brillio

Improving Predictive Code Quality Using Machine Learning

Introduction

Software Engineering has become an important aspect of driving product-led competitive advantage in the market for organizations.

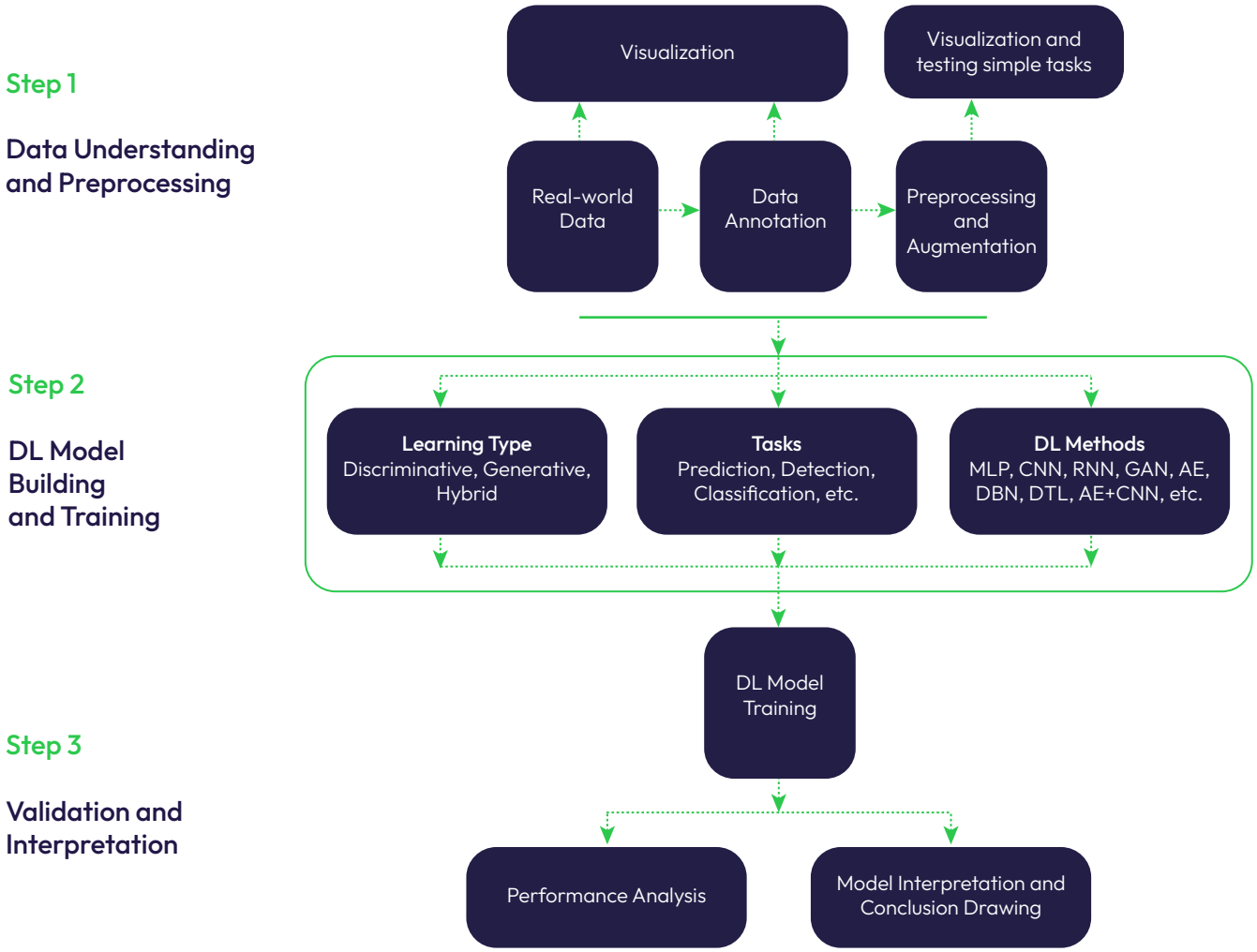
However, poor software quality reduces customer satisfaction. High-quality software, on the contrary, can prevent the need for repairs and reworks by more than 70%. This also reduces the associated application maintenance and support costs.

Over the years, we have witnessed significant advancements in Machine Learning (ML) and Deep Learning (DL) techniques, specifically in the domain of image, text, and speech processing. These advancements, coupled with readily available open-source code, its associated artifacts, and accelerated hardware, have facilitated the use of ML and DL techniques to address software engineering problems.

This is done through quality matrices, which are dynamically collected and shape the developer model. These techniques can be applied for software testing, source code quality analysis, program synthesis, code completion, and vulnerabilities analysis that involves source code analysis. Source code analysis involves tasks that take the

source code as input, process it, and/or produce source code as output. Source code representation, code quality analysis, testing, code summarization, and program synthesis are applications that involve source code analysis.

The below diagram illustrates the standard steps **as per source code analysis.**



> **Source code defects correction annual cost is around US $312 billion.**
>
> -Cambridge University

Challenges Without Predictive Code Quality

- **The Decline in Code Health:** A common theme across software engineering is that seldomly some property of the current code causes a decline in quality which can be identified through ML.
- **The Plethora of Tools:** Each code analysis tool in the market is designed with many redundant features, and choosing the right tool is a complex task.
- **Traditional Software Development:** Traditional software development and the development of ML systems are inherently different. Phases of ML development are very exploratory in nature and highly domain and problem dependent.
- **Effective Feature Engineering:** Features represent the problem-specific knowledge in the pieces extracted from data; the effectiveness is low without predictive analysis
- **Metrics to Measure:** There are rarely any metrics to measure the quality and take preventive decisions
- **Code Quality Trends:** Scope for improvements without code quality trends

Solution Overview

There are several ML techniques that are used for every category. It is evident from the reference list that supports vector machine (SVM), and decision tree (DT) is the most frequently employed ML techniques.

On the DL side, the recurrent neural network (RNN) family (including long short-term memory (LSTM) and gated recurrent unit(GRU)) are the most deployed.

In the following section, we will summarize the method and then deep dive into each category and sub-category while breaking down the entire workflow of a code analysis task into fine-grained steps.

Although there are different workflows to achieve the predictive metrics with software engineering, two important fundamental types of workflows are described in this article which help in providing predictive code quality metrics are as below:

Code Analysis Workflow Defect Prediction Workflow

Most used ML techniques

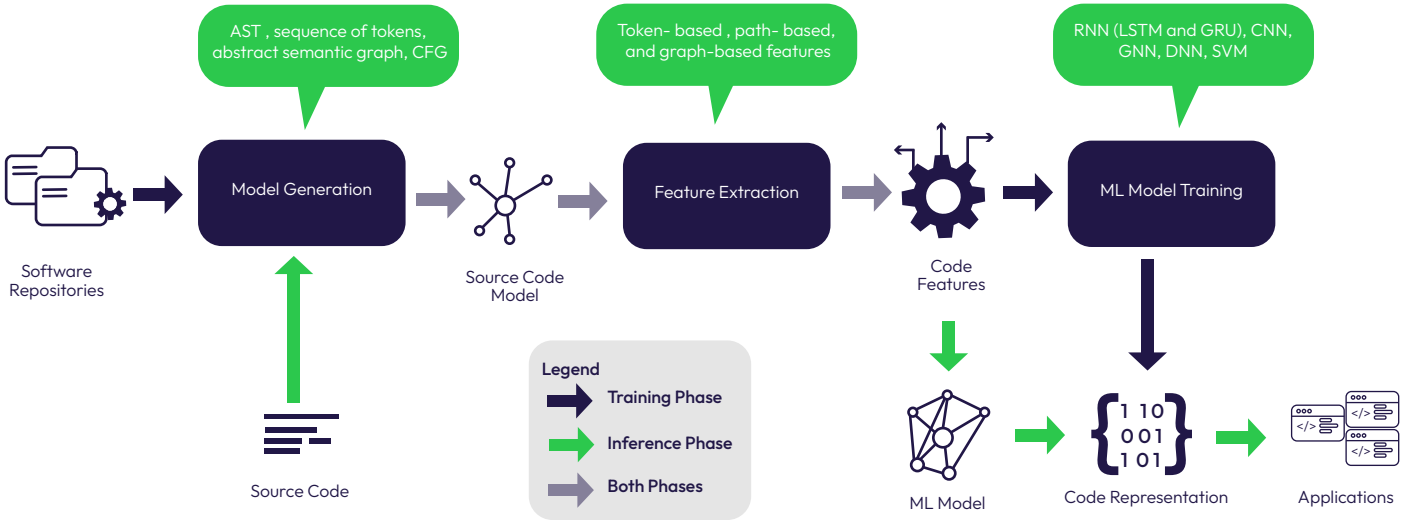
RNN	Recurrent Neural Network
DNN	Deep Neural Network
LSTM	Long Short-Term Memory
SVM	Support Vector Machine
CNN	Convolusion Neural Network
GNN	Graph Neural Network
ANN	Artificial Neural Network

Software metrics allow measurement and evaluation, controlling the software product and processes improvement. They are essential resources to improve quality and cost control during software development.

Code Analysis Workflow: Solution Approach

Raw source code cannot be fed directly to a DL model. Code representation is the fundamental activity to make source code compatible with DL models by preparing a numerical representation of the code to further solve a specific software engineering task.

Most techniques extensively utilize syntax, structure, and semantics. The activity transforms source code into a numerical representation making it easier to further use the code by ML models to solve specific tasks such as code pattern identification, method name prediction, and comment classification.



The above diagram provides an overview of a typical pipeline associated with code representation. In the training phase, many repositories are processed to train a model, which is then used in the inference phase.

Source code is pre-processed to extract a source code model (such as an abstract syntax tree (AST) or a sequence of tokens), which is fed into a feature extractor responsible to mine the necessary features (for instance, ASTpaths and tree-based embeddings).

Then, an ML model is trained using the extracted features. The model produces a numerical (i.e., a vector) representation that can be used further for specific software engineering applications such as defect prediction, vulnerability detection, and code smells detection.

The different stages are as follows-

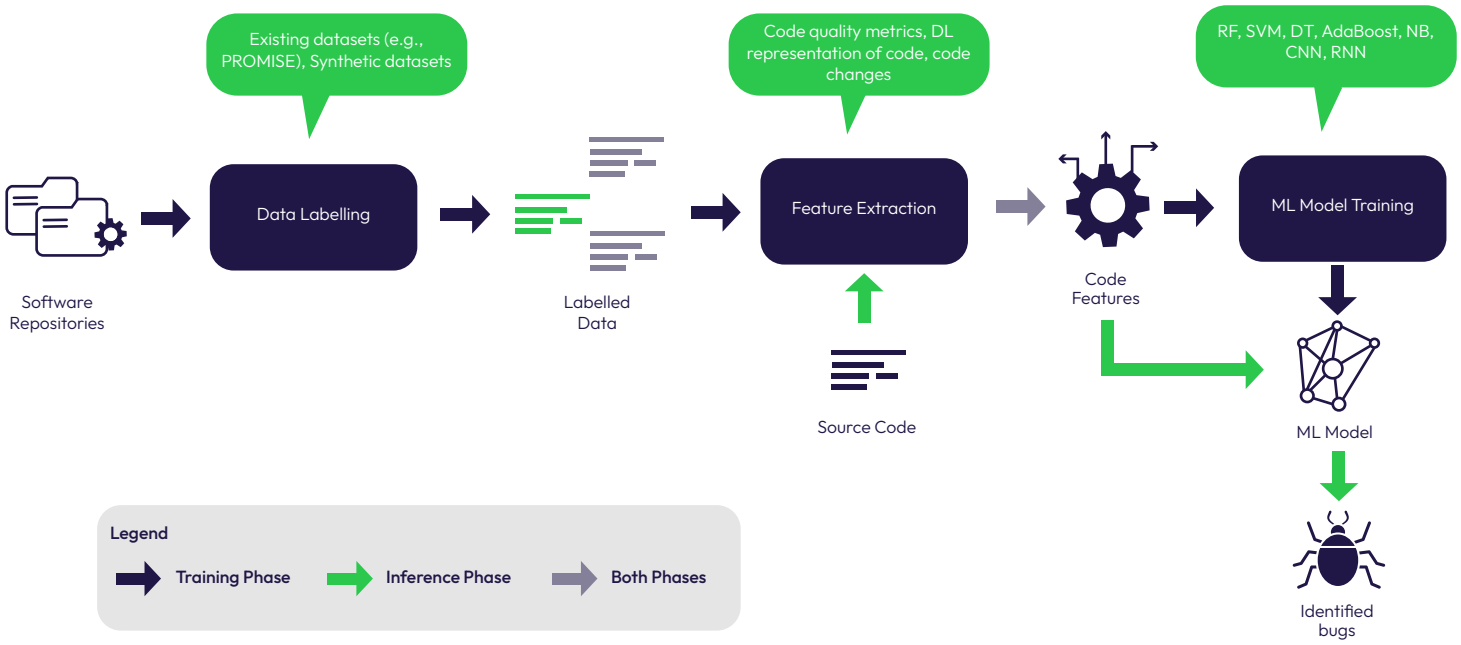
- **Model generation:** Code representation efforts start with preparing a source code model. Most of the research generates AST.
- **Feature extraction:** Relevant features need to be extracted from the prepared source code model for further processing. Another set uses the graph-based feature and uses DL to learn features automatically.
- **ML model training:** Many of the studies rely on the RNN-based DL model. A typical output of a code representation technique is the vector representation of the source code. The exact form of the output vector may differ based on the adopted mechanism. Often, the code vectors are application-specific, depending upon the nature of the features extracted and the training mechanism.

Defect Prediction Workflow: Solution Approach

The below diagram depicts a common pipeline used to train a defect prediction model. The first step of this process is to identify the positive and negative samples from a dataset where samples could be a type of source code entity such as classes, modules, files, and methods.

Next, features are extracted from the source code and fed into an ML model for training.

Finally, the trained model can classify different code snippets as buggy or benign based on the encoded knowledge.



- **Data Labeling:** To train an ML model for predicting defects in source code, a labeled dataset is required. Most of the research recommends using the PROMISE dataset, while others use the Continuous Integration (CI) dataset and synthetic dataset.
- **Feature Extraction:** The most common features to train a defect prediction model are the source code metrics such as Lines of Code, Number of Children, Coupling Between Objects, and Cyclomatic Complexity.

  Other methods are Principal Component Analysis (PCA)—to limits the number of features and Sequential Forward Search (SFS) to extract relevant source code metrics.

  The method named Transfer Learning Code Vectorizer generates features from source code by using a pre-trained code representation dl model. Another approach for detecting defects is capturing the syntax and multiple levels of semantics in the source code.

  A tree-based LSTM model by using source code files as feature vectors. Subsequently, the trained model receives an AST as input and predicts if the file is clear from bugs or not. Specifically, DTL-DP visualizes the programs as images and extracts feature out of them by using a self-attention mechanism.

- **ML Model Training:** To train the model, most of the research used traditional ML algorithms such as Decision Tree, Random Forest, Support Vector Machine, and AdaBoost. In contrast to the above studies, researchers used DL models such as CNN and RNN-based models for defect prediction. Moreover, by using DL approaches, authors achieved improved accuracy for defect prediction, and they pointed out bugs in real-world applications.

> **Although the potential for success is enormous, delivering business impact from AI initiatives takes much longer than anticipated.**
>
> -Chirag Dekate,
> Senior Director Analyst, Gartner

Benefits

Any industry can use predictive code quality to reduce risks, optimize operations and increase revenue. Some of the standard benefits and references from various industries that had the benefit with other similar solutions are:

Overcome Vulnerabilities: Using these models, developers can decide the design patterns to ensure that the source code complies with best practice standards and to discover vulnerabilities such as race conditions, malware, memory leaks, buffer overflows, format string exploits, and security. Combining multiple analytics methods can improve pattern detection and prevent criminal behavior.

"Commonwealth Bank uses ML to predict the likelihood of fraud activity for any given transaction before it is authorized – within 40 milliseconds of the transaction initiation".

Zero Defects: These workflows can be leveraged:

- Processing data which allows interpreting the prediction result and enables quality-based process-integrated decision support.
- **Qualitative Metrics:** Predictive metrics are more advanced and one of the critical keys which can allow measurement and evaluation, controlling the software product and processes improvement.
- **Predictive Delivery:** These methods can provide more accurate estimates of the time, effort, and cost associated with software development. It can help developers and development managers understand whether teams can meet their deadlines, improve process flows, identify the most common issues through metrics.
- **Visualizations For Attention:** Most industries use predictive models to improve service and performance, detect and prevent fraud, and to better understand consumer behavior. Also "Decision Tree" models help to predict the most common journeys among customers and prospects for a specific timeframe.

"Staples gained customer insight by analyzing behavior, providing a complete picture of their customers, and realizing a 137% ROI".

Below is a quick navigation guide for any organization that wishes to maximize the benefits of adopting predictive code quality-

- **Determine What You're Trying to Achieve:** Before you even write a line of code, work out exactly why you are writing that code. What ML technology is the best suitable for predicting your code quality. What outputs are you hoping to produce and what are you going to do with them once you get them? Visualize the automated code reviews and analysis.

- **Improving Your UX/UI:** Software Engineering and development teams contribute more towards UI/UX, which is crucial on the business front. Predictive code quality can help maintain high-end user satisfaction and can remediate any defects. Predictive code quality mainly showcases the practices and coding style an individual or a team follows. While a good quality code can be customized to your users' needs quite easily, a low-quality code can lead to frequent bugs and errors, degrading user satisfaction in the long run.

Conclusion

Detecting security vulnerabilities in software before they are exploited has been a challenging problem for decades. Traditional code analysis methods have been proposed but are often ineffective and inefficient. This paper introduces an approach to developing a predictive code quality framework using ML techniques. Moreover, this paper describes mostly used workflows for predictive code quality at a high level. Still surveying to find the most suitable ML techniques for advanced software engineering. However, most existing static code analysis tools and style checkers need to parse the code and often even link it to fully apply their analysis.

References

https://drops.dagstuhl.de/opus/volltexte/2021/14431/pdf/OASIcs-SLATE-2021-14.pdf
https://ojs.bibsys.no/index.php/NIK/article/view/26/22

Appendix

List of ML techniques widely used for source code analysis:

ABOUT BRILLIO

At Brillio, our customers are at the heart of everything we do. We were founded on the philosophy that to be great at something, you need to be unreasonably focused. That's why we are relentless about delivering the technology-enabled solutions our customers need to thrive in today's digital economy. Simply put, we help our customers accelerate what matters to their business by leveraging our expertise in agile engineering to bring human-centric products to market at warp speed. Born in the digital age, we embrace the four superpowers of technology, enabling our customers to not only improve their current performance but to rethink their business in entirely new ways. Headquartered in Silicon Valley, Brillio has exceptional employees worldwide and is trusted by hundreds of Fortune 2000 organizations across the globe.

https://www.brillio.com/
Contact Us: info@brillio.com

		Category											
		Code completion	Code representation	Code review	Code search	Dataset mining	Program comprehension	Program synthesis	Quality assessment	Refactoring	Testing	Vulnerability analysis	Grand Total
AB	AdaBoost								1	1	2	1	5
AE	Autoencoder								1				1
ANN	Artifical Neural Network								1	2	6		9
ARM	Association Rule Mining							1					1
BERT	BERT		1					1					2
Bi-GRU	Bidirectional GRU		1										1
Bi-LSTM	Bi-LSTM						4	1				1	6
Bi-RNN	Bidirectional RNN						1						1
BiNN	Bilateral Nerual Network								1				1
BMN	Best Matching Neighbours	1											1
BN	Bayes Net	1											1
BP-ANN	Backpropagation ANN											1	1
BR	Binary Relevance								1				1
CART	Classification and Regression Trees								1				1
CNN	Convolution Neural Network		3	1	1				2		2	1	10
Code2Vec	Code2Vec		4				1						5
CoForest-RF	Co-Forest Random Forrest										1		1
CSC	Cost-Sensitive Classifier								2				2
DBN	Deep Belief Network										1		1
DDQN	Double Deep Q-Networks										1		1
DNN	Deep Neural Network		1		1		1	2	2			1	8
Doc2Vec	Doc2Vec							1				2	3
DT	Decision Tree	1					3		10		9	6	29
ELM	Extreme Learning Machine								1		1		2
EN-DE	Encoder-Decoder	1	1				9	9					20
FR-CNN	Faster R-CNN										1		1
GAN	Generative Adversarial Network							1					1
GB	Gradient Boosting						1				1		2
GBT	Gradient boosted trees							1					1
GD	Gradient Descent							1					1
GED	Gaussian Encoder-Decoder							1					1
GEP	Gene Expression Programming										1		1
GGNN	Gated Graph Neural Network							2					2
GINN	Graph Interval Neural Network		1										1
Glove	Global Vectors for Word Representation		1										1
GNN	Graph Neural Network		3										3
GPT-C	Generative Pretrained Transformer for Code							1					1
GRU	Gated Recurrent Unit	1	1				4						6
HAN	Hierarchical Attention Network		1				1						2
HC	Hierarchical Clustering							1					1
HMM	Hidden Markov Model	2											2
KM	KMeans								1	1			2
KNN	K Nearest Neighbours			1			2					1	4
LDA	Linear Discriminant Analysis						1						1
LLR	Logistic Linear Regression										1		1
LOG	Logistic regression					1		2	2	2	1		8
LR	Linear Regression									1	7	1	9
LSTM	Long Short Term Memory	6			1		8	8			4	1	28
MLP	Multi Level Perceptron										1		1
MMR	Maximal Marginal Relevance						1						1
MNN	Memory Neural Network	1											1
MTN	Modular Tree-structured RNN	1	1										2
NB	Naïve Bayes						1	1	8	1	3	1	15
NLM	Neural Language Model						1						1
NMT	Neural Machine Translation	1						5				1	7
NN	Neural Network							1	1				2
Node2Vec	Node2Vec										1		1
ResNet	Residual Neural Network							1					1
RF	Random Forrest	1						2	7		3	5	18
Ripper	Ripper								1				1
RL	Reinforcement Learning							1					1
RNN	Recurrent Neural Network	2	1		1		3	5	1		1		14
SA	Simulated Annealing										1		1
Seq2Seq	Sequence-to-Sequence										1		1
SLP	Single Layer Perceptor										1		1
SMT	Statistical Machine Translation							1					1
SVM	Support Vector Machine		1				4	2	5	2	9	10	33
SVR	Support Vector Regression								1				1
TF	Transformer	2					2						4
VSL	Version Space Learning										1		1
Word2Vec	Word2Vec							1			1		2