Detecting Security Bad Smells of Software by using Data Mining

Sara Alraddadi (✉ sara.alraddadi@outlook.com)  
King Abdulaziz University

Research Article

Keywords:

DOI: https://doi.org/

License: ☎️ This work is licensed under a Creative Commons Attribution 4.0 International License.  
Read Full License

Additional Declarations: No competing interests reported.
Detecting Security Bad Smells of Software by using Data Mining

Sara Alraddadi

Faculty of Computing and Information Technology, King Abdulaziz University, Street, Jeddah, Saudi Arabia.

Corresponding author(s). E-mail(s): sara.alraddadi@outlook.com;

Abstract
Bad smells of code can lead to significant software vulnerabilities that negatively affect the security attribute of software and make the software easily attacked. Method implementation is the lowest code level that produces bad smells in software. Several programming guidelines have been introduced to reduce the number of bad code smells. Some are related to security, while others are related to other software quality attributes. We can detect some of these guidelines by using quality metrics (e.g., the number of lines of code and the number of parameters). This project proposes applying data mining classifiers to automatically detect security code smells and non-security code smells based on software quality metrics. We developed eight models using eight classifiers: Logistic Regression, Random Forest, XG Boost, J48, SVM, Naive Bayes, LightGBM, and Neural Network. We evaluated the eight models using several performance measurements (e.g., accuracy and confusion metrics). The results showed that it could detect the security code smell from the non-security code smell with the highest accuracy of 95%.

Keywords: Code smells, Security code smells, Data Mining, Software quality assurance, Static analysis, Java code metrics.

1 Introduction
Code smells indicate that basic implementation principles were not followed in source code [1]. The code smell signifies the presence of a wide range of quality issues in the software. Implementation, design, and architecture are the
main subcategories that fall under the code smell in Java [2]. Some developers apply sub-optimal implementation decisions, negatively affecting the overall maintainability of software [1]. Many studies have shown how code smells are introduced and evolved and their influence on program comprehension [3]. Security smells may lead to security weakness and potentially to security breaches by recurring coding patterns [4].

Table 1 illustrates an example of method implementation code smells that affect the security of code. The row in Table 1 highlights that having an empty catch block is a risky practice to be implemented in Java and is associated with many defects. The code cannot handle exceptions in case of the presence of empty catch blocks; consequently, program failure could occur [5]. This method is seen as a security smell since it allows external attack errors/exceptions to go unnoticed. CWE-703: Improper Check or Handling of Exceptional Conditions is connected to this security smell [6].

<table>
<thead>
<tr>
<th>Security Code smell</th>
<th>Vulnerability</th>
<th>Definition</th>
<th>Possible Security attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>try { doExchange(); } catch (RareException e) { // this can never happen }</td>
<td>Empty catch block</td>
<td>The catch block in a try-catch statement in Java can be left empty by the developer.</td>
<td>This method is considered a security smell because it allows external attack errors/exceptions to go unrecognised. This security smell is linked to CWE-703: Improper Check or Handling of Exceptional Conditions. Additionally, if exception handling is done incorrectly or not, it might result in an inaccurate state or reveal additional information that can be useful to attackers [6].</td>
</tr>
</tbody>
</table>

Researchers have utilized the static analysis concept to develop tools that detect code smells. Some code smell detection tools have been proposed [7] [4], these tools have been developed in different ways, which might provide different interpretations or detection of code smells. Therefore, their results might not be highly accurate [8]. For example, some tools use different rules and metrics for code smell detection, affecting the number of detected code smells. Other researchers apply data mining techniques to detect code smells.
However, these techniques do not distinguish which quality attribute affects by a code smell.

According to Mike McConnell’s article [6], the longer a defect remains in a program or process, the more it will cost to fix. This also involves security defects/vulnerabilities. Acknowledge this fact; developers are doing their best to prevent them during the early stages of the software development cycle.

Therefore, identifying security code smell is essential to reduce the vulnerabilities and threats [4] [9]. Many researchers have developed tools to detect code smells [7] [4]. These tools have been developed in different ways, which might provide different interpretations or detection of code smells. However, their results might not be highly accurate. For example, some tools use different rules and metrics for code smell detection, affecting the number of detected code smells. According to [8] and [3], new approaches (e.g., data mining) should be used to develop detection tools and should consider not only software metrics but also the context, domain, size, and design of source code.

In this paper, we use data mining and software quality metrics to develop eight models that detect bad smell code concerning software security. First, we collect the dataset and label it. Then, we prepossess the data and split the data into 80% (Train Data) and 20% (Test Data). Finally, we train the model by using eight state-of-the-art classifiers. We evaluate the models and collect the confusion metrics. The accuracy results (Logistic Regression: 90.85%, Random Forest: 96.01%, XgBoost: 95.30%, SVM: 95.16%, J48: 93.02%, Naive Bayes: 77.13%, LightGBM: 94.72%, Neural Network: 95.91%) show the capability of data mining to detect the bad security smell in the source code, hence reducing the vulnerabilities in the source code.

The main contributions of the paper are:

- Predict bad security smell using eight different classifiers on method quality metrics.
- Build dataset based on method quality metrics associated with annotating security and non-security code smells.
- Evaluation and analysis of the eight models

This paper is organized as follows. Section 2 provides information about state-of-the-art (Related Work). Section 3 explains in detail the build of a dataset using the QSored database and eight well-known classifiers (Methodology). Section 4 discusses the 40 experiments we built for evaluation (Empirical Study). Section 5 answers our research questions (Results). Finally, Section 6 and 7 presents our Conclusion and discuss Future Work.

2 Related Work

Many researchers have worked on code smells. We classify them into three categories: 1) Building a static analysis tool to predict security code smell, 2) Applying data mining to detect any kind of code smells, and 3) Creating a dataset for all kinds of code smells.
2.1 Building a static analysis tool to predict security code smell

The article [4] documented the seven different smells in IaC. These smells include 1) granting admin privileges by default, 2) empty passwords, 3) hard-coded secrets, 4) invalid IP address binding, 5) suspicious comments (such as TODO or FIXME), 6) use of HTTP without TLS and 7) use of weak cryptography algorithms. These smells were identified through qualitative analysis on Puppet scripts in open-source repositories, making these smells limited to these puppet scripts and not verifiable on other scripts or languages. Their new research [10] mentioned that article [4] is not exhaustive and may not capture security smells that exist for other languages.

The author Rahman et al., who made the article [4], has also conducted a literature analysis [11] to detect security smells in the Python programming language.

In the article [12], the author was able to detect security smells in Android by accomplishing literature research and creating a tool that statistically identifies the security smells.

The article [6] focused on using Mining Software Repository in order to identify security smell in java code. Moreover, their analysis discovered that the probability of security smell existing in java code correlates with ordinal LOC, ordinal commit count, and ordinal author count.

The article [7] did an empirical study to understand better the relationship between code smells and vulnerabilities using Apache Tomcat software. The vulnerabilities were identified by using static code analysis. Also, the article [13] empirically examined the relationship between code smells and vulnerabilities. They have used metrics to classify code smells in the source code and then empirically examined the correlation between code smells and vulnerabilities. These articles back our claims by showing study results of the relationship between code smells and security vulnerabilities.

2.2 Applying data mining to detect any kind of code smells

Up until now, research has mainly zeroed in on testing different types of data mining to find one or two code smells in most cases [14] [15]. These distinguishing pieces of proof have been finished utilizing single model calculations, for example, choice trees, neural organizations and backing vector machines.

According to [14] [15], the random forest model has high scores in identifying code smells. It has been acknowledged as one of the best in identifying code smells in java projects in studies completed by researchers’ efficiency [14] [15]. The results of these studies highlighted that data mining brings good accuracy and efficiency [14] [15].

The article [16] used data mining approaches to research code smell detection. This study’s performance has been based on accuracy, F-Measure and
Detecting Security Bad Smells of Software by using Data Mining

AUCROC. Additionally, the article highlights that the data mining model’s performance depends on the dataset being used.

The article [8] had a large-scale study conducted where it applied 32 various machine learning algorithms to detect four code smell types, i.e., Data Class, Large Class, Feature Envy and Long Method. This study was done to understand better how well data mining techniques can be used for code smell detection. The study highlighted that nearly all classifiers in terms of accuracy and F-Measure had exceeded 95%. The best performance was from J48 and Random Forest.

This research paper [17] offers a novel approach to data mining-based code smell detection by recommending the use of WekaNose. While the approach is different, it is still experimental, and the article was missing techniques and analysis required for experiments.

2.3 Creating dataset for all kinds of code smells

2.3.1 Landfill: an Open Dataset of Code Smells with Public Evaluation:

The paper [18] LANDFILL is a Web-based platform for exchanging datasets related to code smell. It includes examples of five bad smells utilized from 20 open-source repositories. It contains two sorts of bad smells: three code smells (i.e. Blob, Feature Envy and Long Method) and two test smells (i.e. General Fixture and Eager Test). The goal of LANDFILL is to encourage the collecting and sharing of code smell datasets in the software engineering and MSR research communities.

2.3.2 QScored: A Large Dataset of Code Smells and Quality Metrics:

The paper [19] presents the QScored database as it uniquely contains information such as code quality about 86,000 C# and Java GitHub repositories. The information on code quality covers seven kinds of detected architecture smells, 20 kinds of design smells, eleven kinds of implementation smell, and 27 commonly used code quality metrics used at project, package, class and method levels.
3 Methodology

Figure 1 depicts an overview of our methodology. First, we build our dataset using the QScored database [19]. Then, we annotate each instance (e.g., security bad smell or non-security bad smell). We preprocess the dataset to remove unwanted data, such as duplicated data, and to handle missing data. After that, we split the data into 80% Train Data and 20% Test Data. To achieve our goals, we apply eight well-known classifiers (Logistic Regression, Random Forest, XgBoost, SVM, Naïve Bayes, J48, LightGBM, and Neural Network) for experiment purposes and build eight trained models. These models will show the ability to classify security and non-security bad smells. The Test Data will deliver the final real-world data check to ensure that the Trained Model is capable and efficient in detecting security smells.

3.1 Collect Dataset

We use the QScored database [19] to create our dataset (SecurityCodeSmell.csv). Figure 2 shows that the QScored database contains 86,000 C# and Java GitHub repositories. The code quality data includes seven types of detected architecture smells, twenty types of design smells, eleven types of implementation smell, and 27 widely used code quality metrics at project, package, class, and method levels.
Based on the DesigniteJava tool [20], there are several metrics of the code: 1) solution-level metrics, 2) project-level metrics, 3) Class-level Metrics, and 4) method-level metrics. The method-level metrics are one of the static measurements of source code quality attributes such as security. In contrast, other metrics (e.g., class-level metrics) are related to other aspects of the software. For example, class-level metrics are related to the software design. Therefore, we only consider the method-level metrics. Therefore, we only consider the data from the following table that is only related to java code:

**Table 2: QScored database terms and definitions for method-level metrics**

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Total number of source-code lines in the method.</td>
</tr>
<tr>
<td>CC</td>
<td>Cyclomatic Complexity: is a commonly used metric to measure the complexity of a method.</td>
</tr>
<tr>
<td>PC</td>
<td>Parameter Count.</td>
</tr>
</tbody>
</table>

In the Table 3, DesigniteJava has defined thresholds for method-level metrics [20]. Whenever the metric value crosses the corresponding threshold, Designite marks it as a metric violation. We have three method-level metrics 1) Method LOC, 2) Method CC, and 3) Method PC.

For the Method LOC, if the metric value is greater than or equal to 100, it considers security code smell (Long Method). Moreover, for the Method CC, if the metric value is greater than or equal to 8, it considers security code smell (Complex Method). Finally, for Method PC, if the metric value is greater than or equal to 5, it considers security code smell (Long parameter List).

**Table 3: DesigniteJava defines the thresholds for method-level metrics**

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Threshold Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>100</td>
</tr>
<tr>
<td>CC</td>
<td>8</td>
</tr>
<tr>
<td>PC</td>
<td>5</td>
</tr>
</tbody>
</table>

Then, in order to view the dataset files, we have used software called SQLPro Studio, which can be downloaded from [21].

**3.1.1 Adding additional attribute**

In order to detect security code smell from non-security code smell, we had to add two columns in the dataset labelled as length of identifier and length of statement. The length of identifier is related to the long identifier smell, and length of statement is related to the long statement smell. We have extracted the values of length of identifier and length of statement from the implementataion description of the long identifier and log statement.
3.1.2 Missing data handler

As we have added two new columns to the dataset, it appears to have missing data. So, to fix that, we have decided to fill the missing data with zero, average value, negative one, standard deviation value, and the above value. So in total, we got five different datasets:

1. Missing Values Filled with Zero
2. Missing Values Filled with Average Value
3. Missing Values Filled with Negative One
4. Missing Values Filled with Standard Deviation
5. Missing Values Filled with Value Above

3.1.3 Our security code smell dataset

In the dataset, we have six columns, including the target columns, and there are 25300 rows. Figure 3 shows that we have, in total, six columns in our dataset. Furthermore, the data type for the target is an object, and for Method_cc, Method_loc, Method_pc, length of identifier, and length of the statement have type int64. Finally, we have ensured that we don’t have non-null values in our dataset.

![Table 4](image)

**Table 4:** Additional details about our security code smell dataset

Table 4 indicates that the data metrics for each of the features show distinct patterns that could be the underlying factors in separating all the five classes. For all the features except Method_pc, the standard deviation is high, indicating a large degree of variation in the features. Interestingly, each feature has a large range between the min value and the max, even for the 75 percentile value and the max. This indicates potential outliers in the features dataset, which may be worth accounting for or could help more complex models (mainly neural networks) tease out relationships. The most prominent of these skewed distributions are the final two feature columns, ‘length of identifier’ and ‘length of statement’, which contain a large number of 0’s. Exploring these data distributions and relationships is key to understanding the differences between each labelled class and also will aid in improving model performance.
Detecting Security Bad Smells of Software by using Data Mining

<table>
<thead>
<tr>
<th>count</th>
<th>Method_cc</th>
<th>Method_loc</th>
<th>Method_pc</th>
<th>length of identifier</th>
<th>length of statement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25300.000000</td>
<td>25300.000000</td>
<td>25300.000000</td>
<td>25300.000000</td>
<td>25300.000000</td>
</tr>
<tr>
<td>mean</td>
<td>13.097628</td>
<td>82.220079</td>
<td>2.990474</td>
<td>8.319051</td>
<td>37.498538</td>
</tr>
<tr>
<td>std</td>
<td>25.420636</td>
<td>160.311431</td>
<td>3.196113</td>
<td>17.183639</td>
<td>88.639647</td>
</tr>
<tr>
<td>min</td>
<td>1.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>25%</td>
<td>1.000000</td>
<td>10.000000</td>
<td>1.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>50%</td>
<td>7.000000</td>
<td>39.000000</td>
<td>2.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>75%</td>
<td>15.000000</td>
<td>107.000000</td>
<td>5.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>max</td>
<td>1242.000000</td>
<td>8600.000000</td>
<td>129.000000</td>
<td>93.000000</td>
<td>962.000000</td>
</tr>
</tbody>
</table>

### 3.2 Label the Dataset

The QScored database uses eleven code smells to annotate each row (see Table 5). Some of these smells are not related to metrics (may need static analysis tools). For example, to detect if a code has an empty catch, we need to use a static analysis tool to reveal it. Since our work is to experiment with method quality metrics for prediction, we only consider instances with bad smells related to method quality metrics. Therefore, this is where our work is to detect whether an instance should be classified as security or non-security. The following section explains the labels associated with method quality metrics.

However, not all implementation method metrics are related to security. Table 5 shows which metric is used to indicate security code smells and which one is not.

**Table 5: Implementation method-related code smells**

<table>
<thead>
<tr>
<th>Code Smells</th>
<th>Measured by Method Metrics?</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract function call from constructor</td>
<td>No</td>
<td>It can not be detected through quality metrics. So, we need to use a static analyser.</td>
</tr>
<tr>
<td>Empty catch block</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Magic number</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Complex conditional</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Duplicate code</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Missing default</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Complex Method</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Long identifier</td>
<td>Yes</td>
<td>We can use the quality metrics to measure these code smells.</td>
</tr>
<tr>
<td>Long method</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Long Parameter List</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Long statement</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### 3.2.1 Security bad smell

- **Long Method:**
  A long method is a method that has lines of code greater than or equal to 100 [20]. It is considered one of the security bad smells as more lines of code and hence security bugs[22].
  
  The first problem that may appear in a long method is that a method does more than one task (e.g., validate inputs and manipulate inputs). The second issue that may emerge in a long method is duplicated code. These issues become worse during maintenance. Having a long method may affect security during maintenance [23][24].

- **Complex Method:**
  The software metric that is used in identifying how complex a program is referred to as cyclomatic complexity, and it is a quantitative measure. For example, the number of independent paths through source code provides the complexity of the code [25]. If there is a high number of if-statements and loops in a source code, then cyclomatic complexities are more likely to occur, which can also affect the security of a program [26]. A number of studies have found that increased complexity in systems is the main reason for vulnerabilities to arise in software systems [26][27]. The cyclomatic complexity of a source code should not exceed or be equal to 8 [20].

- **Long Parameter List:**
  A Long Parameter lists, also known as data clumps, occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data[24]. It is considered one of the bad smells when a method has five or more parameters [20] as it negatively affects the maintainability of software system[23], which may lead to security issues.

### 3.2.2 Non-security bad smell

- **Long Identifier:**
  This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) with more than 30 characters in a program. The longer the identifiers, the more potential they are to be meaningful and, thus, the more understandable the program. Although it is considered a bad smell [24], this smell is not related to security.

- **Long Statement:**
  A Long statement smell occurs when a statement is overly long. We consider a statement lengthy when it exceeds 100 characters. A very lengthy statement indicates that the programmer is on the wrong track. If there is a need to add a new condition, the programmer must locate and modify all
the relative codes. Therefore, a method with too many statements is a technical liability and is susceptible to several issues. Apart from all that, long statements are harmless to security.

3.3 Preprocess the Dataset

When we review the first 25000 instances of the dataset, we found that 3477 instances have bugs. Examples of the dataset bugs for the Complex Method, Long Method and Long Parameter List are shown in Tables 6, 7, and 8, which shows that there are differences between the implementation description and the method metrics: CC, LOC and PC.

Table 6: Some of the QScored database bugs (Complex Method)

<table>
<thead>
<tr>
<th>Project Id</th>
<th>Implementation Name</th>
<th>Implementation Description</th>
<th>Method CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8bdff1a3b-b03c-4de4-a9c1-dec534dc7ef</td>
<td>Complex Method</td>
<td>Cyclomatic complexity of the method is 13</td>
<td>2</td>
</tr>
<tr>
<td>88930ce79-bd7d-4a30-9677-0f145c51087</td>
<td>Complex Method</td>
<td>Cyclomatic complexity of the method is 8</td>
<td>10</td>
</tr>
<tr>
<td>df4eb7d0-a820-4b31-abdd-f8e81f6b016</td>
<td>Complex Method</td>
<td>Cyclomatic complexity of the method is 5</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 7: Some of the QScored database bugs (Long Method)

<table>
<thead>
<tr>
<th>Project Id</th>
<th>Implementation Name</th>
<th>Implementation Description</th>
<th>Method LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8278e991-1a90-474e-b452-7c7f372d11ff</td>
<td>Long Method</td>
<td>The method has 136 lines of code.</td>
<td>12</td>
</tr>
<tr>
<td>09b3b61c-6857-4979-9b8d-06d456e05bf</td>
<td>Long Method</td>
<td>The method has 140 lines of code.</td>
<td>82</td>
</tr>
<tr>
<td>f1951298-2b92-4a39-96f1-a14b885c9a3</td>
<td>Long Method</td>
<td>The method has 122 lines of code.</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 8: Some of the QScored database bugs (Long Parameter List)

<table>
<thead>
<tr>
<th>Project Id</th>
<th>Implementation Name</th>
<th>Implementation Description</th>
<th>Method PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>a78a531-3306-4639-9687-31e8233e658</td>
<td>Long Parameter List</td>
<td>The method has 5 parameters.</td>
<td>8</td>
</tr>
<tr>
<td>9f644f9a-47f5-424a-a5d4-6cfd12dfe43d</td>
<td>Long Parameter List</td>
<td>The method has 14 parameters.</td>
<td>19</td>
</tr>
<tr>
<td>lee6fe4a-1ac5-4c43-9ab1-defe906bfe9</td>
<td>Long Parameter List</td>
<td>The method has 13 parameters.</td>
<td>8</td>
</tr>
</tbody>
</table>
We use the first 5000 instances that do not have bugs. Also, the dataset has null values, so we applied different techniques (discussed in the evaluation section).

Since the dataset is inaccurate, we use the first 5000 instances that do not have bugs for each target: Security code smell (Complex Method), Security code smell (Long Method), Security code smell (Long Parameter List), Non-security code smell (long identifier), and Non-security code smell (long statement), all shown in Figure 4. Also, the dataset has null values, so we applied different techniques (discussed in the Evaluation section).

![Fig. 4: Our five targets values](image)

### 3.3.1 Normalisation

In some cases, the data variance is too much. In this case, one small change in the dataset will lead to a lower accuracy, which is unacceptable. Due to this, we perform normalisation of the data where we can have a good distribution of the data with low variance. Normalise can be of different types. One of those is Min-Max normalisation, which takes a dataset’s minimum and maximum values and normalises the dataset using those values. In our dataset, we have performed this to have normal distribution to guarantee we get high accuracy.

### 3.4 Split Data

We have a total of 25000 lines of the dataset. The split was performed randomly, and we set 80% of the dataset as train data and 20% as test data.

### 3.5 Apply Data Mining Classifiers

We use eight classifiers to build eight models. Each one of them can detect security code smell. In this section, we are going to discuss each classifier in detail.

1. Logistic Regression classifier:
   Logistic Regression classifier [28], is one of the fundamental data mining algorithms. By default, it is constrained to binary classification using a logistic function to separate two classes via a threshold. Our data is simple and easy to understand, so Logistic Regression can be an excellent fit
to predict the outcome. The modelling methodology assumes that these inputs are linearly separable and normally distributed.

2. Random Forest Classifier:
The Random Forest algorithm [29] creates trees from the subset of data and then takes the average from each subset to find the final output. In the case of Random Forest, if we can increase the number of trees, it would help to prevent overfitting. Using Random Forest Classifier for our dataset takes less time to train and can also provide higher accuracy in a larger dataset.

3. XgBoost Classifier:
XgBoost is a classification [30] technique that ensembles multiple models to predict the outcome using a boosting method. The most significant advantage of XgBoost uses Boosting process to develop various models where the strong learners take information from weak learners to learn the dataset better, which helps the overall model boost.

4. SVM:
SVM stands for Support Vector Machine [31]. It has two variations, one for regression and one for classification. Here, we use the Support Vector Classifier (SVC), built for classification. Using SVC in our multi-class classification problem, it will be easier to predict the classes if we can make separate planes for each target level.

5. J48:
J48 [32] is nothing but a Decision Tree classifier. This is the base model for creating Random Forest or any other algorithm following tree structure. Its algorithm is straightforward to implement and takes a small amount of time to train. Moreover, this model gives good accuracy as it considers the output directly while calculating the entropy for each split.

6. Naive Bayes:
Naive Bayes is a probabilistic model characterizing the probability of an item belonging to a particular class [33]. The way to create Naive Bayes models in Python is by calling GaussianNB class.

7. LightGBM Classifier:
The LightGBM classifier [34] is also a gradient boosting algorithm like XgBoost. However, the XgBoost grows trees level-wise across depths, and LightGBM grows each leaf first to the maximum depth specified. This is computationally faster as the pruning time of the final returned tree is faster when only one node has to be searched from root to leaf [35].

8. Neural Network:
Neural Network [36] is one of the most sophisticated data mining algorithms. The architecture of the Neural Network includes a neuron which
Detecting Security Bad Smells of Software by using Data Mining

is one of the most critical parts of the model. One of the Neural Network advantages is that it can make decisions and does not need to be reprogrammed [37].

4 Empirical Study

We did 40 experiments for evaluation. We used eight classifiers to create 40 models. 1) Experiment 1: missing values filled with zero, 2) Experiment 2: missing values filled with the average value. 3) Experiment 3: missing values filled with a negative one, 4) Experiment 4: missing values filled with standard deviation, 5) Experiment 5: missing values filled with value Above. Each of the five experiments contains eight sub-experiments.

4.1 Evaluation

The ultimate goal of this project is to decrease the number of vulnerabilities that could appear because of security bad smells concerning method quality metrics. The two main hypotheses are: 1) a data mining classifier can predict security bad smells using method quality metrics, and 2) filling missing values with zero in the dataset increases the accuracy of a trained model to predict security smell from method quality metrics.

Each type of code smell is defined according to its impact, scope, and granularity. The implementation types of code smells are usually restricted to specific scope and impact. For instance, the implementation type of code smell is related to the following methods [38] [39] [19]:

- Abstract function call from constructor
- Complex conditional
- Complex Method
- Empty catch block
- Long identifier
- Long method
- Long Parameter List
- Long statement
- Magic number
- Duplicate code
- Missing default

4.1.1 Retrieve the Dataset

Datasets taken directly from the source may have inconsistencies, errors, or both that are not ready to be considered in the data mining process. At first, inquired about retrieving all the data, which exceeded 26 million Java code related only. Since we are only focusing on the Java programming language, we only have extracted the Java-related projects.

After that, in order to extract data from the QScored database, we have to use SQL query and then store the results in a CSV file. The way we
have extracted the dataset. First, by extracting the Java-related projects from these tables, 1) project, 2) implementation_smell, 3) class_metrics, 4) method_metrics, and 5) componentmetric. In the Query, we used "distinct" to prevent data duplication. Lastly, we stored all implementation code smells related to java in one CSV format file called (SecurityCodeSmell.csv)

4.1.2 Performance Measures

To analyze the results, we used confusion matrix acronyms (True Positive, True Negative, False Positive, and False Negative) as follows:

Classification models produce multiple categorical outputs. The majority of error measures will calculate the overall amount of error in our model, but we are incapable of identifying specific instances of errors in our model. The model may misclassify some categories more often than others, but we are unable to detect this with a common accuracy measure [40].

With the Confusion Matrix table layout, it’s simple to see where the predictions aren’t true. In other words, the "confusion" in the model can be a result of the matrix function to depict the data model’s incapability to classify data correctly, thus the name confusion matrix. These performance measures are used to estimate the prediction model’s performance [41].

4.1.3 Our research questions

• RQ1: Are there some data mining classifiers that can detect security bad smell with respect to method quality metrics with high accuracy?
• RQ2: Which dataset (with missing values) of method quality metrics will train models with high accuracy?

To answer the first research question (a data mining classifier can detect security code smell and non-security code smell), we applied the eight well-known classifiers (see section 3.5). Furthermore, we use the dataset filled with zero missing values as part of our experimental study. We used the Python programming language and the Google Colab tool (https://github.com/Sara-Alraddadi/Detecting-Security-Bad-Smells-of-Software-by-using-Machine-Learning).

To answer the second RQ, we use five strategies: 1) Missing Values Filled with Zero, 2) Missing Values Filled with Average Value, 3) Missing Values Filled with Negative One, 4) Missing Values Filled with Standard Deviation, and 5) Missing Values Filled with Value Above to handle missing values in our dataset.

In both RQs, we utilized four performance measurements (Accuracy, Precision, Recall, and F1 Score) as part of our evaluation. Accuracy will help to indicate which classifier is better than the other. Precision helps to tell if a
model can indicate a positive class. Recall helps to tell if a model can indicate a negative class. F1-score helps identify whether the data mining model is good enough for both positive and negative classes.

5 Results

5.1 RQ1: Are there some data mining classifiers that can detect security bad smell with respect to method quality metrics with high accuracy?

To answer RQ1, the data mining classifiers can detect security bad smells with respect to method quality metrics with high accuracy. Figure 6 shows the result of the accuracy of the eight classifiers. The average of all used classifiers is (92.26%). Thus, the data mining classifiers can be used to predict security code smells. The highest accuracy is (96.01%) achieved by the Random Forest model that’s because the Random Forest algorithm takes the average or mean value of the yield of various trees to find the final output. It can also decrease the dataset’s overfitting and increase accuracy [42].

Table 9 shows the exact predictions made by Random Forest, XgBoost, SVM, J48, LightGBM and Neural Network models. The models made the wrong prediction on the third row shown in Table 9 because the same row has two bad smells, and then the model will end up choosing one of them. Additionally, it is possible to have two bad smells in one method. The cyclo-matic complexity method mostly has more lines of code which tend to cause long method code smell, so there is a relationship between complex method and long method [22].

Table 9: The Random Forest, XgBoost, SVM, J48, LightGBM and Neural Network models data prediction

<table>
<thead>
<tr>
<th>Method_{cc}</th>
<th>Method_{loc}</th>
<th>length of identifier</th>
<th>length of statement</th>
<th>Actual Result</th>
<th>Predicted Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>121.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Security code smell (Long Method)</td>
<td>Security code smell (Long Method)</td>
</tr>
<tr>
<td>14.0</td>
<td>111.0</td>
<td>1.0</td>
<td>0.0</td>
<td>Security code smell (Long Method)</td>
<td>Security code smell (Long Method)</td>
</tr>
<tr>
<td>32.0</td>
<td>126.0</td>
<td>4.0</td>
<td>0.0</td>
<td>Security code smell (Long Method)</td>
<td>Security code smell (Long Method)</td>
</tr>
<tr>
<td>4.0</td>
<td>24.0</td>
<td>2.0</td>
<td>0.0</td>
<td>Non security code smell (Long Statement)</td>
<td>Non security code smell (Long Statement)</td>
</tr>
</tbody>
</table>

On the other hand, the Naive Bayes model achieves the lowest accuracy. Table 10 shows the outcome of the predictions made by the Naive Bayes model. The reason why the Naive Bayes model made the wrong prediction in the first row is most likely due to the assumption of feature independence made by the modeling procedure, which is not necessarily true for this dataset [43].
Table 10: The Naïve Bayes data prediction

<table>
<thead>
<tr>
<th>MethodLOC</th>
<th>MethodLOC</th>
<th>MethodLOC</th>
<th>length of identifier</th>
<th>length of statement</th>
<th>Actual Result</th>
<th>Predicted Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>121.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Security code smell (Long Method)</td>
<td>Security code smell (Complex Method)</td>
</tr>
<tr>
<td>14.0</td>
<td>111.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Security code smell (Long Method)</td>
<td>Security code smell (Complex Method)</td>
</tr>
<tr>
<td>32.0</td>
<td>126.0</td>
<td>4.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Security code smell (Complex Method)</td>
<td>Security code smell (Complex Method)</td>
</tr>
<tr>
<td>4.0</td>
<td>24.0</td>
<td>2.0</td>
<td>0.0</td>
<td>156.0</td>
<td>Non security code smell (Long Statement)</td>
<td>Non security code smell (Long Statement)</td>
</tr>
</tbody>
</table>

Figure 5 illustrates the confusion metrics for eight classifiers, showing that the non-security code smell (Long Statement) and non-security code smell (Long Identifier) classes are relatively easy to distinguish. In contrast, the security code smell (Complex Method) and security code smell (Long Method) classes are the most confusing for each model. The tree-based models (Random Forest, XGBoost, and Light GBM) and neural networks perform very well on this dataset for code smell detection. It seems models with more tunable parameters perform better, though choosing the best performing may be situational dependent. For example, suppose you want more computational speed in generating output. In that case, the Light GBM may be the best to use, whereas if you want better performance biased to a particular class, one of the other models may be better to select.
5.2 RQ2: Which dataset (with missing values) of method quality metrics will train models with high accuracy?

To answer RQ2, we have considered only using the dataset with missing values imputation with zero as it is a safer approach and achieves the best performance compared to the other datasets.

5.2.1 Missing Values Filled with Zero

The first method we tried to handle missing values was filling them with zeros. Then we trained all eight data mining models using the zero-filled data and evaluated their performance in terms of performance metrics, including accuracy, precision, recall, and f1-score. The comparative analysis of the different models we used is shown in Figure 6. The figure shows that the Random Forest model provides the highest accuracy of 96.01% as compared to other models. Also, we can see that the Naive Bayes model has the lowest accuracy of
77.13%. We can see a similar trend in terms of other performance metrics. For instance, the Random Forest has the highest score of 96.42%, and the Naive Bayes has the lowest score of 80.79%. Also, Random Forest provides superior performance in terms of recall and f1-score with values of 96.00% and 95.96%, respectively.

Similarly, the Naive Bayes model provides the worst performance in terms of recall and f1-score with values of 77.13% and 75.12%, respectively. This suggests that when we filled the missing values with zeros, Random Forest provided the best performance of all metrics, and the Naive Bayes model performance was worst in all metrics in this case. From Figure 6, we can also see that all other models are providing a comparable performance in terms of different metrics except Logistic Regression, which is performing relatively better than the Naive Bayes. However, its performance is comparatively much lower than other models.

![Fig. 6: Performance comparison of different data mining models in terms of accuracy, precision, recall, and f1-score, when missing values are filled with zeros.](image)

**5.2.2 Missing Values Filled with Average Value**

Here we present the performance evaluation of eight data mining models on our dataset when missing values are filled with the average value. All models are evaluated in terms of performance metrics, including accuracy, precision, recall, and f1-score; the results are shown in Figure 7. It is clear from the figure that the LightGBM model provided the best performance in terms of all metrics; specifically, it provided an accuracy of 95.47%, a precision of 95.68%, recall of 95.47%, and an f1-score value of 95.45%. Also, Figure 7 highlights that the logistic regression models provided the worst performance when missing data is filled with the average value. As it provided the lowest accuracy of 60.53% and we can see that the performance of logistic regression is lowest in terms of other metrics as well, i.e., it provides a precision of 58.50%, recall score of 60.53%, and f1-score of 57.03%. Also, from Figure 7, we can see that Random Forest, XgBoost, J48, and Neural Network are providing a comparable performance as that of LightGBM. However, the performance of Naive Bayes and SVM models is relatively low. The best performing model in case of missing values filled
with average value is, therefore, LightGBM and the worst model is logistic regression.

Fig. 7: Performance comparison of different data mining models in terms of accuracy, precision, recall, and f1-score, when missing values are filled with average value.

5.2.3 Missing Values Filled with Negative One

This section presents the experimental results of eight data mining classifiers when the missing values are replaced with $-1$. The results of all models in terms of different performance metrics (that are used above) are presented in Figure 8. The figure shows that the Random Forest model is providing superior performance in terms of all metrics, i.e., it provides an accuracy of 95.45%, a precision of 95.95%, recall of 95.45%, and an f1-score of 95.40%. The worst performing model, in this case, is the Neural Network, which provides the lowest performance in terms of all metrics as we are getting an accuracy of 67.59%, a precision of 66.62%, a recall of 67.58%, and an f1-score of 64.87%. Also, from Figure 8, we can see that XgBoost, SVM, J48, and LightGBM are providing a comparable performance as that of Random Forest. However, the performance of Logistic Regression and Naive Bayes models is relatively less. Also, we can see that the performance of the Logistic Regression model is comparatively better than the Naive Bayes model and much better than the Neural Network.
Fig. 8: Performance comparison of different data mining models in terms of accuracy, precision, recall, and f1-score, when missing values are filled with negative one (i.e., −1).

### 5.2.4 Missing Values Filled with Standard Deviation

In addition to filling the missing values with 0, -1, and average values, we also filled them using standard deviation. In this section, we evaluate all data mining models used for smell classification when missing values are filled with standard deviation. As we did before, we evaluated the performance of all models in terms of different performance metrics (i.e., accuracy, precision, recall, and f1-score). The results are depicted in Figure 9; the figure highlights that the Random Forest model is providing the best performance in terms of all metrics as compared to all other models. Similarly, Naive Bayes models perform worst on data in which missing values are filled with standard deviation. Superficially, Random Forest provides an accuracy of 95.99%, a precision of 96.41%, a recall of 95.98%, and an f1-score of 95.95%.

On the other hand, the Naive Bayes model provides the lowest accuracy of 83.64%, the precision of 87.18%, recall of 83.63%, and f1-score of 82.22%. Also, if we look at Figure 9, we see that all other models (except Naive Bayes and Logistic Regression) provide comparable performance as that of Random Forest. Also, it is clear that the performance of Logistic Regression is slightly better than the Naive Bayes model.

![Performance comparison of different data mining models](image)

Fig. 9: Performance comparison of different data mining models in terms of accuracy, precision, recall, and f1-score, when missing values are filled with standard deviation.

### 5.2.5 Missing Values Filled with Value Above

Here we present the evaluation of data mining models when missing values are filled with the neighboring value. For this analysis, we have filled the missing values with the value immediately above them. All models are evaluated with the same metrics as we did for other cases; the experimental results are shown in Figure 10. It is clear from the figure that the XgBoost model is providing the best performance in this case, while the Naive Bayes model is providing...
the lowest performance (which is true in terms of all metrics). Specifically, the XgBoost model provides the best accuracy of 75.73%, a precision of 75.43%, a recall of 75.73%, and an f1-score of 75.12%. On the other hand, the Naive Bayes model provides the lowest accuracy of 37.09%, the precision of 34.42%, recall of 37.09%, and f1-score of 30.56%. Figure 10 suggests that the performance of most of the models is not very good in this case, i.e., when missing values are filled with the value above (this trend can be verified by a comparative analysis of Figure 10 and Figures 6, 7, 8, and 9).

![Fig. 10: Performance comparison of different data mining models in terms of accuracy, precision, recall, and f1-score, when missing values are filled with the value above.](image)

We conclude that we have evaluated the performance of eight data mining models on different missing values imputation techniques in terms of accuracy, precision, recall, and f1-score. We have filled the missing values with zero, −1, average value, standard deviation, and the value immediately above the missing value. From our experiments, we conclude that the Random Forest model has provided the best performance in most of the cases. On the other hand, the Naive Bayes classifier has provided the worst performance against all data imputation techniques (this can be easily verified with a comparative analysis of Figure 6, 7, 8, 9, and 10). Also, from this, we have seen that the performance of Logistic Regression and Neural Network classifiers has been relatively lower than the other models in most cases. We see that models have provided relatively better performance when missing values are imputed with zero value and standard deviation. Similarly, we see that missing value filling with the value above has resulted in a relatively lowest performance of all models. Based on our experiments and analysis, we remark that missing values imputation with zero is a safer approach since it is not as impacted by data drift as a value like the standard deviation would be.

## 6 Conclusion

In this paper, we detected five method-level code smells, three of which are security smells using classification approaches via data mining techniques. The ultimate goal of this paper is to reduce the number of vulnerabilities that may
occur due to bad security smells related to code quality metrics of methods by
detecting security code smells and Non-security code smell. We achieve this by
evaluating a data mining classifiers that will be able to detect security smells
and non-security smells by using quality metrics.

Through the research has been discovered that the QScored database con-
tains bugs. So, to achieve accurate accuracy, we manually removed all the bugs.
As we have added two new columns to the dataset, it appears to have missing
data. So, to fix that, we have decided to fill the missing data with zero, aver-
age value, negative one, standard deviation value, and the above value. So in
total, we got five different datasets. Based on our experiments and analysis,
we remark that missing values imputation with zero is safer approach since it
is not as impacted by data drift.

Then we have developed eight data mining classifiers: Logistic Regression,
Random Forest, XGBoost, J48, SVM, Naïve Bayes, LightGBM, and Neural
Network. The Random Forest, Neural Network, XGBoost, and SVM, respec-
tively, had the best accuracy performance and exceeded 95%.

7 Future Work

For future work, we plan to use static analysis instead of a QScored database to
be able to look at the source code instead of the quality metrics. Doing so will
allow us to detect not only complex methods, long methods, long parameter
lists, long identifiers, and long statements but also abstract function calls from
a constructor, Empty catch blocks, Magic numbers, Complex conditional, and
Missing defaults.

As future analyses to improve models should focus on identifying the fea-
ture distributions for the security code smell (Complex Method) class and the
security code smell (Long Method) class, as these two classes were the most
often confused.

An instance that may have more than one bad smell is neither a dataset
problem nor the models’ problem. The data mining will detect only one of
them. So, we may need to use the assembling technique as part of future work.

Funding: No funds, grants, or other support was received.

Data Availability: The datasets and source code generated during and/or
analysed during the current study are available in the Zenodo repository, https://
doi.org/10.5281/zenodo.7312130

Declarations

Conflict of Interest
Non-financial interests: none.
References


Detecting Security Bad Smells of Software by using Data Mining


[23] Singh, R., Bindal, A., Kumar, A.: Long method and long parameter list
Detecting Security Bad Smells of Software by using Data Mining


models, methodologies and applications to object detection. Progress in Artificial Intelligence 9(2), 85–112 (2020)


