEE, 2018.

[20] Rongcun Wang, Rubing Huang, and Binbin Qu. Network-based analysis of software change propagation. *The Scientific World Journal*, 2014, 2014.

[21] Megan Bailey, King-Ip Lin, and Linda Sherrell. Clustering source code files to predict change propagation during software maintenance. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 106–111, 2012.

[22] Annie Tsui Tsui Ying. *Predicting source code changes by mining revision history.* PhD thesis, University of British Columbia, 2003.

[23] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[24] Elman Mansimov, Omar Mahmood, Seokho Kang, and Kyunghyun Cho. Molecular geometry prediction using a deep generative graph neural network. *Scientific reports*, 9(1):1–13, 2019.

[25] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.

[26] Zhenchao Sun, Hongzhi Yin, Hongxu Chen, Tong Chen, Lizhen Cui, and Fan Yang. Disease prediction via graph neural networks. *IEEE Journal of Biomedical and Health Informatics*, 25(3):818–826, 2020.

[27] Abhishek Tomy, Matteo Razzanelli, Francesco Di Lauro, Daniela Rus, and Cosimo Della Santina. Estimating the state of epidemics spreading with graph neural networks. *Nonlinear Dynamics*, pages 1–15, 2022.

[28] Artur M Schweidtmann, Jan G Rittig, Andrea Konig, Martin Grohe, Alexander Mitsos, and Manuel Dahmen. Graph neural networks for prediction of fuel ignition quality. *Energy & fuels*, 34(9):11395–11407, 2020.

[29] Connor W Coley, Wengong Jin, Luke Rogers, Timothy F Jamison, Tommi S Jaakkola, William H Green, Regina Barzilay, and Klavs F Jensen. A graph-convolutional neural network model for the prediction of chemical reactivity. *Chemical science*, 10(2):370–377, 2019.

[30] Pietro Bongini, Monica Bianchini, and Franco Scarselli. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450:242–252, 2021.

[31] Yutaro Iiyama, Gianluca Cerminara, Abhijay Gupta, Jan Kieseler, Vladimir Loncar, Maurizio Pierini, Shah Rukh Qasim, Marcel Rieger, Sioni Summers, Gerrit Van Onsem, et al. Distance-weighted graph neural networks on fpgas for real-time particle reconstruction in high energy physics. *Frontiers in big Data*, page 44, 2021.

[32] Yantao Shen, Hongsheng Li, Shuai Yi, Dapeng Chen, and Xiaogang Wang. Person re-identification with deep similarity-guided graph neural network. In *Proceedings of the European conference on computer vision (ECCV)*, pages 486–504, 2018.

[33] Dan Lin, Jianzhe Lin, Liang Zhao, Z Jane Wang, and Zhikui Chen. Multilabel aerial image classification with a concept attention graph neural network. *IEEE Transactions on Geoscience and Remote Sensing*, 60:1–12, 2021.

[34] Weijing Shi and Raj Rajkumar. Point-gnn: Graph neural network for 3d object detection in a point cloud. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1708–1716, 2020.

[35] Pengshuai Yin, Jiayong Ye, Guoshen Lin, and Qingyao Wu. Graph neural network for 6d object pose estimation. *Knowledge-Based Systems*, 218:106839, 2021.

[36] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7370–7377, 2019.

[37] Wang Gao, Yuan Fang, Lin Li, and Xiaohui Tao. Event detection in social media via graph neural network. In *International Conference on Web Information Systems Engineering*, pages 370–384. Springer, 2021.

[38] Jasmijn Bastings, Ivan Titov, Wilker Aziz, Diego Marcheggiani, and Khalil Sima'an. Graph convolutional encoders for syntax-aware neural machine translation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1957–1967, 2017.

[39] Anson Bastos, Abhishek Nadgeri, Kuldeep Singh, Isaiah Onando Mulang, Saeedeh Shekarpour, Johannes Hoffart, and Manohar Kaul. Recon: relation extraction using knowledge graph context in a graph neural network. In *Proceedings of the Web Conference 2021*, pages 1673–1685, 2021.

[40] Wenqi Fan, Yao Ma, Qing Li, Jianping Wang, Guoyong Cai, Jiliang Tang, and Dawei Yin. A graph neural network framework for social recommendations. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[41] Xiangde Zhang, Yuan Zhou, Jianping Wang, and Xiaojun Lu. Personal interest attention graph neural networks for session-based recommendation. *Entropy*, 23(11):1500, 2021.

[42] Marcelo Prates, Pedro HC Avelar, Henrique Lemos, Luis C Lamb, and Moshe Y Vardi. Learning to solve np-complete problems: A graph neural network for decision tsp. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4731–4738, 2019.

[43] Zhihao Xing and Shikui Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem. *IEEE Access*, 8:108418–108428, 2020.

[44] Yujiao Hu, Zhen Zhang, Yuan Yao, Xingpeng Huyan, Xingshe Zhou, and Wee Sun Lee. A bidirectional graph neural network for traveling salesman problems on arbitrary symmetric graphs. *Engineering Applications of Artificial Intelligence*, 97:104061, 2021.

[45] Henrique Lemos, Marcelo Prates, Pedro Avelar, and Luis Lamb. Graph colouring meets deep learning: Effective graph neural network models for combinatorial

problems. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 879–885, 2019.

[46] Martin JA Schuetz, J Kyle Brubaker, Zhihuai Zhu, and Helmut G Katzgraber. Graph coloring with physics-inspired graph neural networks. *arXiv preprint arXiv:2202.01606*, 2022.

[47] Jan Toenshoff, Martin Ritzert, Hinrikus Wolf, and Martin Grohe. Graph neural networks for maximum constraint satisfaction. *Frontiers in artificial intelligence*, 3:580607, 2021.

[48] Martin JA Schuetz, J Kyle Brubaker, and Helmut G Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, 2022.

[49] Jia Li, Zhichao Han, Hong Cheng, Jiao Su, Pengyun Wang, Jianfeng Zhang, and Lujia Pan. Predicting path failure in time-evolving graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1279–1289, 2019.

[50] Yang Yao, Bo Gu, Zhou Su, and Mohsen Guizani. Mvstgn: A multi-view spatial-temporal graph network for cellular traffic prediction. *IEEE Transactions on Mobile Computing*, 2021.

[51] Guangyin Jin, Min Wang, Jinlei Zhang, Hengyu Sha, and Jincai Huang. Stgnn-tte: Travel time estimation via spatial–temporal graph neural network. *Future Generation Computer Systems*, 126:70–81, 2022.

[52] Hao Zhou, Dongchun Ren, Huaxia Xia, Mingyu Fan, Xu Yang, and Hai Huang. Ast-gnn: An attention-based spatio-temporal graph neural network for interaction-aware pedestrian trajectory prediction. *Neurocomputing*, 445:298–308, 2021.

[53] Defu Cao, Jiachen Li, Hengbo Ma, and Masayoshi Tomizuka. Spectral temporal graph neural network for trajectory prediction. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1839–1845. IEEE, 2021.

[54] Mahdi Khodayar and Jianhui Wang. Spatio-temporal graph deep neural network for short-term wind speed forecasting. *IEEE Transactions on Sustainable Energy*, 10(2):670–681, 2018.

[55] Guangyin Jin, Qi Wang, Cunchao Zhu, Yanghe Feng, Jincai Huang, and Jiangping Zhou. Addressing crime situation forecasting task with temporal graph convolutional neural network approach. In *2020 12th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, pages 474–478. IEEE, 2020.

[56] Sijie Yan, Yuanjun Xiong, and Dahua Lin. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[57] Byung-Hoon Kim, Jong Chul Ye, and Jae-Jin Kim. Learning dynamic graph representation of brain connectome with spatio-temporal attention. *Advances in Neural Information Processing Systems*, 34:4314–4327, 2021.

[58] Robert L. Glass, Iris Vessey, and Venkataraman Ramesh. Research in software engineering: an analysis of the literature. *Information and Software technology*, 44(8):491–506, 2002.

[59] Korea Standards Association et al. Ks x iso/iec 9126-1: Software engineering-product quality-part 1: Quality model. *Industrial Standards Council*, 2017.

[60] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[61] Tibor Bakota, Péter Hegedűs, Gergely Ladányi, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A cost model based on software maintainability. 2012.

[62] Samuel Ajila. Software maintenance: an approach to impact analysis of objects change. *Software: Practice and Experience*, 25(10):1155–1181, 1995.

[63] Muhammad Shahid and Suhaimi Ibrahim. Change impact analysis with a software traceability approach to support software maintenance. In *2016 13th International Bhurban conference on applied sciences and technology (IBCAST)*, pages 391–396. IEEE, 2016.

[64] Albert-László Barabási. Network science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1987):20120375, 2013.

[65] Frank Harary and Gopal Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

[66] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[67] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[68] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[69] A Vinay, Desanur Naveen Reddy, Abhishek C Sharma, S Daksha, NS Bhargav, MK Kiran, KNB Murthy, and S Natrajan. G-cnn and f-cnn: Two cnn based architectures for face recognition. In *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, pages 23–28. IEEE, 2017.

[70] Xiaoqi Wan, Hui Song, Lingen Luo, Zhe Li, Gehao Sheng, and Xiuchen Jiang. Pattern recognition of partial discharge image based on one-dimensional convolutional neural network. In *2018 Condition Monitoring and Diagnosis (CMD)*, pages 1–4. IEEE, 2018.

[71] Jianfeng Zhao, Xia Mao, and Lijiang Chen. Speech emotion recognition using deep 1d & 2d cnn lstm networks. *Biomedical signal processing and control*, 47:312–323, 2019.

[72] Yajie Miao, Mohammad Gowayyed, and Florian Metze. Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 167–174. IEEE, 2015.

[73] Rabih Zbib, Lingjun Zhao, Damianos Karakos, William Hartmann, Jay DeYoung, Zhongqiang Huang, Zhuolin Jiang, Noah Rivkin, Le Zhang, Richard Schwartz, et al.

Neural-network lexical translation for cross-lingual ir from text and speech. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 645–654, 2019.

[74] RF Gibadullin, M Yu Perukhin, and AV Ilin. Speech recognition and machine translation using neural networks. In *2021 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)*, pages 398–403. IEEE, 2021.

[75] Lintang Adyuta Sutawika and Ito Wasito. Restricted boltzmann machines for unsupervised feature selection with partial least square feature extractor for microarray datasets. In *2017 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, pages 257–260. IEEE, 2017.

[76] Yu-Heng Lai, Wei-Ning Chen, Te-Cheng Hsu, Che Lin, Yu Tsao, and Semon Wu. Overall survival prediction of non-small cell lung cancer by integrating microarray and clinical data with deep learning. *Scientific reports*, 10(1):1–11, 2020.

[77] Jan Funke, Fabian Tschopp, William Grisaitis, Arlo Sheridan, Chandan Singh, Stephan Saalfeld, and Srinivas C Turaga. Large scale image segmentation with structured loss based deep learning for connectome reconstruction. *IEEE transactions on pattern analysis and machine intelligence*, 41(7):1669–1680, 2018.

[78] Zhi Zhou, Hsien-Chi Kuo, Hanchuan Peng, and Fuhui Long. Deepneuron: an open deep learning toolbox for neuron tracing. *Brain informatics*, 5(2):1–9, 2018.

[79] Jerzy W Bala and John Robert Anderson. *Machine Learning: A Multistrategy Approach, Volume IV*, volume 4. Morgan Kaufmann, 1994.

[80] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[81] Sepp Hochreiter and Jürgen Schmidhuber. Lstm can solve hard long time lag problems. *Advances in neural information processing systems*, 9, 1996.

[82] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[83] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[84] Paul Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, Colorado Univ at Boulder Dept of Computer Science, 1986.

[85] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[86] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[87] Kristin P Bennett and Olvi L Mangasarian. Neural network training via linear programming. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[88] Jinn-Moon Yang and Cheng-Yan Kao. A robust evolutionary algorithm for training neural networks. *Neural Computing & Applications*, 10(3):214–230, 2001.

[89] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[90] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[91] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[92] Christopher Olah. Understanding lstm networks. 2015.

[93] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[94] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[95] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

[96] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *J. Mach. Learn. Res.*, 21(70):1–73, 2020.

[97] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1269–1278, 2019.

[98] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[99] Leon Moonen, Stefano Di Alesio, Thomas Rolfsnes, and Dave W Binkley. Exploring the effects of history length and age on mining software change impact. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 207–216. IEEE, 2016.

[100] Abdul Ghaaliq Lalkhen and Anthony McCluskey. Clinical tests: sensitivity and specificity. *Continuing education in anaesthesia critical care & pain*, 8(6):221–223, 2008.

[101] Abdel Aziz Taha and Allan Hanbury. Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. *BMC medical imaging*, 15(1):1–28, 2015.

[102] Mohammad Hossin and Md Nasir Sulaiman. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1, 2015.

[103] Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *Icml*, volume 97, page 179. Citeseer, 1997.

[104] Qiuming Zhu. On the performance of matthews correlation coefficient (mcc) for imbalanced dataset. *Pattern Recognition Letters*, 136:71–80, 2020.

[105] James Fogarty, Ryan S Baker, and Scott E Hudson. Case studies in the use of roc curve analysis for sensor-based estimates in human computer interaction. In *Proceedings of Graphics Interface 2005*, pages 129–136, 2005.

[106] Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M Chen, and Yogachandran Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.

[107] Christoph Bergmeir and José M Benítez. On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191:192–213, 2012.

[108] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[109] William Jay Conover. *Practical nonparametric statistics*, volume 350. john wiley & sons, 1999.