On the Defect Prediction for Large Scale Software Systems – From Defect Density to Machine Learning

Satya Pradhan
Cisco Systems, Inc.
San Jose, CA, USA
satpradh@cisco.com

Venky Nanniyur
Cisco Systems, Inc.
San Jose, CA, USA
vnanniyu@cisco.com

Pavan K. Vissapragada
Cisco Systems, Inc.
San Jose, CA, USA
pavissap@cisco.com

Abstract—As the software industry transitions to software-as-a-service (SAAS) model, there has been tremendous competitive pressure on companies to improve software quality at a much faster rate than before. The software defect prediction (SDP) plays an important role in this effort by enabling predictive quality management during the entire software development lifecycle (SDLC). The SDP has traditionally used defect density and other parametric models. However, recent advances in machine learning and artificial intelligence (ML/AI) have created a renewed interest in ML-based defect prediction among academic researchers and industry practitioners. Published studies on this subject have focused on two areas, i.e. model attributes and ML algorithms, to develop SDP models for small to medium sized software (mostly opensource). However, as we present in this paper, ML-based SDP for large scale software with hundreds of millions of lines of code (LOC) needs to address challenges in additional areas called "Data Definition" and "SDP Lifecycle." We have proposed solutions for these challenges and used the example of a large-scale software (IOS-XE) developed by Cisco Systems to show the validity of our solutions.

Keywords—*Software defect prediction, software quality, software quality analytics, machine learning, large scale software*

I. Introduction

With the advent of cloud technologies and the rapid transition from traditional license-based software to the software-as-a-service (SaaS) model, customer needs are changing at a much faster rate than before [1]. Companies don't have the luxury of validating software in customer labs for months before deployment in live environments. Today's cloud environment requires services to be deployed directly to the cloud, and customers expect only hours of turnaround time for critical software fixes. Customers also demand shorter deployment cycles for traditional on-premise (aka on-prem) deployments. As a result, development teams are challenged to achieve much higher quality before releasing software to customers. At the heart of this challenge is the Software Defect Prediction (SDP) that enables software development teams to take data-driven, proactive actions to improve quality before delivering the software to end users. This reduces software bugs in the customer environment, takes fewer cycles of deployment testing, and improves deployment time.

The software defect prediction has received considerable attention over the years. Different SDP models have been developed using quality attributes including code metrics (e.g., lines of code, complexity), process metrics (e.g., number of changes, recent activity), or defect metrics (e.g., bug backlog, defect arrival rate, disposal rate) [2]. A detailed overview of different parametric techniques developed over the years have been described in [2,3,4]. With the advances in machine learning (ML) and artificial intelligence (AI), these technologies have been used extensively to predict defects for opensource software [5-8]. The objectives of these studies were to find the best set of quality attributes and the ML algorithm that provides the lowest prediction error for a given dataset.

The datasets used in ML-based SDP studies used observations that include metrics derived for source code files or software components [5-8]. Defining observations in the dataset is a relatively straight forward task for small and medium sized software (less than few millions of LOC). However, it is quite challenging for large commercial software with hundreds of millions of LOC. An example of such large-scale system is Cisco's IOS-XE software that consists of 2200+ software components and tens of thousands of source files. This software supports many product families and hundreds of products for routing, switching, wireless, and network management portfolios. New IOS-XE functionality is released to customers on a regular basis every four months. To get the dataset for developing an SDP model for such large-scale software, we could have used the source files, components, products, product families, or releases as observations. Given so many options, the right selection for the observation is one of the most important aspects of the ML-based SDP. We call this the "Data Definition" challenge.

Furthermore, the SDP models in open literature use information available before the software is released to users and predict the number of defects that could be found at the customer site within a 6- or 12-month period [2]. The results of defect prediction just before the release can be used for evaluating release readiness. But it is not enough to manage quality during the entire software development life cycle (SDLC), which typically takes 4-8 months for large-scale commercial software. It is essential to have continuous quality management to reduce the software defect management cost [15] and deliver best-in-class software [16]. Therefore, the SDP needs to provide prediction models that enable continuous quality management. We call this the "SDP Lifecycle" challenge.

The software defect prediction for large-scale software should include four major components:

(a) Data definition
(b) Quality attributes selection
(c) ML algorithm
(d) SDP lifecycle

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00056

Based on our knowledge, all the studies published in open literature focus on the selection of quality attributes and ML algorithms (b and c). None of them address the challenges associated with the data definition and SDP lifecycle (a and d). As shown in this paper, to develop an accurate and useful defect prediction model for large scale software, one must consider all four aspects.

This paper is organized into several sections. Related work on defect prediction using machine learning is presented in Section II. This is followed by detailed descriptions of the SDP challenges and proposed solutions for large-scale systems in Section III. The SDP using the classical defect-density method and ML algorithms is presented in Sections IV and V, respectively. Section VI summarizes the results for the unique challenges posed by large-scale software. Threats to the validity of our work are presented in Section VII. Finally, summary and future work are described in Section VIII.

II. Related Work

One of the focus areas in ML-based SDP is to find the best machine learning algorithm that gives the lowest prediction error. As shown in different studies, the ML algorithm that gives the best SDP model depends on the dataset [5-8]. The work by Hammouri, et al. studied the performance accuracy and capability of three supervised ML algorithms – Naïve Bayes, Artificial Neural Networks, and Decision Tree [5]. The effectiveness of the different algorithms was compared using accuracy, precision (positive predictive value), recall (true positive rate or sensitivity), F-measure, and RMSE (root mean square error). In the datasets used in [5], the decision tree classifier gave better accuracy as compared to other methods. The study in [6] used software quality metrics to predict defects using Bayesian Network, Random Forest, Support Vector Machine, and Decision Tree. For four datasets used by the authors, it was concluded that Support Vector Machine is a better model when compared to other methods. Similarly, the work in [7] compared four different models – Naïve Bayes, Neural Networks, Associated Rules and Decision Tree – and found Naïve Bayes to be a better algorithm. More examples on the use of ML in SDP are presented in [8].

Selection of the best ML algorithm is very important for developing a defect prediction model. However, the choice of the correct attributes or independent variables in the dataset is equally important for developing SDP model using parametric as well as ML-based statistical methods. The study by Chou provides a detailed description of different static code and object-oriented metrics used for defect prediction [9]. The static code metrics include LOC, Halstead metrics, and McCabe cyclomatic complexity. The object-oriented metrics include Chidamber & Kemerer (CK) metrics suite and Metrics for Object-Oriented Design (MOOD) [34]. The study by D'Ambros et al. reviewed an extensive body of work on metrics for defect prediction and grouped them in terms of process metrics, previous defects, source code metrics, and metrics measuring entropy of changes [2].

The study by Singh and Chug used several common metrics such as LOC, object-oriented metrics (cohesion, coupling, and inheritance), and hybrid metrics, which are a combination of object oriented and procedural metrics [10]. Another study on ML-based SDP determined that the most effective metrics for defect prediction are Response for Class (ROC), Line of code (LOC) and Lack of Coding Quality (LOCQ) [11]. A detailed analysis of different metrics for effective defect prediction is presented in [5]. As shown in various studies, the ML algorithm and quality attributes for developing the best SDP model depend on the dataset.

III. Challenges for Large Scale Systems

There are many large-scale software solutions in today's industry that contain hundreds of millions of lines of code (LOC). Cisco IOS, Microsoft Windows, Microsoft Office, Oracle, and Apple iOS are examples of widely used large-scale software. The software used in this study is the IOS-XE software developed by Cisco Systems, Inc. It consists of 200+ millions LOC and 2200+ software components supporting 180+ products and 50+ product families. New functionalities (aka features) are developed by a global team of 3000+ engineers and released to customers every 4 months. There are 300+ change sets (new features or bug fixes) added to this software every day. Images of the software are built, and automated testing is performed every day to maintain the health of the source code repository. Depending on the size and complexity, it takes between 4 and 8 months to design, code, test, and release the software to customers. Many different tests (e.g. unit, functional, regression, system integration, solution, security, performance, scale, and user experience, etc.) are performed before software is delivered to customers. The organization responsible for IOS-XE is divided into several subgroups for different product portfolios (e.g. routing, switching, wireless, industrial IoT, and network management). Each subgroup consists of several hundred smaller teams consisting of 5-15 engineers. Close to 20% of the teams use the agile scrum for development. The remaining teams use a continuous integration approach. Sometimes teams use a waterfall approach for developing large technology infrastructure. This section presents our approach to address the challenges associated with the data definition and SDP lifecycle challenges for large, complex software.

A. Data Definition

In general, a data point consists of three components:

- Data Element is the entity on which data is collected. A software component is an example of a data element.
- Variable is a characteristic of interest for the data element. For example, "number of open defects" is a variable that represents a characteristic of a software component.
- Observation is a measurement collected for a particular data element. Measuring the number of open defects for a component on a particular date is an observation.

For many ML applications, the choice of the data element comes from the very definition of the problem. For example, a customer is a data element for an ML model that predicts if a customer will default on credit card payments. Similarly, a defect is a data element for an ML model to predict if a defect found in a test duplicates another previously known defect. However, it is not a straightforward task to define the data element for defect prediction. Most of the published work on

SDP uses software modules as the data element. Software modules are typically an independent and interchangeable part of the code that contains everything necessary to execute only one aspect of a desired functionality [17]. In our experience, companies typically follow this definition as a guideline and define software modules (aka components) that makes it easier to organize and manage source code.

Depending on the way source code is organized, there are many possible approaches to define the data element for the SDP. Source Files (SF) are the basic level of organization in the source code management (SCM) system. Source Directory (SD) is the next level in SCM. Several directories are then used to form a logical group called a software component. Several software components are grouped together as a product to support a specific use case or hardware platform. For example, ISR-4221 (Integrated Services Routers-4221) is an example of a product developed by Cisco to be used as a branch office router. Software for multiple related products are grouped together to form a product family (PF). For example, the ISR4000 family of routers includes products like ISR-4221, ISR-4321, ISR-4331, ISR-4431, and ISR-4451. Software for multiple product families are grouped together to support specific solutions deployed in customer networks. For example, software defined access (SDA) is a solution used by enterprises for managing employee access to the Internet. The SDA solution uses different products such as wireless access points, wireless LAN controllers, access switches, routers, and network management systems.

Note that the above example of a source code organization is very specific to one of the large business entities within Cisco. The complexity and structure of the source code organization could vary widely between companies and even between groups within the same company. The software elements from the most granular level of a source file to the most general level of solutions are:

a) Source file (SF)
b) Source directory (SD)
c) Components
d) Products
e) Product family (PF)
f) Solutions

A simplified view of different software elements in a large-scale software is shown in Fig.1. The actual source code structure is extremely complex with many more interconnections between different blocks as compared to the details shown in the figure.

Based on the way the source code is organized and defects are tracked in IOS-XE, code level metrics are easier to estimate for all six data elements defined above. However, the defect-based and test metrics are easier to measure for components, products, product families, and solutions. Because of the difficulty in measuring defect and test metrics, we did not use source file or directory as the data element. Furthermore, while selecting data elements, we need to ensure that there is a sufficient number of data points for ML-based model development. If we use product family as the data element, we will have nearly 300 data points in the dataset. If we use solutions as the data element, we will have less than 25 data points. Because of the small number of data points, we do not use product family or solution as data elements. That leaves only components and products as two possible choices for the data element. As shown in Section VI, we have used both data elements for model development and selected the one that gave a lower prediction error.

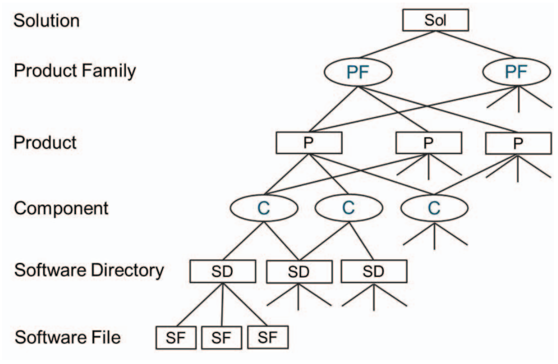

Fig. 1. A simplified hierarchical structure in a source code management (SCM) system.

B. SDP Lifecylce

Software development is a continuous process involving designing, coding, testing, and bug fixing. Depending on the size of the software, the testing and bug fixing process could take many months and involve different types of test activities (e.g. unit, functional, system integration, and solution, etc.) For most of the software development processes, these activities can be organized into three logical groups (Fig.2). The first logical group ends with the milestone called Code Complete (CC), when the source code for a new software feature is complete and ready for formal integration into the code base. Activities like design, coding, unit testing, static analysis, and code review are completed before CC. The next logical phase includes the integration of code with other features in the software and regression testing. These activities end at a logical milestone called Feature Complete (FC). The next set of activities include system integration, solution, performance, scale, security, and other testing required to validate the software before delivering it to end users. This milestone is called the First Customer Availability (FCA).

All the studies in open literature developed the SDP model at FCA and predicted the number of defects expected to be found at the customer environment. This is not sufficient for continuous quality management during the entire SDLC. Therefore, we have developed different SDP models at the three logical milestones described above. These models are:

a) CC Model – Uses the code metrics available at CC as model attributes and predicts the number of defects to be found between CC and FC.
b) FC Model – Uses the code, defect, and process metrics available at FC to predict the number of defects to be found between FC and FCA.
c) FCA Model – Uses the code, defect, and process metrics available at FCA to predict the number of defects to be found by the customers or end users within 6 months from FCA.

The attributes and target variables for each of these models are summarized in Fig.2. More details on the model attributes are discussed in Section V.

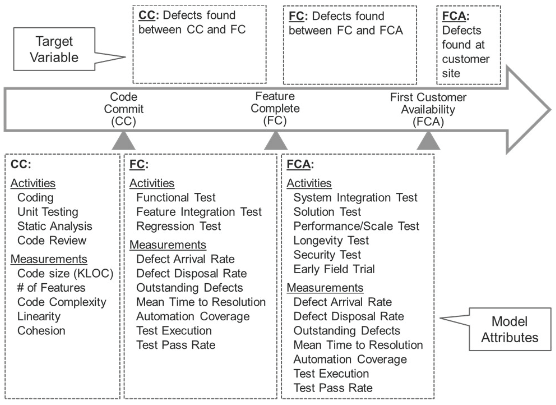

Fig. 2. Logical points for model development during SDLC.

IV. Defect Prediction using Defect Density

Defect density is one of the simplest methods for software defect prediction. It is measured as [18]:

$$Defect\ Density\ (DD) = \frac{Number\ of\ Known\ Defects}{Software\ Size}$$

The number of known defects is the count of total defects reported against a software entity (product or component) by internal teams and customers over a period of time. Software size is typically measured by Lines of Code (LOC) or Function Points (FP). LOC is a widely used method for software sizing because of its simplicity in measurement. FP is considered a better representation of software size, but its adoption has been very limited because of the difficulties in FP measurement, particularly for large-scale systems [19,20]. Therefore, we have used LOC to measure software size.

Software products are continuously modified by adding, updating, and deleting code for bug fixes as well as new features. The software size is represented by the code change, which is a function of the lines of code added, updated and, deleted [2]. We used two different approaches to estimate the software size:

KLOC-A = Added + Updated + Deleted

KLOC-B = Added + Updated – Deleted

where KLOC is kilo LOC. "Added", "Updated", and "Deleted" are KLOC added, updated, and deleted, respectively.

For a given amount of code change in a software release, the simplest defect prediction model can be described as:

Model-1 → $(Defect)_{Rel} = (DD)_{Rel} \times (KLOC)_{Rel}$

where, $(Defect)_{Rel}$, $(DD)_{Rel}$, and $(KLOC)_{Rel}$ are the number of predicted defects, defect density, and KLOC, respectively, for the release. We have observed that 70-80% of bugs found during a release are related to new features introduced in the release. The remaining defects are related to the functionalities introduced in previous releases. Therefore, the above defect prediction model that uses *DD* and *KLOC* for the current release may not be effective. Therefore, we added the contribution of previous release in the prediction model:

Model-2 → $(Defect)_{Rel} = (DD)_{Rel} \times (KLOC)_{Rel}$
$+ (DD)_{Prev} \times (KLOC)_{Prev}$

where, $(DD)_{Prev}$, and $(KLOC)_{Prev}$ are defect density and KLOC, respectively, for the previous release. The objective of the model development is to use the learning dataset to estimate $(DD)_{Rel}$ and $(DD)_{Prev}$, which are the defect densities for the current and previous releases, respectively.

We collected KLOC and the total number of defects found in IOS-XE software described in Section III for the last 8 releases. The data for first 6 releases was used to develop (train) the model, and the data from last two releases was used to test the model. The training dataset consisted of 1122 observations and the testing dataset consisted of 374 observations. The target variable in the dataset is the total number of defects found from the beginning of the release activity until six months after FCA. We used the linear regression algorithm to estimate the defect density for four different scenarios:

- Model-1 with KLOC-A
- Model-1 with KLOC-B
- Model-2 with KLOC-A
- Model-2 with KLOC-B

The estimated defect density and corresponding R-square and P-values are presented in Table I. The P-value for all four models is very small (below 0.05). The R-Square value for both choices for Model-2 are 0.66, which is slightly higher than the R-square value for Model-1.

TABLE I. Comparison of Defect Density Models

Model Description	$(DD)_{Rel}$	$(DD)_{Prev}$	R^2	P-Value
Model-1 + KLOC-A	4.4	NA*	0.63	< 0.0001
Model-1 + KLOC-B	3.9	NA*	0.64	< 0.0001
Model-2 + KLOC-A	3.1	0.7	0.66	< 0.0001
Model-2 + KLOC-B	2.7	0.6	0.66	< 0.0001

* NA = Not Applicable

We applied the models developed in Table I to the test dataset and estimated the prediction error for each model. As shown in Table II, all the models have high prediction errors that are close to 28%. Given very similar model errors (Table I) and prediction errors (Table II) for all four scenarios, it is difficult to decide on the best model. As described earlier, 20-30% of defects found in a release are from features that are developed in the previous release. Therefore, we selected Model-2 with KLOC-A as the defect density model for comparison with other ML-based models developed in this paper.

TABLE II. Error Comparison for Different defect density models

Model Description	Estimation Error
Model-1 + KLOC-A	28.2%
Model-1 + KLOC-B	28.6%
Model-2 + KLOC-A	28.0%
Model-2 + KLOC-B	28.7%

V. Defect Prediction using Machine Learning

As discussed in Section III-B, we are proposing three defect prediction models at CC, FC and FCA to help model-based quality management during the entire software development life cycle. The details of the attribute selection, exploratory data analysis (EDA), model development, and model validation are presented below. The dataset used in this section is the same dataset used for the defect density model in Section IV and uses products as the data element (Fig. 1).

A. Attibute Selection and EDA

Selecting the right set of attributes is one of the most important steps in SDP [2]. We used code, process, and defect metrics for model development at three different points during the SDLC (Fig. 2).

Code Complete (CC) Model

Our defect prediction model at CC uses software size information (KLOC and number of features) from current and previous releases. The attributes for the CC model are:

- New feature KLOC for the current release
- New feature KLOC for the previous release
- Bug fix KLOC for the previous release
- Number of features added in the current release
- Number of features added in the previous release

We used three different scenarios for using KLOC information in the model:

- **Scenerio-1**: Added, Modified, and Deleted KLOCs for current as well as previous releases are considered as separate attributes
- **Scenerio-2**: KLOC = Added + Modified – Deleted
- **Scenerio-3**: KLOC = Added + Modified + Deleted

There are six attributes in Scenario 1. There are two KLOC attributes, one for the current release and another for the previous release, in each of scenarios 2 and 3. The target variable for this model is the total number of defects found during the period between CC and FC (Fig. 1).

FC Model

In addition to the code metrics used for the CC model, defect and process metrics for testing and bug fixing activities are available at FC. The attributes for the FC model consist of:

- New feature KLOC for the current release
- New feature KLOC for the previous release
- Bug Fix KLOC for the previous release
- Number of features added in the current release
- Number of features added in the previous release
- Outstanding defects for different severities at FC
- Number of defects found during different test activities for different severities before FC
- Defect MTTR (mean-time-to-resolution) for different severities at FC

The target variable for the FC model is the total number of defects found during all testing activities between FC and FCA (Fig.2). We performed data cleansing activities to remove attributes with null values, duplicates, and outliers. This resulted in 123 attributes that were used for exploratory data analysis (EDA). We used multi-collinearity analysis using Stepwise VIF (Variance Inflation Factor) to remove attributes that were highly corelated [21]. In this analysis, the higher the value of VIF, the greater the correlation between the variables. Values greater than 5 are considered moderate to high correlation, and values of 10 or higher are considered very high correlation. To keep enough attributes for model development, we selected two different datasets that have VIF ≤ 5 and VIF ≤ 10.

The distribution of VIF for all the attributes is shown in Fig. 3. As expected, most of the 123 attributes are strongly corelated with other attributes and are removed from the dataset. The final dataset with VIF ≤ 10 has 37 attributes and the dataset with VIF ≤ 5 had 27 attributes.

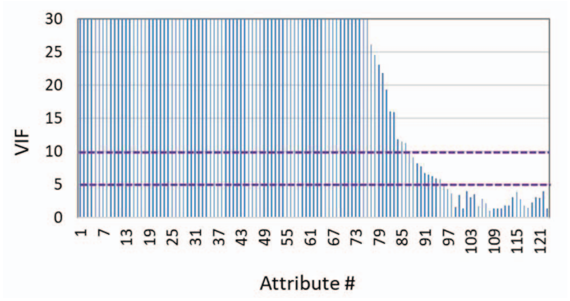

Fig. 3. Distribution of VIF for all the attributes in the dataset for defect prediction at FC.

FCA Model:

The attributes for the FCA model include all attributes for the FC model measured at the FCA, all release quality criteria, security metrics, and other metrics used to measure development and test effectiveness between FC and FCA. These are:

- Software size metrics (KLOC and number of features)
- Defect arrival metrics consisting of the number of defects found in different testing activities for different severities
- Defect disposal metrics representing number of bugs disposed for different severities
- Defect mean time to resolution (MTTR) metrics for different severities
- Metrics in release quality criteria [14] that must be met before software is given to customers
- Defect backlog metrics for different severities that are open at FCA
- Test effectiveness metrics including % of defects found and % of escapes for each test activity
- Security metrics, including the number of defects found and number of outstanding security defects

The target variable for the FCA model is the number of defects that will be found in the customer environment within six months from the FCA date.

As in the case of the FC model, we performed data cleansing to remove attributes with null values, duplicates, and outliers. This resulted in 119 attributes, which were then used in EDA using stepwise VIF. The result of this analysis is shown in Fig.4. As in the case of FC model, we selected two different datasets for model development. The dataset with

VIF ≤ 10 has 35 attributes and the dataset with VIF ≤ 5 had only 15 attributes.

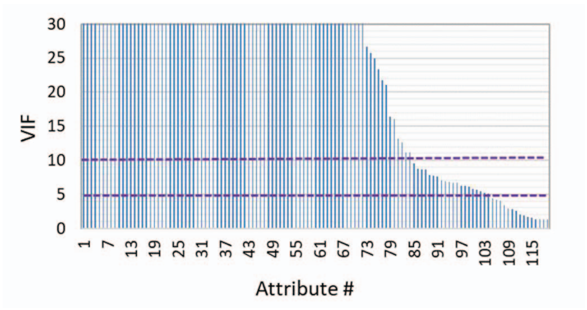

Fig. 4. Distribution of VIF for all attributes in the dataset for defect prediction at FCA

B. *ML Algorithm and Model Development*

We used three supervised ML algorithms to develop the defect prediction models [22-24]:

- Linear Regression
- Random Forest Regression
- XGBoost Regression

These algorithms were implemented using standard model building functions available in the Python library:

- sklearn.linear_model.LinearRegression()
- sklearn.ensemble. RandomForestRegressor()
- xgboost. XGBRegressor()

Then, we estimated three commonly used measures for model evaluation: root mean squared error (RMSE), mean absolute error (MAE), and mean absolute percentage error (MAPE) [25].

The SDP models at CC were developed using different ML algorithms for three scenarios described in Section V-A. The model error for these algorithms are presented in Table III. The XG Boost algorithm in Scenario-2 gives the best results. The KLOC in Scenario-2 is calculated as “added + updated – deleted.” This choice gave the best performance in terms of RMSE and MAE. Note that the MAPE for this case is somewhat higher than a few other cases. The actual value of the predicted variable is used in the denominator for MAPE [25]. Therefore, for very small actual values, the MAPE would be high and biased toward it. As a result, we put more emphasis on RMSE and MAE in selecting the best model.

TABLE III. ERROR COMPARISON FOR DIFFERENT ALGORITHMS USED FOR MODEL DEVELOPMENT AT CODE COMPLETE

Model & Scenario	RMSE	MAE	MAPE
Scenario - 1			
Linear Regression	223	87	661
Random Forest	97	41	141
XG Boost	108	45	127
Scenario - 2			
Linear Regression	161	76	720
Random Forest	107	45	98
XG Boost	**96**	**42**	**132**
Scenario - 3			
Linear Regression	162	77	713
Random Forest	108	46	103
XG Boost	97	42	117

Tables IV and V present the modelling error for defect prediction at FC and FCA, respectively. As shown in Table IV, the Linear Regression model with VIF≤ 10 gave the best model for FC with lowest MAE and RMSE. For the FCA model, Random Forest with VIF≤ 5 is the best model (Table V).

TABLE IV. ERROR COMPARISON FOR DIFFERENT ALGORITHMS USED FOR MODEL DEVELOPMENT AT FC

Model	MAE	RMSE	MAPE
Linear Reg, VIF ≤ 10	**12**	**32**	**98**
Random Forest, VIF ≤ 10	12	41	30
XGBoost, VIF ≤ 10	14	46	46
Linear Reg, VIF ≤ 5	45	172	237
Random Forest, VIF ≤ 5	21	57	77
XGBoost, VIF ≤ 5	19	51	98

TABLE V. ERROR COMPARISON FOR DIFFERENT ALGORITHMS USED FOR MODEL DEVELOPMENT AT FCA

Model	MAE	RMSE	MAPE
Linear Reg, VIF ≤ 10	20	59	231
Random Forest, VIF ≤ 10	12	39	100
XGBoost, VIF ≤ 10	12	36	120
Linear Reg, VIF ≤ 5	20	64	226
Random Forest, VIF ≤ 5	**12**	**32**	**125**
XGBoost, VIF ≤ 5	13	35	138

C. *Defect Prediction Error*

The best CC, FC, and FCA models found in the last section were used in the testing dataset to predict the number of defects. The total defect prediction error for all three models are shown in Table VI.

TABLE VI. DEFECT PREDICTION ERRORS FOR DIFFERENT MODELS

Model	Defect Prediction Error
CC Model	14.3%
FC Model	2.8%
FCA Model	11.7%

Among all three models, the CC model has the highest prediction error of 14.3%. The FC model gave the least error of 2.8%. Different prediction errors could be the results of different model attributes available for model development. Software size information (KLOC and number of features) are used as attributes for the CC model. The absence of other code metrics (e.g. complexity, linearity, modularity, static analysis, etc.) might be the reason for the higher prediction error for the CC model. Given the size and complexity of our code base, it is difficult and too expensive to measure these code metrics. Furthermore, because the CC model is mostly used for resource planning between CC and FC, this level of accuracy for the CC model is acceptable for our application. A lower prediction error of 2.8% for the FC model shows the effectiveness of defect and process metrics for SDP.

VI. SDP FOR LARGE SCALE SYSTEMS

As discussed in Section I, there are four important areas to be considered in ML-based SDP for large scale software. Two of the areas, the choice of attributes and ML algorithms, were discussed in last section. The results for other two areas, "Data definition" and "SPD lifecycle" challenges, are discussed in this section.

A. Data Definition Challenge

As described in Section III-A, selecting the right data element for training and testing the dataset is an important step in the SDP, particularly for large scale systems. We developed defect prediction models using products and components as data elements. The details of the product-based defect prediction models were presented in Section V. The component-based defect prediction models were developed using the same procedure as used for the product-based models. The defect prediction errors for both product-based and components-based models at CC, FC, and FCA are compared in the following table.

TABLE VII. DEFECT PREDICITION ERRORS FOR PRODUCT-BASED AND COMPONENT-BASED MODELS

Model	Defect Prediction Error	
	Product-Based Model	Component-Based Model
CC Model	14.3%	13.1%
FC Model	2.8%	0.6%
FCA Model	11.7%	1.4%

As shown in Table VII, the component-based models yield a much lower prediction error for both FC and FCA models. Even at CC, the component-based model has a slightly lower prediction error than the product-based model. As discussed in Section V-C, using size metrics (e.g. KLOC and number of features) only in the CC model is not enough to achieve a lower prediction error. For applications that require better prediction accuracy at CC, we recommend using additional code metrics (e.g. complexity, linearity, cohesion, and static analysis, etc.) as model attributes.

Given the consistent lower prediction error for all three models, we conclude that the component-based model is better than the product-based model for the dataset used in this study. This conclusion could be different for other datasets. However, based on the results in table VII, it can be concluded that the choice of data element is one of the important factors in the SDP for large scale software. Researchers and practitioners developing ML-based SDP models should evaluate model performance for different data elements before selecting the best model for implementation.

B. SDP Lifecycle Challenge

As proposed in Section III-B, the accuracy of defect prediction could depend on the number of models used during the entire SDLC. We considered three different scenarios to provide an empirical validation of our hypothesis. These are:

(i) Single Defect Density Model – This method developed a single model for the entire SDLC and used software size metrics to predict the number of internal as well as customer found defects.

(ii) Defect Density Based Multiple ML Models – This approach used size metrics and developed three different models at CC, FC, and FCA.

(iii) Size and Process Metrics Based Multiple ML Models – In this approach, we developed three different models at CC, FC, and FCA using size and other process metrics as discussed in Section IV and V.

TABLE VIII. COMPAISON OF DEFECT PREDICTION ERROR IN DIFFERENT SCENARIOS FOR SDLC CHALLENGE

Scenario	Description	Defect Prediction Error
(i)	Defect Density Model	14.1%
(ii)	Defect density Based Multiple ML Model	13.0%
(iii)	Size and Process Metrics Based Multiple ML Model	8.7%

Table VIII compares the error in overall defect prediction over the entire SDLC for the three scenarios described above using component-based dataset. The scenarios (i) and (ii) show high prediction errors when size is used as the only attribute. When size and process metrics are used for FC and FCA models, the overall prediction error is much lower (8.7%). Furthermore, comparison of (i) and (iii) shows significant reduction in prediction error when multiple models, along with process metrics, are used as attributes.

VII. THREATS TO VALIDITY

Although the results presented in Section IV, V and VI are promising, there are some factors that threaten the validity of our conclusions. This section summarizes both internal and external threats to the validity.

Internal validity is the ability of a study to establish a causal link between independent and dependent variables regardless of what the variables are believed to represent [26]. We used software size (KLOC and number of features) as attributes for the CC model. Because of the difficulty and high cost of measurement, we could not include other code metrics (e.g. cyclomatic complexity, linearity, and modularity, etc.) as the model attributes. Because the absence of some code metrics could affect the accuracy of the CC model, our conclusions related to data element and SDP lifecycle challenges are not impacted by this threat.

External validity refers to how the results of a study generalize [26]. The results in our study are for IOS-XE software developed by the Enterprise Networking (EN) group, which is one of the largest and most diverse groups in Cisco. While the results from the EN give evidence of the ability to generalize, results from the implementation in other large software will increase confidence on the general applicability of our approach.

VIII. SUMMARY AND FUTURE WORK

Published studies on ML-based software defect prediction (SDP) focus on the choice of model attributes and algorithms to achieve the best prediction model. We identified two additional focus areas that are essential for the SDP in large scale software systems. We call these the "Data Definition" and "SDP Lifecycle" challenges. We developed several SDP models for different scenarios for large-scale software (IOS-XE) developed by Cisco Systems. As the results show, our proposed solutions to the Data Definition and the SDP Lifecycle challenges give a significantly lower prediction error. It should be noted that our solution is specific to the dataset used in the study and may not directly apply to other large software systems. However, our general conclusion on the importance of these challenges should be applicable to all large software. Researchers and industry practitioners developing ML-based SDP models for large software should address all four challenges before finalizing a model for their application.

The next step in our work is to develop a detailed operational procedure to use these models in managing quality activities during upcoming software releases. We will also continue to use our methodology to develop and implement ML-based SDP models in other large software developed by Cisco. In addition, we will be working on incorporating additional code metrics to improve prediction accuracy.

REFERENCES

[1] Pettey, C., "Moving to a Software Subscription Model," Gartner Research, May 2018, downloaded on 4/4/2019 from https://www.gartner.com/smarterwithgartner/moving-to-a-software-subscription-model/

[2] D'Ambros, M., Lanza, M., and Robbes, R., "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, 2012, Vol. 17, No. 4-5, pp. 531-577.

[3] Catal, C, "Software fault prediction: A literature review and current trends," *Expert Systems with Applications*, Volume 38, Issue 4, April 2011, Pages 4626-4636.

[4] Radjenovic, D., Hericko, M., Torkar, R., and Zivkovic, A., "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, Volume 55, Issue 8, 2013, pp. 1397-1418.

[5] Hammouri, A., Hammad, M., Alnabhan, M., and Alsarayrah, F., "Software Bug Prediction using Machine Learning Approach," *International Journal of Advanced Computer Science and Applications (IJACSA)*, Vol.9, No.2, 2018, pp.78-83.

[6] Prasad, M.C.M., Florence, L., and Arya, A., "A Study on Software Metrics based Software Defect Prediction using Data Mining and Machine Learning Techniques," *International Journal of Database Theory and Application*, Vol.8, No.3, 2015, pp.179-190.

[7] Yousef, A. H., "Extracting Software Static Defect Models using Data Mining," Ain Shams Engineering Journal, (2015) 6, pp. 133-144.

[8] Malhotra, R., "A Systematic Review of Machine Learning Techniques for Software Fault Prediction," *Applied Soft Computing,* Volume 27, February 2015, Pages 504-518.

[9] Chou, C. H., "Metrics in Evaluating Software Defects," *International Journal of Computer Applications*, Vol. 63, No.3, February 2013, pp.23-29.

[10] Singh, P. D., and Chug, A., "Software defect prediction analysis using machine learning algorithms," *7th International Conference on Cloud Computing, Data Science & Engineering Confluence*, IEEE, 2017.

[11] Okutan, Ahmet, and Olcay Taner Yıldız. "Software defect prediction using Bayesian networks." *Empirical Software Engineering*, Vol. 19, No.1, 2014, pp.154-181.

[12] *The Elusive Agile Enterprise: How the Right Leadership Mindset, Workforce and Culture Can Transform Your Organization*, Forbes Insight in Association with Scrum Alliance, 2018.

[13] Knaster, R., and Leffingwell, D., *SAFe Distilled – Applying the Scaled Agile Framework for Lean Enterprises*, Addison-Wesley, 2019.

[14] Pradhan, S., and Nanniyur, V., "Back to Basics – Redefining Quality Measurement for Hybrid Software Development Organizations," *The 30th IEEE International Symposium on Software Reliability Engineering (ISSRE 2019)*, Berlin, Germany, Oct 28-31, 2019.

[15] Dawson, M., Burrell, D., Rahim, E. and Brewster, S., "Integrating Software Assurance into the Software Development Life Cycle (SDLC)," *Journal of Information Systems Technology and Planning*, 2010, Vol 3, pp.49-53.

[16] Jones, C., *Software Assessments, Benchmarks, and Best Practices*, Addison-Wesley Information Technology Series, 2000.

[17] "Modular Programming," WikiPedia, retrieved on 4 Sept, 2019 from https://en.wikipedia.org/wiki/Modular_programming

[18] Rahmani, C. and Khazanchi, D., "A Study on Defect Density of Open Source Software," The 9th IEEE/ACIS International Conference on Coputer and Information Science, ICIS 2010, pp. 679-683.

[19] Walkerden, F., and Jeffery, R., "Software Cost Estimation: A Review of Models, Process, and Practice," Advances in Computers, Vol. 44, 1997, pp.59-125.

[20] Basavaraj M., Shet K., "Software Estimation using Function Point Analysis: Difficulties and Research Challenges," *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, Editor: Sobh T., 2007, pp.111-116, Springer, Dordrecht.

[21] Akinwande, O., Dikko, H.G., and Agboola, S., "Variance Inflation Factor: As a Condition for the Inclusion of Suppressor Variable(s) in Regression Analysis", *Open Journal of Statistics*, January 2015, pp.754-767.

[22] Olsen, D. L., and Delen, D., *Advanced Data Mining Techniques*, Springer, 1st Edition, page 138, ISBN-13: 978-3540769163, 2008.

[23] Kulkarni V Y, Sinha P K, "Random Forest Classifiers: A Survey and Future research Directions", *International Journal of Advanced Computing*, Vol 36, Issue 1, 1144-1153.

[24] Chen, T. and Guestrin, C., "Xgboost: A Scalable Tree Boosting System," *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining*, 2016, pp.785-794.

[25] Botchkarev, A, "A New Typology Design of Performance Metrics to Measure Errors in Machine Learning Regression Algorithms," *Interdisciplinary Journal of Information, Knowledge, and Management*, 2019, Vol. 14, No. 1, pp. 45-79.

[26] Perry, D., Porter, A., and Votta, L., "Empirical Studies of Software Engineering: A Roadmap," *Proceedings of the Conference on The Future of Software Engineering*, ACM New York, USA, 2000, pp.345-355.