Probability of Failure-free Operations with Software for Defect Management

A. Vivek Yoganand¹, G. Premnath², R. Sathya³, S. Velmurugan⁴

¹, ², ³, ⁴ Assistant Professor, Computer Science & Engineering
Veltech Multitech Dr. Rangarajan Dr. Sakunthala Engineering College, Avadi, Chennai-600 062

Abstract-- In this paper, an empirical study that links software defect consistency with defects detection and prevention is respond. Various measurement issues such as validity, reliability, and other challenges in measuring process consistency at the project level are discussed. Even a relatively modest improvement in the consistency of implementing these practices is associated with a significant reduction in field defects prevention. A software developer tests the program developed to determine if it behaves as desired. In each of these cases, there is an attempt by a tester to determine if the human thoughts, actions, and products behave as desired. Behavior that deviates from the desirable is possibly of an error. Numerous studies have applied to learn the software defect management problem, i.e. predicting which modules will experience a failure during operation based on software metrics. However, the defect-prediction operations can measure that the resulting classifiers often predict the faulty (minority) class less accurately. Here, we investigate how to prevent alternatives (under, and oversampling) for software defect prediction using defect management study.

Index Terms-- Test Metric, Risk Management, Defect Prevention, Orthogonal defect Classification, Defensive Design, Risk Impact.

ACRONYMS
SDP Software defect prediction
AUC Area under the curve
NB Naive Bayes with the log filter
PD Probability of detection
PF Probability of false alarm
DNC Dynamic version of AdaBoost.NC
BN Bayesian Nets

INTRODUCTION
Defect management is an integral part of a development and test process in many software development organizations. It is a sub process of the development process. It entails the following:
A. Defect prevention
B. Discovery
C. Recording and Reporting
D. Classification
E. Resolution
F. Prediction

Defect prevention is achieved through a variety of process and tools. For example, good coding techniques, unit test plans, and code inspection are all important elements of any defect prevention process. Defect discovery is the identification of defects in response to failures observed during dynamic testing or found during static testing [11][06]. Discovery a defect often involves debugging the code under test.

Defects found are classified and recorded in a database. Classification becomes important in dealing with the defects.

For example, defects classified as high severity are likely to attend to first by the developers than those classified as low severity. A variety of defect classification schemes exist. Orthogonal defect classification, popularly known as ODC, is one such scheme. Defect classification assist as organization in measuring statistics such as the types of defect, their frequency, and their location in the development phases and document. These statistics are then input to the organization’s process improvement team that analyzes the data, identifies areas of improvement in the development process, and recommends appropriate actions to higher management [12].

Each defect, when recorded, is marked as open indicating that is needs to be resolved. One or more developers are assigned to resolve the defect. Resolution requires careful scrutiny of the defect, identifying a fix if needed, implementing the fix, testing the fix, and finally closing the defect indicating that it has been resolved. It is not necessary that every recorded defect be resolved prior to release. Only defects that are considered critical to the company’s business goals, that included quality goals, are resolved; others are left unresolved until later.

Defect predication is another important aspect of defect management. Organization often does source code analysis to predict how many defects an application might contain before it enters the testing phase. Despite the imprecise nature of such early predications, they are used to plan for testing resources and release dates [09]. Advanced statistical techniques are used to predict defects during the test process. The Predictions tend to be more accurate than early predications due to the availability of the defect data and the use of the sophisticated models. The defect discovery data, including time to discover and type of defect, is used to predict the count of remaining defects. Once again this information is often imprecise, though nevertheless used in planning [08].
Several tools exist for recording defects, and computing and reporting defect-related statistics. Bugzilla, open source, and FogBugz, commercially available, are two such tools. They provide several features for defect management, including defect recording, classification, and tracking. Several tools that compute complexity metrics also predict defects using code complexity.

In this paper the following topics are discussed
A. Defect management and terminology
   A. Identify Critical Risks
   B. Minimize Expected Impact
   C. Techniques to prevent defects.
   D. Techniques to Minimize Impact
   E. Failures free Algorithm.
   F. Defect identification measures.
   G. Summary and conclusion.

Major defect prevention and management are covered with the help of

I. A Defect management and Terminology
The best approach to defects is to eliminate them altogether. While that would be ideal, it is virtually impossible given current technology. In the meantime, developers need strategies to find defects quickly and minimize their impact.

Fig 1) a multi-step approach to defect management

   Defect Prevention          Deliverable Baseline          Defect Discovery          Defect Resolution
   Management Reporting

   Process Improvement

i. Defect Prevention
Identifying and implementing the best defect prevention techniques (which is a large part of identifying the best software development processes) should be a high priority activity in any defect management program.

ii. Defect Prevention Process
Defect prevention should begin with an assessment of the critical risks associated with the system. Getting the critical risks defined allows people to know the types of defects that are most likely to occur and the ones that can have the greatest system impact. Strategies can then be developed to prevent them. The major steps for defect prevention are as follows:

Example: Two requirements are given below, each of which leads to a different program.
Requirement 1: It is required to write a program that inputs two integers and outputs the maximum of these.
Requirement 2: It is required to write a program that inputs a sequence of the integers and outputs the sorted version if this sequence.

Based on the above requirements it’s easy to implements the domain model. It has three sorts of elements

< A – 3 15 12 55. >
<D 232 78. >
<A. >

The first elements contains a sequence of four integers to be sorted in ascending order, the second one has a sequence to be sorted in descending order, and third one has an empty sequence to sorted in ascending order.

A. Identify Critical Risks
The first step in preventing defects is to understand the critical risks facing the project or system. The best way to do this is to identify the types of defects that pose the largest threat. In short, they are the defects that could jeopardize the successful construction, delivery and/or operation of the system. These risks can vary widely from project to project depending on the type of system, the technology, the users of the software, etc [15][17]. These risks might include:

- Missing a key requirement
- Critical application software that does not function properly
- Vendor supplied software does not function properly
- Performance is unacceptably poor
- Hardware malfunction
- Hardware and/or software does not integrate properly
- Hardware new to installation site
- Hardware not delivered on-time
- Users unable or unwilling to embrace new system
- User’s inability to actively participate in project
- Etc.

It should be emphasized that the purpose of this step is not to identify every conceivable risk, but to identify those critical risks that merit special attention because they could jeopardize the success of the project.

Estimate Expected Impact.
Once the critical risks are identified, the financial impact of each risk should be estimated. This can be done by assessing the impact, in dollars, if the risk does become a problem combined with the probability that the risk will become a problem. The product of these two numbers is the expected impact of the risk. The expected impact of a risk (E) is calculated as E = P * I, where:

P=Probability of the risk becoming a problem
I=Probability of the risk becoming a problem

Once the expected impact of each risk is identified, the risks should be prioritized by the expected impact and the degree to which the expected impact can be reduced. While guess work will constitute a major role in producing these numbers, precision is not important.
What will be important is to identify the risk, and determine the risk’s order of magnitude. Large, complex systems have many critical risks. Whatever can be done to reduce the probability of each individual critical risk becoming a problem to a very small number should be done [30]. Doing this increases the probability of a successful project by increasing the probability that none of the critical risks will become a problem.

One should assume that an individual critical risk has a low probability of becoming a problem only when there is specific knowledge justifying why it is low. For example, the likelihood that an important requirement was missed may be high if developers have not involved users in the project. If users have actively participated in the requirements definition, and the new system is not a radical departure from an existing system or process, the likelihood may be low.

One of the more effective methods for estimating the expected impact of a risk is the annual loss expectation (ALE) formula [09][25]. This is discussed below:

- The occurrence of a risk can be called an “event.”
- Loss per event can be defined as the average loss for a sample of events.
- The formula states that the ALE equals the loss per event multiplied by the number of events.

For example, if the risk is that the software system will abnormally terminate, then the average cost of correcting an abnormal termination is calculated and multiplied by the expected number of abnormal terminations associated with this risk.

For the annual calculation, the number of events should be the number of events per year.

B. Minimize Expected Impact

The expected impact may be strongly affected not only by whether or not a risk becomes a problem, but also by how long it takes for a problem to become recognized and how long it takes to be fixed once recognized. In one reported example, a telephone company had an error in its billing system that caused it to under bill its customers by about $30 million. By law, the telephone company had to issue corrected bills within thirty days, or write-off the under billing. By the time the telephone company recognized it had a problem, it was too late to collect much of the revenue.

Expected impact is also affected by the action that is taken once a problem is recognized. Once Johnson and Johnson realized it had a problem with TYLENOL tampering, it greatly reduced the impact of the problem by quickly notifying doctors, hospitals, distributors, retail outlets, and the public of the problem. While the tampering itself was not related to a software defect, software systems had been developed by Johnson and Johnson to quickly respond to drug related problems. In this case, the key to Johnson & Johnson’s successful management of the problem was how it minimized the impact of the problem once the problem was discovered.

Minimizing expected impact involves a combination of the following three strategies:

- **Eliminate the Risk:** While this is not always possible or desirable, there are situations where the best strategy will be simply to eliminate the risk altogether. For example, reducing the scope of a system, or deciding not to use the latest unproven technology are ways to eliminate certain risks altogether.

- **Reduce the Probability of a Risk Becoming a Problem:** Most strategies will fall into this category. Inspections and testing are examples of approaches that reduce, but do not eliminate, the probability of problems.

- **Reduce the Impact if there is a Problem:** In some situations, the risk cannot be eliminated, and even when the probability of a problem is low, the expected impact is high. In these cases, the best strategy may be to explore ways to reduce the impact if there is a problem. Contingency plans and disaster recovery plans would be examples of this strategy.

From a conceptual viewpoint, there are two ways to minimize the risk. These are deduced from the annual loss expectation formula. The two ways are to reduce the expected loss per event, or reduce the frequency of an event. If both of these can be reduced to zero, the risk will be eliminated. If the frequency is reduced, the probability of a risk becoming a problem is reduced. If the loss per event is reduced, the impact is reduced when the problem occurs.

There is a well known engineering principle that says that if you have a machine with a large number of components, even if the probability that any given component will fail is small, the probability that one or more components will fail may be unacceptably high. Due to this phenomenon, engineers are careful to estimate the mean time between failures of the machine. If the machine cannot be designed with a sufficiently large mean time between failures, the machine cannot be made. When applied to software development, this principle would say that unless the overall expected impact of the system can be made sufficiently low, do not develop the system.

Appropriate techniques to reduce expected impact are a function of the particular risk.

C. Techniques to prevent defects.

The expected impact may be strongly affected not only by whether or not a risk becomes a problem, but also by how long it takes for a problem to become recognized and how long it takes to be fixed once recognized. In one reported example, a telephone company had an error in its billing system that caused it to under bill its customers by about $30 million. By law, the telephone company had to issue
corrected bills within thirty days, or write-off the under billing. By the time the telephone company recognized it had a problem, it was too late to collect much of the revenue.

Expected impact is also affected by the action that is taken once a problem is recognized. Once Johnson and Johnson realized it had a problem with Tylenol tampering, it greatly reduced the impact of the problem by quickly notifying doctors, hospitals, distributors, retail outlets, and the public of the problem. While the tampering itself was not related to a software defect, software systems had been developed by Johnson and Johnson to quickly respond to drug related problems. In this case, the key to Johnson & Johnson’s successful management of the problem was how it minimized the impact of the problem once the problem was discovered.

a. Test metrics

The term metric refers to a standard of measurement. In software testing, there exit a variety of metrics. Figure (2) shows a classification of various types of metrics briefly discussed in this section. Metrics can be computed at the organizational, process, project, and product levels. Each set of measurements has its value in monitoring, planning, and control [19] [20].

Fig (2) Types of metrics used in software testing and their relationships.

Level 1: Organization
Level 2: Establishes test processes
Level 3: Used in projects
Level 4: To test products

Regardless of the level at which metrics are defined and collected, there exist four general core areas that assist in the design of metrics. These are schedule, quality, resources, and size. Schedule-related metrics measure actual completion times of various activities and completion. Quality-related metrics measure quality of a product or a process. Resources-related metrics measure items such as cost in dollars, manpower, and tests executed. Size-related metrics measure size of various objects such as the source code and number of tests in a test metrics.

Minimizing expected impact involves a combination of the following three strategies:

● Eliminate the Risk: While this is not always possible or desirable, there are situations where the best strategy will be simply to eliminate the risk altogether. For example, reducing the scope of a system, or deciding not to use the latest unproven technology are ways to eliminate certain risks altogether.

   ● Reduce the Probability of a Risk Becoming a Problem: Most strategies will fall into this category. Inspections and testing are examples of approaches that reduce, but do not eliminate, the probability of problems.

   ● Reduce the Impact if there is a Problem: In some situations, the risk cannot be eliminated, and even when the probability of a problem is low, the expected impact is high. In these cases, the best strategy may be to explore ways to reduce the impact if there is a problem. Contingency plans and disaster recovery plans would be examples of this strategy.

From a conceptual viewpoint, there are two ways to minimize the risk. These are deduced from the annual loss expectation formula. The two ways are to reduce the expected loss per event, or reduce the frequency of an event. If both of these can be reduced to zero, the risk will be eliminated. If the frequency is reduced, the probability of a risk becoming a problem is reduced. If the loss per event is reduced, the impact is reduced when the problem occurs.

There is a well known engineering principle that says that if you have a machine with a large number of components, even if the probability that any given component will fail is small, the probability that one or more components will fail may be unacceptably high. Due to this phenomenon, engineers are careful to estimate the mean time between failures of the machine. If the machine cannot be designed with a sufficiently large mean time between failures, the machine cannot be made. When applied to software development, this principle would say that unless the overall expected impact of the system can be made sufficiently low, do not develop the system.

Appropriate techniques to reduce expected impact are a function of the particular risk.

D. Techniques to Minimize Defect

Some techniques that can help you minimize the impacts of defects are as follows:

● Quality Assurance: Quality Assurance techniques are designed to ensure that the processes employed are adequate to produce the desired result and that the process is being followed.

● Training and Education (work force): It goes without saying that the better trained a work force is, the higher the quality of its work. Many defects are simply the result of workers not understanding how to do their job. Computer technology is significantly more complex today than it was just a few years ago. Moreover, the complexity will increase significantly in the coming years. Thus, it appears that the training needs at most organizations will increase sharply in the coming years.
Training and Education (customers): As more and more people use computers, and as the complexity of systems grows, the problem of training the end user will become more and more acute. Unlike the problem of training workers, more creative strategies will be required to train customers -- especially when customers are not technically sophisticated. One computer manufacturer reported that a customer when asked to send in a copy of a floppy disk, sent in a Xerox copy. In another instance, a customer complained that a floppy disk would not go into the drive. It was later determined that the customer did not realize that he had to remove the floppy that was already in the drive, before another could be inserted. While these anecdotes are extreme, the problem of effectively training even sophisticated customers to use complex software is far from trivial. Many software vendors have recognized this problem and developed strategies to address it (more elaborate HELP facilities, cue cards, audio training tapes delivered with the product, tutorials, etc.)

Methodology and Standards: As the nature of a process becomes understood, it evolves from art to science. At some point in this evolution, it becomes appropriate to standardize the process. This has occurred in the past with the development of standard life cycles, design approaches, etc. This is occurring today with many diverse efforts – various [14][19] IEEE standards, ISO 9000, etc. As the root cause of defects become understood, consideration should be given to developing or enhancing an organization’s methodology and standards to produce a repeatable process that prevents the defects from re-occurring.

Defensive Design: While there are many variations of defensive design, the concept generally refers to designing the system so that two or more independent parts of the system must fail before a failure could occur. As technology gets more and more complicated, there should be significantly more emphasis on designing systems defensively to prevent, discover, and minimize the impact of defects. While some organizations have been doing this for years, it is a new concept to many organizations and the industry provides very little guidance on how to do it. Design techniques to improve reliability should receive more attention as the complexity of technology grows. These techniques usually involve designing the system so that two components of the system must be in error before a major system problem can occur.

Defensive Code: The concept of designing a program to prevent, discover, and minimize the impact of defects is not new. It is however not widely practiced. Like defensive design, the concept of defensive code involves adding code to a program so that two parts of the program must fail before a major problem can occur. One form of defensive code, assertions, has been around for many years, but has received relatively little attention. An assertion is code that tests for expected conditions and brings unexpected conditions to the attention of the programmer or users. This area too deserves to receive more attention as the complexity of technology grows. The best defect prevention techniques will be the ones that reduce the expected impact the most. This, in turn, will be a function of the nature of the risks and systems within an organization. Very critical software (e.g., NASA’s space shuttle software, health care equipment) and widely distributed software (e.g., Microsoft Windows) may need to use all of the above techniques and more to adequately reduce the overall risk of highly critical software.

II. FAILURES FREE ALGORITHM (DEFECT PREDICTION WITH BN).

A Bayesian net (BN) is a directed acyclic graph together with an associated set of probability tables. The nodes represent uncertain variables and the arcs represent the causal/relevance relationships between the variables. We have adopted the convention in this paper that a dotted margin around a node indicates that it is a “link” node. The BN of Fig. 1 forms a causal model of the process of inserting, finding and fixing software defects. The variable ‘effective KLOC (Effective line of Code) implemented’ represents the complexity-adjusted size of the functionality implemented: as the amount of functionality increases the number of potential defects rises.

Figure options

In this version of the model, KLOC is used as a surrogate for Function Points (FP). Function points are the preferred measure of program size since they can be estimated on the basis of a functional specification. However, most of the companies involved in the validation of this model did not use FPs and, since the validation was retrospective, KLOC measures were readily available.
The 'probability of avoiding defect in development'

<table>
<thead>
<tr>
<th>C</th>
<th>COBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>IDENTIFICATION</td>
</tr>
<tr>
<td>000200</td>
<td>DIVISION</td>
</tr>
<tr>
<td>000300</td>
<td>PROGRAM-ID</td>
</tr>
<tr>
<td>000400</td>
<td>HELLO WORLD</td>
</tr>
<tr>
<td>000500</td>
<td>ENVIRONMENT</td>
</tr>
<tr>
<td>000600</td>
<td>DIVISION</td>
</tr>
<tr>
<td>000700</td>
<td>CONFIGURATION</td>
</tr>
<tr>
<td>000800</td>
<td>SOURCE-COMPUTER</td>
</tr>
<tr>
<td>000900</td>
<td>PH-COBOL</td>
</tr>
<tr>
<td>001000</td>
<td>OBJECT-COMPUTER</td>
</tr>
<tr>
<td>010000</td>
<td>FILE</td>
</tr>
<tr>
<td>010100</td>
<td>RETURN</td>
</tr>
<tr>
<td>010200</td>
<td>PROCEDURE</td>
</tr>
<tr>
<td>100100</td>
<td>MAIN-LOGIC</td>
</tr>
<tr>
<td>100200</td>
<td>SECTION</td>
</tr>
<tr>
<td>100300</td>
<td>BEGIN</td>
</tr>
<tr>
<td>100400</td>
<td>DISPLAY &quot; &quot; LINE 1  POSITION 1</td>
</tr>
<tr>
<td>100500</td>
<td>DISPLAY &quot;Hello world!&quot; LINE 15 POSITION 10</td>
</tr>
<tr>
<td>100600</td>
<td>STOP</td>
</tr>
<tr>
<td>100700</td>
<td>MAIN-LOG-EXIT</td>
</tr>
<tr>
<td>100800</td>
<td>EXIT</td>
</tr>
</tbody>
</table>

The ‘probability of avoiding defect in development’ determines ‘defects in’ from ‘Potential defects given specification and documentation adequacy’. This number represents the number of defects (before testing) that are in the new code that has been implemented. However, inserted defects may be found and fixed: the residual defects are those remaining after testing.

There is a probability table for each node, specifying how the probability of each state of the variable depends on the states of its parents. Some of these are deterministic (in the sense that they introduce no new uncertainty): for example ‘Residual defects’ is simply the numerical difference between ‘Defects in’ and ‘Defects fixed’. In other cases, we can use standard statistical functions: for example the process of finding defects is modelled as a sequence of independent experiments, one for each defect present, using the ‘Probability of finding a defect’ as a characteristic of the testing process.

**Defects found = B(Defects inserted, Prob finding a defect)**

Where $B(n, p)$ is the Binomial distribution for $n$ trials with probability $p$.

Some nodes are defined as ranked nodes. These have a discrete set of states such as: “very low”, “low”, “medium”, “high”, “and very high”. Such nodes are useful when capturing expert judgment, where a simple, qualitative description is required. However, because this judgment is also intended to indicate degree, it is represented by an underlying real number in the range $[0,1]$. An example is the ‘Quality of spec and doc PRE’ node in Fig. 4. Its children incorporate the $[0,1]$ value of their parent into expressions which determine their conditional probability tables. However, the correspondence between qualitative degree and quantitative value is not always straightforward. For example, the ‘Quality of spec and doc POST’ node in Fig. 2 uses a partitioned expression on one of its parents to create distinct conditional probability tables corresponding to each parent state.

Various computer languages balance brevity and clarity in different ways; as an extreme example, most assembly languages would require hundreds of lines of code to perform the same task as a few characters in APL. The following example shows a comparison of a “hello world” program written in C, and the same program written in COBOL - a language known for being particularly verbose.

Another increasingly common problem in comparing KLOC metrics is the difference between auto-generated and hand-written code. Modern software tools often have the capability to auto-generate enormous amounts of code with a few clicks of a mouse. For instance, GUI builders automatically generate all the source code for a GUI object simply by dragging an icon onto a workspace. The work involved in creating this code cannot reasonably be compared to the work necessary to write a device driver, for instance. By the same token, a hand-coded custom GUI class could easily be more demanding than a simple device driver; hence the shortcoming of this metric. For variables without parents the table just contains the prior probabilities of each state of code.

The BN represents the complete joint probability distribution – assigning a probability to each combination of states of all the variables – but in a factored form, greatly reducing the space needed. When the states of some variables are known, the joint probability distribution can be recalculated conditioned on this ‘evidence’ and the updated marginal probability distribution over the states of each variable can be observed.

The quality of the development and testing processes is represented in the BN of Fig. 2 by four variables over the 0 to 1 interval:

- Probability of avoiding specification defects,
- Probability of avoiding defects in development,
- Probability of finding defects,
- Probability of fixing defects.

The BN in Fig. 2 is a simplified version of the BN at the heart of a decision support system for software defects, discussed below. None of these probability variables (or the ‘Effective KLOC implemented’ variable) are entered directly by the user: instead, these variables have further parents modeling the causes of process quality.
III. EXPERIMENTAL METHODOLOGY

This section describes the data sets, learning algorithms, and evaluation criteria used in this study. The data sets we chose vary in imbalance rates, data sizes, and programming languages. The chosen learning algorithms cover different types of methods in class imbalance learning.

A. Data Sets

All ten SDP (Software defect prediction) data sets listed in Table I come from practical projects, which are available from the public PROMISE repository [27] to make sure that our predictive models are reproducible and verifiable, and to provide an easy comparison to other papers. The data sets are sorted in order of the imbalance rate, i.e., the percentage of defective modules in the data set, varying from 6.94% to 32.29%. Each data sample describes the attributes of one module or method (hereafter referred to as module), plus the class label of whether this module contains defects. The module attributes include McCabe metrics,

| TABLE I: PROMISE DATA SETS, SORTED IN ORDER OF THE IMBALANCE RATE (DEFECT%: THE PERCENTAGE OF DEFECTIVE MODULES) |
|-----|-----|-----|-----|
| name | language | examples | attributes | defect% |
| m2  | C++   | 161    | 39       | 32.29   |
| k2  | C++   | 522    | 21       | 20.49   |
| jml | C     | 10885  | 21       | 19.35   |
| kc1 | C++   | 2109   | 21       | 15.45   |
| pc4 | C     | 1458   | 37       | 12.20   |
| pc3 | C     | 1563   | 37       | 10.23   |
| cm1 | C     | 498    | 21       | 9.83    |
| kc3 | Java  | 458    | 39       | 9.38    |
| mw1 | C     | 403    | 37       | 7.69    |
| pc1 | C     | 1109   | 21       | 6.94    |

Halstead metrics, lines of code, and other attributes. It is worth mentioning that a repeated pattern of an exponential distribution in the numeric attributes is observed in these data sets, formed by many small values combined with a few much larger values. Some work thus applied a logarithmic filter to all numeric values as a preprocessor, which appeared to be useful for some types of learners [3]. For example, the log filter was shown to improve the performance of Naive Bayes significantly, but contributed very little to decision trees [37]. The data sets cover three programming languages. Data set jm1 contains a few missing values, which are removed before our experiment starts. Missing data handling techniques could be used instead in future work.

B. Evaluation Criteria

Due to the imbalanced distribution of SDP data sets and various requirements of software systems, multiple performance measures are usually adopted to evaluate different aspects of constructed predictors. There is a trade-off between the defect detection rate and the overall performance, and both are important.

To measure the performance on the defect class, the Probability of Detection (PD), and the Probability of False Alarm (PF) are usually used. PD, also called recall, is the percentage of defective modules that are classified correctly within the defect class. PF is the proportion of non-defective modules misclassified within the non-defect class. Menzies et al. claimed that a high-PD predictor is still useful in practice, even if the other measures may not be good enough [17], [27].

For more comprehensive evaluation of predictors in the imbalanced context, G-mean [18], and AUC [29] are frequently used to measure how well the predictor can balance the performance between two classes. By convention, we treat the defect class as the positive class, and the non-defect class as the negative class. A common form of G-mean is expressed as the geometric mean of recall values of the positive and negative classes. A good predictor should have high accuracies on both classes, and thus a high G-mean. In the SDP context,

\[ G = \sqrt{P(1 - F)} \]

It reflects the change in PD efficiently [50]. AUC estimates the area under the ROC curve, formed by a set of (PF, PD) pairs. The ROC curve illustrates the trade-off between detection and false alarm rates, which serves as the performance of a classifier across all possible decision thresholds. AUC provides a single number for performance comparison varying in [0,1]. A better classifier should produce a higher AUC. AUC is equivalent to the probability that a randomly chosen example of the positive class will have a smaller estimated probability of belonging to the negative class than a randomly chosen example of the negative class. Because the point \((PF = 0, PD = 1)\) is the ideal position, where all defects are recognized without mistakes, the measure balance is introduced by calculating the Euclidean distance from the real (PF, PD) point to (0, 1), and is frequently used by software engineers in practice [3]. By definition,

\[ \text{balance} = 1 - \sqrt{\frac{(0 - PF)^2 + (1 - PD)^2}{2}} \]

In our experiment, we compute PD and PF for the defect class. Higher PDs and lower PFs are desired. We use AUC, balance to assess the overall performance. All of them are expected to be high for a good predictor. The advantage of these five measures is their insensitivity to class distributions in data [9], [31].

IV. CLASS IMBALANCE LEARNING FOR SDP

In this section, we first compare the performance of the five class imbalance learning methods based on the parameter searching strategy and the two existing SDP methods. The results will show their advantages and disadvantages. Based on the observations, we then improve them further.

A. Study about AUC
For each data set, we build seven predictive models following the algorithm settings described in the previous section. We use balance, G-mean, and AUC, as the criterion for determining the best parameter of class imbalance learning methods under our training scheme. We use the Student’s t-test at a confidence level of 95% for the statistical significance test.

For the defect class, Fig. 2 presents the scatter plots of (PD, PF) points from the seven training methods on the ten SDP data sets. Each plot has a different parameter searching criterion applied to the class imbalance learning methods, which are balance, G-mean, and AUC respectively. Each point is the average of 100 s-independent runs. The classifier with more points distributed in the bottom right has a higher defect detection rate and less performance loss on the non-defect class.

We can gain the following results from Fig. 2. In terms of the defect detection rate (PD), NB outperforms all the five class imbalance learning models, which shows its effectiveness in finding defects. Although RUS-bal appears to be better at PD than other class imbalance learning models, it is still not as good as NB in most cases. In terms of the false alarm rate (PF), although RF is the best, it performs the worst in PD, which makes it hardly useful in practice. To understand which measure is better to be the criterion for choosing parameters of class imbalance learning methods for SDP, we produce bar graphs in Fig. 4, displaying the average performance values of PD, [02], [11]

RUS Random undersampling
RUS-bal Balanced version of random undersampling
THM Threshold-moving
SMB SMOTEBoost
BNC AdaBoost.NC

![Fig. 4. Scatter plots of (PD, PF) points of the seven training methods on the ten SDP data sets.](image)

(a) Balance criterion;

<table>
<thead>
<tr>
<th>Table IV: The Optimal Parameter Obtained from the Training Scheme Based on Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
</tr>
<tr>
<td>mc2</td>
</tr>
<tr>
<td>kc2</td>
</tr>
<tr>
<td>jm3</td>
</tr>
<tr>
<td>kc1</td>
</tr>
<tr>
<td>pc4</td>
</tr>
<tr>
<td>pc3</td>
</tr>
<tr>
<td>cm1</td>
</tr>
<tr>
<td>kc3</td>
</tr>
<tr>
<td>mw1</td>
</tr>
<tr>
<td>pc1</td>
</tr>
</tbody>
</table>

Our parameter searching strategy, as described in Fig. 1, can be adopted for tackling different problems. A general trend is that more imbalanced data sets need the algorithm to focus on the minority class more aggressively. For instance, in RUS, the percentage of the majority-class examples left for training in pc1 (the most imbalanced data) is much lower than that in mc2 (the least imbalanced data); in THM, the best misclassification cost of the minority class in pc1 is much higher than that in mc2.

To sum up, among the seven SDP and class imbalance learning methods, Naive Bayes is the winner according to PD, and AdaBoost.NC is the winner based on the overall performance. From the viewpoint of the problem nature, the robustness of Naive Bayes to class imbalance in SDP implies that the extracted features are appropriate for describing the attributes of software code modules. Although the prior probability of the defect class is low, the statistical distribution of this class can be represented quite well by those features. Its posterior probability is thus rectified by summing the information from multiple features. Moreover, as claimed in [3], the defects may be actually associated in some log-normal way to the features. AdaBoost.NC is less aggressive in finding defects, as it tries to maintain the performance balance between classes. For highly imbalanced data, RUS-bal and Naive Bayes tend to be good choices. Random Forest is shown to be ineffective, as the bootstrapping training strategy and the tree learner are sensitive to class imbalance [9], [53]. The other class imbalance learning methods, RUS, THM, and SMB, are all better than Random Forest. Based on the above results, Naive Bayes is recommended when a high hit rate of defects is more important (even at the cost of higher false alarm rates); otherwise, AdaBoost.NC could be a good choice.

**A. Dynamic AdaBoost.NC**

Given the complementary strengths of Naive Bayes and AdaBoost.NC, it would be ideal to find a predictor that combines their advantages. For practical algorithm application, it is also desirable to reduce the number of pre-defined parameters. In Section IV-A, we applied a parameter searching strategy to each method to obtain an appropriate setting. In this section, we propose a dynamic version of AdaBoost.NC that can adjust its parameter...
automatically during training, with the goal of improving or at least maintaining the effectiveness of AdaBoost.NC without the exhaustive search for the best parameter. Similar to the parameter searching strategy we described before, we still split data into three parts: a training set, a validation set, and a testing set. We make use of the sequential training framework of AdaBoost.NC to adjust its main parameter $\lambda$ each time we build the individual classifier, based on a chosen accuracy performance criterion (we use the balance measure in this section). We set an initial value of $\lambda$, such as 9, before the training starts. If the next classifier has a better balance on the validation set, we increase $\lambda$ by 1 to emphasize the minority class further. Otherwise, we reduce $\lambda$ by 1. By doing so, the minority class performance can be boosted as much as possible without hurting the overall performance. The main steps of the algorithm are described as follows.

**Z** Data set to be learnt

**\infty** Decrement value used in the parameter searching strategy

**$\lambda$** Penalty strength for encouraging ensemble diversity

**ambd** Ambiguity term that assesses the ensemble diversity

Given data set $\{ (x_i,y_i), \ldots (x_n,y_n) \}$ and a chosen performance criterion $\text{Acc}$, initialize data weights $D_0(x_i) = 1/m$; penalty term $p_0(x_i) = 1$; plenty strength $\lambda$.

1. Training epoch $t=1, 2, \ldots, T$:
2. Step 1) Train weak classifier $f_t$ using distribution $D_t$.
3. Step 2) Get weak classifier $f_t$; $X \rightarrow R$.
4. Step 3) Calculate the penalty value for every example $x_i$.
   $$p_t(x_i) = 1 - \text{ambd}(x_i)$$
5. Step 4) Calculate $f_t$’s weight $\omega_t$ by error and penalty
   $$\omega_t = \frac{1}{2} \log \frac{\sum_{x_i \in \text{Pos}} D_t(x_i) (p_t(x_i))^2 (1 + k_t(x_i)y_i)}{\sum_{x_i \in \text{Neg}} D_t(x_i) (p_t(x_i))^2 (1 - k_t(x_i)y_i)}$$
   which is equivalent to
   $$\omega_t = \frac{1}{2} \log \left( \frac{\sum_{x_i \in \text{Pos}} D_t(x_i) (p_t(x_i))^2}{\sum_{x_i \in \text{Neg}} D_t(x_i) (p_t(x_i))^2} \right)$$
6. For discrete label outcome.
7. Step 5) If $\text{Acc}(f_t) \geq \text{Acc}(f_{t-1})$, then $\lambda = \lambda + 1$;
   else $\lambda = \lambda - 1$
8. Step 6) Update data weights $D_{t+1}$ and obtain new weights $D_{t+1}$ by error and penalty:
   $$D_{t+1}(x_i) = \frac{D_t(x_i) (1 + k_t(x_i)y_i)}{Z_t}$$
   Where $Z_t$ is a normalization factor.
9. Output the final ensemble:
   $$\text{Ensemble} = \text{sign}(\sum \omega_t f_t(x_i))$$

An ambiguity term ($\text{ambd}$ in step 3) decomposed from the classification error function is introduced into the weight-updating rule of Boosting (in step 6). It assesses the average difference between ensemble and its individuals, and is used to penalize training examples causing low diversity. The parameter $\lambda$ controls the strength of applying the penalty, which is given an initial value before the algorithm start and then adjusted according to the measure “Acc” (step 5). Both accuracy and ensemble diversity are taken into account through the sequential training.

The improved AdaBoost.NC (denoted by “DNC”) is compared to the original AdaBoost.NC (BNC) and to the Naïve Bayes (NB) methods, based on Student’s t-test at a confidence level of 95%. Its performance outputs and the significance test results are shown in Table VI. The superscripts (subscripts) of b and n indicate that DNC’s performance is significantly better (worse) than BNC (for b) and NB (for n) respectively.

In terms of the defect detection rate (PD), although BNC is still worse than Naïve Bayes in 9 out of 10 cases, it outperforms BNC in 5 cases significantly, and is competitive in the rest. It shows that AdaBoost.NC can find more defects by changing its parameter dynamically during training than that with a fixed parameter.

In terms of the overall performance, DNC performs significantly better than or at least comparably to BNC and Naïve Bayes in all cases according to balance. According to G-mean, DNC performs significantly better than or comparable to BNC and Naïve Bayes in 9 out of 10 cases. According to AUC, DNC outperforms BNC in 3 cases, is outperformed by BNC in 4 cases, and is comparable to BNC in 3 remaining cases; DNC performs significantly better than or comparable to Naïve Bayes in 8 cases. These results show that, in general, DNC has better or at least comparable overall performance to BNC and Naïve Bayes. It can improve PD and overall performance without fixing the best parameter prior to learning.

V. CONCLUSIONS

The objective of SDP is to find as many defective software modules as possible without hurting the overall performance of the constructed predictor (e.g. without increasing the false alarm rate). The imbalanced distribution between classes in SDP data is a main cause of its learning difficulty, but has not received much attention. This paper studied whether and how class imbalance learning can facilitate SDP. We investigated five class imbalance learning methods, covering three types (undersampling, threshold-moving, Boosting-based ensembles), in comparison with two top-ranked predictors (Naïve Bayes, and Random Forest) in the SDP literature. They were evaluated on ten real-world SDP data sets with different range of data sizes and imbalance rates. To ensure the results presented in this paper are of practical value, five performance measures were considered, including PD, Pr, balance, G-mean, and AUC.

To fully discover the potential of using class imbalance learning methods to tackle SDP problems, we first searched for the best parameter setting for each method based on the balance, G-mean, and AUC measures, because determining how much degree the defect class...
should be emphasized is crucial to their final performance. Then, random under sampling, balanced random under sampling, threshold-moving, AdaBoost.NC, and SMOTEBoost were compared with Naive Bayes with the log filter, and Random Forest. The results show that AdaBoost.NC presents the best overall performance among all in terms of balance, G-mean, and AUC. The balanced random under sampling has a better defect detection rate (PD) than the other class imbalance learning methods, but it is still not as good as Naive Bayes. The balance and G-mean measures are shown to be better performance criteria than AUC for deciding algorithm parameters.

To further improve AdaBoost.NC and overcome the parameter setting issue, we proposed a dynamic version of AdaBoost.NC that can adjust its parameter automatically. It shows better PD and overall performance than the original AdaBoost.NC. It offers the advantage of reduced training time and more practical use, as no pre-defined parameters of emphasizing the minority class are required. Future work from this paper includes the investigation of other base classifiers. Currently, this paper only considered C4.5 decision trees. In addition, it is important to look into more practical scenarios in SDP, such as learning from data with very limited defective modules and many unlabeled modules (semi-supervised), and defect isolation to determine the type of defects (multi-class imbalance).

REFERENCES


AUTHORS

Vivek Yoganand is a faculty member at VelTech Multitech Institute of Computer Engineering Education and Research, Chennai, India. He teaches Software Testing and Operating System Techniques. He received his B.E in Computer Science Engineering from Anna University, Chennai (India), and M.Tech in Multimedia Technology from SRM University, Chennai (India) and MBA in International Business from Bharathiar University, Coimbatore (India) apart from that he completed Bachelor of General Law from Annamalai University, Coimbatore (India). His current research interest is in software testing, image processing and reliability.

Premnath is a faculty member at Veltech Multitech Institute of Computer Engineering Education and Research, Chennai (India). He teaches Software Engineering and Computer Networks. He received his B.E in Computer Science and Engineering from Madras University, Chennai (India) and M.Tech in Computer Science and Engineering from Vellore Institute of Technology (VIT) University, Vellore (India). His current area of research interest is in Computer Networks and Software Engineering

Sathya is a faculty member at VelTech Multitech Institute of Computer Engineering Education and Research, Chennai, India. She teaches data ware housing and Data mining and Software Engineering. She received his B.Tech in Information Technology from Anna University, Chennai (India), and M.E Computer science and Engineering from Anna University, Chennai (India). Her current research interest is in web mining, Database, Data analysis.

Velmurugan is a faculty member at VelTech Multitech Institute of Computer Engineering Education and Research, Chennai, India. He teaches data ware housing and Data mining and Software Engineering. He received his B.E in electronics communication and engineering from Anna University, Chennai (India), and M.E Computer science and Engineering from Anna University, Chennai (India). His current research interest is in web mining, Database, Data analysis.