Mathematical Biosciences and Engineering

http://www.aimspress.com/journal/mbe

MBE, 19(4): 3701–3719.
DOI: 10.3934/mbe.2022170
Received: 20 October 2021
Revised: 19 January 2022
Accepted: 26 January 2022
Published: 09 February 2022

*Research article*

# Mathematical foundations based statistical modeling of software source code for software system evolution

**Raghavendra Rao Althar[1,2], Abdulrahman Alahmadi[3], Debabrata Samanta[4,*], Mohammad Zubair Khan[3,*] and Ahmed H. Alahmadi[3]**

[1] Department of Data Science, CHRIST University, Bangalore, Karnataka, India
[2] Specialist-QMS, First American India Private Ltd., Bangalore, Karnataka, India
[3] Department of Computer Science and Information, Taibah University, Madinah, Saudi Arabia
[4] Department of Computer Science, CHRIST University, Bangalore, India

* **Correspondence:** Email: debabrata.samanta369@gmail.com, mkhanb@taibahu.edu.sa.

**Abstract:** Source code is the heart of the software systems; it holds a wealth of knowledge that can be tapped for intelligent software systems and leverage the possibilities of reuse of the software. In this work, exploration revolves around making use of the pattern hidden in various software development processes and artifacts. This module is part of the smart requirements management system that is intended to be built. This system will have multiple modules to make the software requirements management phase more secure from vulnerabilities. Some of the critical challenges bothering the software development community are discussed. The background of Machine Learning approaches and their application in software development practices are explored. Some of the work done around modeling the source code and approaches used for vulnerabilities understanding in software systems are reviewed. Program representation is explored to understand some of the principles that would help in understanding the subject well. Further deeper dive into source code modeling possibilities are explored. Machine learning best practices are explored inline with the software source code modeling.

**Keywords:** software development; requirements management; Bidirection encoders transformers; knowledge graphs; transfer learning

## 1. Introduction

Software development landscape houses various data that can provide valuable insights into the way software development processes operate. With a wide variety of problems bothering software development processes, these insights can play a crucial role in solving some of those. Software security is a critical need for software industries; it will be necessary for the software industry to explore these insights. Software systems operate with a specific pattern; software source code can have many stories to tell about the application. Many of the testing and scanning activities that happen within software development processes can also provide handy information. If put together in a meaningful way and fed back to the practitioners at the right time, all this information can help build secured software systems. Security issues found early in the project life cycle can help mitigate them much more comfortably and with low cost. So shifting the focus of security to the left of the software development phases will be vital. In this work, there is a focus on the requirements management phase. They enable the requirements manager with all possible information to operate with a security mindset right at the requirements elicitation phase. The software's naturalness is a much-discussed subject, where source code is more predictable than the natural language text itself. In this work, exploration will be done on statistical modeling of the source code to see how patterns can be learned from the source code. This study will help the requirements management team with critical insights about the application. The necessary security-related requirements are taken care of right at the requirements management phase.

## *1.1. Motivation*

Software development is extensive information processing work; a fair amount of data is generated across the life cycle phases. Traditionally the focus lies in delivering the needs of the customer based on their specifications. Complex software systems were also making these processes focused on what we have on hand during development. Not much attention is paid to looking back at the patterns of the processes underneath the software development cycle. Growing technology also has kept the software development practitioner on the toe to adapt in no time. The business also is focusing on taking the products and services to the market in a short time. Addressing the issues is reactive only if the users raise major red flags. With this approach keeping lights on being the only focus, many weaknesses have crept into the system over time. If these weaknesses are deep into the root, it poses challenging questions about the possibility of getting out of this situation.

## *1.2. Objective*

As part of the management strategy, there is a lack of enough attention to leveraging the operations' learnings and patterns. Lacking collaboration among the team members and their customer counterparts is also not helping the situation. It is very tough to bring discipline into necessary activities like a disciplined approach for coding practices. There is no streamlined mechanism for periodically enhancing these guidelines and educating the team on the latest updates. Though the knowledge is available across the industry and within the organization, this seems overwhelming to the practitioners. Getting the correct information at the right time in the hands of the right people is not happening. Objective of this study is to explore the mechanics to combine information from all the useful sources and provide it to practitioners in a way that can be used as and when needed

## *1.3. Contribution*

Work explores the gaps associate with the statistical modeling of the software artifacts. Exploration extends to understanding the program representation constructs. Source code being a critical construct of software system, understanding the pattern hidden will be handy information. These learning will help to build intelligent systems for information processing and consumption.

## 2. Background of computational intelligence

With the explosion of computational resources in recent decades, possibilities have skyrocketed in computational intelligence. Though the field has been around for ages, primarily in statistics, there has been a revolution around building this intelligence in various domains. Progress in the deep learning arena has made it possible to experiment with complex scenarios. However, there has been steep growth in data-based operations. Some corporations are still waiting to see if these make sense [1]. Or there is limited use of the computational intelligence restricting only to direct business-related use cases. Traditionally, computational intelligence found its root in mathematical modeling; in the recent past, there has been a significant effort to keep applications at the forefront as new frameworks are devised. Inspirations from biological neurons in the world of deep learning were a significant breakthrough. The evolution of computational intelligence can be seen from fuzzy logic to neural networks, evolutionary computation, learning theory, and probabilistic models.

## 3. Review of literature

In work [2], the author highlights that due to increasing software security vulnerabilities that compromise the software systems, there is a significant need for focus. Their work proposes to use a machine learning approach for developing a vulnerability detection system by studying from the open-source code source available for C and C++. Data sets used were both the existing labeled vulnerability dataset and the manually labeled one [3]. The labeling was done on open-source functions from three different static analyzers that indicated potential exploitation in the system. Lexed source code interpretation is made with deep feature representation learning, which is fast and scalable, using the dataset. Outcomes were evaluated on real software packages and the NIST SATE IV benchmark dataset. Authors confirm that their study shows that the deep feature representation learning on source code is a promising approach in vulnerabilities detection for the software source code. In work [4], unsupervised word2vec methods are trained for a large set of the unlabelled dataset to explore the embedding. Still, the performance on the classification was minimal compared to embeddings, which were learned from random initialization. In work [5], natural language processing-based work is explored with the background of source code sharing common aspects concerning writing the code and also for work done on the programming language. In work [6], the author highlights the importance of detecting the vulnerabilities and removing them from the software. They emphasize that the previous studies have focused on prediction models for checking vulnerabilities of the software components. Still, there is a more profound need to ensure that these approaches are made more accurate and useful. In this work, the author uses n-gram analysis and feature selection algorithm for predicting vulnerable software components [7, 8]. In this case, features are the tokens in the software source code files, like Java class files. Different feature selection algorithms from machine learning are employed to reduce the feature space for searching optimal functions. The authors have evaluated their approach on Java-based android applications and confirm that the proposed technique demonstrates higher precision, accuracy, and recall for predicting vulnerable classes. In work [9], n-gram analysis is employed to identify faulty GUI (Graphical User Interface) features of the software testing domain. Challenges associated with the vulnerabilities related prediction model are studied in work [10]. For vulnerable software components exploration, though machine learning application is the primary thought process, where the features are based on the study, work like [11, 12] focuses on classification approach based on ensemble methods.

In work [13], author highlight that its essential to detect vulnerable components of web application so that the resources needed for verification can be effectively allocated. In the prior studies, public and private datasets are used to propose various prediction models for vulnerabilities. The authors have developed web services for vulnerabilities prediction in software hosted on the Azure cloud computing platform in this work. The authors investigated many machine learning approaches available in the Azure machine learning studio and found an excellent overall performance using the Multi-layer perceptron model. Software metrics data are fed into the system as web form, used by web services that do vulnerabilities prediction. Prediction outcomes are shown on the web form for testing purposes. Drupal, Moodle, and PHPMyAdmin project datasets are used in these experiments. Artificial Neural Networks have shown promising performance, and web services format helps in complex systems. In work [14], a static analysis tool is leveraged to detect vulnerabilities with machine learning approaches. Work [15] explore static and dynamic code attributes for vulnerabilities detection in web applications. Metrics and text mining combinations are employed in work [16] for vulnerabilities prediction purposes. In work [17], authors focus on identifying the source of all software vulnerabilities. The vulture tool built by authors works on existing open-source databases and the archive ones to make sure all these vulnerabilities are mapped to the software components. The ranking is provided for the vulnerable components, which becomes the base information to investigate why the vulnerabilities get into the system. Authors highlight that the components that exhibited the low rate of vulnerabilities in the past are less prone to vulnerabilities currently. Those components which had a similar structure of imports or function calls show the likelihood of vulnerabilities. The authors use this base observation in their vulture tool, which yields a correct prediction for half of the components under the forecast; among these, two-thirds come out as accurate. This prediction gives the project managers a more significant advantage to focus their effort and attention on the parts that matter. Work [18] emphasizes that there is no single metric for correlating failures across multiple projects to predict vulnerabilities. Studies in [19] show how the security issues undergo evolution over time.

In work [20], the author highlights that the software community has been showing interest in security and vulnerability prediction metrics. Cohesion, coupling, and complexity kind of metrics have been proposed. In this work, the author suggests code metrics from dependency graphs to predict vulnerable code components. To validate the efficiency of the metrics presented in the work prediction model is tested on Firefox Javascript engine. The authors confirm that dependency graphs prove to be a promising approach for the early indication of vulnerabilities through this experiment. Work [21] focus on the trends around vulnerabilities research in the modern world. Work [17] introduced new metrics for building vulnerability prediction models. These metrics were function calls and imports. Literature review has helped to firm up on the following aspects of machine learning application in software development. Focus on security needs of the software development processes is evident from the literature review. Deep feature representation learning on source code is a promising approach in vulnerabilities detection for the software source code. Natural Language Processing methodologies and their application to the software development artifacts will be beneficial. Some of the customized tool explored in literature are useful in understanding software development landscape. Exploration also covers the key metrics that can be leveraged in these machine learning approaches.

## 4. Research gap for statistical modeling

One of the prominent gaps is the lack of a wide variety of metrics used to validate the models built on data. Table 1 provides details of some of the validation metrics.

**Table 1.** Model validation metrics.

| Metrics | Definition | Application |
|---|---|---|
| **Accuracy** | (True Positive + True Negative) / (True Positive + False Positive + False Negative + True Negative) | Well suited for classification problem with a well-balanced data set |
| **Precision** | (True Positive) / (True Positive + False Positive) | Suited in case if the confidence on the prediction is very important |
| **Recall** | (True Positive) / (True Positive + False Negative) | Suited in case if identification of positive events are a priority |
| **F1 score** | 2*(precision * re-call) / (precision + recall) | In case of equal focus on precision and recall |
| **Log loss / Binary Crossentropy** | Log loss is useful in the case of binary classifiers; in the case of logistic regression, this provides an optimization Objective. Refer to Eq (4.1). | In cases of classifier output being Prediction probabilities |
| **Categorical Crossentropy** | Log loss is applicable for multiclass problems where probabilities are assigned to each class. The logarithmic Loss function is given in Eq (4.2). | In cases of multiclass prediction probabilities |
| **AUC** | Area Under Curve provides indication | Useful when ranking prediction is important rather than actual values |

$$BinaryCrossEntropy = -(ylog(p) + (1-y)log(1-p)) \tag{4.1}$$

'p' is the probability of predicting '1'.

$$CategoricalCrossEntropy = \frac{-1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} * log(p_{ij}) \tag{4.2}$$

'N' samples belong to 'M' class, $y_{ij}$ is '1' if sample' i' belongs to class 'j'; otherwise, it will be '0'. $p_{ij}$ will be the probability of classifier predicting samples' i' as belonging to class 'j'. Binary Cross Entropy is one of the metrics used in classification approach which compares each of the predicted probabilities to actual class output which can take value of 0 or 1. Further the penalization of the probabilities is done based on the distance from the expected value. This penalization is depicted in terms of the score. Categorical Cross Entropy is another metric used in case of tasks that involve multi-class classification. This is designed to identify a single class to which an example belongs based on quantification of differences between probability distributions.

This experimentation will help develop models that can be stronger in all perspectives and provide an accurate representation of the data domain. In the software development domain, some of the work is done leveraging the metrics and measures, but there is no extensive study in this area that needs attention. Some of the works have utilized regression models for exploring the time required for fixing software issues, but these need to be explored with more advanced applications of regression models.

Time's influence on the progression of the software-related problems and leveraging the hidden knowledge and using it for forecasting the future process states needs a focused study. Availability of the data and its quality are key concerns; there is a need to devise a mechanism to check the influence of the time range of data on the outcomes [22, 23]. There is a need to focus on approaches that can make it up for not depending on too much data that spreads across a long time range. Many of the studies focus on the generic influence of the features on the outcome factor. Still, more focused attention is needed to assess how each feature influences outcome and then fine-tunes the list of the features for those that matter. Dependency on the project-specific data is another constraint that needs to be handled to ensure that a fair amount of generic learning is taken across various projects catering to different domains. This learning will help to make the data science experiments cost-effective for the organization. Some of the works have tried anomaly detection methods for software development process optimization; in these cases, false-positive rates have been a cause of concern and need attention. Many of the works lack validation in the industrial settings, which do not provide accurate information about how these solutions fairing in the real world. Graph-based data representation is taking the forefront, and explorations are done in the software development domain, but this area needs more investigation. In modeling the security-related vulnerabilities in software development processes, there is a lack of effort to handle the security control-related information that is not prevalent in high frequency but is essential as their exploitation may cause huge problems [24]. Threat and vulnerabilities exploration has traditionally been made in software development operations, but there is quite a bit of dependency on the experts; these have the potential to be automated and made more deterministic exploration. There is a need for balance between validation of the experiments across industrial settings and general public data. Since there is a substantial amount of natural language data involved in software development, including the source code, which exhibits quite a bit of naturalness, there is a need for in-depth exploration of text analytics-based approaches. Most of the approaches have been restricted to black-box modeling, and there is a need to establish transparency of the models and make them explainable. These data models' characteristics will help build the approaches effectively to cater to all the needs of the data domain. In the domain of security requirements modeling, the primary challenge of understanding the security categories' operational definitions is essential to build consistency in experiments and its outcome. Since the data is the fuel for data science experiments, extracting the required information is a practical challenge. There is a need for approaches that can cut down the data hungriness of the data models [25, 26]. Software requirements engineering needs further exploration with the latest machine learning and deep learning approaches. In software source code, a more effective mechanism for labeling the source code is required to cater to the appropriate data supply to deep learning models. Much of the work has focused on the retrospective identification of security issues. Still, not much work is done on proactively identifying possibilities of the security issues creeping into the software development processes as the software systems are enhanced.

## 5. Program representation

Program representation formats are AST (Abstract Syntax Tree), control-flow graphs, and program dependency graphs. Understanding these formats goes a long way for progressing in the statistical modeling of software source code. AST helps to represent the software source code in a tree format for a programming language. The construct of the source code is used as a building block for tree construction. Representation only considers essential aspects like content and structure and does not represent everything in the code. To define a syntax-based format for a case of a 'if' command, it will be a single node that branches out to the further divisions. ASTs are a primary source for program analysis where the property of programs is studied to optimize and correct them. Abstract interpretation is another crucial aspect that needs to be understood to understand source code with statistical computation. In abstract interpretation, an approximation of the program's semantics is made to infer information about attributes like control flow and data flow without getting into granularities of the computation [27, 28]. Abstract interpretation will help to understand the software program's pattern and not the precise nature, as accurate inference needs a large amount of computation power and time. Statistical modeling can be built to understand the nature of the software systems by leveraging these patterns. (CFG) Control Flow Graph is a graphical representation of the control flow during program execution. CFG is dependent on the process underneath the operation of the software program. CFG is a directed graph that demonstrates all the path that is involved in the program. Entry and exit blocks are associated with CFG, where the flow between the nodes is shown in between. It's essential to explore the mechanisms where inferences can be derived from graph information. One of the transformations is to put together crucial information by randomly moving through the graph; this move can be fixed in length and help build machine learning features. The pattern learned from those can be vectorized, which will help derive the embeddings of the data hidden in the graphs data. Graph data can also be used in image data, and image comparison can be made to establish the code's attributes. PDG (Program Dependency Graphs) is the mechanism for graph representation where data and control dependencies are brought out straightforwardly [29].

## 6. Exploration in to source code modeling

Work [30] studies the application of machine learning for big code and its naturalness. This work is an extensive survey of all the work done in this area. One of the approaches would be to build the knowledge repositories of the pattern learned from the domain under study; these patterns are gathered in the knowledge bases. New projects can be validated with this knowledge base as a reference. In the process knowledge base gets calibrated over some time. Capabilities of the NLP (Natural Language Processing) can be leveraged to establish dependencies and interconnection between the code's methods and functions. Apart from security vulnerabilities computation, technical debt identification is another prominent area that will be benefitted. Based on the learning from software's naturalness as highlighted in work [31], code summarization and code search-related use cases can be resolved. This characteristic comes in handy while we conduct predictions on the code components. Though programming language looks complex and flexible, code is influenced by the programmer. This characteristic has a significant influence on making code more natural than that of the natural language. It's the limited vocabulary of the programming language that is used is that makes the code natural. Based on this naturalness, we would derive the codebase's statistical properties and used them for our prediction purpose. Computational intelligence built on code will also help cut down the time and effort required by the code reviewer. The reviewer can focus on a crucial part of the review rather than worrying about the syntax part, which the system can take care of. Code commits another critical information that must be mined as part of prediction model building that can hold quite a fair amount of data on the potential vulnerabilities. Approaches used for the application monitoring are handy systems to study more critical features from the software system that can complement the features learned from source code [32].

Another set of features can be generated from the pull requests associated with the codebase; API calls from these can be studied to figure out the flaws compared to the best practices. These changes made can be compared with the critical information that is available for the system. This information helps to understand the dependencies. Training can be conducted with two approaches: creating the example cases and making the model learn, and parsing into publicly available datasets. Rule-based learning can be a starting point to build the training dataset, which can be further leveraged to build on the supervised machine learning models. Based on the study of publicly available datasets, it's possible to understand the code commits and see what kind of code changes are done in the past. These would also assist in the recommendation of the fixes for the code. One of the critical factors is that the more granular the information is in the code, commits better it is for pattern understanding. But since the commits restrict the issue fixing and some form of improvisation to the code and information about the same is not documented in granular form, this poses the challenge and calls for better approaches to identify these granular aspects. Controlling the false positives is one of the critical areas pointed out by other researchers [33]. One of the crucial challenges of extending the models across the domain is the highly specialized language that is difficult to comprehend. Another important consideration is it's not enough to bank on the most used features in the code because the ones outside the list will also have extensive usage and cannot be ignored. The influence of time on the data source used for learning is an important area of exploration. The codebases used for education, particularly from the open-source, need to be assessed for relevancy to the day. This area may need the assistance of the machine learning and deep learning approaches that account time factor [34].

The effort invested in the source code's statistical modeling also will facilitate the building of the software tools. It's the combination of source code, code-related documentation, and publicly available data that can together help make sense of the pattern hidden in the source code. Static analysis tools and machine learning approaches can be looked at hand in hand to build an effective prediction mechanism on the code components. Previous software engineering methods have based themselves on the formal structure of the code; to advance in the progressive direction, it's essential to understand the software program's statistical nature. Besides other factors highlighted towards the software's naturalness, programmers' intention of keeping the software maintainable also keeps the common theme as a base for programming. Probabilistic modeling makes up a promising approach for building together with the uncertainty seen in source code. This method also helps build comprehensive knowledge-making use of text-based sources like requirements documents and other documentation related to the software program. Difficulties involved in the porting of programs from one language to a different language also need focused exploration. Learning from the software traces and flows will provide the right amount of information for pattern recognition [35].

Generation of the code as tokens is one form for modeling with a tree-based syntax representation model and graph structures generated with semantic models. N-gram modeling has seen its implementation in code components, like in language modeling, where previous n-1 tokens are used as bases for predicting the nth token. But the challenge with code modeling is the difficulty to hold the context similar to that of language modeling. So there will be a need to account for context information. LSTM are specialized in retaining the key information from the input data based on its importance to the context of the situation. Hence rather than learning everything from the input, this specific focus helps to capture what is more relevant for the scenario under study. RNN also helps to understand the context from the input data as they act as encoders to retain the input sequence in the form of an internal state. Though the input sequence is not relied upon the state information stored in the RNN can help in focusing on the context of the input data. For accounting, the contextual information, LSTM (Long Short Term Memory) networks, and RNN (Recurrent Neural Networks) are appropriate. Though neural networks can reveal hidden patterns and retain long-term contextual information, the amount of data required for the same makes it very challenging [36]. In contrast to sequential modeling, syntactic tree representation has its advantage. The hierarchy's representation is done in a structured format of a tree drilling down to the features. Abstract Syntax Tree is one such representation, though this representation also needs extensive computation. Graph representation of the source code puts together the critical aspects of sequential learning and tree-based learning. Graph-based modeling finds its challenge as there is no specific starting point for the same. In the language modeling-based approach for software source code modeling, context is accounted by code semantics understanding, but it is a complex area of exploration.

## 7. Application of machine learning for code modeling

Since the source code holds many features, managing those contributing to bringing out the pattern is tricky. Lasso regression can control the less significant features low co-efficient for the variables. Running forward propagation and backward propagation also helps identify the correct pattern in complex data distributions. Literature shows the use of precision, recall, and accuracy metrics extensively; log loss metrics can be explored to understand the models better. During probabilistic modeling, when the Bayes models are built, probabilities reduce to close zero resulting in the probabilities of the events getting rounded off to zero [37]. In case of the likelihood of the event becoming zero, Laplace smoothing technique can be used to ensure the probability equation holds the information of all the events so that influence of all the features in the modeling processes can be retained. One hot encoding and response encoding method of coding can be considered for the case of categorical features. The influence of each of these methods can be studied based on the domain's data, and appropriate techniques can be used. In the KNN (K-Nearest Neighbour) classifier being used to classify the vulnerable code components, the calibrated classifier with cross-validation is one parameter used within this model. This parameter takes in a classifier, which acts as a calibrated classifier within KNN and also can take a variety of methods as its input like 'sigmoid'. Experimenting with the variety of kernels to choose one that suits well for the data is another critical area of exploration that can provide ways in complex domains like code modeling. In data science kernels are utilized to create a kernel trick which helps transforming linear classifier to solve non-linear problem. Kernel functions are applied on each of the data points with intent of mapping non-linear observations in higher dimensional space, so that the data points can be separated and studied. Abstract Syntax Tree way of representing the code can be modeled in a decision tree-based machine learning approach. This model helps to study the pattern in the code well. In decision tree-based modeling, the homogeneity of the code can be modeled utilizing the Gini index-based metrics. This approach helps to establish the typical pattern in code. Gini index and entropy are opposite, which provides information on how regular the pattern is within the data. In particular, these come handy for decision tree-based modeling to control the extent to which the features can be accommodated with the experiment. Entropy helps to understand the information gained from each feature and select the right features based on their importance. Truncating and pruning the decision trees to be experimented with to get the right depth tree based on the domain's need. Under the bagging modeling approach, there are various ways to test; we may combine multiple models [38]. These different models can take a different set of samples and perform modeling. These sampling can either be with or without the replacement of the data points. Other approaches to creating bagging models are to perform row sampling or column sampling. In row sampling, different cuts of samples are selected, while in the case of column sampling, additional features are chosen. Based on these, multiple models are created, and then the prediction of all these models in the bag is used to make a final overall prediction.

XG Boost is one of the best in class machine learning approaches that provide exemplary performance and speed due to its features like parallelization, out-of-memory computation, and cache optimization. XG Boost's critical aspect is its ability to regularize, auto prune, and treat missing values. CHAID (Chi-square Automatic Interaction Detector) is another approach that helps to identify the best feature from the data. The formula (Eq 7.1) for chi-square is as follows, where y' is the actual value and '$y'$' is the expected value.

$$Chi - square = \sqrt{\frac{(y - y')^2}{y'}} \tag{7.1}$$

With difficulty involved in large-scale source code presentation with supervised learning, unsupervised methods like clustering and segmentation will have a significant role to play in source code modeling. Clustering study of the software source code provides base for many of the source code related use cases like automated coding, recovery of the artifacts, understanding the software systems and establishing the traceability and so on. Assessment of the inter and intracluster homogeneity is a central theme in the clustering. Characteristics of the source code that influence this clustering approach needs to be explored closely. K-Means clustering is one of the methods that can provide an unsupervised format to experiment. The cost function in the case of K-Means clustering can be represented as in Eq (7.2).

$$J = \sum_{i=1}^{n} ||(x_i - \mu_{k(i)}||^2. \tag{7.2}$$

'i' in the equation refer to 'ith' data point. 'k' refers to clusters. $\mu_{k(i)}$ represents the co-ordinate of cluster center for k(i).

Silhouette analysis helps to identify the quality of the cluster formed from the data points. Metrics based on cohesiveness and dissimilarity are used to arrive at the proper cluster formation. Figure 1 shows details of silhouette analysis. Figure 1 highlights that though there is no cohesiveness, one common cluster is shown, which is not appropriate. In the second part of Figure 3, clusters are demonstrated correctly based on their cohesiveness and dissimilarity. The third part of the figure shows two small clusters even though there is cohesiveness within the group. Equation (7.3) denotes the computation of the silhouette metrics.

$$s(i) = \frac{b(i) - a(i)}{max(b(i), a(i))}. \tag{7.3}$$

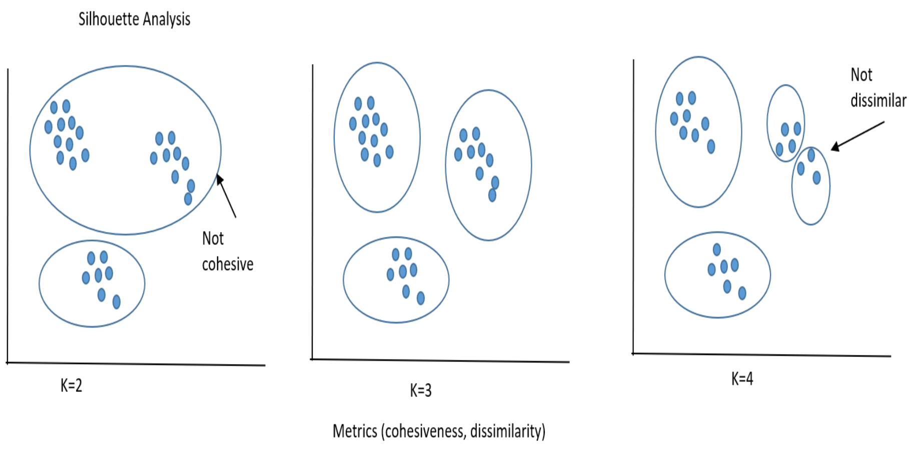

**Figure 1.** Sihouette analysis representation.

a(i)–cohesion: average distance from own cluster b(i)–separation: average distance from neighbor cluster s(i) value of 1 is good Ideally a(i) should be smaller than b(i) 'i' represents individual datapoint.

Hopkins statistics provides a means to assess the randomness of the data in the cluster. Lower the Hopkins value highly random the data, denoting the difficulty of working with that data. Another critical logic is identifying relation, persona, and intent among the features created from the data point. The relation is straightforward, the persona is the characteristics of the data points, and intent is the data distribution's objective. These aspects will provide means to establish a data pattern. Hierarchical clustering helps build the cluster within the data space, and the same can be denoted on a dendrogram to identify the correct number of clusters within data. Since we are dealing with the complex data landscape of software source code, it is essential to experiment with some of the dimensionality reduction techniques like LDA (Latent Dirichlet Allocation), t-SNE (t Distributed Stochastic Neighbor Embedding), and PCA (Principal Component Analysis).

One another aspect that needs attention is the multi-collinearity between the features that are gathered for modeling. This aspect will have to be monitored closely in software source code-related features to understand how each of the features influences overall prediction and influence each other. In the case of multi-collinearity, there are possibilities that the influence of features among themselves will mask the impact on the final prediction. Variance Inflation Factor is one method to keep a check on the multi-collinearity of the features. Simultaneously, machine learning approaches focus on understanding the pattern hidden in the data from the domain under study. The distribution from which the data is coming plays a crucial role in the prediction models' success. The data domain distribution is influenced by factors like the period in which the data generation has happened, the number of changes that have occurred in the domain, the impact of those changes on the data, and the scope of the data. These factors can influence the nature of the data generated in the process, and further, these data hold the inherent pattern in the domain. However, machine learning is about understanding patterns hidden in the process through data, a mathematical representation of the process. Data visualization is an essential aspect of getting a good sense of the data domain. In the case of a larger experiment done in this research, there is a focus on providing visualization capabilities for the end-users to derive knowledge from the data. In the overall system intended to be built, a smart requirements manager that will be available for requirements managers in the software development process, visualization capabilities built on top of the knowledge base created from various modules can be seen [39].

Some visualization capabilities like a word cloud, n-gram analysis-based visualization, and knowledge graph visualizations will be built. The module discussed in this paper is the source to learn the pattern from the software development landscape, particularly from the software's source code. NLP (Natural Language Processing) pipelines must be architected to model the text source data associated with the source code. These data would be the code commit comments, code review, other review comments, testing-related information, and additional textual information captured during the software development process. A study of the source code's time-series pattern will provide helpful information around the possible security vulnerabilities that can creep into the information system shortly. Security vulnerabilities are the primary focus area in this research. In simple terms figuring out the vulnerable code components will be one of the critical explorations [40].

Time series exploration of the source code is influenced by the influence of maintainability of the software over some time. The new updates to software and their impact on security will have to be kept in check. Impact analysis is an essential exercise in software development; a similar thought process must be used in source code modeling. The impact of various phenomena happening during software development has to be accounted for. Exploring the machine learning arena to its critical areas in the source code modeling will be as follows. Regression modeling will help assess the extent of vulnerable codes in the software system, the extent of contribution by these vulnerabilities in inducing the security threats, the number of security issues, and other quantitative aspects that can be modeled. The classification approach in machine learning will be explored to bucket the vulnerable code components into various security concerns. These categories can be established based on the security objective set for the software domain under study [41]. This objective may pop out of the necessary security triad as a starting point: confidentiality, integrity, and availability. These security objectives can also be done by learning from the historical source code management data and other data sources. Clustering is another prominent area of machine learning, where the grouping of clusters will help combine the typical characteristics of the data domain. This data domain visibility in clusters will help devise the plan of action focusing on each cluster rather than a generic plan at a high level. In the case of security vulnerabilities modeling, clustering of the code components will help generate knowledge about the software system vulnerabilities in the categories and treat them according to their nature [42]. Figure

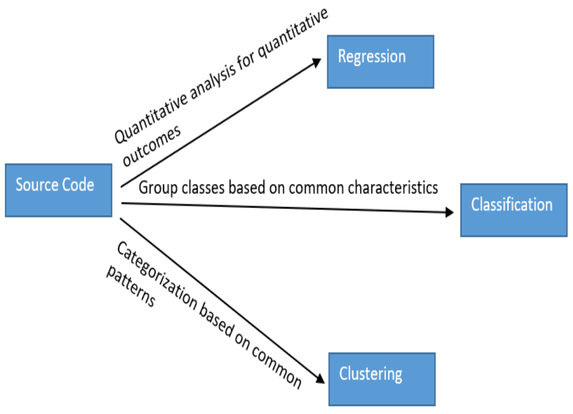
Source Code
Quantitative analysis for quantitative outcomes
Regression
Group classes based on common characteristics
Classification
Categorization based on common patterns
Clustering

**Figure 2.** Machine learning in software source code modeling.

2 represents the machine learning approach and its application for software source code modeling. Machine learning applications in software source code will have to experiment with a semi-supervised learning approach. Most of the experiments will be unsupervised where the work has to be done without labeled data, so that clustering may be a handy area of exploration. Supervised learning ability also can be incorporated if there is the possibility of creating some handmade features and labels. For instance, vulnerable code parts can be labeled and used for training purposes. In the regression arena, metrics-related code components can be derived to perform supervised learning. The mix of these supervised and unsupervised approaches can provide a semi-supervised platform for modeling software source code. The reinforcement learning area can be explored, considering that these semi-supervised learning systems are established. They keep working on the new data that will help calibrate the system as it continues to operate. Once the system has enough data and capabilities of learning, aspects of the reinforcement system can be built into these matured systems of prediction. Figure 3 shows the basic setup for Reinforcement learning. Table 2 provides the details of reinforcement learning variables.

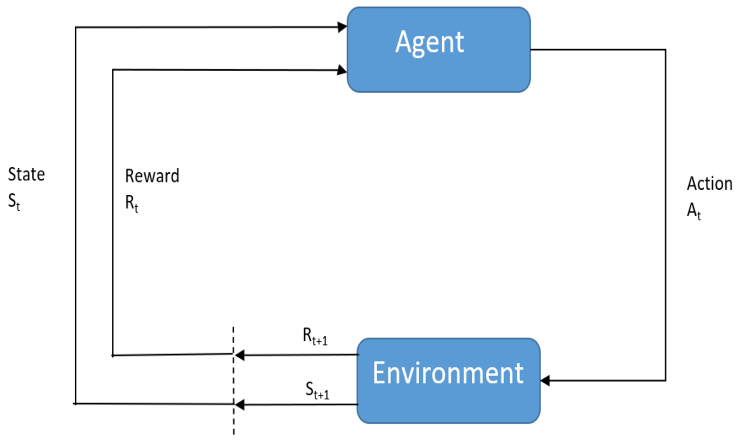

**Figure 3.** Reinforcement learning setup.

**Table 2.** Reinforcement learning variables.

| Reinforceme Learning Variable | Dtescription |
|---|---|
| Environment | World in which agent operate |
| Reward | Environmental feedback |
| State | Situation of agent at a point of time |
| Policy | Agent state and action mapping approach |
| Value | Point to be awarded in future for an agent for its action in particular state |

In the aspiration of making the machine learning models transparent and interpretable, understanding the gradient descent mechanism working underneath machine learning models is essential. Gradient descent helps arrive at a cost function while building a mathematical relationship between all features and their outcome. Gradient descent targets to optimize the cost function so that the best possible connection is constructed to get an adequate level of accuracy for the models. The below Eq (7.4) gives the cost function in gradient descent.

$$C = \sum 1/2(\hat{y} - y)^2 \tag{7.4}$$

The Eq (7.5) below describes what gradient descent does: 'b' is the next position of the optimization function, while 'a' represents his current position. The minus sign refers to the minimization part of gradient descent. The gamma is a waiting factor, and the gradient term ($\Delta f(a)$) is the direction of the steepest descent.

$$b = a - \gamma \Delta f(a) \tag{7.5}$$

To minimize cost-function J(w, b); and reach its local minimum by tweaking its parameters (w and b). Figure 4 shows the horizontal axes representing the parameters (w and b), while the cost function J(w, b) is represented on the vertical axes. Gradient descent is a convex function. Figure 4 shows gradient descent. As part of the classification or clustering approach, the basic idea of modeling will be

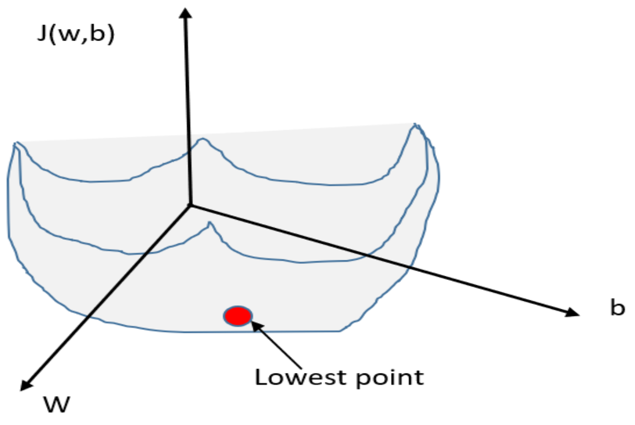

**Figure 4.** Gradient descent.

to structure the data space into families of common characteristics. One of the primary ways to do this is by assessing the distance between the data points. For instance, in the KNN (K-Nearest Neighbor) approach, the data points' similarities occupy the data space, so the distance between data points is critical. Distance between data points can be measured in multiple ways. Euclidean distance is the most commonly used, based on the need of the domain and scenarios other distances like Manhattan, Minkowski, hamming, cosine similarity can be experimented with. Data availability is a most significant challenge; approaches are available to leverage available data to build effective models. K-fold cross-validation is one such handy approach that helps break the available data into blocks of data and use the blocks as a combination of test and training data in various ways to explore the pattern in the data. Imbalanced data set is another challenge in the real world, particularly in security vulnerabilities, because security vulnerabilities are fewer in number in the overall landscape of software development. Apart from accuracy, recall, precision, and F1 score, which is widely used to evaluate the models' performance, there is a need to explore ROC (Receiver Operating Characteristics) and AUC (Area Under Curve) metrics to assess the models effectively. Log-loss metrics also provide more granular insight into how the model works and help make necessary tweaks to the model. Advanced regression modeling like lasso regression and ridge regression will refine the regression use case scenarios of the security vulnerabilities modeling with source code. Lasso regression technique is specialized in using shrinkages, these are the data values that get shrunk towards the mean value or similar central values. In comparison to regular regression methods lasso uses the regularization approach which helps in more accurate predictions.

## 8. Analysis of current research

Dynamic variation in the threat and vulnerabilities in the industry will be a challenge to handle. Some of the threats may not frequently impact, but when they do occur, they have a significant effect; making sure that area is addressed is crucial. Machine learning models need periodic calibration to keep them relevant at any point in time. Changing landscape of technology will also add to the challenge of keeping up with the security needs. Integrating multiple models will be a challenge as the proposed architecture intends to provide a central system for decision-making. Data associated with the process will be sensitive, and access to data would be a challenge; utmost care may have to be taken while handling and processing the sensitive data. The Explainability of the models will be a challenge to handle, as a need to convince the practitioners of the recommendations provided by the architecture. Extension of the work to other domains would involve more profound work. Exploration of the naturalness of the software source code has provided better insights on leveraging the patterns hidden in the source code to improve coding practices. Statistical modeling in software development also provides opportunity to address productivity and quality related issues in software development. Using data for decision making process in the software development companies can be facilitated with multiple thought processes discussed in the paper.

## 9. Conclusions

In this work, the focus has been to establish context around the need for statistical modeling in software development. Many of the concerns faced by the software industries are explored. Correlations are drawn for the software development processes and their challenges and sources of data that they hold. Some of the change management mechanisms in the software development and opportunities that they present are covered. Assessment of research gaps firms up on the critical challenges in the software development processes and the areas that need attention. Some of the challenges faced by software development companies are related to rapid developments happening in technology landscape, increasing customer demands with limited time to market, conflicts across various stakeholders of the software development teams and many other. Paper has explored the opportunities that are available in software development practices where data science can be leveraged. Research gaps involved in statistical modeling of software development practices are the key theme of discussion. Representation of the software source code to understand the hidden patterns are explored and further extended to check the applications of machine learning and its benefits. Software program representation is explored as this helps to build the context for software source code modeling. Understanding this representation will help make the data models correctly understand the software program structure; this helps mature these models and align them to the problem areas' real needs. Some of the experiences with the utilization of machine learning are discussed in light of their applicability for software modeling and considering the source code being the center of discussion. There is a need for focused research on exploring the practical data science or applied data science methods to bring out the best of both software development and data science. These explorations will also help companies to enhance the data knowledge of the technical team. It helps bring confidence among the leadership team to consider data an essential component of their strategic road map. All this will enhance the pace of data science applications in fields like software engineering, where applications can be increased to the next level.

**Acknowledgements**

There is no funding available for this research work.

**Conflict of interest**

The authors of this manuscript declared that they do not have any conflict of interest.

**References**

1. A. Ahmad, C. Feng, M. Khan, A. Khan, A. Ullah, S. Nazir, et al., A systematic literature review on using machine learning algorithms for software requirements identification on stack overflow, *Secur. Commun. Networks*, 2020.
2. R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, et al., Automated vulnerability detection in source code using deep representation learning, in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, IEEE, (2018), 757–762.
3. H. El-Hadary, S. El-Kassas, Capturing security requirements for software systems, *J. Adv. Res.*, **5** (2014), 463–472. https://doi.org/10.1016/j.jare.2014.03.001
4. K. Chen G. S. Corrado, T. Mikolov, I. Sutskever, J. Dean, Distributed representations of words and phrases and their compositionality, *Adv. Neural Inf. Process. Syst.*, (2013), 3111–3119.
5. Y. Kim, Convolutional neural networks for sentence classification, in *Proc. 2014 Conf. Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, (2014), 1746–1751. https://doi.org/10.3115/v1/D14-1181
6. Y. Pang, X. Xue, A. S. Namin, Predicting vulnerable software components through n-gram analysis and statistical feature selection, in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, IEEE, (2015), 543–548. https://doi.org/10.1109/ICMLA.2015.99
7. S. Bettaieb, S. Y. Shin, M. Sabetzadeh, L. Briand, G. Nou, M. Garceau, Decision support for security-control identification using machine learning, in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Springer, (2019), 3–20.

8. R. Malhotra, A. Chug, A. Hayrapetian, R. Raje, Analyzing and evaluating security features in software requirements, in *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, IEEE, (2016), 26–30.

9. Y. Pang, X. Xue, A. S. Namin, Feature selections for effectively localizing faulty events in gui applications, in *2014 13th International Conference on Machine Learning and Applications*, IEEE, (2014), 306–311.

10. B. Murphy, P. Morrison, K. Herzig, L. Williams, Challenges with applying vulnerability prediction models, in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, ACM–Association for Computing Machinery, 2015.

11. Y. Pang, X. Xue, A. S. Namin, Trimming test suites with coincidentally correct test cases for enhancing fault localizations, in *2014 IEEE 38th Annual Computer Software and Applications Conference*, IEEE, (2014), 239–244.

12. X. Xue, Y. Pang, A. S. Namin, Identifying effective test cases through k-means clustering for enhancing regression testing, in *2013 12th International Conference on Machine Learning and Applications*, IEEE, **2** (2013), 78-–83.

13. C. Catal, A. Akbulut, E. Ekenoglu, M. Alemdaroglu, Development of a software vulnerability prediction web service based on artificial neural networks, in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, Cham, (2017), 59–67.

14. I. Medeiros, N. Neves, M. Correia, Dekant: a static analysis tool that learns to detect web application vulnerabilities, in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, (2016), 1–11,

15. L. K. Shar, L. C. Briand, H. B. K. Tan, Web application vulnerability prediction using hybrid program analysis and machine learning, *IEEE Trans. Dependable Secure Comput.*, **12** (2014), 688–707.

16. Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, S. Li, Combining software metrics and text features for vulnerable file prediction, in *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE, (2015), 40–49.

17. S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in *Proceedings of the 14th ACM conference on Computer and communications security* , (2007), 529–540,

18. N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in *Proceedings of the 28th international conference on Software engineering*, (2006), 452–461.

19. Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, C. Zhai, Have things changed now? an empirical study of bug characteristics in modern open source software, in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, (2006), 25–33,

20. V. H. Nguyen, L. M. S. Tran, Predicting vulnerable software components with dependency graphs, in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, (2010), 1–8.

21. C. Catal, B. Diri, A systematic review of software fault prediction studies, *Expert Syst. Appl.*, **36** (2009), 7346–7354. https://doi.org/10.1016/j.eswa.2008.10.027

22. A. Hayrapetian, R. Raje, Empirically analyzing and evaluating security features in software requirements, in *Proceedings of the 11th Innovations in Software Engineering Conference*, (2018), 1–11. https://doi.org/10.1145/3172871.3172879

23. T. Li, Identifying security requirements based on linguistic analysis and machine learning, in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, (2017), 388–397.

24. R. Jindal, R. Malhotra, A. Jain, Automated classification of security requirements, in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, (2016), 2027–2033. https://doi.org/10.1109/ICACCI.2016.7732349

25. J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu, A novel neural source code representation based on abstract syntax tree, in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE/ACM, (2019), 783–794.

26. U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning distributed representations of code, in *Proceedings of the ACM on Programming Languages*, **3** (2019), 1–29. https://doi.org/10.1145/3341688

27. Z. Mushtaq, G. Rasool, B. Shehzad, Multilingual source code analysis: A systematic literature review, *IEEE Access*, **5** (2017), 11307–11336. https://doi.org/10.1109/ACCESS.2017.2710421

28. H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, A. Ghose, Automatic feature learning for predicting vulnerable software components, *IEEE Trans. Software Eng.*, **47** (2021), 67–85. https://doi.org/10.1109/TSE.2018.2881961

29. X. Sun, X. Liu, B. Li, Y. Duan, H. Yang, J. Hu, Exploring topic models in software engineering data analysis: A survey, in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, IEEE/ACIS, (2016), 357–362.

30. M. Allamanis, E. T. Barr, P. Devanbu, C. Sutton, A survey of machine learning for big code and naturalness, *ACM Comput. Surv. (CSUR)*, **51** (2018), 1–37.

31. A. Hindle, E. T. Barr, M. Gabel, Z. Su, Z. Su, P. Devanbu, On the naturalness of software, *Commun. ACM*, **59** (2016), 122–131. https://doi.org/10.1145/2902362

32. J. Tabassum, M. Maddela, W. Xu, A. Ritter, Code and named entity recognition in stackoverflow, preprint, arXiv:2005.01634.

33. C. Ling, Z. Lin, Y. Zou, B. Xie, Adaptive deep code search, in *Proceedings of the 28th International Conference on Program Comprehension*, (2020), 48–59. https://doi.org/10.1145/3387904.3389278

34. K. Goseva-Popstojanova, J. Tyo, Identification of security related bug reports via text mining using supervised and unsupervised classification, in *2018 IEEE International conference on software quality, reliability and security (QRS)*, IEEE, (2018), 344–355.

35. T. D. Oyetoyan, P. Morrison, An improved text classification modelling approach to identify security messages in heterogeneous projects, *Software Qual. J.*, **1** (2021). https://doi.org/10.1007/s11219-020-09546-7

36. L. B. Othmane, G. Chehrazi, E. Bodden, P. Tsalovski, A. D. Brucker, Time for addressing software security issues: Prediction models and impacting factors. data science and engineering, *Data Sci. Eng.*, **2** (2017), 107–124.

37. S. Xu, Y. Xiong, Automatic generation of pseudocode with attention Seq2seq model, in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, (2018), 711–712.

38. Q. L. Nguyen, Non-functional requirements analysis modeling for software product lines, in *2009 ICSE Workshop on Modeling in Software Engineering*, (2009), 56–61. https://doi.org/10.1109/MISE.2009.5069898

39. R. R. Althar, D. Samanta, D. Konar, S. Bhattacharyya, *Software Source Code*, De Gruyter, July 2021.

40. R. R. Althar, D. Samanta, Application of machine intelligence-based knowledge graphs for software engineering, in *Methodologies and Applications of Computational Statistics for Machine Intelligence*, (2021), 186–202.

41. R. R. Althar, D. Samanta, The realist approach for evaluation of computational intelligence in software engineering, *Innovations Syst. Software Eng.*, **17** (2021), 17–27. https://doi.org/10.1007/s11334-020-00383-2

42. R. R. Althar, D. Samanta, Computational Statistics of Data Science for Secured Software Engineering, in *Methodologies and Applications of Computational Statistics for Machine Intelligence*, (2021), 81–96.