TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING PULCHOWK CAMPUS DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

THESIS NO: 073/MSCS/664

INTELLIGENT CODE. SMELL. DETECTION SYSTEM USING DEEP LEARNING

by

Sanjaya Subedi

A THESIS

SUBMITTED TO DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE MASTER'S DEGREE IN COMPUTER SYSTEM AND KNOWLEDGE ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

LALITPUR, NEPAL

August, 2021

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING PULCHOWK CAMPUS DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

THESIS NO: 073/MSCS/664

INTELLIGENT CODE. SMELL. DETECTION SYSTEM USING DEEP LEARNING

by

Sanjaya Subedi

A THESIS

SUBMITTED TO DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE MASTER'S DEGREE IN COMPUTER SYSTEM AND KNOWLEDGE ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

LALITPUR, NEPAL

August, 2021

INTELLIGENT CODE SMELL DETECTION SYSTEM USING DEEP LEARNING

by
Sanjaya Subedi
073/MSCS/664

Thesis Supervisor
Assistant. Prof. Dr. Basanta Joshi

A thesis submitted in partial fulfillment of the requirements for the
degree of Masters of Science in Computer System and Knowledge Engineering

Department of Electronics and Computer Engineering
Institute of Engineering, Pulchowk Campus

Tribhuvan University
Lalitpur, Nepal

August, 2021

COPYRIGHT©

DECLARATION

I declare that the work hereby submitted for Master of Science in Computer System and Knowledge Engineering (MSCSK) at IOE, Pulchowk Campus entitled "**Intelligent Code Smell Detection System Using Deep Learning**" is my own work and has not been previously submitted by me at any university for any academic award.

I authorize IOE, Pulchowk Campus to lend this thesis to other institution or individuals for the purpose of scholarly research.

Sanjaya Subedi
073/MSCS/664

August, 2021

RECOMMENDATION

The undersigned certify that they have read and recommended to the Department of Electronics and Computer Engineering for acceptance, a thesis entitled "**Intelligent Code Smell Detection System Using Deep Learning**", submitted by **Sanjaya Subedi** in partial fulfillment of the requirement for the award of the degree of "**Master of Science in Computer System and Knowledge Engineering**".

.....

Supervisor: Asst. Prof. Dr. Basanta Joshi,
Department of Electronics and Computer Engineering,
Institute of Engineering, Tribhuvan University

.....

External Examiner: Mr. Om Bikram Thapa
CTO, Vianet Communications Pvt. Ltd

.....

Committee Chairperson: Assoc. Prof. Dr. Nanda Bikram Adhikari,
Department of Electronics and Computer Engineering,
Institute of Engineering, Tribhuvan University

Date: August, 2021

DEPARTMENTAL ACCEPTANCE

The thesis entitled "**Intelligent Code Smell Detection System Using Deep Learning**", submitted by Sanjaya Subedi in partial fulfillment of the requirement for the award of the degree of "Master of Science in Computer System and Knowledge Engineering" has been accepted as a bonafide record of work independently carried out by him in the department.

.....

Prof. Dr. Ram Krishna Maharjan
Head of the Department
Institute of Engineering,
Tribhuvan University,
Nepal.

ACKNOWLEDGEMENT

My sincere and deepest gratitude to **Dr. Basanta Joshi** for supervising my research work and providing the kind support, guidance and mentorship.

I would like to convey my special thanks of gratitude to the Department of Electronics and Computer Engineering (DOECE) and to our Head of Department **Prof. Dr. Ram Krishna Maharjan** for providing us with the golden opportunity to explore our interest and ideas in the field of engineering through this thesis. I also would like to provide my sincere gratitude to MSCSKE Project Coordinator **Dr. Nanda Bikram Adhikari** for providing me the opportunity to present this report. I would also like to thank ALL OF THE RESPECTED TEACHERS for their constant support, comments and encouragement to do the research activity and thesis.

Finally, I would like to thank all our teachers and friends who have helped me directly or indirectly for encouraging me with this thesis and all the support.

Sanjaya Subedi
073/MSCS/664
Pulchowk Campus

ABSTRACT

As the software industry is growing day by day, the challenges of software development are also growing. One of the major challenges in software development is the ability of evolution of software itself as per increase its demand, increase the feature, enhancement on itself. The evolution of software itself is not possible or extremely challenging as the code base of software itself is not scalable, maintainable, reliable, testable etc. Such type of code base is commonly considered as low-quality code and software build from such code base is low quality software.

In this research, the goal is to understand the code smells and its importance and develop the intelligent model for code smell detection using one of the most popular machine learning algorithms, LSTM(Long Short-Term Memory), a RNN based approach. The data for training the model has been prepared by collecting the open-source codebase and trained, validate and test the model. The types of code smell considered for this research are Magic Number, Complex Method, Long Identifier, Long Statement, Long Parameter List, Deficient Encapsulation, Unutilized Abstraction, Insufficient Modularization, Broken Hierarchy and Feature Envy. The experimental results show that in the best case, the model produces an accuracy of 91.17%, True Positives 92.34% and False Positives 8.84%.

Keywords: *Code Smell, Code Smell Detection, LSTM-RNN, Software Development Industry, Intelligent Code Smell Detection*

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ML	Machine Learning
NN	Neural Network
AUC	Area Under Cruve
TNR	True Negative Rate
LSTM	Long Short-Term Memory
TP	True Positives
RNN	Recurrent Neural Network
FPR	False Positive Rate
ROC	Receiver Operating Characteristic
TPR	True Positive Rate
IAM	Identity Access and Management
DBN	Deep Belief Network
APT	Advanced Persistent Threats
MSE	Mean Squared Error
CERT	Computer Emergency Response Team
TN	True Negative
FP	False Positive

CHAPTER 1: INTRODUCTION

1.1 Background and Motivation

In software development and programming, the characteristic that can be in placed in the source code which could potentially raise the bug, vulnerabilities or anything unwanted can be termed as code smell. It is not a direct bug but could potentially lead to raise the bug. The code smell patterns do differ by programming language, developer coding standard, standard coding practices and development methodologies.

For the first time this term was used by Fowler, Martin (1999) in his book "Refactoring. Improving the Design of Existing Code" and later popularized by Kent Beck [11]. To understand the typical code smells let's go through the following examples for Java Programming Language:

Example 1:

```
TestObject obj = getObject();

obj.getSomething(); // If object "obj" is null then exception is thrown here, there is
no meaning of checking null later.

if (obj!= null){
        //TODO add missing logic.
}
```

Figure 1: Code smell example1

Example 2:

```
public class MidTermTest(){

public MidTermTest (int smell1, int smell2, int smell3, int smell4, String smell5,
String smell6, String smell7, String smell8, String smell9, String smell10){

        // TODO assign these variables

        }
}
```

Figure 2 : Code smell example 2

In Example 1, we can see that there is clearly smell in code as **obj.getSomething();** will throw the exception if the object obj is null. So, checking null later would not make the code safer to execute. Likewise, in Example 2, A class has a default parametrized constructor of ten variables. It can be clearly imagined how risky would it be to create and object having ten parameters in constructor, instead we could have used getter/setter approach.

As software industry is growing faster and getting bigger day by day, software companies need to update their software or solution's code base frequently to compete the market. And if code smell does exist in their code base it is difficult to scale themselves and it might lead the huge loss of financial and reputation of the software industry.

The major cases of technical debt are code smells, no effective design decisions taken by developers which can negatively affect the maintainability, scalability and Quality of a software system. In last few decades, the research has heavily investigated on

- The evolution of code smells.
- The definition of code smells.
- The understanding, interest and ability of software development team to fix them.
- The effect on source code before and after the fix of code smell [14][15]

Code Smells are categorized in different varieties based on the nature of bug they can potentially raise. Below are the few most common code smells [11]:

- Long Method,
- Divergent Change,
- Parallel Inheritance Hierarchies,
- Lazy Class,
- Incomplete Library Class
- Speculative Generality,
- Middleman,
- Shotgun Surgery,
- Feature Envy,
- Long Parameter List,
- Comments,
- Message Chains,
- Data Clumps,
- Inappropriate Intimacy,
- Large Class,
- Dead Code,
- Duplicate Code,
- Data Class,
- Object-Orientation Abusers etc.

1.2 Problem Statement

Code smells directs the existence of quality compromise impacting various parts of software features such as reliability, maintainability and testability. If the number of existences of code smells are huge in software it is very difficult to evolve and maintain. If software cannot update and evolve then it is way expensive and costly for software industry to re-engineer and re-develop it from beginning. In Software Engineering Industry, lots of research has been carried out for determining the code smells of the code base at the time

of software development, researching on its impact and various dimensions. Rule based or static code analysis tools are quite popular these days. In this research, it aimed to understand the code smells and its importance, along with the research of intelligent code smell detection using one of the most popular machine learning algorithms. As per research and study it is found out code smells can lead us or give a clue on architecture flaw, implementation flaw and design flaw. With the intelligent code smell analysis, detection system, the system would be adaptive and update the rule and sequences as per new changes and would be able to predict or determine the code smell in upfront. Many rule-based code analysis tools do evaluate the code as per the defined and assigned rule however, if new coding standard, design standard or implementation standard got emerged then they cannot incorporate the change as new rule should be developed and implemented. With this research the investigation on the possibilities of building the code smell detection system with can incorporate the new trends as per developer's implementation is carried out.

Code smells are due to poor designs and implementations in software developing which is huge technical debt to the software industry as the quality of produced source code is not good, which will eventually degrade the quality of software. Since the last decades various code smell detection system, software, technologies and tools has been researched and identified and are in practice. However, the studies illustrate the outcome of these tools can be specific and depend on the approach and nature of code base. These days various ML techniques for smell detection have been researched and proposed, aiming to solve the issue of subjective tool providing to a intelligent model the ability to distinguish between smelled and non-smelled source code base. These approaches take metrics as feature/input and validation is done on balanced samples, which would limit the approach for machine learning. To add more value in this part of research, the research on code smell detection using deep learning (LSTM) has been carried out as the part of this thesis.

1.3 Objective

The objectives of this research work are as follows:

- To understand the code smells and its implications in software development

- To collect the data from open-source platform, GitHub and prepare a machine learning model (LSTM model).
- To perform the analysis of the prepared model, by validating automatically and manually.

CHAPTER 2: LITERATURE REVIEW

2.1 Understanding the Evolution of Code Smells

"Refactoring. Improving the Design of Existing Code" a book published by Martin Fowler in basically introduced and defined the term code small and explain the possibilities of code smell in detail. Ahmad Tahmid and Nadia Nahar[10] have done the detail study in the evolution of code smells. As per their research code smells are defects in software potentially directs to the issues in maintaining the software for further evolution. Code smells are inter-related and inter-connected in software system rather than being the single instance. This means, code smells basically do not only exit in single class or methods, however are present interrelating to various class and methods. If a class is calling method of another class, then the called method or the way of calling method can be smelly. These clusters of problem in software causes the maintainability issues in majority while updating, upgrading and evolving the software.

2.2 Smell Detection using Machine Learning

The quality problems with effect the overall software quality is commonly known is code smells. These code smell can be hidden bugs, badly designed architecture, poorly written code, wrongly used design patterns. These smells hinder the evolution is software itself. And if software cannot evolve then it is way expensive and costly for software industry to re-develop it from starch. In Software Engineering Industry, tremendous research has been carried out for determining the code smells of the code base at the time of software development, researching on its impact and various dimensions. Rule based or static code analysis tools are quite popular these days. In this research, we aim to understand the code smells and its importance. Along with the research of intelligent code smell detection using one of the most popular machine learning algorithms, LSTM. As per research and study we find out code smells can lead us or give a clue on architecture flaw, implementation flaw and design flaw. With the intelligent code smell analysis, detection system, the system would be adaptive and update the rule and sequences as per new changes and would be

able to predict or determine the code smell in upfront. Many rule-based code analysis tools do evaluate the code as per the defined and assigned rule however, if new coding standard, design standard or implementation standard got emerged then they cannot incorporate the change as new rule should be developed and implemented. With our research we investigate the possibilities of building the code smell detection system with can incorporate the new trends as per developer's implementation.

The approach we are following is to build the LSTM model, train and validate it. And use that model for determining the code smell for new code bases. For validating the result of the model, we are manually verifying the certain part result to make sure the model is giving the expected result.

These days there are many machine learning techniques are being research for code smell detection. Kreimer [1] has researched on the decision-tree based approach to worked on to determine the code smells like larger classes, long methods. Similarly, Vaucher et al. researched on Bayesian and explained machine learning model to detect Blob class or God class. SVM-Support Vector Machine to detect Blob class was researched and proposed by Maiga et al. and the same thing was researched by Amorim et al. to detect long method, feature envy, long parameter list and Blob class. Fontana et al., did the research in comparing various machine learning techniques (including Forest, LibSVM, ERandom, SMO, J48, JRip, and Baive Bayes) in determining the complexity of code smells, e.g., data class, feature envy, God class and long method. All these machine learning approaches those are researched can provide the inference that machine learning approach in detecting code smell is one of the most interesting research areas and it has still limitations.

Thirupathi Guggulothu and Salman Abdul Moiz [13] researched on two method level code smells implementing a multi-label classification approach. Existing studies used to determine a single type of code smell but, their study detected two code smells whether they exist in the same method or not. For this work, they considered two method datasets which are constructed by single type detectors. These datasets have 395 common instances thus leads to form the disparity while merging process in the existing study. Due to this,

the performances were less in their study. In their research, these common instances are led to construct the multi-label dataset and to avoid the disparity. They experimented, two multi-label classification methods (LC, CC) on the multi-label dataset. The CC method has given best performance than LC based on all three measures. The performance of the proposed study is much better than the existing study. As per their study, accuracy was 91% in average. But it only detects two verities of smells. Tao Lin, Xue FuFu Chen[23] have proposed the code smell detection using CNN, however it also identify limited amount features.

2.4 Review of Algorithm

For this research, the machine learning algorithm that is going to be used is Deep learning (LSTM) algorithm which is basically the most effective flavor of Neural Networks and is the type of RNN with feedback connections. This section presents the brief summary of this approach.

2.4.1 Neural Networks

Neural Networks are commonly used in machine learning based application. A neural network (NN) in Machine Learning is a Machine Learning Model that is developed with the concept of functioning and structure of biological brain. It simply contains computing uniting called nodes like neurons in brain. These neurons receive incoming inputs along with:

- incoming edges,
- multiplies the inputs by corresponding edge weights,
- and then applies a non-linear function called activation function,

to the weighted sum and produces an output as shown in figure below [17]:

$$y(x) = f(w \odot x + b)$$

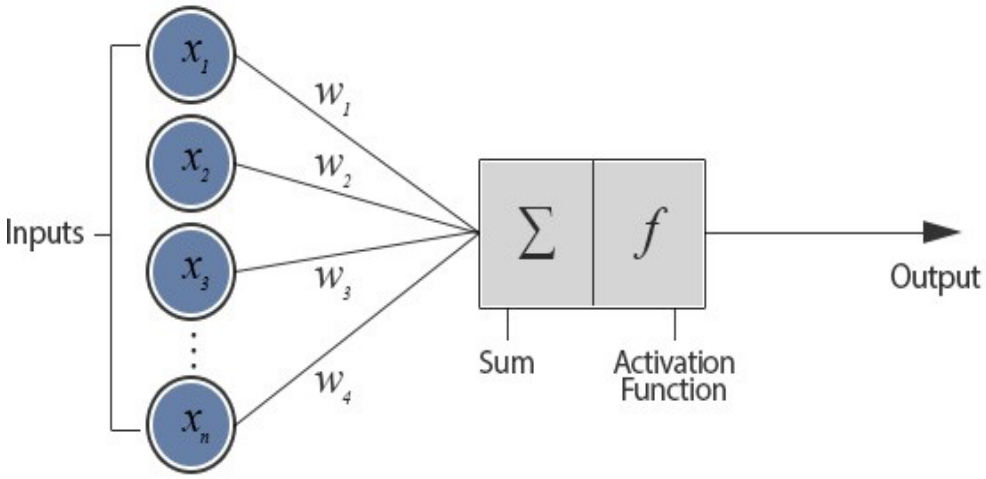

Figure 3 : Working mechanism of a neuron

The output of a neuron is a non-linear function of the weighted sum of its inputs. The non-linearity is introduced by the activation function. Typical activation functions include logistic sigmoid (σ), tanh, and rectified linear units (ReLU) and they are defined as [17]:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$ReLU(z) = max(0, z)$$

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

The linear activation applies identity function as:

$$a(z) = z$$

Neural Network has different variants, e.g., Feedforward Neural Network, Self-Organizing Maps, Convolutional Neural Network, Deep Neural Network, Replicator Neural Network, Recurrent Neural Networks etc. In following section, Recurrent Neural Network with Long Short-Term Memory has been explained.

Recurrent Neural Networks (RNN)

Recurrent neural network is an extension of a feed forward neural network, and it is being widely used in different ML proposes for e.g. natural language processing. RNN is known as powerful algorithm for modeling sequences by having cyclic connections.

Let X=(x1,x2,...,xT) to represent the input vector sequence and hidden vector sequence H=(h1,h2,...,hT). The output vector sequence Y=(y1,y2,...,yT) are calculated with t belongs to [1, T] as follows (Fanzhi Meng, Fang Lou, Zhihong Tian and et.al, 2018):

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Where W and b is the weight matrix and bias term, xt is the input vector at time t, ht-1 is the state at time t-1, σ is a nonlinearity activation function.

Long short-term memory (LSTM)

RNN become powerless when time series is very long. Long short-term memory (LSTM) is well suited to classify the time series because it employs the LSTM cell to learn the historical experience (Fanzhi Meng, Fang Lou, Zhihong Tian and et.al, 2018).

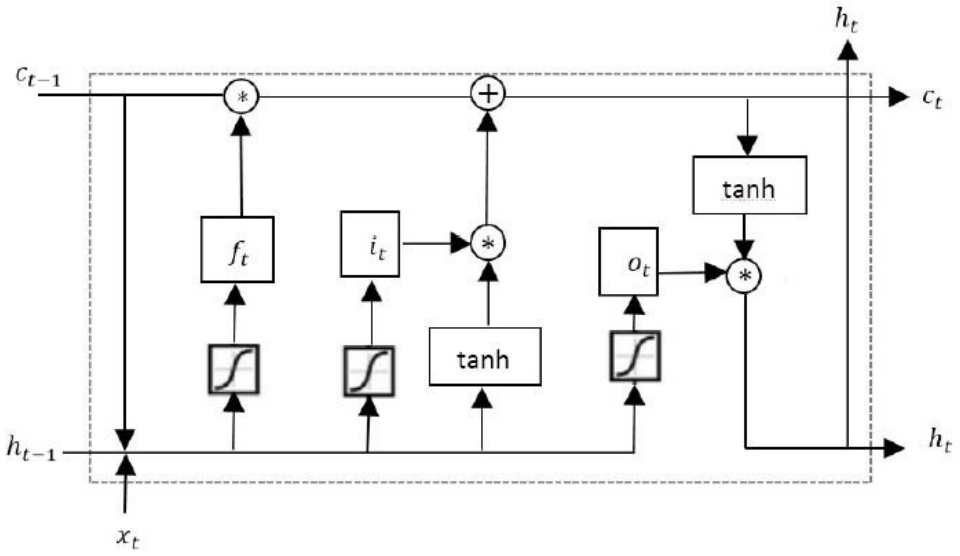

Figure 4 : Long Short-Term Memory Cell

The information flow in LSTM-RNN is controlled by three gates (f, i, o). Firstly, the input gate and the cell state at time t are given as follows (Fanzhi Meng, Fang Lou, Zhihong Tian and et.al, 2018):

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)$$

$$c_t = f_t c_{t-1} + i_c \tanh(W_{xc}x_t + W_c h_{t-1} + b_c)$$

The forget gate decides whether the previous memory ht−1 is passed, it can be calculated as follow (Fanzhi Meng, Fang Lou, Zhihong Tian and et.al, 2018):

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)$$

The output gate decides whether the output of memory cell is passed. The calculation process of ht can be described as follows (Fanzhi Meng, Fang Lou, Zhihong Tian and et.al, 2018).

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o)$$

$$h_t = o_t \tanh(c_t)$$

Where,

a) σ is the function of logistic sigmoid
b) xt is the input vector at time t.
c) Wci, Wcf and Wco are weight matrices.
d) i and c are input gate and cell state respectively.
e) f and o are forget gate and output gate respectively.

CHAPTER 3: METHODOLOGY

3.1 Overview

In this section, a detail methodology of LSTM based approach to identify code smells is mentioned. Figure 5 presents the overview of the work approach. Based on a data set of codes, we generate large number of training data with contains both good code and smelled code. These samples are used to train LSTM model whose output indicates whether the input code is smelled or non-smelled. We have discussed the detail implementation and design in section below.

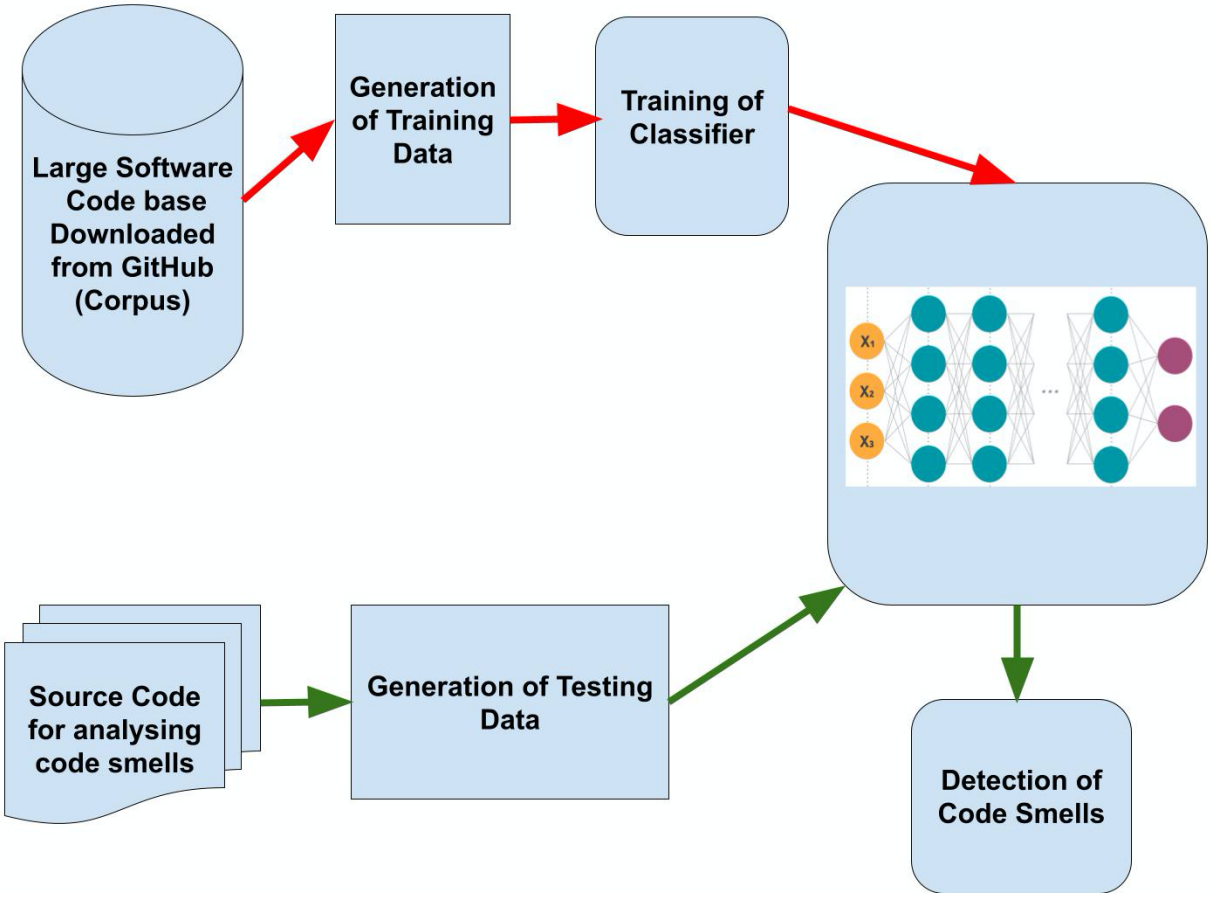

Figure 5 : Overview of the System Approach

3.2 Generating the Training Data

To train the LSTM networks, following steps are be carried out:

Step 1(Data Preparation): Downloaded the Java based open source projects those are available in GitHub. Those are cloned from GitHub by filtering on the basis of number of file(>1000) and number of stars(>1000) provided to that projects. The popular open-source java project like Hadoop, Spring, Zookeeper etc. are also considered as the part of code smell analysis.

Step 2(Code Split): All the downloaded source code then needs to split to generate code fragments. The splitting of code is done in two levels:

- Class Level: It is the fragments of code that contains the class level information of code file.
- Method Level: It is the fragments of code that contains the method level information of the code file.

To achieve this, we developed the mechanism to split the code in method level and class level as per java conventions. This code fragments do not contain additional comments, imports etc. Following is the sample of extracted code fragment at class level:

```
class NameFilter implements FilenameFilter {
        @Override
        public boolean accept(File directory, String fileName) {
            try(Integer.parseInt(fileName);) {
                int value = Integer.parseInt(fileName);
                if (value < 1) {
                    return false;
                }
            }
            return new File(directory, fileName).isDirectory();
        }
    }
```

The code is split based on class and methods. Following figure is an example of one of the classes.

```
 5
 6⊕import java.net.URISyntaxException;
16
17 public class EncoderUtilTest {
18
19⊖    public String getFilePath(String file) {
20        ClassLoader classLoader = getClass().getClassLoader();
21        try {
22            String configurationPath = classLoader.getResource("etc/" + file).toURI().getPath();
23            return configurationPath;
24        } catch (URISyntaxException e) {
25            return "";
26        }
27
28    }
29
30⊖    @SuppressWarnings("unchecked")
31    public void compareRows(List<Map<String, Object>> initialRows,
32            List<Map<String, Object>> modifiedRows, List<String> keysToCompare,
33            Map<String, Object> accessPair) {
34        for (int index = 0; index < initialRows.size(); index++) {
35            Map<String, Object> map = initialRows.get(index);
36            for (Entry<String, Object> entry : map.entrySet()) {
37                if (keysToCompare.contains(entry.getKey())
38                        || entry.getKey().equalsIgnoreCase("msg")) {
39                    if (entry.getKey().equals("group")) {
40                        if (entry.getValue() instanceof List<?>) {
41                            if (((List<String>) entry.getValue()).isEmpty()) {
42                                Assert.assertEquals(entry.getValue(),
43                                        modifiedRows.get(index).get(entry.getKey()));
44                            } else {
45                                Assert.assertNotEquals(entry.getValue(),
46                                        modifiedRows.get(index).get(entry.getKey()));
47                            }
48                        }
49                    } else {
50                        Assert.assertNotEquals(entry.getValue(),
51                                modifiedRows.get(index).get(entry.getKey()));
52                    }
53                } else if (entry.getKey().equalsIgnoreCase("_participating_events")) {
54                    List<Map<String, Object>> events = (List<Map<String, Object>>) entry.getValue();
55                    compareRows(events, (List<Map<String, Object>>) modifiedRows.get(index)
56                            get("_participating_events"), keysToCompare, accessPair);
```

Figure 6: Class file

This class has:

- 1843 lines of code
- 35 methods

This class is split into two data sets class and methods

- Class data file contains all the data at class level without imports, comments and other extra information other than code.
- Method data file contains each method of the class.

Step 3(Static Code Smell): These projects are scanned via static code analysis tool. In our case we used the tool Designite(https://www.designite-tools.com/) as the static code analysis tool as this would automatically dump the code smells in csv which is easier for further processing.

The format of csv is as below:

- *Project Name,Package Name,Type Name,Method Name,Code Smell*
- *hadoop,com.hadoop.core.entity.store,DBStorageTest,executeDbStorageTest,Long Method*

The output of code smell from static tool analysis is as below, it is a CSV data with code smell name

```
Project Name,Package Name,Type Name,Method Name,Code Smell
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRows,Complex Method
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRows,Long Statement
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRowsWithModifiers,Complex Method
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRowsWithModifiers,Long Parameter List
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRowsWithModifiers,Long Statement
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRowsWithAccessPair,Complex Method
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRowsWithAccessPair,Long Statement
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRowsWithAccessPairForTimechart,Complex Method
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,compareRowsWithAccessPairForTimechart,Long Parameter List
ParserLib,com.parserlib.shared.encodings,EncoderUtilTest,testSimpleJson,Long Statement
```

Figure 7 : Static code smell

Considering the following method, this can be easily determined as complex methods as multiple level iterations, multi-level if-else statement are used. This method is too complex to read and understand. So as per standard practice and principle of clean code, code should be as simple and as human readable as possible. This method is detected as smell code block and type of smell is Complex Method, Long Statement etc. with is provided by smell list mentioned above.

```
@SuppressWarnings("unchecked")
public void compareRows(List<Map<String, Object>> initialRows,
        List<Map<String, Object>> modifiedRows, List<String> keysToCompare,
        Map<String, Object> accessPair) {
    for (int index = 0; index < initialRows.size(); index++) {
        Map<String, Object> map = initialRows.get(index);
        for (Entry<String, Object> entry : map.entrySet()) {
            if (keysToCompare.contains(entry.getKey())
                    || entry.getKey().equalsIgnoreCase("msg")) {
                if (entry.getKey().equals("group")) {
                    if (entry.getValue() instanceof List<?>) {
                        if (((List<String>) entry.getValue()).isEmpty()) {
                            Assert.assertEquals(entry.getValue(),
                                    modifiedRows.get(index).get(entry.getKey()));
                        } else {
                            Assert.assertNotEquals(entry.getValue(),
                                    modifiedRows.get(index).get(entry.getKey()));
                        }
                    }
                } else {
                    Assert.assertNotEquals(entry.getValue(),
                            modifiedRows.get(index).get(entry.getKey()));
                }
            } else if (entry.getKey().equalsIgnoreCase("_participating_events")) {
                List<Map<String, Object>> events = (List<Map<String, Object>>) entry.getValue();
                compareRows(events, (List<Map<String, Object>>) modifiedRows.get(index)
                        .get("_participating_events"), keysToCompare, accessPair);
            } else if (entry.getKey().equalsIgnoreCase("children")) {
                List<Map<String, Object>> events = (List<Map<String, Object>>) entry.getValue();
                List<String> encryptedKeys = new ArrayList<>(keysToCompare);
                encryptedKeys.add("group");
                compareRows(events,
                        (List<Map<String, Object>>) modifiedRows.get(index).get("children"),
                        encryptedKeys, accessPair);
            } else {
                if (keysToCompare.contains(entry.getKey())) {
                    Assert.assertNotEquals(entry.getValue(),
                            modifiedRows.get(index).get(entry.getKey()));
                } else {
                    if (entry.getValue() == null) {
                        if (modifiedRows.get(index).get(entry.getKey()) instanceof String) {
                            Assert.assertTrue("null".equalsIgnoreCase(
                                    (String) modifiedRows.get(index).get(entry.getKey())));
                        } else {
                            Assert.assertEquals(entry.getValue(),
                                    modifiedRows.get(index).get(entry.getKey()));
                        }
                    } else {
                        Assert.assertEquals(entry.getValue(),
                                modifiedRows.get(index).get(entry.getKey()));
                    }
                }
            }
        }
    }
```

Figure 8: Complex method example

Step 4(Label Data Generation): Once Step 2 and 3 are completed smell code fragments and non-smells code fragments is are generated. This is done by extracting that information from the csv data that we generated from static code analysis and the methods and classes fragments that we get from Step 2.

We collect smell code pieces as negative samples and non-smell code pieces as positive as positive sample, which are referred as negative and positive samples respectively. Following are the examples of negative sample and positive sample:

Negative Sample:

```
@Deprecated
protected void enableDebugLog(boolean enabled, String tag) {
		Log.warn(TAG, "BaseGameActivity.enabledDebugLog(bool,String) is " +
"deprecated. Use enableDebugLog(boolean)");
		enableDebugLog(enabled);
}
```

Positive Sample:

```
protected StatisticsChartGenerator createChartGenerator() {
		StatsValueRetriever statsValueRetriever =
StatsValueRetriever.DEFAULT_RETRIEVERS.get(getGraphType());
		if (statsValueRetriever == null) {
			throw new RuntimeException("Unknown GraphType: " +
getGraphType() + ". See the StatsValueRetriever class for the list of acceptable types.");
		}
		GoogleChartGenerator retVal = new
GoogleChartGenerator(statsValueRetriever);
		if (getTagNamesToGraph() != null) {
			Set<String> enabledTags = new HashSet<String>(

	Arrays.asList(MiscUtils.splitAndTrim(getTagNamesToGraph(), ",")));
			retVal.setEnabledTags(enabledTags);
		}
		return retVal;
	}
```

Once static code smell and code split is done, each code fragment is validated against the smell list. If code smell is present, then that code fragment will be as negative sample and if code smell is not present then that code fragment will be positive samples. The fragments are generated as shown below:

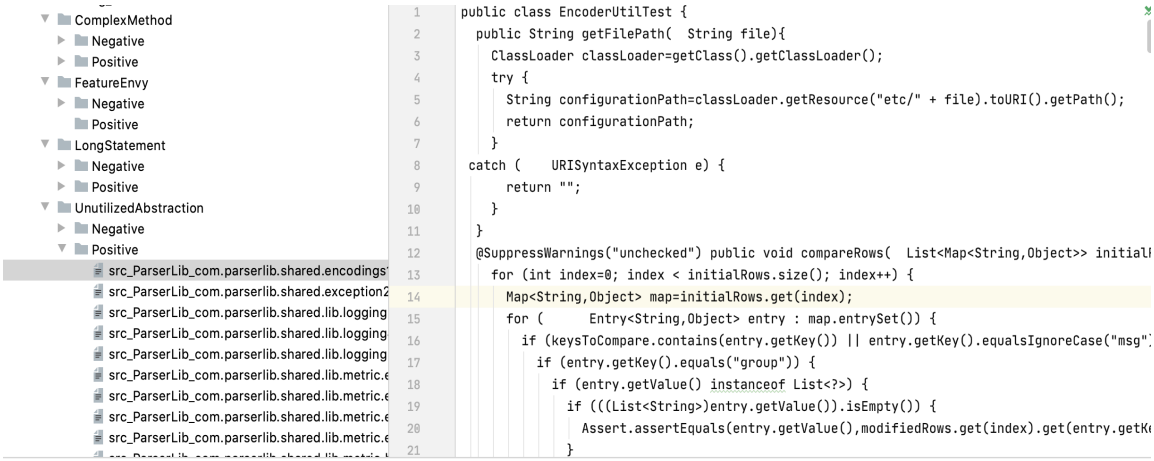

Figure 9: Learning Data (Label data)

Step 5(Tokenization): In this step the collections of positive and negative samples are tokenized to generate the corresponding integer vectors, symbols, or discrete tokens. The tokenization of these samples is done via a free and open-source tool (tokenizer) which converts code portion to numbers. These results are feed as input to train and validate the model. Figure 6. Below describes the overall overview of training the model.

This positive and negative sampled data is further tokenized and that is the input data for LSTM. Following is the sample of tokenized data:

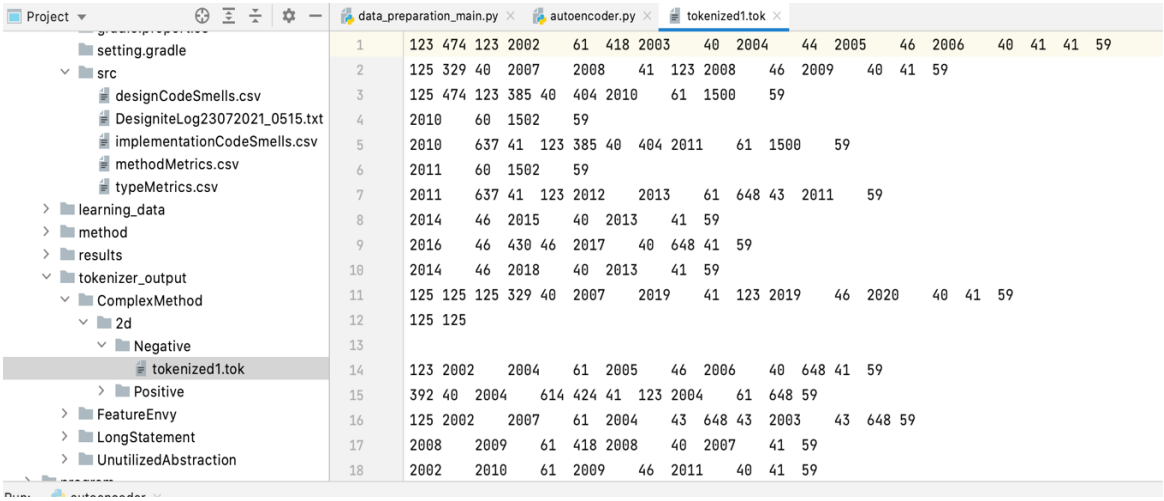

Figure 10: Tokenized Data

These tokenized data is the input for LSTM model, for training, validating and testing the results.

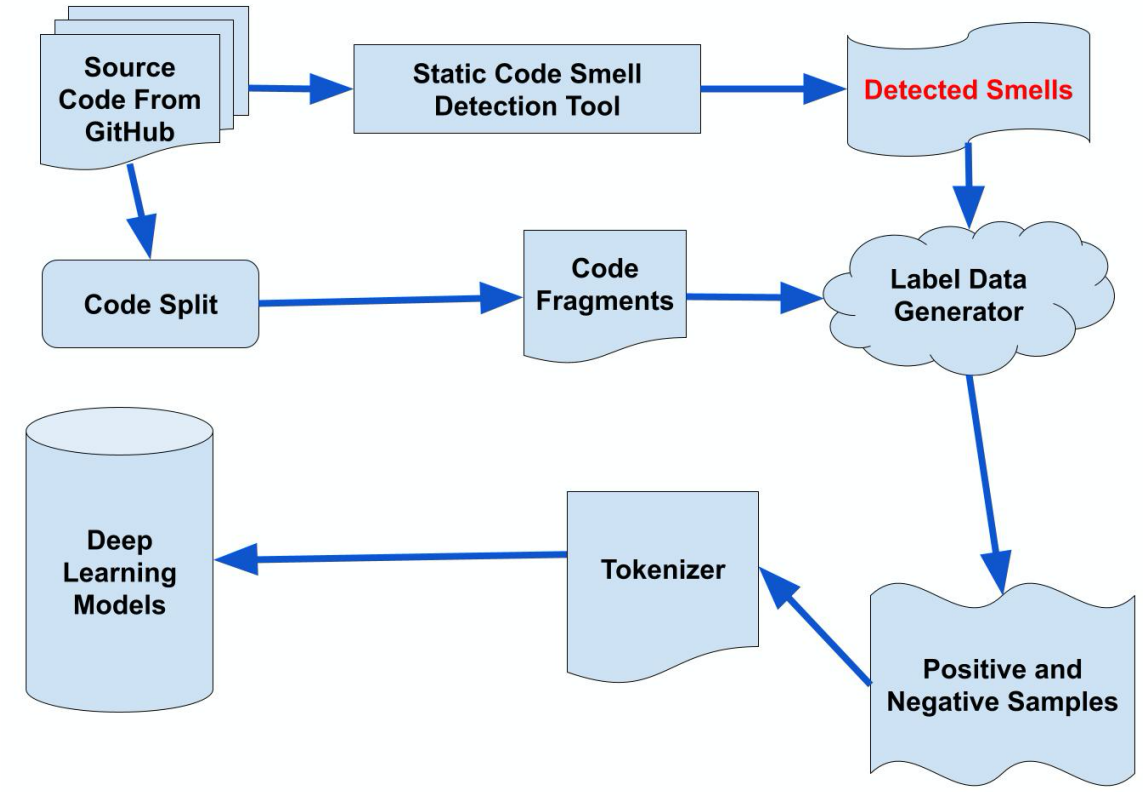

Figure 11 : Overview of training data generation and feeding it to Deep Learning model

3.3 Model Training and Code Smell Detection

The LSTM algorithm will be used for modeling code smells and non-code smells. The reason behind using LSTM is they are very useful for learning sequences and their ability to maintain long term memory.

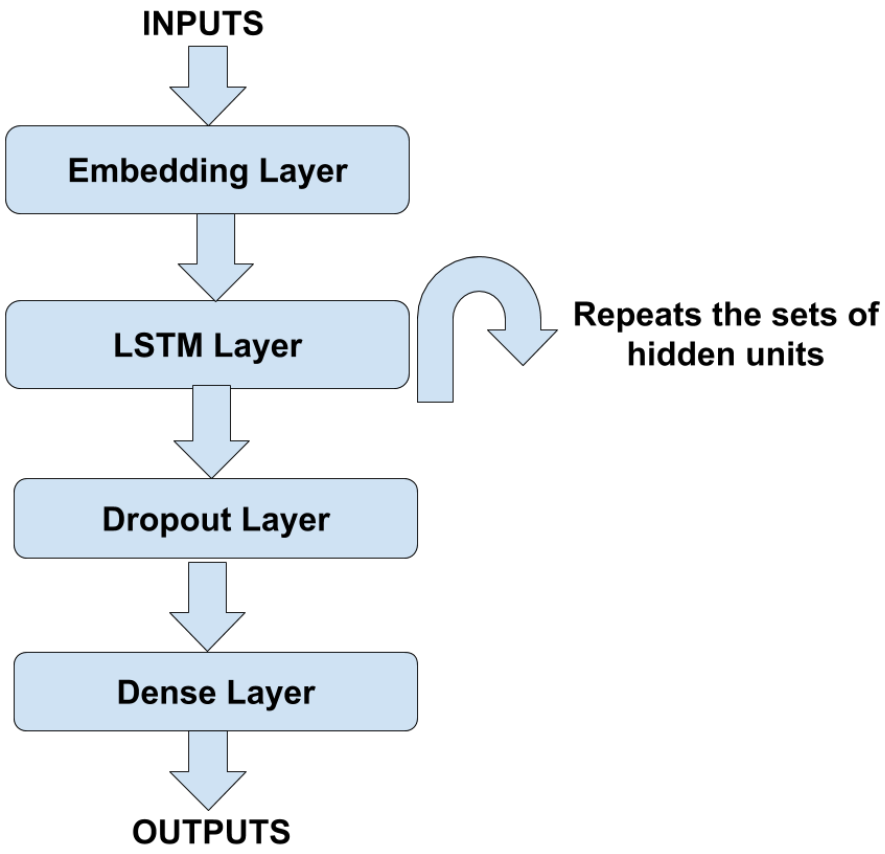

Figure 12 : General flow of LSTM-RNN Model

Consider the input X= {xu1,xu2,...,xut } is the input feature vector for positive and negative code samples u, where u = {u1, u2, u3,…….,un} is the set of n samples. Then the model maps input to a hidden state sequence hu1 , hu2 , . . ., huT . The hidden state hut is computed as a

function of xu ,xu ,...,xu . Conditioning hu on a sequence rather than the current input alone allows us to capture temporal patterns in code smells, and to build an increasingly accurate model of the correct code smells over time.

In LSTM RNN architecture (Hochreiter and Schmidhuber 1997), the hidden state hu at time t is a function of a long-term memory cell, cu. In a deep LSTM with L hidden layers,

the final hidden state, the output of hidden layer L, hu = hu , depends on the input sequence and cell states as follows:

$$
\begin{aligned}
\mathbf{h}_{l,t}^{u} &= \mathbf{o}_{l,t}^{u} \odot \tanh(\mathbf{c}_{l,t}^{u}) \\
\mathbf{c}_{l,t}^{u} &= \mathbf{f}_{l,t}^{u} \odot \mathbf{c}_{l,t\text{-}1}^{u} + \mathbf{i}_{l,t}^{u} \odot \mathbf{g}_{l,t}^{u}, \text{ and} \\
\mathbf{g}_{l,t}^{u} &= \tanh\left(\mathbf{W}_{l}^{(g,x)}\mathbf{h}_{l\text{-}1,t}^{u} + \mathbf{W}_{l}^{(g,h)}\mathbf{h}_{l,t\text{-}1}^{u} + \mathbf{b}_{l}^{g}\right) \\
\mathbf{f}_{l,t}^{u} &= \sigma\left(\mathbf{W}_{l}^{(f,x)}\mathbf{h}_{l\text{-}1,t}^{u} + \mathbf{W}_{l}^{(f,h)}\mathbf{h}_{l,t\text{-}1}^{u} + \mathbf{b}_{l}^{f}\right) \\
\mathbf{i}_{l,t}^{u} &= \sigma\left(\mathbf{W}_{l}^{(i,x)}\mathbf{h}_{l\text{-}1,t}^{u} + \mathbf{W}_{l}^{(i,h)}\mathbf{h}_{l,t\text{-}1}^{u} + \mathbf{b}_{l}^{i}\right) \\
\mathbf{o}_{l,t}^{u} &= \sigma\left(\mathbf{W}_{l}^{(o,x)}\mathbf{h}_{l\text{-}1,t}^{u} + \mathbf{W}_{l}^{(o,h)}\mathbf{h}_{l,t\text{-}1}^{u} + \mathbf{b}_{l}^{o}\right)
\end{aligned}
$$

Where,

hu0,t = xu. Vector gul,t is a hidden representation based on the current input and previous hidden state. Vectors ful,t, iul,t and oul,t, modulate how cell-state information is propagated across time, how the input is incorporated into the cell state, and how the hidden state relates to the cell state, respectively. The flow of LSTM training (Fangfang Yuan et al., 2018) is shown below:

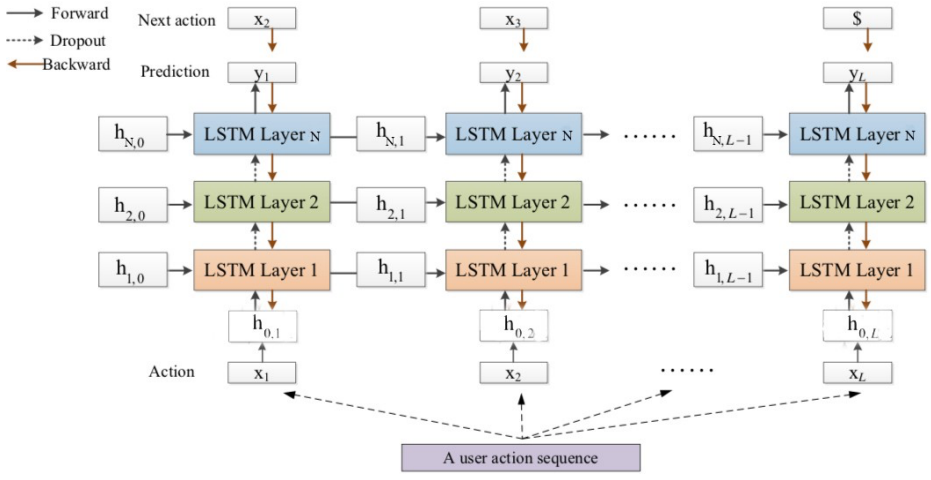

Figure 13 : Flow of LSTM training

At training phase model will be trained on generated code smells. The model will be trained to reproduce input as output. i.e. the reconstruction error is minimized during the training phase. The output vector is defined as:

$$y_t^u = W_{hy} h_t + b_t$$

In the testing phase, code smells are not detected when a poor reconstruction of an input is made by the model.

3.5 Code Smell Score Generation

Suppose hu is the hidden state at time t, then the model will output the parameters θ for a probability distribution over the observation, xu . The smell score for code fragment u at time t, Au is given by:

$A_u = -\log P_\theta(x_u|h_u)$

For code smell pattern, it is likely to produce low probability density, resulting high score. So, if

xu is code smell, the value produced will be large.

3.6 Validating and Verifying

To validate the model is working as expected following approaches are carried out:

Validating with standard tools output: The result of model be validated against the result of standard rule-based code smell detection software.

Validating with manual results: As validating against standard rule-based code smell detection software would not suffice the requirement because the input of for the source is also from that software so to ensure the model is working as expected, it is proposed to

take few hundreds of samples do the manual verification and compare the result with standard rule-based set and model result.

3.7 Code Smell types for this research

Following are the code smells chosen for this research [19]:

Magic Number: In a code, if there is the use of any number directly for comparing, performance logical operations then they are considered as smell and termed as Magic Number. The directly assignment of number always have high changes of occurring bugs and it is challenging to maintain. In the Figure shown below, we can see various operation done by hardcoding the numbers.

```
@Override public BigInteger convert(String value,ConversionContext ctx){
  ctx.addSupportedFormats(getClass(),"[-]0X.. (hex)","[-]0x... (hex)","<bigint> -> new BigInteger(bigint)");
  if (value == null) {
    return null;
  }
  String trimmed=value.trim();
  if (trimmed.startsWith("0x") || trimmed.startsWith("0X")) {
    LOG.finest("Parsing Hex createValue to BigInteger: " + value);
    return new BigInteger(value.substring(2),16);
  }
 else   if (trimmed.startsWith("-0x") || trimmed.startsWith("-0X")) {
    LOG.finest("Parsing Hex createValue to BigInteger: " + value);
    return new BigInteger('-' + value.substring(3),16);
  }
  try {
    return new BigInteger(trimmed);
  }
 catch (  Exception e) {
    LOG.log(Level.FINEST,"Failed to parse BigInteger from: " + value,e);
    return null;
  }
}
```

Figure 14: Magic Number Example

Complex Method: The method or function defined and declared in code with contains complex code for example nested if-else statements, for loops, while loops etc. generally even more than two-three levels. The more complex are the method the understandability and readability of the code is more complex which has high tendency to produce the bug.

```
private Set<String> calculateChangedKeys(Map<String,PropertyValue> valueMap,Map<String,PropertyValue>
  Set<String> result=new HashSet<>();
  if (this.valueMap != null) {
    for (    Map.Entry<String,PropertyValue> en : valueMap.entrySet()) {
      if (!newValues.containsKey(en.getKey())) {
        result.add(en.getKey());
      }
    }
  }
  for (  Map.Entry<String,PropertyValue> en : newValues.entrySet()) {
    if (valueMap != null) {
      if (!valueMap.containsKey(en.getKey())) {
        result.add(en.getKey());
      }
      if (!Objects.equals(valueMap.get(en.getKey()),en.getValue())) {
        result.add(en.getKey());
      }
    }
 else {
      result.add(en.getKey());
    }
  }
  return result;
}
```

Figure 15: Complex Method example

Long Identifier: As shown in figure below, the name of methods, declared variables are too long. This can make the readability of code harder which are identified as Long Identifier.

```
@Test public void testConstructionPropertiesAndDisabledBehavior() throws IOException {
  JavaConfigurationPropertySource localJavaConfigurationPropertySource=new JavaConfigurationPropertySource();
  StringWriter stringBufferWriter=new StringWriter();
  System.getProperties().store(stringBufferWriter,null);
  String before=stringBufferWriter.toString();
  try {
    assertThat(localJavaConfigurationPropertySource.isEnabled()).isTrue();
    System.setProperty("tamaya.defaultprops.disable","true");
    localJavaConfigurationPropertySource=new JavaConfigurationPropertySource();
    assertThat(localJavaConfigurationPropertySource.isEnabled()).isFalse();
    assertThat(localJavaConfigurationPropertySource.getProperties()).isEmpty();
    assertThat(localJavaConfigurationPropertySource.toString()).contains("enabled=false");
    System.getProperties().clear();
    System.getProperties().load(new StringReader(before));
    System.setProperty("tamaya.defaults.disable","true");
    localJavaConfigurationPropertySource=new JavaConfigurationPropertySource();
    assertThat(localJavaConfigurationPropertySource.isEnabled()).isFalse();
    System.getProperties().clear();
    System.getProperties().load(new StringReader(before));
    System.setProperty("tamaya.defaultprops.disable","");
    localJavaConfigurationPropertySource=new JavaConfigurationPropertySource();
    assertThat(localJavaConfigurationPropertySource.isEnabled()).isTrue();
    System.getProperties().clear();
    System.getProperties().load(new StringReader(before));
    System.setProperty("tamaya.defaults.disable","");
    localJavaConfigurationPropertySource=new JavaConfigurationPropertySource();
    assertThat(localJavaConfigurationPropertySource.isEnabled()).isTrue();
  }
  finally {
    System.getProperties().clear();
    System.getProperties().load(new StringReader(before));
  }
}
```

Figure 16: Long Identifier Example

Long Statement: The smells due to excessive large statement are Long Statement smells. They are mostly responsible for making code non-readable.

```
@Test public void callToConvertAddsMoreSupportedFormatsToTheContext() throws Exception {
  ConversionContext context=new ConversionContext.Builder(TypeLiteral.of(Boolean.class)).build();
  BooleanConverter converter=new BooleanConverter();
  converter.convert("",context);
  assertThat(context.getSupportedFormats()).contains("true (ignore case) (BooleanConverter)",
    "false (ignore case) (BooleanConverter)");
}
```

Figure 17: Long Statement Example

Long Parameter List: If any method has the parameters more than three then it is considered as Long Parameter List. These parameter always tends to create confusion and potentially leads to error the software.

```
public DefaultConfigurationContext(ServiceContext serviceContext,List<PropertyFilter> propertyFilters,
  List<PropertySource> propertySources,Map<TypeLiteral<?>,List<PropertyConverter<?>>> propertyConverters,
  MetadataProvider metaDataProvider){
  this.serviceContext=Objects.requireNonNull(serviceContext);
  this.immutablePropertyFilters=Collections.unmodifiableList(new ArrayList<>(propertyFilters));
  this.immutablePropertySources=Collections.unmodifiableList(new ArrayList<>(propertySources));
  this.metaDataProvider=Objects.requireNonNull(metaDataProvider);
  this.metaDataProvider.init(this);
  propertyConverterManager=new PropertyConverterManager(serviceContext);
  for (  Map.Entry<TypeLiteral<?>,List<PropertyConverter<?>>> en : propertyConverters.entrySet()) {
    for (    @SuppressWarnings("rawtypes") PropertyConverter converter : en.getValue()) {
      this.propertyConverterManager.register(en.getKey(),converter);
    }
  }
}
```

Figure 18: Long Parameter Example

Deficient Encapsulation: If abstraction is done and always re-define and implemented the abstracted methods. This leads to Deficient Encapsulation.

Unutilized Abstraction: If any abstraction is not used by its child class or not accessible due to access modifier then it is Unutilized Abstraction.

Insufficient Modularization: If abstraction is not fully implemented and implementing further will raise to Deficient Encapsulation, complexity is referred as Insufficient Modularization.

Broken Hierarchy: With child class do not fundamentally share the parent features then it leads to Broken Hierarchy.

Feature Envy: If any method or variable are used from other class which is irrelevant then it violates the rule of putting data and behavior in a single class.

```
class Mark{
                int a =10;
                int b = 20;
                //..
        }

class Random{

        private int getToalMarks(Mark m) {
                return m.a + m.b;
        }
}
```

3.8 Tools and Programming Language

The following tools and programming language will be used in the research work:

- Java, Python
- Microsoft Excel
- Sublime Text
- Eclipse, PyCharm IDE

3.9 Evaluation

3.9.1 Evaluation matrices

Evaluation is done using following parameters:

- true positive rate (TPR),
- false positive rate (FPR),
- precision
- accuracy.

And are calculated using following equations:

- TPR = TP / (TP + FN)

- FPR = FP / (TN + FP)
- Precision = TP / (TP + FP)
- Accuracy = (TP + TN) / (TP + FN + FP + TN)

3.9.1 ROC Curve

The following figure illustrates the ROC curve and it gives the information in how the change in parameter influences the system's performance. It's quality is described by AUC(Area under Curve), which is higher the score higher is the result.

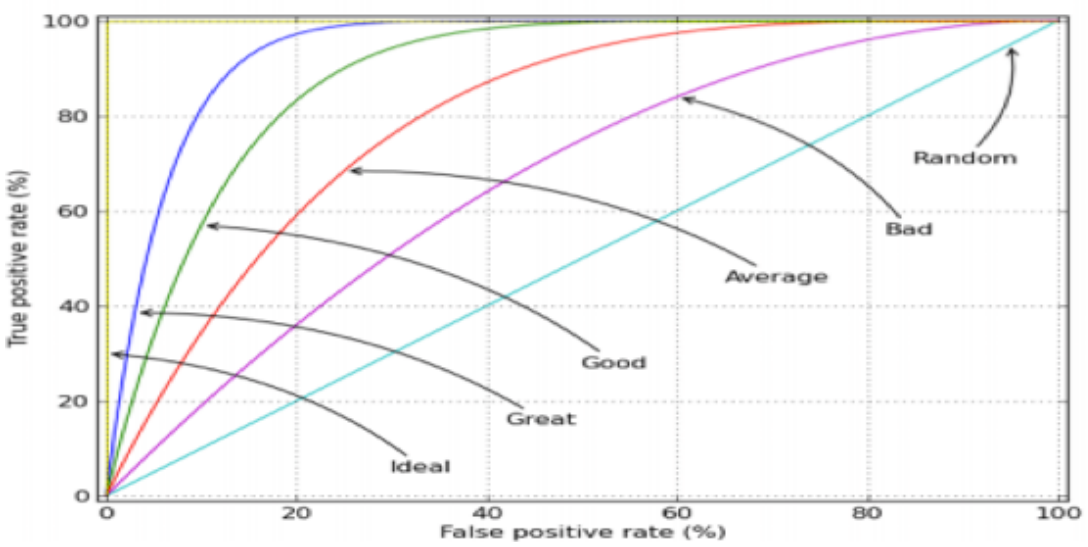

Figure 19 : Illustration of ROC curve

CHAPTER 4: EXPERIMENTAL RESULT, ANALYSIS

4.1 Data Preparation

The data for preparing the model is the code base of free and open-source projects those are publicly available in GitHub. From those code base we extracted out positive (non-smelled code) and negative (smelled code) samples by the use of static code analysis and develop the model from it, train and validate the model. The following is the data distribution:

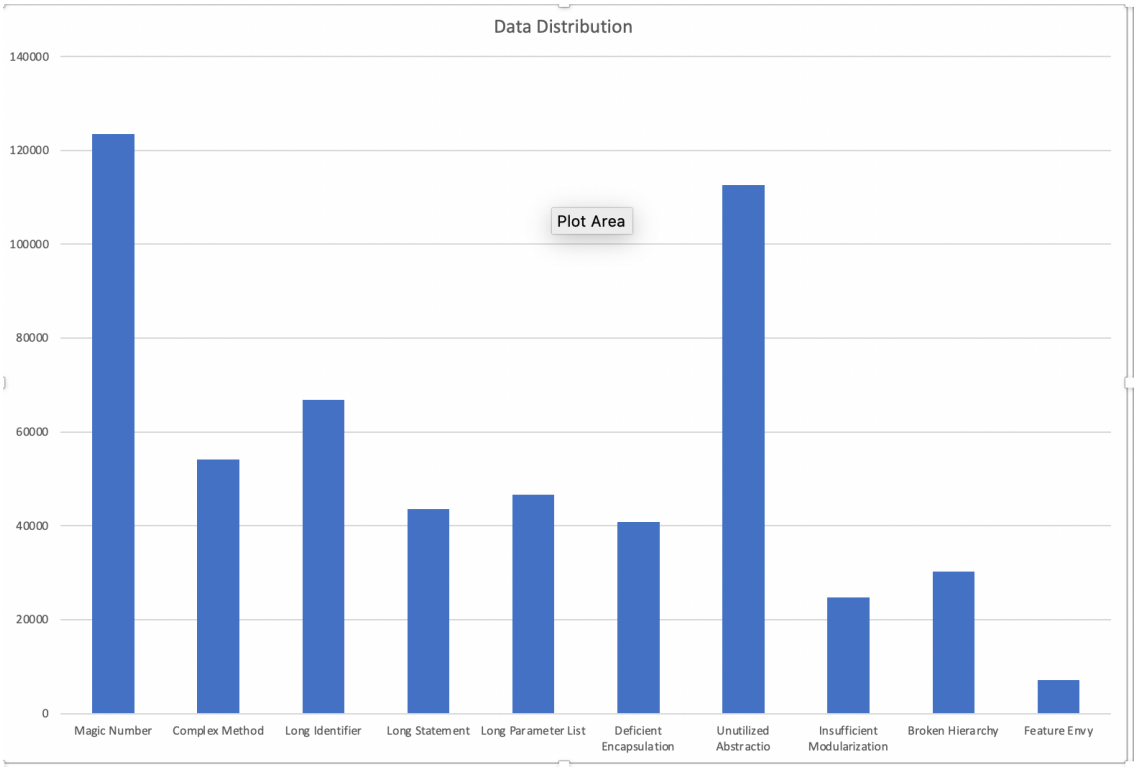

Figure 20: Code smell data distribution

The code smells, non-smells, training, validating and testing are as follows:

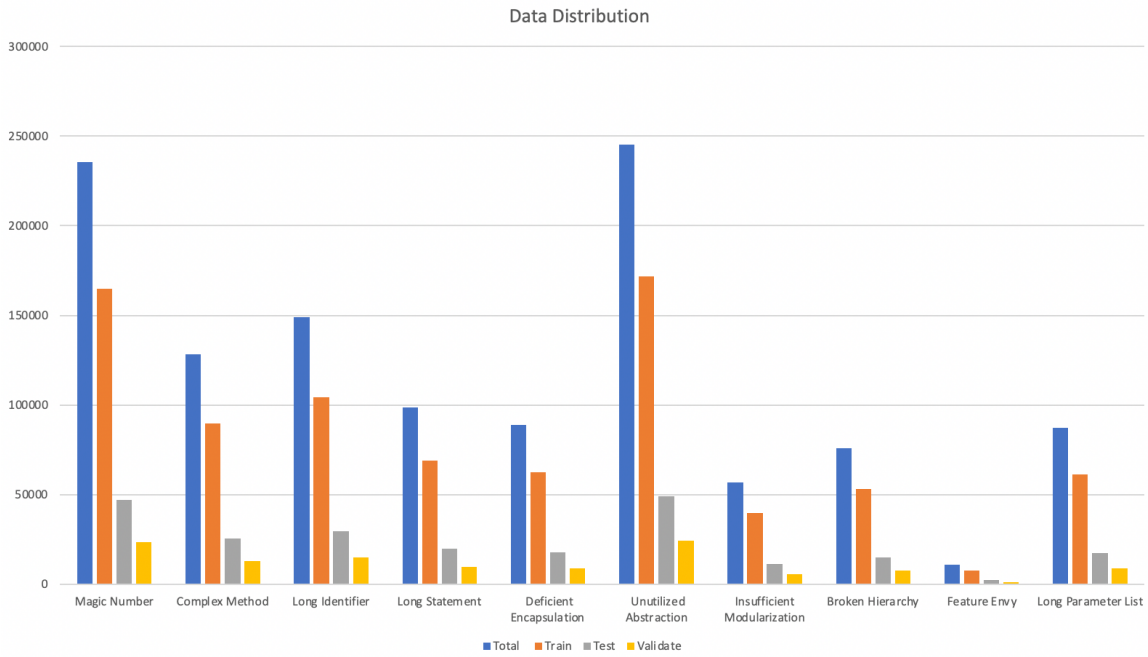

Figure 21: Data Distribution as per code smells

Total Samples	1,176,730
Total Training Samples	823,711
Training Validating Samples	235,346
Training Testing Samples	117,673
Manually Validating Samples	~150

Table 1 : Data Statistics

4.2 Experimental Results

4.2.1 Model Training and validation

The model was trained with training set (70% of total instances) and validated with validation set (10% of total instances). Following sections describe further details about the training phase.

We trained the model with following parameters:

- **batch_size:** 100,
- **learning rate:** 0.0001,
- **activation function:** ReLU,

- **Optimizer:** Adam.

Model Loss

The following figure illustrates the loss during train and validation during training.

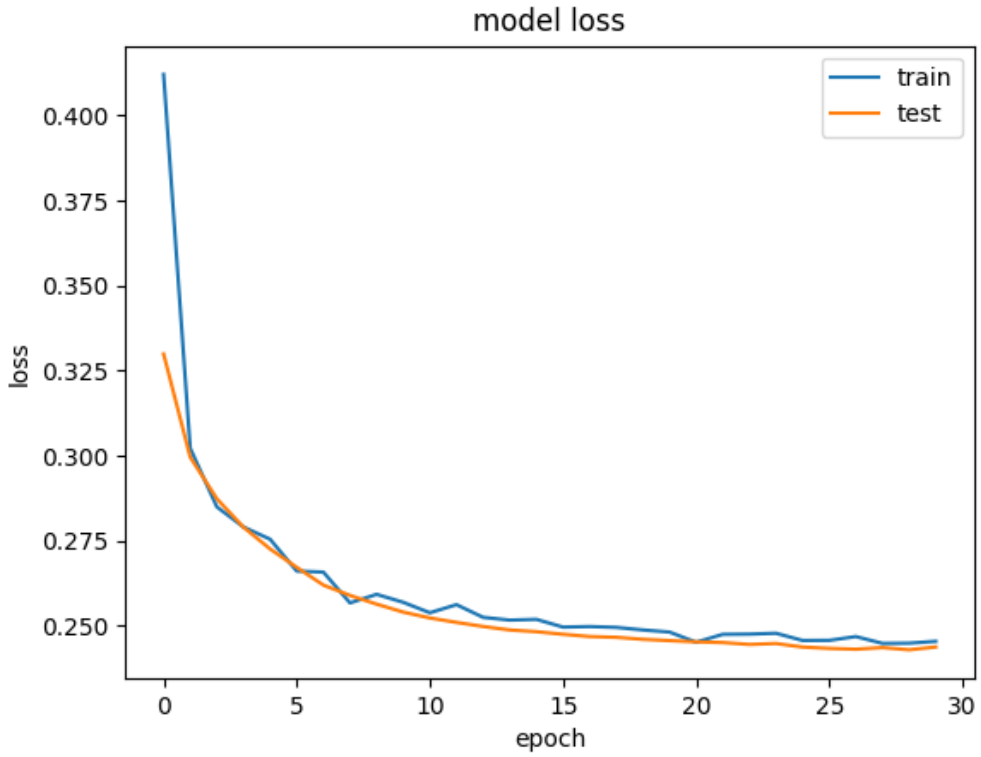

Figure 22 : During the training the loss in training and validation

Model-Loss for different learning rates is as below:

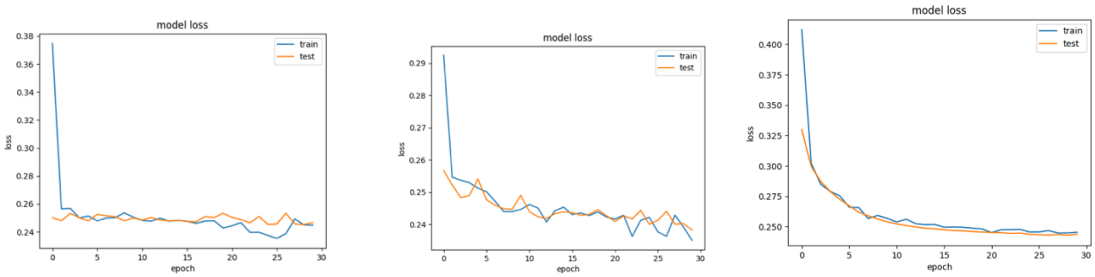

Figure 23 Model Loss with learning rate 0.01, 0.001 and 0.0001 respectively

Model training time

The model was run up to 50 epochs, and it was stopped when there is not significant difference in the validation loss value between successive iterations. Table below shows information about environment and training time.

Environment: macOS (cores i5, 16GB RAM)

Training Time: Approx. 12 hrs

4.2.2 Model Evaluation

Once the model is trained, to evaluate the performance and effectiveness of model 20% of the data are used for the purpose of testing.

This model is trained with both positive and negative sampled tokens, so the network learns their behavior. Once model is prepared it is tested with other java project by tokenizing the input.

91) **Confusion Matrix**

The confusion matrix is for the testing data an

	Confusion Matrix	
	Actual	**Predicted**
Non Code smell	50,204	5,239
Code Smell	5,120	50,456

Table 2 : Confusion Matrix

From the confusion matrix, following parameters are calculated:

Performance Metrics	**Percentage**
Accuracy	91.17
Recall (TPR)	92.34
False Positive Rate (FPR)	8.84
True Negative Rate (TNR)	91.15

Table 3 Accuracy, Recall, FPR, TNR

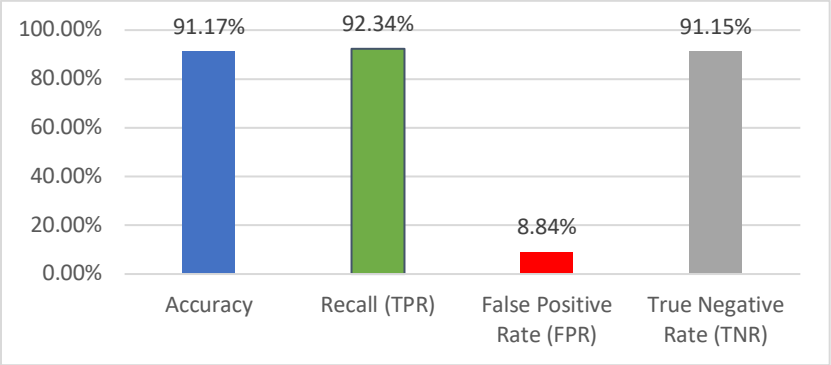

Figure 24 : Accuracy, TP and FP rate

2) ROC Plot

This ROC plot is as shown in Figure below:

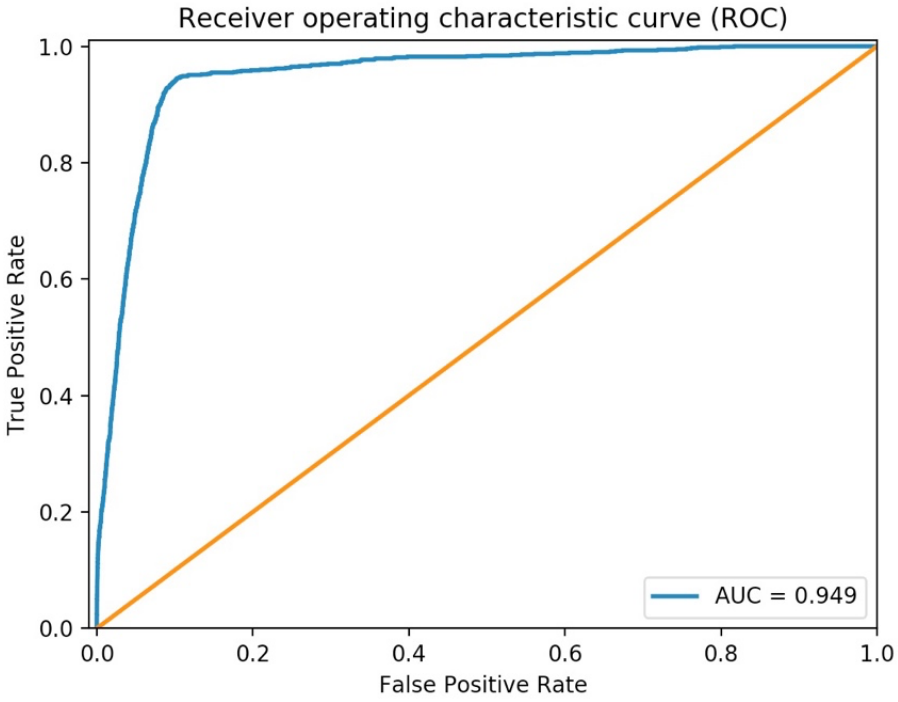

Figure 25: Combined ROC Curve of the code smells

ROC curve of each samples code smells are as below:

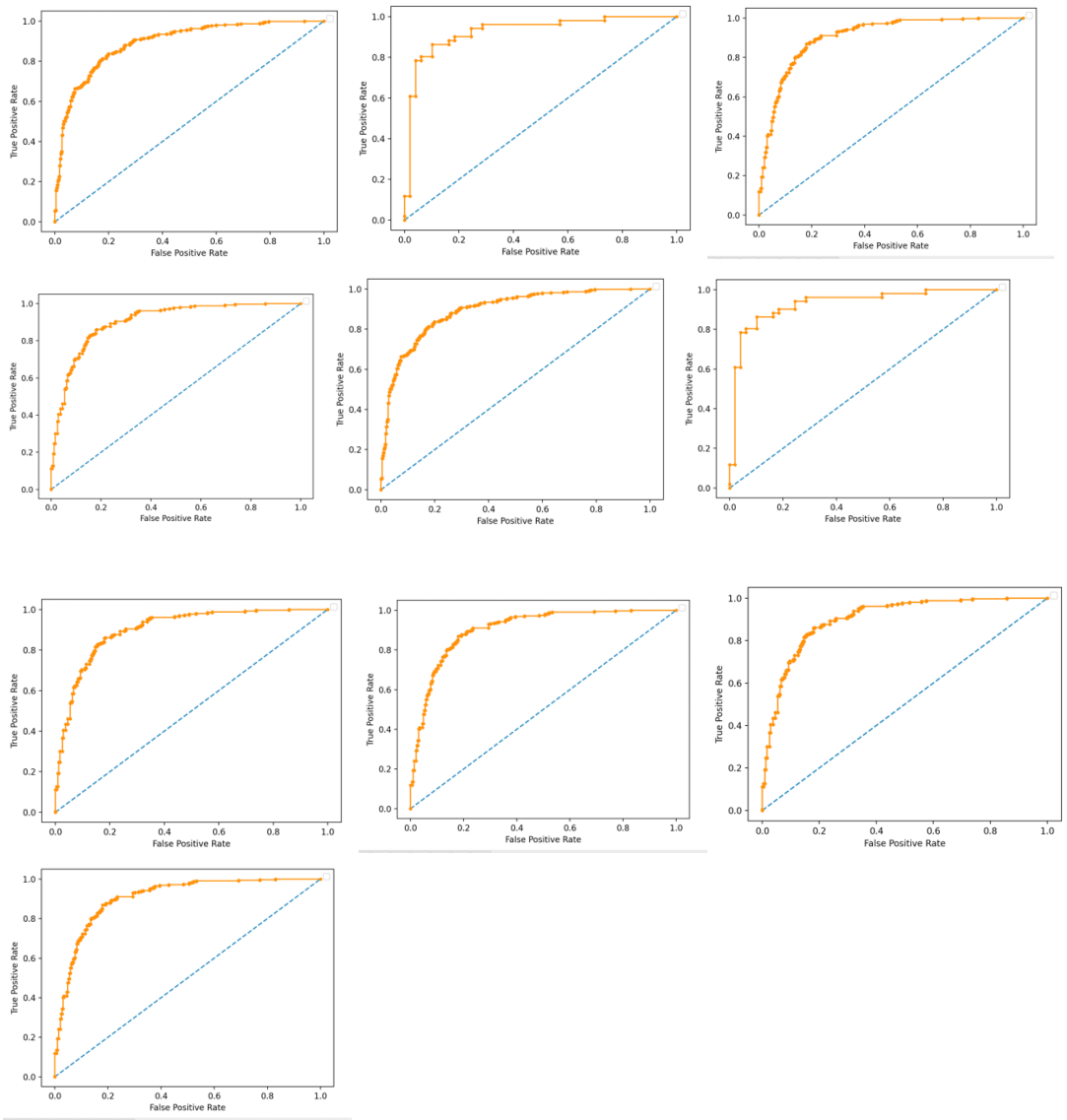

Figure 26: ROC curve of each code smells (Magic Number, Complex Method, Long Identifier, Long Statement, Long Parameter List, Deficient Encapsulation, Unutilized Abstraction, Insufficient Modularization, Broken Hierarchy and Feature Envy respectively)

3) Experiment with all the code smell types

Following chart shows TRP and FPR with different code smell types.

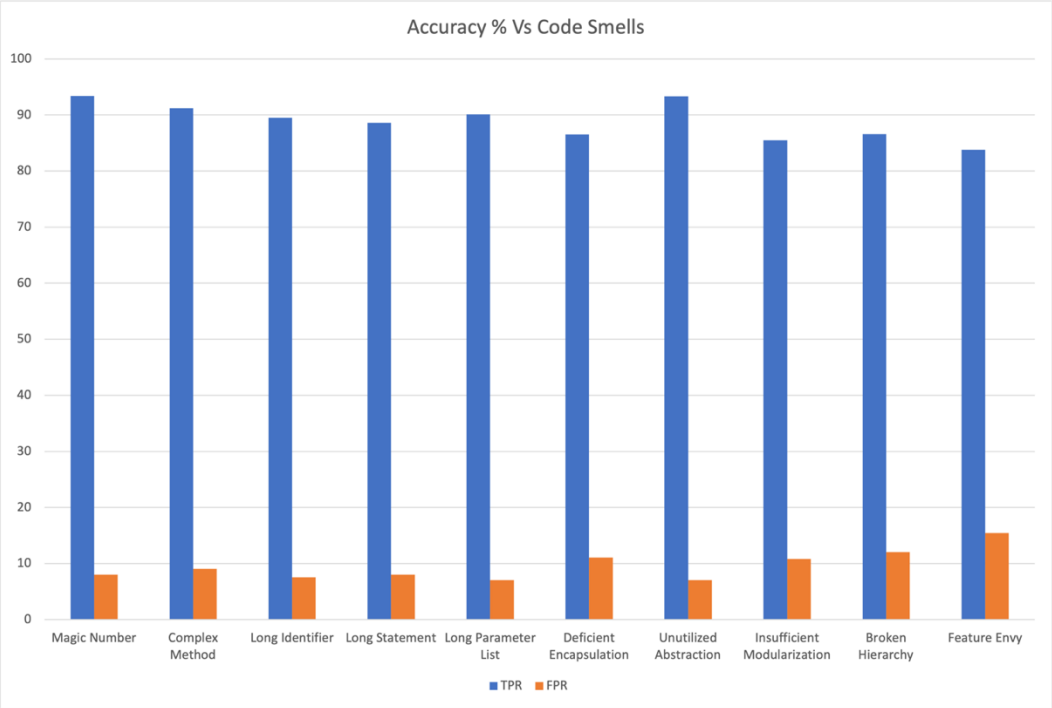

Figure 27: TPR and FPR as per code smells

4.3 Comparison with another research

Following table shows comparison of results with results on different papers done in same dataset.

SN	Paper	Model	Accuracy(%)	TPR(%)	FPR(%)
1	[17]	SVM	87.79	81.06	12.18
2	[20]	Decision-Tree	93.85	92.46	6.86
3	[23]	CNN	89.88	91.86	9.07
4	Proposed	LSTM - RNN	91.1	92.34	8.84

Table 4 : Result Comparison with other research

Following chart shown below do the various comparison based on accuracy, recall, false positive rate and true negative rate comparison.

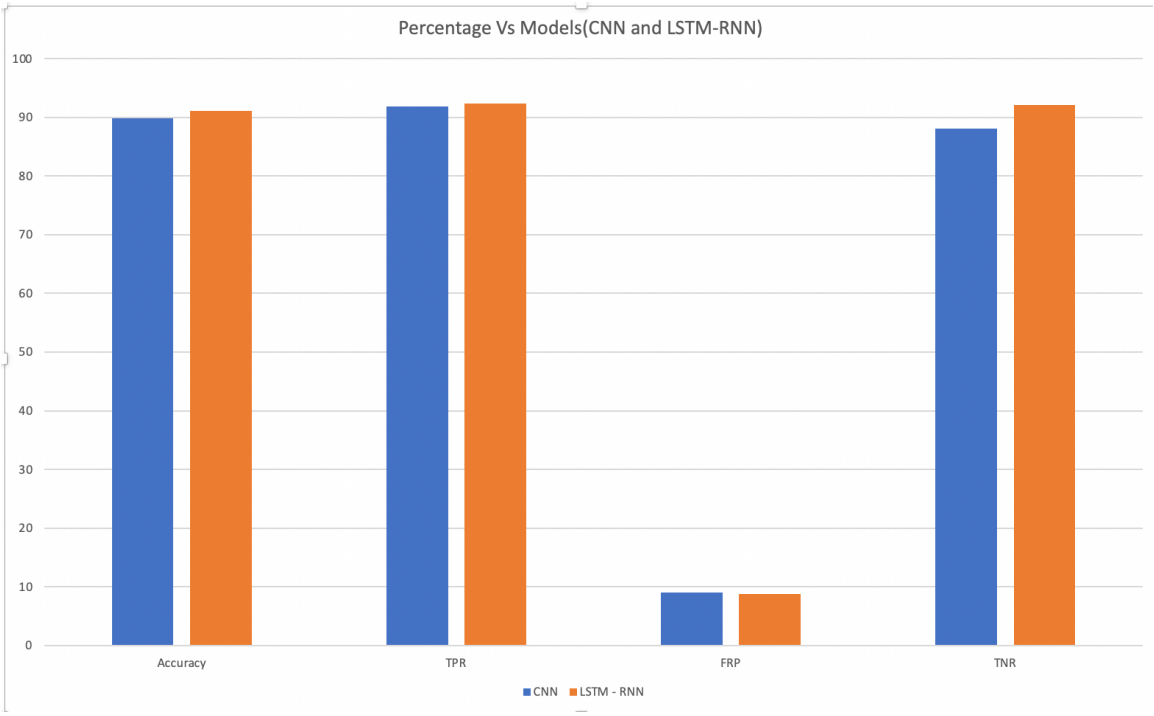

Figure 28: Comparison of performance of different models

4.4 Manual Validation and Verification

The few samples of codes fragments where manually validated against the result predicted by LSTM model. The following was the results:

- Total samples taken were 150
- Out of 150, 20 code fragments were smelly and other 130 were non-smelly.
- The same code fragments were fed to LSTM model for prediction.
- The result was as follow:
    - Smell detected: 26, i.e 6 samples were false positive.
- As per this observation following is observer result:

	Results	
	Actual	**Predicted**
Non Code smell	130	124
Code Smell	20	26

Table 5 : Manual verification result

The accuracy of the model was 84.5%. The accuracy is relatively less then automatic verification as the sample taken are less and also imbalance.

4.5 Testing the model with C# code base

Since the model was build using java code base, however the code smells those are considered are the smells those generically present in Object Oriented Programming language so to analyze the generic validity of the model, test was carried out with C# sample code. The following was the result:

	Actual	**Predicted**
Non Code smell	2088	2228
Code Smell	1196	1056

Table 6: Result with C# code base

From this test, the accuracy observed was 73.71% . This accuracy is less in comparison with Java itself as the syntax, keywords and pattern to write the code in C# is bit different then Java. However, it indicates it is possible to use this model for other similar object oriented programming languages as well.

4.6 Analysis of the results

The proposed thesis work uses LSTM-RNN machine learning model for detecting code smell. With the static code analysis tool, the sampled data were having huge difference in number of smelly code and non-smelly code. The number of non-smelly code were huge so to maintain the same ration non-smelly code were taken only as per required. The different code smells used were Magic Number, Complex Method, Long Identifier, Long Statement, Long Parameter List, Deficient Encapsulation, Unutilized Abstraction, Insufficient Modularization, Broken Hierarchy and Feature Envy. Since, the model prepare is to smelly or non-smelly code, the meeting point of TNR and TPR is called as threshold value. The experimental results on test data shows that in the best case scenario accuracy of 91.1%, TPR 92.34% and FPR 8.84 %. The main objectives in code smell detection is to increase TPR and decrease FPR. Since, code smell data are limited, they should be merely missed. However, it was found that while increasing detection rate for positives cases,

false positives are also increased.

Model performance was also analyzed by changing learning rates. On higher learning rates (0.01, 0.001) model stopped training more quickly and it didn't get converged. On too low learning rate (0.00001) model converge too slow. On moderate learning rate 0.0001, the results are found good.

After the classification of test data, some of the sequences are analyzed manually, to verify the results. The following table has some true positive, false positive and true negative cases.

Also, the manual validation and validation with another programming language (C#), shows that the prepared model can predict the code smells. Though accuracy is less in this case, it is because of data set.

CHAPTER 5: CONCLUSIONS AND FUTURE ENHANCEMENTS

5.1 Conclusions

The thesis work has proposed a method to collect, process and analyze code smell of various open-source projects. This research has accommodated the ten different code smells (Magic Number, Complex Method, Long Identifier, Long Statement, Long Parameter List, Deficient Encapsulation, Unutilized Abstraction, Insufficient Modularization, Broken Hierarchy and Feature Envy). It also proposes that it is possible to detect code smell in intelligent way using LSTM machine learning model. The validation done manually also did add more value for the authentication of the result. And the comparison of result with other related research shows that the accuracy and performance of this research is good and acceptable for implementation. Though, this research is only bound to one programming language, JAVA and has considered only ten code smell among many others but this is only the initiation for further work. Each test data is fed to the network and calculated the reconstruction error. The test data are predicted as normal, or code smelled based on the reconstruction error. The predicted smells and actual smells of test data are compared to form confusion matrix. Model's fitness is evaluated based on the different metrics. The evaluation result in testing data shows that model has True Positives 92.34%, False Positive 8.84% and Accuracy 91.17%. The objective of code smell detection is to increase true positives and decrease false positive rate. However, it was found that while increasing true positives, false positives are also increased, and accuracy decreased. The performance parameters: true positive rate, false positive rates and accuracy etc. can be controlled by choice of threshold value of classification. We can relate choice of threshold value as per implementation level time and data set of code base.

5.2 Limitations and Future Enhancements

While doing this thesis work, a lot of effort have also been spent data preparation and preparing the LSTM model for each code smell types. The model has been prepared for each code smell types as well as whole in aggregate, tune them as for more better result. As this research is limited to only one programming language, JAVA so in future it can be made for multi-language like C#, Python, C/C++ etc. Also, this research can be continued

for transfer learning. i.e. training the model in one specific language and using this for predicting the code smells of other programming language.

REFERENCES

[1] Amandeep Kaur, Sushma Jain and Shivani Goel "A Support Vector Machine Based Approach for Code Smell Detection" 2017 International Conference on Machine Learning and Data Science (MLDS)

[2] Mouna Hadj-Kacem and Nadia Bouassida "A Hybrid Approach to Detect Code Smells using Deep Learning" ENASE 2018 - 13th International Conference on Evaluation of Novel Approaches to Software Engineering

[4] Asif Imran and Tevfik Kosa "Design Smell Analysis for Developing and Established Open Source Java Software" 11 Oct 2019, Cornell University

[5] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia and Rocco Oliveto "Toward a Smell-aware Bug Prediction Model" IEEE Transactions on Software Engineering Year: 2019 | Volume: 45, Issue: 2 | Journal Article | Publisher: IEEE

[6] Antoine Barbez, Foutse Khomh and Yann-Gaël Guéhéneuc "Deep Learning Anti-Patterns from Code Metrics History" 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)

[7] Bjorn Latte, Soren Henning, Maik Wojcieszak "Clean Code: On the Use of Practices and Tools to Produce Maintainable Code for Long-Living Software" EMLS 2019: 6th Collaborative Workshop on Evolution and Maintenance of Long-Living Systems @ SE19,
Stuttgart, Germany

[8] Sevilay Velioğlu, Yunus Emre Selçuk "An automated code smell and anti-pattern detection approach" 2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)

[9] E. van Emden, L. Moonen "Java quality assurance by detecting code smells" Ninth Working Conference on Reverse Engineering, 2002. Proceedings.

[10] Ahmad Tahmid, Nadia Nahar and Kazi Sakib "Understanding the Evolution of Code Smells by Observing Code Smell Clusters" 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

[11] Fowler, Martin (1999). “Refactoring. Improving the Design of Existing Code”. Addison- Wesley. ISBN 978-0-201-48567-7.

[12] Tufano Michele, Palomba Fabio, Bavota Gabriele, Oliveto Rocco, Di Penta Massimiliano, De Lucia Andrea, Poshyvanyk Denys (2015). "When and Why Your Code Starts to Smell Bad". 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. pp. 403–414. CiteSeerX 10.1.1.709.6783

[13] Thirupathi Guggulothu and Salman Abdul Moiz "Code Smell Detection using Multilabel Classification Approach" School of Computer and Information Sciences, University of Hyderabad, Hyderabad-500 046, Telangana, India

[14] Meyer, Daniel: Metrics-Based Code Smell Detection in Highly Configurable Software Systems Master’s Thesis, University of Magdeburg, 2015.

[15] Amandeep Kaur and Gaurav Dhiman "A Review on Search-Based Tools and Techniques to Identify Bad Code Smells in Object-Oriented Systems"

[16] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, Andrea De Lucia "Detecting Code Smells using Machine Learning Techniques: Are We

There Yet?" Conference: 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering, At Campobasso, Italy

[17] Amandeep Kaur; Sushma Jain; Shivani Goel - A Support Vector Machine Based Approach for Code Smell Detection, 2017 International Conference on Machine Learning and Data Science (MLDS)a

[18] A. Singh, "Anomaly Detection for Temporal Data using Long Short-Term Memory (LSTM)," 2017.

[19] Fanzhi Meng et al., "Deep Learning based Attribute Classification Insider Threat Detection for Data Security," 2018 IEEE Third International Conference on Data Science in Cyberspace, 2018.

[20] Mohammad Y. Mhawish, Manjari Gupta2 - Generating Code-Smell Prediction Rules Using Decision Tree Algorithm and Software Metrics, International Journal of Computer Sciences and Engineering

[21] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. “House of Cards: Code Smells in Open-Source C# Repositories". in ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 424-429. 10.1109/ESEM.2017.57.

[22] Muhammad Ilyas Azeema, Fabio Palomba, Lin Shi, Qing Wanga - Machine learning techniques for code smell detection: A systematic literature review and meta-analysis

[23] Tao Lin, Xue FuFu Chen, Luqun Li "A Novel Approach for Code Smells Detection Based on Deep Leaning" EAI International Conference on Applied Cryptography in Computer and Communications

APPENDIX – I

MSCKE_Final_Thesis_Report_for_SI/073mscke_664_sanjay_...

ORIGINALITY REPORT

17%
SIMILARITY INDEX

PRIMARY SOURCES

1	flipkarma.com Internet	317 words — 4%
2	arxiv.org Internet	154 words — 2%
3	Fanzhi Meng, Yunsheng Fu, Fang Lou. "A network threat analysis method combined with kernel PCA and LSTM-RNN", 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI), 2018 Crossref	140 words — 2%
4	Balaram Sharma, Prabhat Pokharel, Basanta Joshi. "User Behavior Analytics for Anomaly Detection Using LSTM Autoencoder - Insider Threat Detection", Proceedings of the 11th International Conference on Advances in Information Technology, 2020 Crossref	94 words — 1%
5	kth.diva-portal.org Internet	82 words — 1%
6	citeseerx.ist.psu.edu Internet	38 words — 1%
7	mafiadoc.com Internet	32 words — < 1%