DETECTING INACCURATE STACK TRACES IN BUG REPORTS

MEHER KIRAN BHEREE
Bachelor of Technology, GITAM University, 2015

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

DETECTING INACCURATE STACK TRACES IN BUG REPORTS

MEHER KIRAN BHEREE

Date of Defence: August 16, 2022

Dr. J. Anvik Thesis Supervisor	Associate Professor	Ph.D.
Dr. W. Osborn Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. H. Cheng Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. J. Sheriff Chair, Thesis Examination Committee	Assistant Professor	Ph.D.

Dedication

A thesis is written in honour of my late father, Subhadhra Rao Bheree, who constantly encouraged and supported me in my academic endeavours.

Abstract

The generally held opinion in the software engineering community is that incorrect information in bug reports is often found in non-structural fields such as *bug descriptions* and *steps to reproduce*. However, structural information such as software stack traces can be inaccurate, increasing the project costs due to wasted time in fixed faults. Regarding the occurrence of inaccurate stack traces in bug reports, there is little empirical evidence. Therefore, we seek to provide such evidence by conducting an empirical study on the bug reports containing stack traces from the Eclipse and Apache projects.

We propose an approach to classify the stack traces as either “Accurate” or “Inaccurate” by comparing the file names found in a stack trace in a bug report and the corresponding commit history for its fix. Thus, we determine the occurrence of inaccurate stack traces and identify the frequently occurring exception types that appears in the inaccurate stack traces for each project.

Finally, we investigate training three supervised machine learning algorithms (Naive Bayes, Support Vector Machines and Logistic Regression), on features extracted from stack traces to create recommender that labels stack traces in bug reports as either ”Accurate” or ”Inaccurate”. The Logistic Regression algorithm was found to perform better with a *F1-score* up to *87%* for the investigated Eclipse projects and *96%* for the investigated Apache projects.

Acknowledgments

I express my sincere gratitude to my supervisor Dr. John Anvik, whose guidance, support and encouragement have been priceless throughout my research. I also want to thank my committee members, Dr. Wendy Osborn and Dr. Howard Cheng for their constant support and motivation.

Many thanks to Dr. John Sheriff for being the Examination chair of my thesis defence. I am extremely grateful to the School of Graduate Studies (SGS) and its member staff for helping me financially throughout my Master's in Canada.

It is a great pleasure to thank my parents, Jyothi Yedla (sister), Shiva Amireddy (brother) and other family members, who guided me positively and made me feel confident in my abilities. Special thanks to my wife Sudha Atti for her unconditional support and understanding.

To conclude, I had a great experience with everyone in the Sibyl lab for all the virtual conversations and social gatherings. And I would like to thank Ajay Tedlapu, Abha Goel, Shahul Shaik, Manvitha Pandiri, Vaishnavi Alluri, Mrudula Bangaru, Jaya Peddinti and Prashanth Chinthapalli for all the support in this very intense Master's journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many large software companies rely on issue tracking systems (e.g., Bugzilla[1], Jira[2]) to manage feature requests and bug reports. A feature request directly or indirectly relates to a user requirement, such as upgrading software or adding new functionality to the existing software. Bug reports are created by a user or tester when they perceive that the software is no longer delivering the desired outcome with the specified input values, which is referred to as a software failure. Depending on the degree of the failure, the "bug reporter" or "bug submitter" may need to report the failure and classify the failure in the issue tracking system. As a result, more flaws in the software projects may be discovered and repaired. Often bug reports are comprised of a variety of information like bug descriptions, stack traces, patches, steps to reproduce, screenshots and code samples [2]. They include a detailed explanation of the failure and, on rare occasions, a pointer to where the error in the code might be found. This allows users to alert developers to issues they discovered while using a piece of software.

According to Boehm and Basili, up to 70% of the cost of software is spent on debugging and maintenance, whereas 30% is spent on development [3]. Software debugging is a complex process, and it frequently necessitates searching through millions of lines of code to find the source of an issue. Thus, developers often spend their time debugging the issue rather than developing a new piece of code to add or change functionality. Some bug reports contain incomplete or incorrect information provided by the end-user in the bug

[1] https://www.bugzilla.org/

[2] https://www.atlassian.com/software/jira

report. This can lead to delays in the debugging process, thereby increasing software project maintenance costs.

A previous study by Bettenburg *et al.* [2] examined different sections in the bug report and interviewed 156 Developers from three different software projects. Their study shows that the steps to reproduce are the essential piece of information, and stack traces are the next most important. Steven *et al.* [5] investigated bug reports from Facebook, Apache, and Eclipse and noticed that users provide screenshots, stack traces and test cases in less than 10% of all the bug reports. Therefore, bug reports should evolve to contain more structural information. In another research, Chaparro *et al.* [4] focused on the steps to reproduce in bug reports, identifying and assessing the quality of these steps automatically and providing feedback to the reporters, which they can use to improve the bug report.

In short, previous studies have focused on the bug description and steps to reproduce and determined that they often contain incorrect information, but little prior work has investigated the accuracy of stack traces. Thus, in this work, we downloaded the bug reports and commit history for eight open source Java-based projects. We compare the names of committed files from the commit history to the names of stack trace files. If the file names match, we consider the stack trace to be Accurate; otherwise, it is Inaccurate.

1.1 Motivating Example

Stack traces in bug reports can be helpful as they assist the developers in the debugging process [13]. A stack trace can narrow down the list of files likely to contain the defect. Also, software vendors, including Microsoft, Apple, and Mozilla, are improving built-in support to send stack traces back to developers when the software crashes.

Although stack traces have several advantages and disadvantages. When we examined specific bug reports from some open-source projects, specifically those from the Eclipse and Apache projects, we found that the stack traces could be inaccurate regarding the files indicating where the code will need to be fixed.

Take, for example, bug report #397872 from the Eclipse project [8], as shown in Figure 1.1. This bug report was filed in January 2013, with a description reporting that there is an *NullPointerException* in the RCP application[3] User interface. Figure 1.1 shows that the reporter provided steps to reproduce, *normal* severity and attached a stack trace. On checking the comments, it is clear that the developer focused on the *WorkbenchPage.java*

file as specified in the stack trace. However, later in November 2013, a different project developer figured out that the actual cause was because of the *addpart()* method from *PartServiceImpl.java* file and provided a fix to it. Finally, this bug report was verified and closed in December 2013, meaning it took the project a year to resolve the issue because of the inaccurate stack trace.

Another example is bug report #15358 from the Cassandra project [9] in October 2019. The report describes a software failure migrating from Cassandra version 3.11.4 to version 4.0-alpha1. The client discovered this fault and reported this bug with *normal* severity, giving a potentially inaccurate stack trace to the problem. As shown in Figure 1.2, the stack trace displays an exception type *IllegalArgumentException* and an ordered list of class tracebacks for the problem. The developer focused on the *IllegalArgumentException* as shown by the comments in the bug report and investigated the classes in the stack trace making code changes. Seven months later, in April 2020, the project determined that this defect was the *heapbuffer()* method in the *DistributedReadWritePathTest* class, which is not referred to in the stack trace, leading to the project taking 183 days to close the bug report.

The above examples show how an inaccurate stack trace can result in extended fixing times, in these cases six months to a year, thereby causing additional costs to the software project's development.

[3] https://www.eclipse.org/articles/Article-RCP-1/tutorial1.html

Bug 397872 - [Workbench] NPE in WorkbenchPage#busyShowView

Status: VERIFIED FIXED
Alias: None
Product: Platform
Component: UI (show other bugs)
Version: 4.2.1
Hardware: PC Linux
Importance: P3 normal (vote)
Target Milestone: 4.2.2+
Assignee: Wojciech Sudol
QA Contact: Wojciech Sudol
URL:
Whiteboard:
Keywords:
Depends on:
Blocks:

Reported: 2013-01-10 10:08 EST by Emily Middleton
Modified: 2015-06-24 12:08 EDT (History)
CC List: 7 users (show)
See Also:

Stack trace:
org.eclipse.e4.core.di.InjectionException: java.lang.NullPointerException
at org.eclipse.e4.core.internal.di.MethodRequestor.execute(MethodRequestor.java:63)
at org.eclipse.e4.core.internal.di.InjectorImpl.invokeUsingClass(InjectorImpl.java:229)
at org.eclipse.e4.core.internal.di.InjectorImpl.invoke(InjectorImpl.java:210)
at org.eclipse.e4.core.contexts.ContextInjectionFactory.invoke(ContextInjectionFactory.java:131)
at org.eclipse.e4.core.commands.internal.HandlerServiceImpl.executeHandler(HandlerServiceImpl.java:171)
at org.eclipse.ui.internal.handlers.LegacyHandlerService.executeCommand(LegacyHandlerService.java:500)
at org.eclipse.ui.internal.ShowViewMenu$3.run(ShowViewMenu.java:141)
at org.eclipse.jface.action.Action.runWithEvent(Action.java:498)
at org.eclipse.jface.action.ActionContributionItem.handleWidgetSelection(ActionContributionItem.java:584)
at org.eclipse.jface.action.ActionContributionItem.access$2(ActionContributionItem.java:501)
at org.eclipse.jface.action.ActionContributionItem$5.handleEvent(ActionContributionItem.java:411)
at org.eclipse.swt.widgets.EventTable.sendEvent(EventTable.java:84)
at org.eclipse.swt.widgets.Widget.sendEvent(Widget.java:1276)
at org.eclipse.swt.widgets.Display.runDeferredEvents(Display.java:3554)
at org.eclipse.swt.widgets.Display.readAndDispatch(Display.java:3179)
at org.eclipse.e4.ui.internal.workbench.swt.PartRenderingEngine$9.run(PartRenderingEngine.java:1029)
at org.eclipse.core.databinding.observable.Realm.runWithDefault(Realm.java:332)
at org.eclipse.e4.ui.internal.workbench.swt.PartRenderingEngine.run(PartRenderingEngine.java:923)
at org.eclipse.e4.ui.internal.workbench.E4Workbench.createAndRunUI(E4Workbench.java:86)
at org.eclipse.ui.internal.Workbench$5.run(Workbench.java:588)
at org.eclipse.core.databinding.observable.Realm.runWithDefault(Realm.java:332)
at org.eclipse.ui.internal.Workbench.createAndRunWorkbench(Workbench.java:543)
at org.eclipse.ui.PlatformUI.createAndRunWorkbench(PlatformUI.java:149)
at org.eclipse.ui.internal.ide.application.IDEApplication.start(IDEApplication.java:124)
at org.eclipse.equinox.internal.app.EclipseAppHandle.run(EclipseAppHandle.java:196)
at org.eclipse.core.runtime.internal.adaptor.EclipseAppLauncher.start(EclipseAppLauncher.java:79)
at org.eclipse.core.runtime.adaptor.EclipseStarter.run(EclipseStarter.java:353)
at org.eclipse.core.runtime.adaptor.EclipseStarter.run(EclipseStarter.java:180)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:601)
at org.eclipse.equinox.launcher.Main.invokeFramework(Main.java:629)
at org.eclipse.equinox.launcher.Main.basicRun(Main.java:584)
at org.eclipse.equinox.launcher.Main.run(Main.java:1438)
Caused by: java.lang.NullPointerException
at org.eclipse.ui.internal.WorkbenchPage.busyShowView(WorkbenchPage.java:1179)
at org.eclipse.ui.internal.WorkbenchPage$9.run(WorkbenchPage.java:3774)
at org.eclipse.swt.custom.BusyIndicator.showWhile(BusyIndicator.java:70)
at org.eclipse.ui.internal.WorkbenchPage.showView(WorkbenchPage.java:3771)
at org.eclipse.ui.internal.WorkbenchPage.showView(WorkbenchPage.java:3747)
at org.eclipse.ui.handlers.ShowViewHandler.openView(ShowViewHandler.java:126)
at org.eclipse.ui.handlers.ShowViewHandler.openOther(ShowViewHandler.java:99)
at org.eclipse.ui.handlers.ShowViewHandler.execute(ShowViewHandler.java:67)
at org.eclipse.ui.internal.handlers.HandlerProxy.execute(HandlerProxy.java:290)
at org.eclipse.ui.internal.handlers.E4HandlerProxy.execute(E4HandlerProxy.java:76)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:601)
at org.eclipse.e4.core.internal.di.MethodRequestor.execute(MethodRequestor.java:56)
35 more

Wojciech Sudol 2013-11-22 14:10:35 EST Comment 11

The problem is caused by the PartServiceImpl.addPart(MPart, MPart) method. When the Help view is focused, the partServiceImpl.getContainer() method returns the MWindow instead of active perspective (because the Help view is inside the right "sticky folder", which in turn is outside the perspective stack). Then EModelService.findElements(*) returns all suitable part containers from all perspectives. Eventually, the newly opened view is added to the first of the containers, which is not always inside the active perspective.
My fix proposition is to accept only part containers from the active perspective.

Review URL: https://git.eclipse.org/r/#/c/18754/

During testing I have not found any side effects of the fix.

Eric Moffatt 2013-11-27 10:45:30 EST Comment 12

Committed (on Wojciech's behalf), thanks !

http://git.eclipse.org/c/platform/eclipse.platform.ui.git/commit/?id=3b3894f712dd09df69050b01fa492960a9705a96

Wojciech Sudol 2013-12-10 08:04:54 EST Comment 13

Verified in I20131209-2000.

Paul Webster 2013-01-11 08:32:32 EST Comment 1

I can reproduce this from an empty workspace.

Eric, do you think this might be because there's a placeholder for call hierarchy in the java perspective that hasn't been instantiated?

PW

Emily Middleton 2013-01-11 09:14:50 EST Comment 2

On further investigation I can reproduce this with views other than Call Hierarchy. Of the Java views, you can also see this with Declaration, Javadoc and JUnit but not Package Explorer or Type Hierarchy. I have not tried views from other categories.

Eric Moffatt 2013-01-16 14:33:22 EST Comment 3

I've just tried this on M20130109-1200 and so far don't see any issues regardless of the view / perspective I use.

Is it possible that there's some missing required bundles in your configuration ? What I'm thinking is that it's possible to include the bundle *defining* the view (so it appears in the dialog) without having all the bundles it needs included (so it crashes on creation).

You're correct though in that if 'compatibilityView' code at the end of the method should be refactored, likely to throw an exception (PartInitException?) if true and just proceed otherwise...

Paul Webster 2013-01-16 14:48:57 EST Comment 4

(In reply to comment #3)
> I've just tried this on M20130109-1200 and so far don't see any issues
> regardless of the view / perspective I use.
>
> Is it possible that there's some missing required bundles in your
> configuration ? What I'm thinking is that it's possible to include the
> bundle *defining* the view (so it appears in the dialog) without having all
> the bundles it needs included (so it crashes on creation).

I was able to reproduce this with the SDK and starting in a completely empty workspace, following the steps from comment #0

Eric, could you try again? If you can't reproduce this maybe it's a linux only problem

PW

Markus Duft 2013-10-03 06:34:34 EDT Comment 5

we have this problem in a custom RCP application since switching from 3.8 to 4.3. you have my vote :)

Eric Moffatt 2013-10-03 15:58:52 EDT Comment 6

Got it...I suspect I didn't select *all* the views...

I'll put this one next on my hit list.

Eric Moffatt 2013-10-07 14:32:08 EDT Comment 7

I just tried the scenario as given in the first comment again with 4.4.0.I20131001-0800 and don't seem to be able to reproduce anymore, can anybody else verify that it's fixed ?

Piotr Aniola 2013-10-23 11:37:59 EDT Comment 8

I am able to reproduce on:
Version: 4.4.0
Build id: I20131001-0800

I will try to fix this.

Piotr Aniola 2013-11-04 11:25:19 EST Comment 9

The issue seems to be that WorkbenchPage.getViewReference returns null for the part corresponding to the Call Hierarchy view.

Piotr Aniola 2013-11-05 11:35:35 EST Comment 10

Note to self:
PartServiceImpl.isInContainer returns false for Call Hierarchy, preventing the corresponding ViewReference to be added to viewReferences. Evenatually, this causes WorkbenchPage.getViewReference to return null.

Figure 1.1: Example of a bug report with an inaccurate stack trace from Eclipse

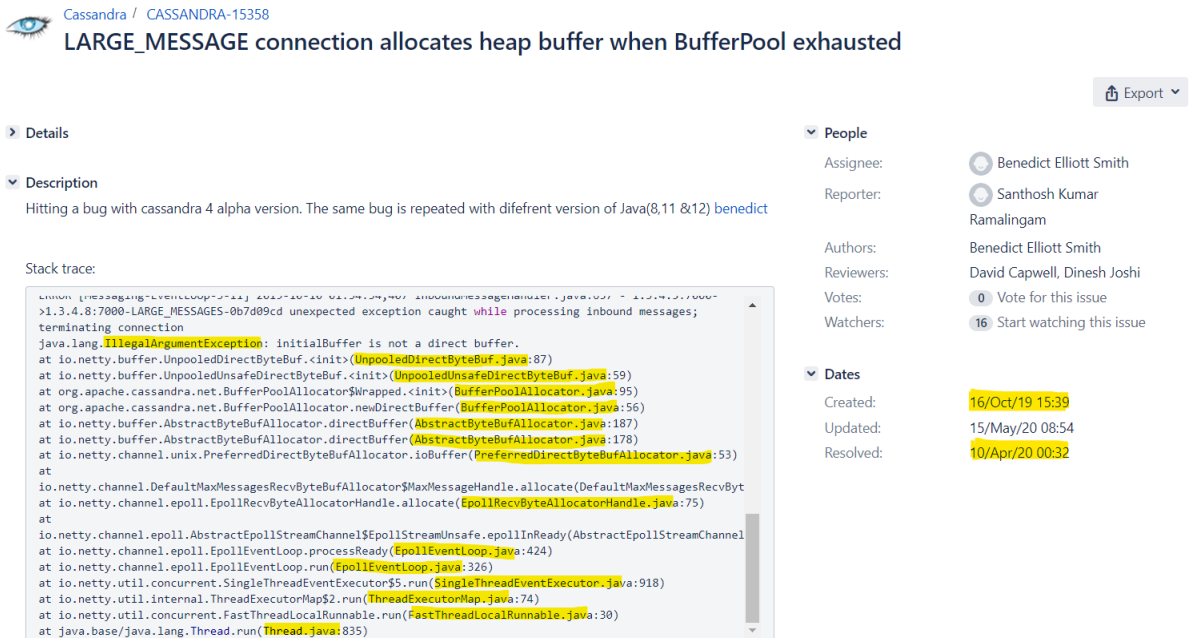

Figure 1.2: Example of a bug report with an inaccurate stack trace from Cassandra.

1.2 Research Questions

In this thesis, we examine the frequency of inaccurate stack traces in bug reports and the use of machine learning classifiers to alert developers to their presence. This leads to the following research questions.

1.2.1 RQ 1: How often do inaccurate stack traces occur in bug reports?

During the debugging process, it is common for a developer to begin debugging with the files that appear in a stack trace. If a stack trace is inaccurate, the developer might lose time examining unnecessary files. In this research, we investigate how often stack traces reference files which are not part of the eventual fix.

1.2.2 RQ 2: What exceptions are more likely to occur in inaccurate stack traces?

While implementing a feature in a software project, a developer may or may not catch thrown exceptions. If the exceptions are not catched, the software will crash, and the thrown exception will appear in the stack trace, indicating at which point the exception was thrown

in the execution. Analyzing the stack traces in a project's bug reports to find what type of exceptions occur commonly in inaccurate stack traces can help developers to be more cautious in their investigations. If a particular exception is frequently associated with an inaccurate stack trace, then if that exception occurs, the developer will know to be more skeptical about the information and look more carefully when debugging.

1.2.3 RQ 3: What is the effectiveness of using different machine learning algorithms in tagging stack traces as accurate or inaccurate?

We investigate using a machine learning classifier to flag stack traces as potentially inaccurate. We examined classifiers created using a variety of machine learning algorithms, specifically classifiers trained using Naive Bayes (NB), Support Vector Machine (SVM) and Logistic Regression (LogR), to discover which machine learning algorithm produces better results.

1.3 Contributions

This thesis makes the following contributions:

1. An approach to identifying inaccurate stack traces in bug reports.
2. The results of an investigation as to the frequency of inaccurate stack traces in bug reports.
3. Identification of the commonly occurring exception types in inaccurate bug reports.
4. The results of an investigation into using machine learning to create a classifier to identify inaccurate stack traces.

1.4 Thesis Organization

This thesis proceeds as follows. In the second chapter, we covered a few background concepts crucial to understanding this work.

In Chapter 3, we discuss related work researching bug report structure, use of bug reports for prediction, and managing crash reports. In Chapter 4, we explain our overall approach.

In Chapter 5, we performed experimental evaluations and present results to our three research questions. In Chapter 6, we discuss the significance of our results. The thesis concludes in Chapter 7 with ideas for future directions and limitations from this work.

Chapter 2

Background

Understanding our approach requires background knowledge about the information found in bug reports, version control systems, the Infozilla tool, the machine learning algorithms used in our investigation and the metrics used to evaluate our recommenders.

2.1 Bug Reports

A sample bug report for JDT[4] that was recorded in Bugzilla is shown in Figure 2.1. Specific content such as free-form text, attachments, and dependencies may be included in each bug report. Various categorical information regarding the bug report is provided through the pre-defined fields such as reporter, creation date, and report identification number. The data for these pre-defined fields is provided when the report is created. Other parameters, including those for the product, component, operating system, version, priority, and severity, are chosen by the reporter when the report is submitted but may also be modified during the report's existence. The assignee i.e., to whom the report is to be assigned, and its current status, either open or closed, often change over time.

The report's title, a thorough explanation of the defect, and other comments are all included in the free-form text. The complete description includes a detailed explanation of the bug's impact and the steps required for a developer to reproduce the bug. The additional comments may contain further information about the issue such as describing potential bug fixes and include links to other bugs that appear to be duplicate reports.

[4]JDT offers the tool plug-ins necessary to construct a Java IDE to support any Java application's development.

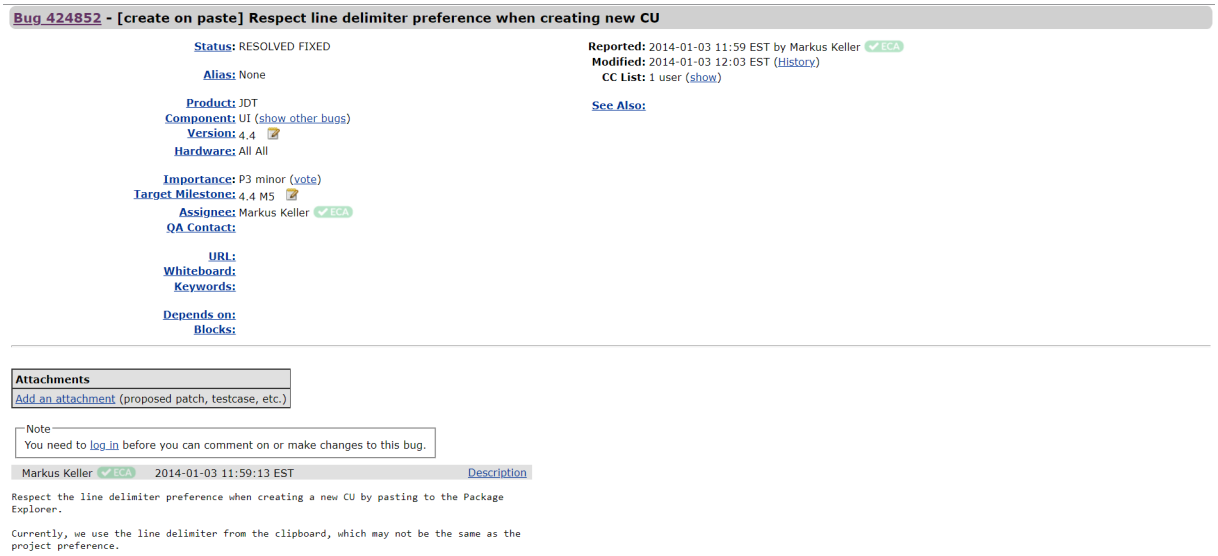

Figure 2.1: An example bug report from JDT project.

Developers and reporters may include structural information in reports, such as screenshots in the form of attachments and stack traces, patches and source code, either in text form or as attachments. The activity log of a bug report offers a historical account of how the report has changed over time, including any reassignments or changes in priority.

2.1.1 Components of Bug Reports

Stack traces:

A typical stack trace, as shown in Figure 2.2, consists of an ordered list of methods or stack frames[5] that were active on the call stack before an exception or error occurred. Each frame contains the fully-qualified name of the method and the exact location of the execution inside the source code through a file name and line number.

Thus, a stack trace holds all the filenames and method names involved in the program flow, from the point of an exception arising back to the start of execution.

[5] A stack frame represents a single function call containing file path, class name and a method name.

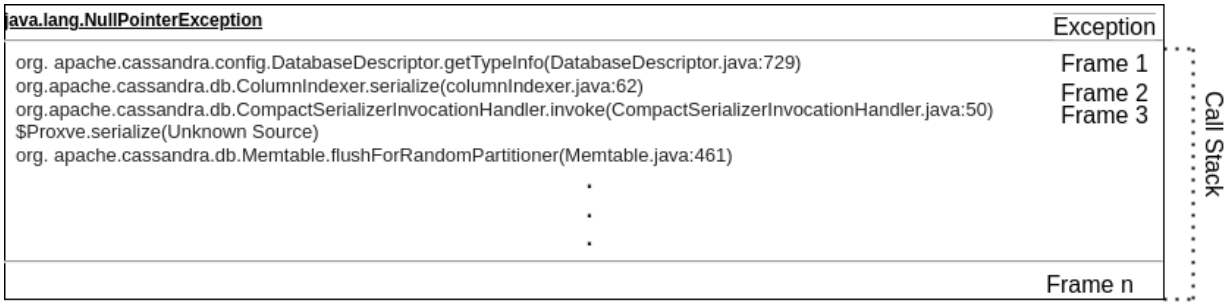

Figure 2.2: An example Stack Trace.

Patches:

A patch is a set of changes to the project files (e.g. source code, configuration) that fixes a defect in a software product. These changes are most often given as a collection of differences between two versions of a file as shown in Figures 2.3.

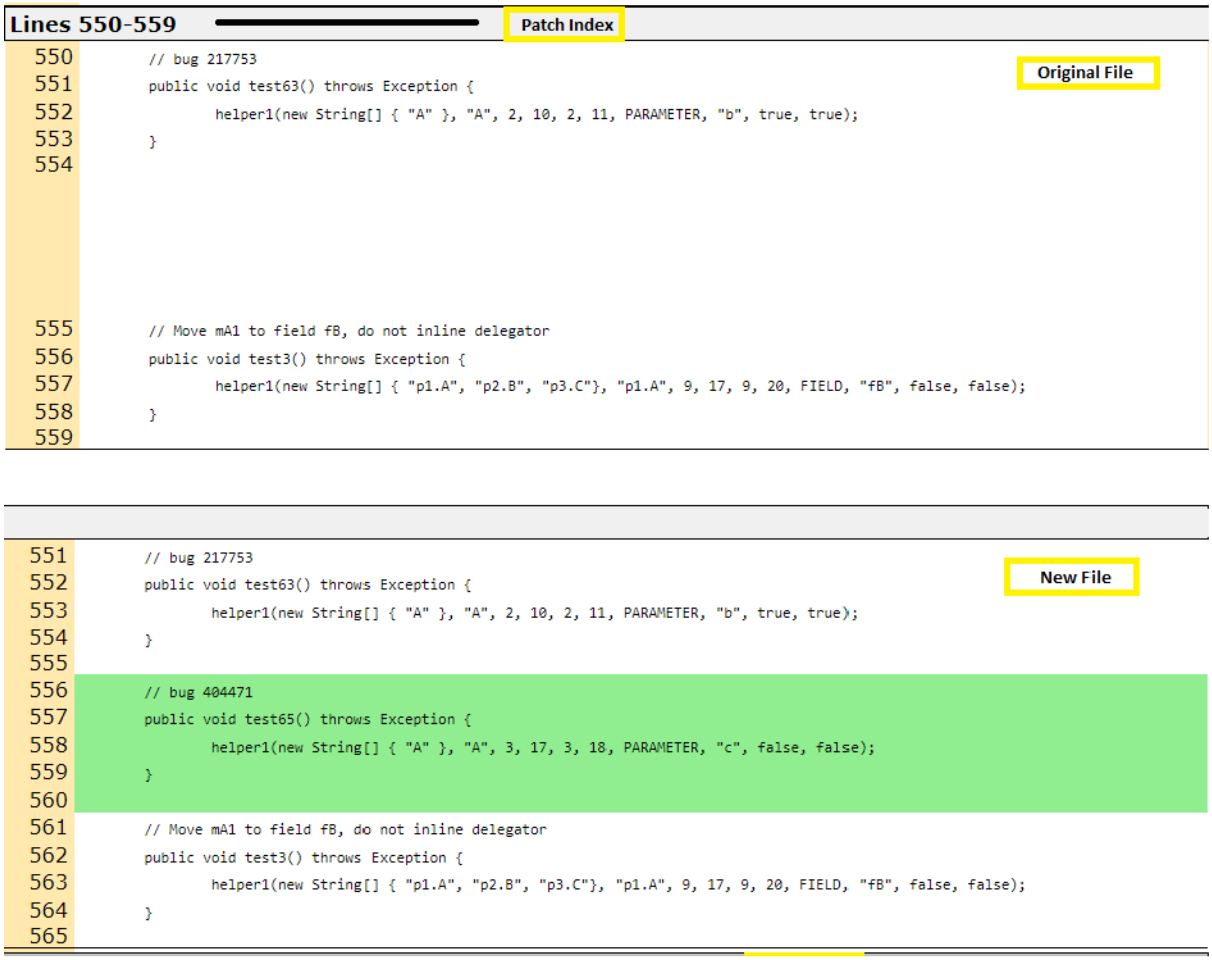

Figure 2.3: An example patch.

Source Code:

A source code is used to demonstrate an issue, indicate the programming context in which a problem originated, describe the environment in which it occurred or even offer a prototype solution to the problem detailed in the bug report. An example is shown in Figure 2.4.

```
-----------------------------------------------------------------
import java.io.Serializable;

public class Bean implements Serializable{

        private String name;

        public String getName() {
                return name;
        }
        public void setName( String name ) {
                this.name = name;
        }
}
-----------------------------------------------------------------
```

Figure 2.4: An example of source code snippet in a bug report.

Steps to reproduce:

Steps to reproduce are used to describe a chain of causality, or give a set of actions to reproduce or fix a problem as shown in Figure 2.5.

```
description:
Problem when switch from file to file.

build version:
2.3.0 v20080319-0800

steps to reproduce:
1. New two reports.
2. Insert a label in the first report.
3. Switch to the second report and switch to the script page.
4. Switch back to the first report.
5. Select the label and switch to script page.

Expected result:
after step4, no error.
after step5, the script page should show script method avaible for the label.
```

Figure 2.5: An example of steps to reproduce.

These steps are the procedures that any other user must follow to encounter the same bug. They should be as detailed as feasible, with pictures or test data to make it easier to read and comprehend.

2.2 Version Control System

A version control system (VCS) is a repository that allows the developer to track code changes, track who made the changes. Also make a copy of the project by forking it, edit it and then merging the changes with the original project. Technically, it signifies a set of commits where a commit is a saving point for code changes made by the developer. Commits provide a point in the project where a developer can go back and fix a bug or modify the code.

2.2.1 Commit History

A project's commit history is a log of commits that allows someone to see the changes that have occurred for every commit made in a specific branch. This allows teams to swiftly follow the evolution of design work over time, as well as revert to or past states of development.

Ideally for bug fixes, the developers mention the bug id in their commit message as a good practice. This practice helps the developers to figure out the location of changes made to fix a particular bug.

2.3 Infozilla Tool

There are two sorts of information extracted from bug reports. The first type is structural information, which includes patches, source code, and stack traces that have a specific structure depending on the programming language and can be retrieved or filtered using regular expression or string operations. The second category, includes unstructured textual content such as summaries and descriptions within bug reports. In contrast to structural

information, it is difficult to determine where the descriptions end and start again.

Unstructured data requires significant work to pre-process to be ready for natural language processing, primarily dependent on the quality of the information provided by users/developers. Structural information, such as stack traces, is generated automatically without human intervention and appears to include more verifiable information.

In this work, we are using the Infozilla tool [13] to extract structural data from bug reports. The Infozilla tool is an open-source project developed by A. Schroter, N. Bettenburg, and R. Premraj [13]. It is designed to extract structural and non-structural software engineering data from unstructured data sources like e-mails, discussions, bug reports, and wiki pages. This tool extracts structural information like stack traces, patches, and source code from the bug reports and produces an XML format output of this information. It can also extract non-structural data like descriptions and comments.

2.4 Machine Learning Algorithms

Machine learning algorithms use computational techniques to maximize a performance criterion based on sample data. These machine learning models have specific parameters and learn information from data to optimize the model's parameters. As more samples are provided for learning, the models learn more from the data and perform better predictions. There are two types of machine learning: supervised learning and unsupervised learning [10].

A *supervised learning* algorithm uses a set of known input and output data. The model learns a function to map from the input data to output and function is appropriately fitted to produce accurate predictions for the outcome of new data.

An *unsupervised learning* algorithm is employed to infer predictions from data sets having input data but no labelled responses. These algorithms identify hidden patterns or data clusters or have the capacity to find similarities and differences in information.

In this work, unsupervised learning is not an option because the algorithm cannot clas-

sify or categorise stack traces into multiple classes using only their features. We chose the labels for the stack traces based on the commit history. Because of this, we employ supervised learning methods and discuss about the following classifiers.

2.4.1 Naive Bayes classifier

A Naive Bayes classifier is based on the Naive Bayes algorithm [10]. In order to find the probability for a label, this algorithm uses the Bayes rule as shown in Equation 2.1.

$$P(label|features) = \frac{P(label) * P(features|label)}{P(features)} \tag{2.1}$$

Given the label, the algorithm then makes the 'naive' assumption that all features are independent. Rather than computing P(features) explicitly, the algorithm calculates the numerator for each label and normalizes them to the sum of one.

Table 2.1: Sample data set

Features			**Labels**
Filename	**Method name**	**Exception**	
classOne.java	*methodOne*	*exceptionOne*	***Accurate***
classTwo.java	*methodTwo*	*exceptionTwo*	***Accurate***
classThree.java	*methodThree*	*exceptionThree*	***Inaccurate***
classTwo.java	*methodTwo*	*exceptionOne*	***Inaccurate***
classOne.java	*methodOne*	*exceptionThree*	***Accurate***
classTwo.java	*methodTwo*	*exceptionOne*	***Accurate***

Consider the sample data set provided in Table 2.1, which has features and labels. Assume we trained a Naive Bayes classifier with this sample data. Let's see how the Naive Bayes classifier can predict the label for a new instance with the features *classTwo*, *methodTwo* and *exceptionThree*.

As shown in the calculations of Figure 2.6, the classifier finds the probability of the labels given the sample data and found probabilities for *P(Accurate)* and *P(Inaccurate)* as 4/6 and 2/6 respectively. Also, the probability of the features (*classTwo*, *methodTwo* and *exceptionThree*) given a specific label (i.e. Accurate or Inaccurate) are calculated and found

the probability that the instance has the Accurate label is **0.08** and Inaccurate label is **0.04**. Thus, the Naive Bayes classifier predicts the label for the new features as ***Accurate***.

$$P(Accurate) = 4/6$$
$$P(Inaccurate) = 2/6$$

$$P(Accurate|(classTwo,methodTwo,exceptionThree)) = P(Accurate) * P(classTwo|Accurate) * P(methodTwo|Accurate) * P(exceptionThree|Accurate)$$
$$\Rightarrow (4/6) * (2/4) * (2/4) * (1/4) = \mathbf{0.08}$$

$$P(Inaccurate|(classTwo,methodTwo,exceptionThree)) = P(Inaccurate) * P(classTwo|Inaccurate) * P(methodTwo|Inaccurate) * P(exceptionThree|Inaccurate)$$
$$\Rightarrow (2/6) * (1/2) * (1/2) * (1/2) = \mathbf{0.04}$$

Figure 2.6: Naive Bayes calculations.

2.4.2 Support Vector Machines

Support Vector Machine (SVM) is a supervised machine learning algorithm for data classifications. SVM creates a *hyper-plane*[6] to separate various classes in a given data set; however, the main principle is to find the optimum *maximum margin separator*[7] that best classifies the data [10].

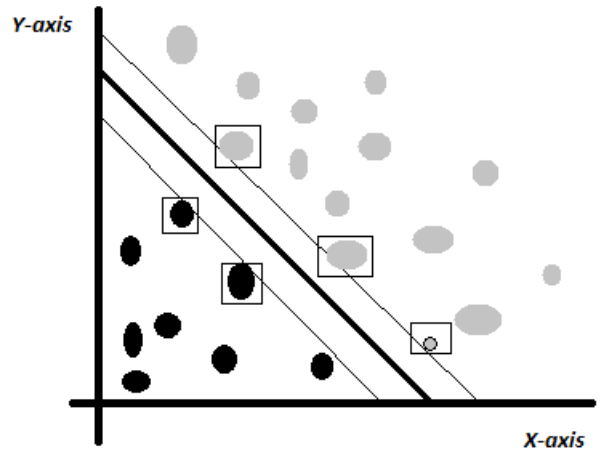

Figure 2.7: Support Vector Machine.

[6] A hyperplane is a plane with one less dimension than its dimensional space.
[7] A decision boundary that is farthest from the training point possible.

In the example shown in Figure 2.7, there are two classes of data (Black and Gray dots). To classify them, SVM finds linear separators (the slim lines) close to the training points and builds a maximum margin separator (the solid line) at the midpoint of the two linear separators. This line provides the greatest distance from the data points of the different classes to classify outliers with greater confidence in the future. The *Support Vectors* (data points inside the boxes) are the points that are close to the separator, and these points play an essential role in deciding the *maximum margin separator*.

Unlike above, in non-linear separation problems, SVM produces a linear separation hyperplane, but by applying the kernel trick, they can embed the data in a higher-dimensional space. Here, the kernel trick is SVM using the kernel functions that can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can find linear separators in the higher-dimensional feature space.

2.4.3 Logistic Regression

Logistic Regression is a classification algorithm used to predict the probability of a categorical variable [10]. Logistic Regression is most used when the data in question has binary output.

$$y = w_0 + w_1.x_1 + .. + w_n.x_n \tag{2.2}$$

In the above linear equation, Equation 2.2, y is the dependent variable and $x_1, x_2, \ldots, x_n$ are the independent variables. Independent variables are used to predict or model the dependent variables. As they can take on any value, they are considered independent variables. On the other hand, dependent variables are the variables we want to predict using the independent variables. Independent variables are commonly referred to as characteristics or qualities, whereas dependent variables are target variables or labels.

$$Logistic(y) = \frac{1}{1 + e^{-}y} \tag{2.3}$$

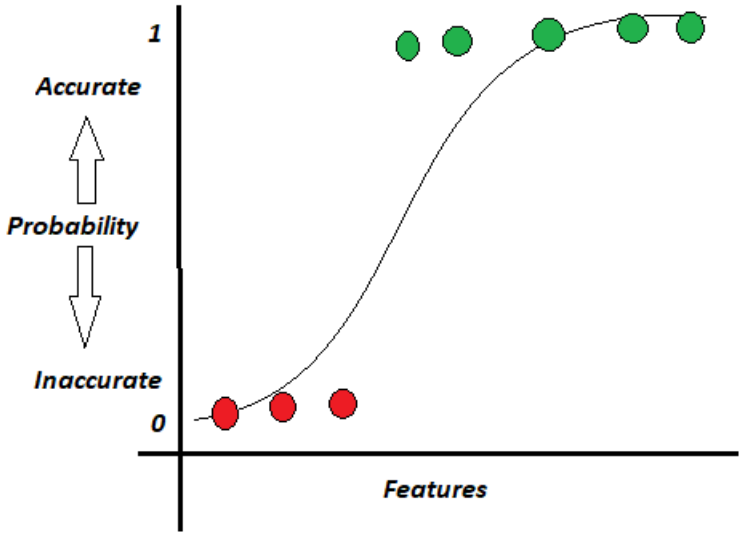

Figure 2.8: Logistic Regression.

Logistic Regression predicts the probability of an occurrence of a binary event utilizing a logit or sigmoid function as shown in Equation 2.3. The process of fitting the weights of this model to minimize a loss on a data set is achieved by the Maximum Likelihood Estimation (MLE) approach. Maximizing the likelihood function determines the parameters most likely to produce the observed data.

The logistic function, often known as the sigmoid function, generates an ”S”-shaped curve that may convert any real-valued integer to a number between 0 and 1, as shown in the Figure 2.8. As the curve advances toward positive infinity, the expected value of y becomes 1, and if it moves toward negative infinity, the predicted value of y becomes 0. If the sigmoid function output is more significant than 0.5, the outcome is 1; otherwise, the result is 0.

2.5 Metrics

One of the most important steps in developing an effective machine learning model is evaluating the performance of the machine learning model. Different metrics are used to evaluate the model’s performance or quality, and these metrics are known as performance

metrics or evaluation metrics. These performance metrics allow us to see how well our model performed with the given data. By adjusting the hyper-parameters [8], we can enhance the model's performance. Each machine learning model strives to generalize well on previously unseen/new data, and performance metrics aid in determining how well the model generalizes on the new data set. Metrics are frequently designed for a specific type of machine learning problem or model. Among the most important and widely used metrics are: Precision, Recall, F1-Score and Accuracy.

2.5.1 Precision

The number of positive class predictions that actually belong to the positive class is determined by precision as shown in Equation 2.4.

$$Precision = \frac{TruePositive(TP)}{TruePositive(TP) + FalsePositive(FP)} \tag{2.4}$$

2.5.2 Recall

The number of positive class predictions made out of all positive class in the data set is measured by recall as shown in Equation 2.5.

$$Recall = \frac{TruePositive(TP)}{TruePositive(TP) + FalseNegative(FN)} \tag{2.5}$$

2.5.3 Accuracy

The ratio of the overall number of right predictions to the total number of predictions for a model is known as accuracy as shown in Equation 2.6.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.6}$$

[8] A measurement whose value is used to control the learning process

2.5.4 F1-Score

The F1-score is defined as the harmonic mean of precision and recall as shown in Equation 2.7. When compared to the Accuracy Metric, it provides a more accurate measurement of cases that were incorrectly classified.

$$F1Score = \frac{(2 * Precision * Recall)}{Precision + Recall} \tag{2.7}$$

2.6 Summary

In this chapter, we described the contents of bug reports, the Infozilla tool used to extract structural information from bug reports, the machine learning techniques we employed in our investigation to create recommendations, and the metrics we used to rate the performance of our recommenders.

Chapter 3

Related Work

Several prior studies have examined bug reports and proposed approaches to improve their quality and use in recommendation systems. Many recommendation systems rely on bug report descriptions as the primary feature source in prediction. Little research has been focused on using the structural information of a bug report in prediction.

This chapter presents research on bug report structure and the use of bug report components for predicting bug report severity, performing bug report triage, and detecting duplicate bug reports. Finally, we cite research related to crash report management.

3.1 Research About Bug Report Structure

Bettenburg *et al.* [2] studied the quality of bug reports and showed a significant gap between what reporters provide and what developers require to fix a bug. Several parameters, such as the length of descriptions, formatting, and the existence of stack traces and attachments, were found to influence the quality of bug reports. Researchers polled 872 developers from the Apache, Eclipse, and Mozilla projects to find out what information matters most to them. The researchers invited the developers to fill out a survey about the most relevant details in bug reports and the issues they had with them. Also, the researchers asked the developers to rank the bug report quality from very poor to very good. According to their findings, the most valuable elements in bug reports are *steps to reproduce* and *stack traces*. The researchers also developed a prototype called *Cuezilla*, which measures the quality of new bug reports and recommends which element to be added to improve the

quality.

Schroter *et al.* [13] examined nearly 3,940 Eclipse bug reports containing stack traces. Their research results show that the mean lifetime for fixing bugs with a stack trace is 2.73 days, whereas for bugs without a stack trace, it is 26.44 days. Also, the researchers found that bug reports containing multiple stack traces might have a higher rate of fixing. The researchers produced these results by considering a small data set of the Eclipse project.

Another research by Shah *et al.* [6] looked at the issues surrounding exception handling from the human perspective. The researchers devised a study to assess different perspectives of software developers to understand better how developers perceive exception handling, the methods they use to deal with exception handling constructs, and the utility of a visualization tool researchers developed for exception handling. Based on the researchers' findings, they proposed a new role for the software development process: the exception engineer, who collaborates closely with software engineers throughout the process.

3.2 Research Using Bug Report Information For Prediction

As indicated by developers and recorded by researchers, the quality of the provided steps to reproduce in the bug reports is a crucial concern with user-written bug reports. In assessing the bug report quality, Chaparro *et al.* [4] focused on steps to reproduce. They proposed an Euler approach, which automatically identifying and assessing the quality of the steps to replicate in a bug report and offered comments to the reporters so that they could improve the problem report. External evaluators evaluated Chaparro's approach feedback. The results show that the approach accurately recognized 98% of the existing steps to reproduce and 58% of the missing ones, and 73% of bug report quality annotations.

Korosh *et al.* [7] builds on earlier research by looking at how categorical features, in addition to stack traces, might be used to predict the severity of issues. According to the researchers, developers usually prioritize defects that need to be fixed based on severity, and bug submitters frequently enter an incorrect severity level for various reasons, prolonging

the bug resolution process. As a result, they implemented a technique that can automatically forecast the bug severity. They experimented on bug reports submitted to Eclipse between 2001 and 2015 and Gnome between 1999 and 2015 to demonstrate that including categorical information in addition to stack traces improves the accuracy of the severity prediction approach from 5% to 20%.

In another research by Korosh *et al.* [12], they investigated using stack traces and categorical aspects (system version, severity, and platform) of bug reports to improve bug report accuracy over bug report descriptions. Their technique uses past bug reports to predict faulty components and products of newly submitted bug reports. They used TF-IDF [9] to weight historical bug report stack traces to feature vectors. These vectors are then sent into a classification algorithm with a subset of bug report category data. This technique also addresses the issue of imbalanced data. When forecasting faulty components, their approach has a 58% accuracy rate on the Eclipse data set and 70% on the Gnome data set.

3.3 Research Managing Crash Reports

Paila *et al.* [14] focused on crash reports that are reported numerous times each day, which causes the development team to put a lot of time into reviewing the crash reports. To solve this issue, the researchers have created an automated technique to evaluate a crash report and locate the incorrect module. The researchers built this approach using a cutting-edge algorithm that analyzes crash reports for exception-based patterns and maps reference assemblies. Several thousand crash reports from four different industrial automation applications have been subjected to this methodology. Results show that the algorithm achieved a high level of accuracy in identifying the incorrect module and subsystem responsible for a crash.

Bergel *et al.* [1] presented methods for automatically creating tests to reproduce stack traces as manually replicating crashes can be costly and time-consuming. According to the

[9] Term frequency–inverse document frequency, it is a natural language processing technique to represent a word in a corpus or collection of documents as numerical statistic.

study, search-based approaches are more difficult since the algorithms have less information to work with, and type checking in dynamic languages without explicit type declarations can only be accomplished during runtime. So, they proposed a genetic algorithm approach for reproducing crashes in Python using only the information given in the stack trace of the fault. An empirical assessment of three distinct trials yielded largely positive outcomes, with great precision and repeatability of the desired crashes.

In another research, Wang *et al.* [11] offered five guidelines for automatically identifying linked crash types. A crash correlation group is a collection of crash categories linked to similar or related bug reports. Using crash correlation groups, the researchers present an approach for locating and ranking problematic files. A mechanism to identify duplicate and related bug reports is also proposed. To discover and resolve associated crash types, developers can combine the suggested crash correlation rules with the new bug localization approach. Triagers can reduce their burden by automatically screening duplicate bug reports using the duplicate bug report identification approach.

3.4 Summary

In this chapter, we discussed prior research on bug report structure in which researchers investigated the significance of components in bug reports. We also talked about research on recommenders using bug report information, such as assessing steps to reproduce, issue severity, and predicting faulty components. Finally, we discussed crash report research, which involves evaluating a crash report to locate the incorrect module and ranking problematic files.

Chapter 4

An Approach to Detecting and Predicting Inaccurate Stack Traces

Our approach to identifying accurate/inaccurate stack traces uses supervised machine learning techniques to predict if a stack trace might help or mislead the developer during the fixing process. We make recommendations based on bug reports that developers have fixed, which contain stack traces, and their corresponding commit history. To create the accurate/inaccurate classifier, we download bug reports and commit history from a software project and extract stack traces from the bug reports. A machine learning algorithm is trained on features extracted from the stack traces in bug reports, and instance labels for a given stack trace are assigned depending on the commit history for a given bug report as shown in Figure 4.1.

Our approach consists of the following steps:

1. Gathering bug reports and commit history from a software project.
2. Mapping the bug reports with commit history.
3. Extracting stack traces from bug reports.
4. Feature extraction from the stack traces.
5. Extraction of file names from the commit history.
6. Assigning accurate/inaccurate labels to the extracted stack traces.

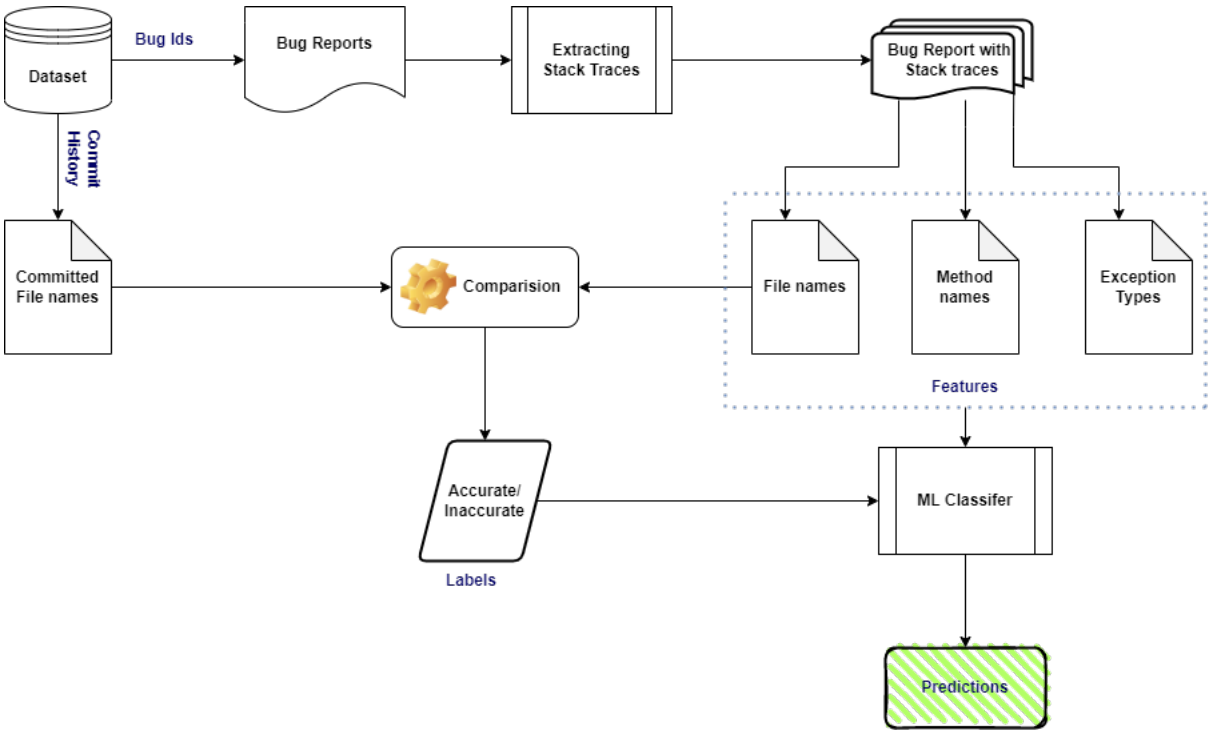

Figure 4.1: Design for training and detecting Accurate / Inaccurate Stack traces

7. Training a supervised machine learning algorithm to create a classifier for identifying inaccurate stack traces.

4.1 Data Source and Preparation

For our approach, we require bug reports containing stack traces that we can link to commits in the project's commit history. From the project commit history, we need the filenames of the committed files where the developer modified the code to solve the defect for a bug report.

Two possible situations exist in linking the bug reports with commits for any software project. First, if the project's version control system is integrated with the project's issue tracking system (e.g. GitHub), then adding the issue number or bug id in the commit message will automatically link the committed files with the bug report.

Second, consider the issue tracking system (e.g. Bugzilla) and version control system which are separate systems (e.g. CVS). In that case, the bug id establishes references to

three different systems, including the issue tracking system, version control system, and wiki [10]. This bug id can be used as an identifier while writing a wiki article, bug report, or VCS commit message and are then instantly transformed into hyperlinks pointing to the tools mentioned. This makes it possible to navigate easily between bug reports and code modifications.

We must distinguish the bug reports that include stack traces from the all the downloaded bug reports. Since stack traces have a start line and a trace line, regular expressions can be used to find and separate them. In this work, we used the Infozilla tool to filter stack traces from bug reports and extracts the reports into an XML file.

4.2 Extracting Features and Committed File Names Extraction

4.2.1 Extracting features from stack traces

As shown in Figure 2.2, a stack trace contains an exception at the beginning, method names, and file names within the frames of the call stack. These stack trace attributes may be helpful for a machine learning algorithm to understand their hidden patterns in predicting the stack traces as accurate or inaccurate. As a result, we decided to extract these attributes from stack traces and pass them as features to train a machine learning algorithms. We can use feature-specific regular expressions or string operations to extract these features.

```
1 tree = ET.parse(#path_to_bug_report_with_stack_trace)
2 root = tree.getroot()
3 for child in root:
4  exception_with_path = child.iter('Exception').text
5  exception=exception_with_path
6            [exception_with_path.rindex('.')+1:]
```

Listing 4.1: Exception extraction from a Stack trace

[10] A kind of website that allows users to update its content directly from the browser and maintains a version history for each editable page, https://en.wikipedia.org/wiki/Wiki.

```
<?xml version="1.0" encoding="UTF-8"?>
<infozilla-output>
  <Patches amount="0" />
  <Stacktraces amount="1">
    <Stacktrace timestamp="1631578486416">
      <Exception>java.lang.NullPointerException</Exception>
      <Reason />
      <Frames>
        <Frame depth="0">org.apache.cassandra.config.DatabaseDescriptor.getTypeInfo(DatabaseDescriptor.java:729)</Frame>
        <Frame depth="1">org.apache.cassandra.db.ColumnIndexer.serialize(ColumnIndexer.java:62)</Frame>
        <Frame depth="2">org.apache.cassandra.db.CompactSerializerInvocationHandler.invoke(CompactSerializerInvocationHandler.java:50)</Frame>
        <Frame depth="3">$Proxy0.serialize(Unknown Source)</Frame>
        <Frame depth="4">org.apache.cassandra.db.Memtable.flushForRandomPartitioner(Memtable.java:461)</Frame>
        <Frame depth="5">org.apache.cassandra.db.Memtable.flush(Memtable.java:440)</Frame>
        <Frame depth="6">org.apache.cassandra.db.Memtable.forceflush(Memtable.java:279)</Frame>
        <Frame depth="7">org.apache.cassandra.db.ColumnFamilyStore.forceFlush(ColumnFamilyStore.java:409)</Frame>
        <Frame depth="8">org.apache.cassandra.db.Table.flush(Table.java:857)</Frame>
        <Frame depth="9">org.apache.cassandra.db.CommitLog.doRecovery(CommitLog.java:408)</Frame>
        <Frame depth="10">org.apache.cassandra.db.CommitLog.recover(CommitLog.java:317)</Frame>
        <Frame depth="11">org.apache.cassandra.db.RecoveryManager.recoverEachTable(RecoveryManager.java:90)</Frame>
        <Frame depth="12">org.apache.cassandra.db.RecoveryManager.doRecovery(RecoveryManager.java:77)</Frame>
        <Frame depth="13">org.apache.cassandra.db.DBManager.&amp;lt;init&amp;gt;(DBManager.java:112)</Frame>
        <Frame depth="14">org.apache.cassandra.db.DBManager.instance(DBManager.java:61)</Frame>
        <Frame depth="15">org.apache.cassandra.service.StorageService.start(StorageService.java:465)</Frame>
        <Frame depth="16">org.apache.cassandra.service.CassandraServer.start(CassandraServer.java:96)</Frame>
        <Frame depth="17">org.apache.cassandra.service.CassandraServer.main(CassandraServer.java:1049)</Frame>
      </Frames>
    </Stacktrace>
  </Stacktraces>
</infozilla-output>
```

Figure 4.2: An example of extracted stack trace using Infozilla tool

In our work, the extracted stack traces from Infozilla are in XML format, as shown in Figure 4.2. We use *xml.etree.ElementTree*[11] library for parsing the stack trace.

This code *child.iter('Exception').text* at line #4 in 4.1, gives the whole exception along with the path as we must pass the exception's start and end indexes to parse the exception name from *exception_with_path*. On the *exception_with_path* string, we use the *rindex* string operation to find the index of the first period (.) from the right side and increment it by one. That incremented index is the start of the exception name, and the end is the last index, i.e. '-1' as after the colon, it will be '-1' by default. Thus we get the exception name.

```
1 for subchild in child.iter('Frame'):
2     frame = subchild.text
3     file_name = frame[frame.find('(')+1:frame.find(':')]
4     method_with_path = frame[:frame.find('(')]
5     method_name = method_with_path
6                   [method_with_path.rindex('.')+1:]
```

Listing 4.2: File name and Method name extraction from a Stack trace

In this code 4.2, *child.iter('Frame')* at line #1, returns a list of frames found in stack traces. To obtain the file name, the expression *frame[frame.find('(')+1:frame.find(':')]* is

[11] A Python library for parsing and creating XML data, https://docs.python.org/3/library/xml.etree.elementtree.html

used, which includes the start and end indexes. Where the start index is determined by finding the first open parenthesis in the frame and incrementing it by one, and the end index is determined by the presence of a colon in the frame.

To obtain the method name, we first used the *find* string operation (*frame[:frame.find('(')]*) at line #3, to parse the frame up until the open parenthesis, which provides the method name along with the path.

Next, we used another expression *method_with_path [method_with_path.rindex('.')+1:]* containing the *rindex* method to get the method name, just like we did with filenames previously.

4.2.2 Extracting filenames from the commit history

To extract the committed file names from the commit history, we use the regular expression shown in Equation 4.1 which captures one word that ends with ".*java*".

$$regex = \texttt{(\textbackslash w+\textbackslash .java)\{1\}} \tag{4.1}$$

Consider the example in Figure 4.3, which displays the commit history for the bug report #10909 from the Cassandra project. In this example, the name *ActiveReportService* would be extracted.

```
* 98cc2c8d6cc27f1a2e675030a13b46fd336812f8 3
Avoid NPE on incremental repair failure CHANGES.txt src/java/org/apache/cassandra/service/ActiveRepairService.java
```

Figure 4.3: Commit History for Cassandra Bug # 10909

4.3 Labelling Stack Traces

To label a stack trace as inaccurate or accurate, we consider the committed file names associated with a bug report that contains a stack trace. To determine if a stack trace is accurate, we check for a match between the files using the the Top-N stack frames and the

files committed to fix the problem. If any file names match the committed file names, we label that stack trace as 'Accurate'; otherwise, it is labelled as 'Inaccurate'.

4.3.1 Walk-through Example

For example, consider bug report #100227 containing a stack trace from the AspectJ project as shown in Figure 4.4 and related commit history is shown in Figure 4.5.

Bug 100227 - [generics][itds] inner class with generic enclosing class

```
/home/user/sgelin3/dev/java/ajc/new_bug/Bug.java [error] Internal compiler error
java.lang.NullPointerException
        at
org.aspectj.ajdt.internal.compiler.lookup.EclipseFactory.fromBinding(EclipseFactory.java:202)
        at
org.aspectj.ajdt.internal.compiler.ast.InterTypeFieldDeclaration.build(InterTypeFieldDeclaration.java:173)
        at
org.aspectj.ajdt.internal.compiler.ast.AspectDeclaration.buildInterTypeAndPerClause(AspectDeclaration.java:1020)
        at
org.aspectj.ajdt.internal.compiler.lookup.AjLookupEnvironment.buildInterTypeAndPerClause(AjLookupEnvironment.java:306)
        at
org.aspectj.ajdt.internal.compiler.lookup.AjLookupEnvironment.completeTypeBindings(AjLookupEnvironment.java:122)
        at
org.aspectj.org.eclipse.jdt.internal.compiler.Compiler.beginToCompile(Compiler.java:302)
        at
org.aspectj.org.eclipse.jdt.internal.compiler.Compiler.compile(Compiler.java:316)
...
```

Figure 4.4: Stack trace from AspectJ Bug # 100227

```
1:org.aspectj.ajdt.core/src/org/aspectj/ajdt/internal/compiler/lookup/EclipseFactory.java
9:tests/src/org/aspectj/systemtest/ajc150/Ajc150Tests.java 13:weaver/src/org/aspectj/weaver/TypeX.java
```

Figure 4.5: Commit History for the AspectJ Bug # 100227

The stack trace notifies that the error is due to a *NullPointerException* and the fault might be traced back as indicated in the stack trace's file names. From the commit history, it is evident that developers made changes to the two files *EclipseFactory.java* and *TypeX.java* to fix the bug. If the committed file names are compared to those appearing in the stack trace, we notice that the file name in the stack trace's first frame, i.e. *EclipseFactory.java* matches with the committed file name. Thus, this stack trace is considered 'Accurate.'

4.4 Training a Machine Learning Classifier

For each project, we obtained the filenames, method names and exceptions from bug reports containing stack traces, using string operations as specified in Section 4.2.1. We assigned labels to each stack trace after comparing the names of the files in the committed files and those in the stack frames. These features and their assigned labels for each stack trace are written to a file in CSV format.

We use the file name as it is crucial in determining if the stack trace is accurate or inaccurate. Along with file names, we also utilize method names and exception types because they may be helpful. To make sure that the file names are unique, we run a script to remove any duplicates using the file path. Likewise, method names can be the same across the classes. Therefore, we use a script to filter the unique method names based on file names and paths.

Having processed the data, we used the pandas[12] library to read and filter the required data from the CSV files, and we utilized the Scikit-learn[13] library to build the machine learning models.

In training the machine learning models, we experimented with different combinations of features (i.e. Filename only, Filename and Method name, Filename and Exception Type, Filename, Method name and Exception Type). We used pandas to extract the relevant feature columns from the CSV file as required and mapped the accurate and inaccurate labels to ”1” and ”0,” respectively, before passing them to the machine learning algorithm. When training the classifier, we randomized the instances in the data set and used 70% of it as training data and 30% as testing data.

We experimented with three machine learning models (Naive Bayes, Support Vector Machines and Logistic Regression) to make predictions on the testing set. We used methods from Scikit-learn package to get the metrics used to evaluate the predictions.

[12] A python library for data manipulation and analysis.

[13] An open source machine learning package that supports both supervised and unsupervised learning

Chapter 5

Evaluation and Results

We evaluated 12,865 and 19,247 bug reports downloaded from the Eclipse and Apache projects. We extracted 1,833 and 2,634 bug reports containing stack traces from the Eclipse and Apache projects. We used the committed file names from the commit history and stack traces file names to determine if the bug report containing the stack trace is accurate or not, and we discovered that 39% of stack traces in Eclipse projects and 14% in Apache projects are inaccurate for the Top-10 stack depth i.e. looking at the top-10 frames of the call stack. We experimented with three machine learning algorithms described in Section 2.4 to construct an accurate/inaccurate stack trace recommender: Naive Bayes, Logistic Regression, and Support Vector Machines. We found that Logistic regression is the best machine learning algorithm. Furthermore, we investigated different feature combinations to train the machine learning algorithms and found *Filename-Exception Type* is the best feature combination.

This chapter presents the results of applying our approach to eight open-source projects. First, we present the details of the data sets used in our investigation. Next, we provide answers to each of our research questions. Recall that our research questions are:

RQ 1: How often do inaccurate stack traces occur in bug reports?

RQ 2: What exceptions are more likely to occur in inaccurate stack traces?

RQ 3: What is the effectiveness of using different machine learning algorithms in tagging stack traces as accurate or inaccurate?

5.1 Data Sets

We used data from open source projects that use the Bugzilla and JIRA issue tracking systems to develop the approach as the data is easily accessible. Specifically, we chose data from the Eclipse and Apache software communities. These software communities were chosen because they host a variety of projects from different application sectors, and previous research has used data from these communities.

5.1.1 Eclipse Data Set

We used data from Xin Ye *et al.* research [15] for the Eclipse data sets. This data set contains bug report information (i.e., bug id, summary, report time) for the following open-source Java projects: AspectJ[14], Birt[15], Eclipse Platform UI [16] and the JDT[17]. It also contains the commit history details for all bug reports, which is crucial for our approach. This is one of the reasons we chose this data set, as the collected commit histories saves us time from not having to collect the commit history for these projects ourselves.

However, Xin's data set does not contain the bug report descriptions. Therefore, we downloaded this information using the Bugzilla API. As Table 5.1 shows, the data set contains a total of 12,865 bug reports, with JDT having the most (4,893) and AspectJ having the least (543).

[14]The AspectJ is an extension of the Java programming language to support aspect-oriented programming. This project provides Eclipse IDE integration.

[15]The Business Intelligence Reporting Tool (Birt) supports the creation of data visualizations, dashboards and reports.

[16]The Eclipse Platform UI provides the fundamental building blocks for Eclipse-built user interfaces.

[17]JDT offers the tool plug-ins necessary to construct a Java IDE to support any Java application's development.

Table 5.1: Eclipse Data Set.

Projects	Bug Reports	Bug Reports with Stack Traces	File Names	Method Names	Exception Types
AspectJ	543	158	481	1049	27
Birt	3530	458	1773	3309	60
Eclipse UI	3899	532	1983	3954	49
JDT	4893	685	2373	4794	42
Total	12865	1833	6610	13106	178

5.1.2 Apache Data Set

We downloaded the Apache data from the following four projects: Cassandra[18], Hadoop[19], Hbase[20] and Spring [21]. We choose these projects because of their vast bug reports size. We downloaded the bug report details (i.e. bug id, comments, description, report time). As shown in Table 5.2, all four Apache projects' bug reports totalled 19,247, with Hbase having the most (8,152) and Hadoop having the fewest (2,395).

To collect the commit history of the projects, we cloned the source code repository for each project and used a `git` command[22] to search the commit logs for bug ids and collect the corresponding file names.

Table 5.2: Apache Data Set.

Projects	Bug Reports	Bug Reports with Stack Traces	File Names	Method Names	Exception Types
Cassandra	4748	903	1702	3272	104
Hadoop	2395	36	202	311	18
Hbase	8152	1264	2483	5345	161
Spring	3952	431	1931	3076	96
Total	19247	2634	6318	12004	379

[18]The Cassandra is a NoSQL database management system designed to handle massive amounts of data.

[19]The Hadoop is a framework that enables the distributed processing of massive data volumes across computer clusters.

[20]The HBase is an open-source, NoSQL, distributed database for use with Hadoop.

[21]The Spring framework is an inversion of control container for Java applications.

[22]git log –grep [regex]

5.1.3 Extracting Stack Traces

The Infozilla tool extracts structural information like stack traces from the bug reports. We wrote a Python script to process all of the bug report information from the Eclipse and Apache data sets and run the data through Infozilla.

The Infozilla tool identifies the bug reports containing stack traces and extracts the reports into an XML file. As shown in the Tables 5.1 and 5.2, we found 1,833 and 2,634 bug reports containing stack traces from the Eclipse and Apache data sets.

These extracted stack traces contain an ordered list of class names, methods and types of exceptions. We filtered these components from the stack traces using a Python script containing regular expressions.

5.2 RQ 1 : How often do inaccurate stack traces occur in bug reports?

To answer RQ1, we compared the file names in the commit history for a given bug report with the filenames in the stack traces. We consider a stack trace accurate if a committed file name matches one of the stack trace file names; otherwise, the stack trace is considered inaccurate as shown in the Tables 5.3 and 5.4.

Usually, a stack trace contains many frames[23], which can be numerous, depending on the type of crash or error. According to Schröter *et al.* [13], 40% of bugs were fixed in the files found in the first frame, and 80% of bugs were fixed in the files found in the first six stack frames. For files found in the top ten stack frames, about 90% of issues were fixed. In our work, we investigated varying frame depths: Top-1, Top-3, Top-5 and Top-10 stack depth.

For the AspectJ project, we found that if only the filename in the first stack frame is considered, then 46% of the stack traces can be considered accurate and 54% are inaccurate. However, when examining the top 3, 5 and 10 stack frames, we find 65%, 68%, and 70% can be considered accurate, respectively.

[23] A stack frame represents a single function call containing file path, class name and a method name.

In the Birt project, we discovered that 76% and 60% of the stack traces could be considered inaccurate when considering the Top 1 and 3 frames, respectively. In comparison, when we look at the Top 5 and 10 stack frames, we discover that 46% and 48% are accurate, and 54% and 52% are inaccurate.

When we looked at the first stack frame for the Eclipse Platform UI project, we found that 72% were incorrect, and only 28% were accurate. Stack traces were determined to be almost evenly split between accurate and inaccurate for the Top-3 stack depth. We observed that 58% and 63% were accurate in the case of Top-5 and Top-10 stack depth, respectively.

We found 74% of stack traces for the JDT to be inaccurate and 26% to be accurate for the first frame in stack depth. However, we noticed that 62% and 66% were accurate for Top-5 and Top-10 Stack depths, respectively.

Table 5.3: Accurate/Inaccurate stack traces of all the bug reports for the Eclipse projects.

Projects	**Bug Reports with Stack Traces**	**Top-1**		**Top-3**		**Top 5**		**Top 10**	
		Accurate	**Inaccurate**	**Accurate**	**Inaccurate**	**Accurate**	**Inaccurate**	**Accurate**	**Inaccurate**
AspectJ	158	72	86	104	54	107	51	111	47
Birt	458	108	350	183	275	210	248	220	238
Eclipse Platform UI	532	147	385	258	274	307	225	336	196
JDT	685	177	508	359	326	423	262	453	232
Total	1833	504	1329	904	929	1047	786	1120	713

Table 5.4: Accurate/Inaccurate stack traces of all the bug reports for the Apache projects.

Projects	**Bug Reports with Stack Traces**	**Top-1**		**Top-3**		**Top 5**		**Top 10**	
		Accurate	**Inaccurate**	**Accurate**	**Inaccurate**	**Accurate**	**Inaccurate**	**Accurate**	**Inaccurate**
Cassandra	903	301	602	393	510	638	265	748	155
Hadoop	36	13	23	16	20	25	11	27	9
Hbase	1264	379	885	621	643	987	277	1169	95
Spring	431	115	316	192	239	293	138	320	111
Total	2634	808	1826	1222	1412	1943	691	2264	370

On considering the Top-1 and Top-3 stack frames for the Cassandra project, 67% and 56% are inaccurate, respectively. In comparison, we discovered that 29% and 17% of stack traces are inaccurate for the Top-5 and Top-10 stack depths, respectively.

For Hadoop, when considering the first stack frame, we discovered that 36% of stack traces are inaccurate and 64% are accurate. For the Top-3 frame, accurate and inaccurate are 44% and 55%, respectively. In contrast, when considering the Top-5 and Top-10 stack

depths, we discovered that 69% and 75% of stack traces are accurate, and 31% and 25% of stack traces are inaccurate.

When considering the first frame in a stack trace for the Hbase project, we identify 30% as accurate and 70% as inaccurate. For the Top-3 frame, accurate and inaccurate are 49% and 51%, respectively. When we examined the Top-5 and Top-10 stack frames, we noticed that 78% and 92% of stack traces are accurate, respectively, whereas 22% and 8% are inaccurate.

In the instance of the Spring project, we discovered that 73% and 55% of stack traces are incorrect when evaluating the first and third stack frames, respectively. However, we determined that 32% and 26% of the Top-5 and Top-10 stack frames are inaccurate.

In both Eclipse and Apache data sets, we discovered that inaccurate stack traces occur more often than accurate stack traces for Top 1 and 3 Stack depth. Top 5 and 10 offer nearly identical results and more accurate stack traces. Overall, 14% of stack traces in the Apache data set are inaccurate when evaluating Top-10 stack depth, whereas 39% of stack traces in the Eclipse data set are inaccurate, as shown in the Tables 5.3 and 5.4.

5.3 RQ 2 : What exceptions are more likely to occur in inaccurate stack traces?

We gathered exception types found in inaccurate stack traces for all four examined stack depths to answer this question from the collected bug reports. We calculated the percentage of occurrence in inaccurate stack traces for each exception type and ranked the exceptions based on their mean percentage values.

Table 5.5 shows the ranked order of exceptions for the Eclipse data set for the top five most frequently occurring exceptions. In inaccurate stack traces, the *NullPointerException* occurred a mean of 35% of time across all stack depths. The next exception that occurred, *IllegalArgumentException*, had an average of 9.6%, which issignificantly lower than *NullPointerException*. *SWTException* and *AssertionFailedError* are next, with an average of

5.8% and 5.6%, respectively. Lastly, *ClassCastException* occurred an average of 4.4% of the time in for all stack depths of inaccurate stack traces.

Table 5.5: The percentage of the exception types occurring in inaccurate stack traces using data from all of the projects in the Eclipse data set.

Rank	Exception Type	Stack depth				Mean
		Top-1	Top-3	Top-5	Top-10	
1	NullPointerException	33	33.7	36.2	36.9	35
2	IllegalArgumentException	10.2	11	8.8	8.3	9.6
3	SWTException	6.4	7.7	5.1	4.1	5.8
4	AssertionFailedError	7.1	5.4	4.6	5.5	5.6
5	ClassCastException	3.8	4.3	4.7	5	4.4

Table 5.6: The percentage of the exception types occurring in inaccurate stack traces using data from all of the projects in the Apache data set.

Rank	Exception Type	Stack depth				Mean
		Top-1	Top-3	Top-5	Top-10	
1	NullPointerException	13.5	11.2	13.2	24.6	15.6
2	AssertionError	11	7.6	13.6	25.4	14.4
3	IOException	8.8	8.1	12.7	23.8	13.4
4	IllegalArgumentException	8.1	7.6	10.4	19.4	11.4
5	IllegalStateException	3.7	3.2	4.9	9.2	5.3

The ranking order of the exceptions in the Apache data set is shown in Table 5.6. Inaccurate stack traces typically showed 15.6% occurrences of the *NullPointerException*, followed by *AssertionError* with 14.4% average. *IOException*and *IllegalArgumentException* have an average of 13.4% and 11.4%, respectively. Finally, *IllegalStateException* appeared 5.3% of times in inaccurate stack traces, which is significantly lower than the previous exception.

5.4 RQ 3 : What is the effectiveness of using different machine learning algorithms in tagging stack traces as accurate or inaccurate?

We use data from the Top-N stack frames (N=1, 3, 5, or 10) to train a classifier that identifies stack traces as accurate or inaccurate. Specifically, we use the file names, method

names and exception types as features. We experimented with different combinations of these features (i.e. Filename only, Filename and Method name, Filename and Exception Type, Filename, Method name and Exception Type) as well as different machine learning algorithms (i.e. Naive Bayes, Logistic Regression, and Support Vector Machine). We trained the machine learning algorithms by dividing the data into 70% training and 30% testing sets.

5.4.1 AspectJ Project

We extracted 158 bug reports from the AspectJ project with stack traces containing 481 filenames, 1,049 method names and 27 exception types, as shown in Table 5.1. These features are used to train all three of the classifier.

As shown in Table 5.7, 92% is recorded as the highest precision using the Naive Bayes algorithm with the *Filename-Exception type* as features for the Top-10 stack frames and 50% being the lowest precision using the Naive Bayes algorithm with the *Filename* feature for the Top-1 stack frame. The lowest recall (23%) occurred when training the Naive Bayes algorithm with the Top-3, Top-5 stack frames and the *Filename* feature set.

As for accuracy, the best result (78%) was found using the Top-3 and Top-10 stack depth and *Filename-Exception Type*,*Filename-Method Name-Exception Type* as feature sets with the Logistic Regression algorithm. The lowest result (43%) was found using the Top-5 and Top-10 and *Filename* feature set with the Naive Bayes algorithm.

Table 5.7: Evaluation metrics for the AspectJ Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	0.5	0.96	0.66	0.54	0.52	0.93	0.67	0.57	0.59	0.97	0.73	0.67	0.59	0.94	0.72	0.67
	Logistic Regression	0.58	0.45	0.51	0.59	0.57	0.46	0.51	0.59	0.7	0.8	0.75	0.75	0.71	0.76	0.73	0.74
	SVM	0.54	0.56	0.56	0.58	0.52	0.54	0.53	0.56	0.67	0.76	0.71	0.71	0.64	0.73	0.68	0.68
Top 3	Naive Bayes	0.82	0.23	0.36	0.46	0.78	0.28	0.42	0.47	0.91	0.44	0.6	0.6	0.87	0.44	0.58	0.58
	Logistic Regression	0.7	0.93	0.8	0.69	0.71	0.92	0.8	0.7	0.77	0.94	0.84	0.77	0.79	0.93	0.85	**0.78**
	SVM	0.7	0.93	0.8	0.69	0.69	0.94	0.79	0.68	0.77	0.94	0.84	0.77	0.79	0.94	0.81	0.71
Top 5	Naive Bayes	0.84	0.23	0.36	0.43	0.82	0.31	0.45	0.47	0.91	0.41	0.57	0.56	0.89	0.47	0.61	0.58
	Logistic Regression	0.72	0.97	0.83	0.72	0.73	0.95	0.83	0.75	0.77	0.95	0.85	0.77	0.79	0.94	0.85	0.77
	SVM	0.72	0.94	0.81	0.71	0.73	0.95	0.83	0.72	0.71	0.96	0.82	0.7	0.78	0.93	0.85	0.76
Top 10	Naive Bayes	0.88	0.25	0.38	0.43	0.87	0.34	0.49	0.48	**0.92**	0.42	0.58	0.56	0.9	0.49	0.63	0.59
	Logistic Regression	0.75	0.97	0.85	0.74	0.75	0.96	0.84	0.73	0.79	0.95	0.86	0.78	0.79	0.95	**0.87**	**0.78**
	SVM	0.76	0.95	0.85	0.75	0.74	0.96	0.84	0.72	0.8	0.94	0.86	0.78	0.75	0.97	0.84	0.74

When considering the different machine learning algorithms, as shown in Figure 5.1, 5.2, 5.3 and 5.4, in all the stack depths and feature combinations, Logistic Regression and

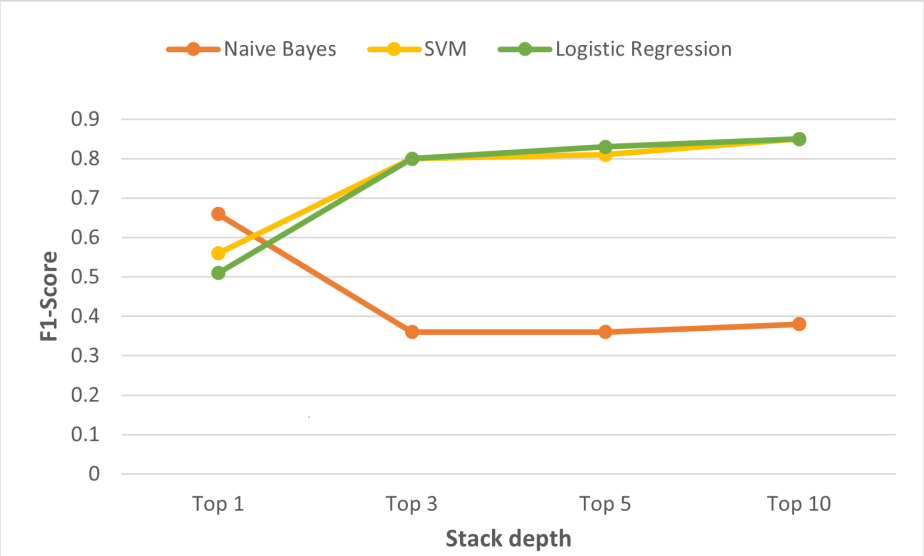

Figure 5.1: F1-score Measure for Top-N stack depth for AspectJ Project with File name as feature.

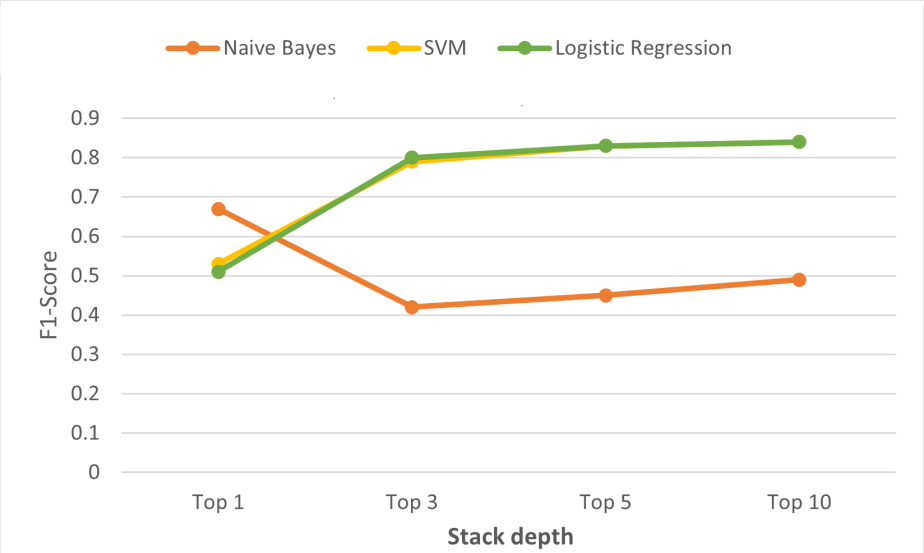

Figure 5.2: F1-score Measure for Top-N stack depth for AspectJ Project with File name, Method name as features.

SVM performed similarly. As the stack depth increases, the classifier performance improves. The F1-score for the Logistic Regression with *Filename* feature increased from 51% to 80% as the stack depth changed from Top-1 to Top-3. In contrast, Naive Bayes decreased from 66% to 36% as we moved from Top-1 to Top-3 stack depths when using *Filename* feature set as shown in Figure 5.1. The best F1-score (87%) was found using the

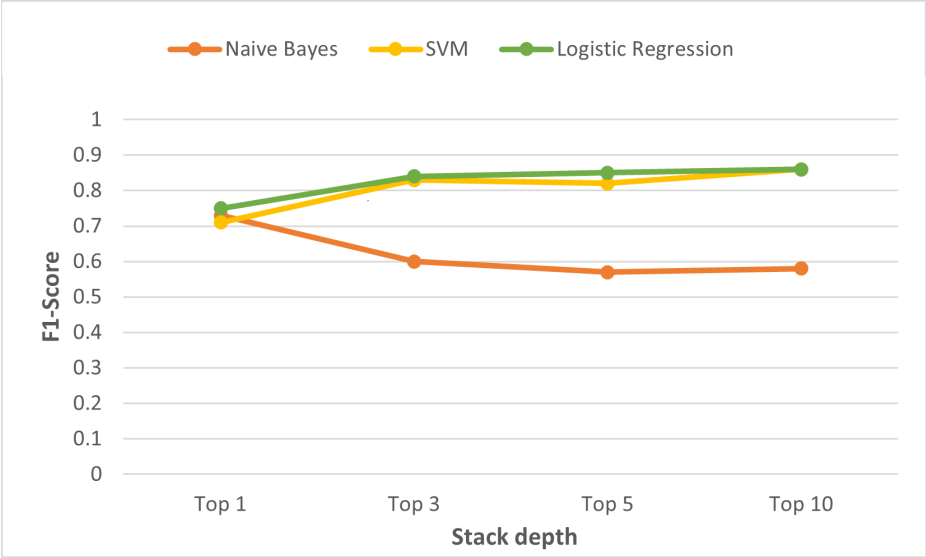

Figure 5.3: F1-score Measure for Top-N stack depth for AspectJ Project with File name, Exception as features.

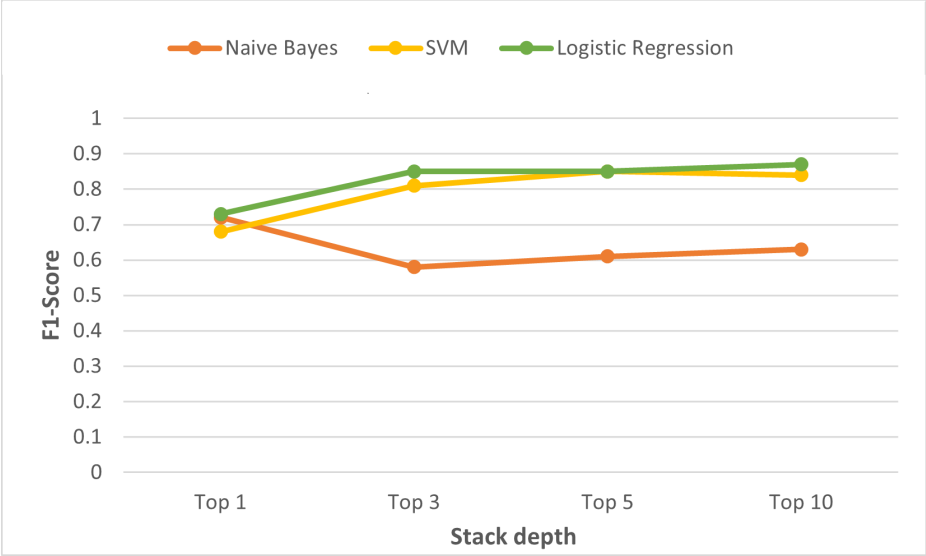

Figure 5.4: F1-score Measure for Top-N stack depth for AspectJ Project with File name, Method name , Exception as features.

Logistic Regression algorithm with the *Filename-Method Name-Exception type* feature set and the Top-10 stack frames. The worst F1-score (36%) was found using the Naive Bayes algorithm with the *Filename* feature set and for the Top-3, Top-5 stack frames.

5.4.2 Birt Project

In the case of the Birt project, we extracted 458 bug reports with stack traces that contained 1,773 filenames, 3,309 method names, and 60 exception kinds.

The Top-1 stack frame has the highest accuracy (82%) using Logistic Regression as an algorithm with *Filename-Exception Type* as feature set as shown in the Table 5.8. The lowest precision of 25% is the recommender using Naive Bayes as an algorithm with the *Filename* feature in the Top-1 stack frame.The best precision (75%) was found using the Logistic Regression and SVM as algorithms with the *Filename-Exception Type* feature set and the Top-1 stack frames. The worst precision (25%) was found using the Naive Bayes algorithm with the *Filename* feature set and the Top-1 stack frames.

The maximum recall is 97% for the Top-3 stack frame with *Filename*, *Filename-Exception Type* type as features and trained on the Naive Bayes algorithm. When using *Filename-Method Name-Exception Type* as features and the SVM as the machine learning algorithm, the lowest recall, 12%, was recorded in the Top-1 stack frame.

Table 5.8: Evaluation metrics for the Birt Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	0.25	0.94	0.39	0.37	0.26	0.89	0.4	0.42	0.35	0.94	0.51	0.61	0.36	0.9	0.51	0.63
	Logistic Regression	0.76	0.03	0.06	0.79	0.49	0.05	0.09	0.78	**0.75**	0.23	0.35	**0.82**	0.69	0.26	0.38	0.81
	SVM	0.6	0.08	0.15	0.79	0.47	0.07	0.11	0.78	**0.75**	0.14	0.24	0.8	0.64	0.12	0.2	0.8
Top 3	Naive Bayes	0.4	**0.97**	0.56	0.45	0.41	0.94	0.57	0.49	0.45	**0.97**	0.62	0.57	0.47	0.95	0.63	0.59
	Logistic Regression	0.53	0.2	.3	0.64	0.5	0.31	0.38	0.64	0.65	0.71	0.68	0.76	0.65	0.7	0.68	0.76
	SVM	0.55	0.23	0.32	0.65	0.52	0.19	0.28	0.64	0.61	0.73	0.67	0.74	0.62	0.59	0.61	0.72
Top 5	Naive Bayes	0.48	0.95	0.64	0.54	0.49	0.95	0.65	0.56	0.54	0.95	0.69	0.63	0.55	0.95	0.69	0.64
	Logistic Regression	0.6	0.61	0.61	0.66	0.6	0.63	0.62	0.67	0.69	0.76	0.72	0.75	0.69	0.77	0.72	0.75
	SVM	0.6	0.61	0.61	0.66	0.59	0.59	0.59	0.65	0.66	0.79	0.72	0.73	0.66	0.73	0.69	0.72
Top 10	Naive Bayes	0.52	0.93	0.67	0.58	0.55	0.9	0.68	0.62	0.57	0.93	0.71	0.66	0.6	0.91	0.72	0.68
	Logistic Regression	0.63	0.69	0.66	0.68	0.63	0.69	0.66	0.68	0.71	0.79	**0.75**	0.76	0.71	0.8	**0.75**	0.76
	SVM	0.61	0.72	0.66	0.66	0.61	0.7	0.65	0.66	0.68	0.83	0.74	0.74	0.67	0.77	0.72	0.72

All three machine learning algorithms performed similarly for all feature combinations and stack depth with the Birt project. As the stack depth increased, the F1-score increased, as indicated in the Figures 5.5, 5.6, 5.7 and 5.8. F1-score ranged between 65% and 68% for the Top-10 stack depth with either *Filename* or *Filename-Method Name* as features, and 71% to 75% for the other two feature combinations. The F1-score for Logistic Regression with the Top 1 stack frame using the *Filename* as features started extremely low at 7% and climbed to 66% as the stack depth increased as shown in Figure 5.5. The highest F1-score

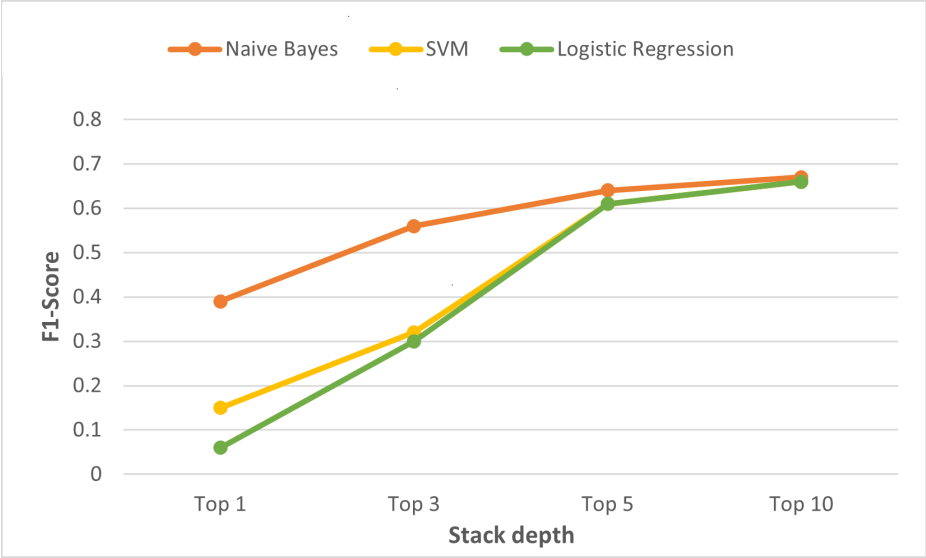

Figure 5.5: F1-score Measure for Top-N stack depth for Birt Project with Filename as feature.

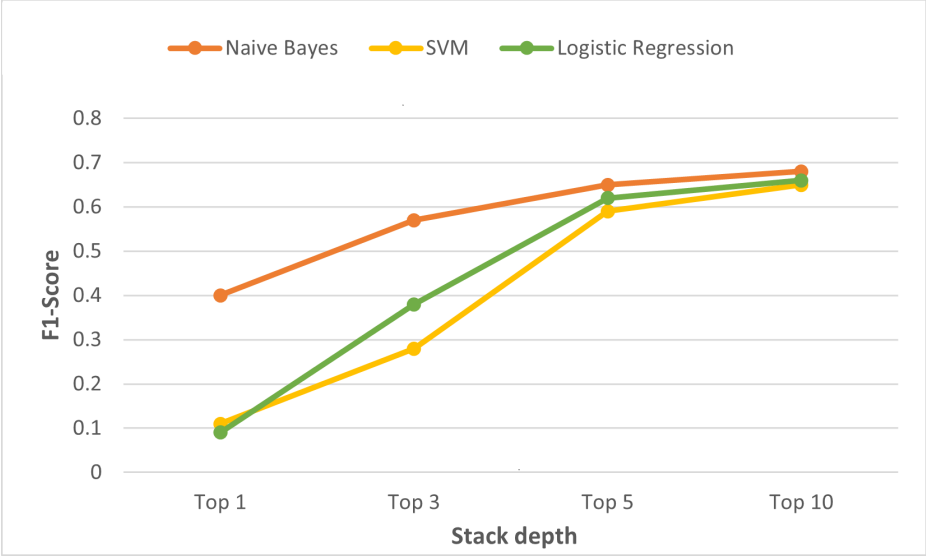

Figure 5.6: F1-score Measure for Top-N stack depth for Birt Project with File name, Method name as features.

of 75% is reported for the Logistic Regression for Top-10 stack frame with both *Filename-Exception Type* and *Filename-Method Name-Exception Type* feature set.

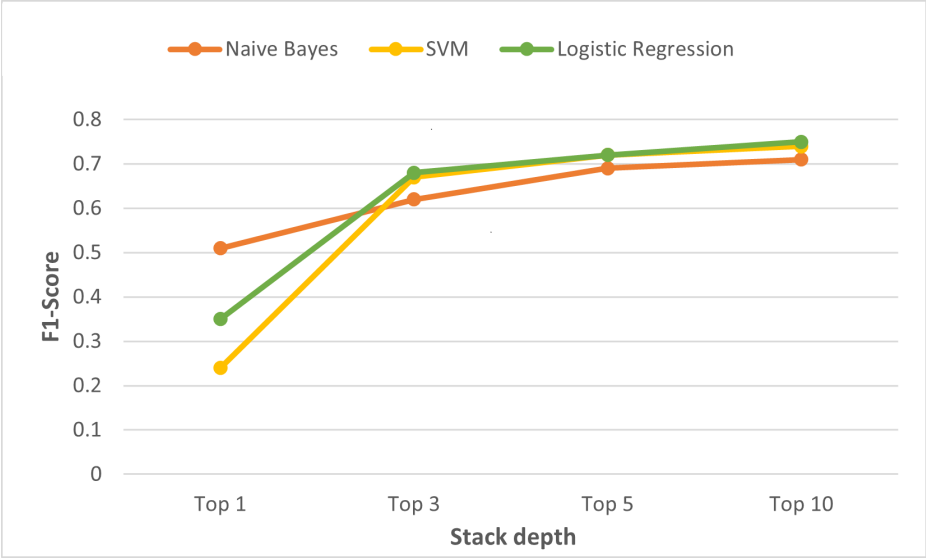

Figure 5.7: F1-score Measure for Top-N stack depth for Birt Project with File name, Exception as features.

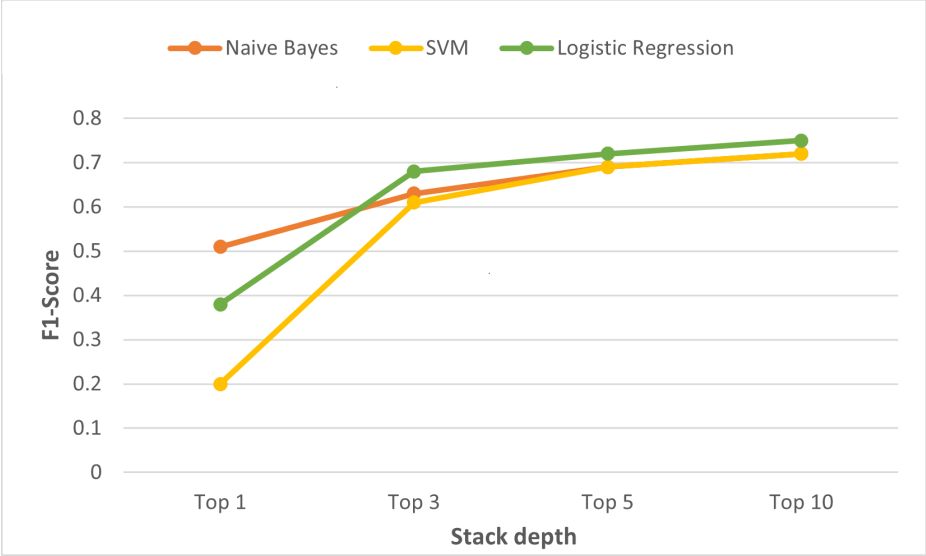

Figure 5.8: F1-score Measure for Top-N stack depth for Birt Project with File name, Method name, Exception as features.

5.4.3 Eclipse Platform UI Project

As shown in Table 5.1, for the Eclipse Platform UI project, we were able to extract 532 bug reports with stack traces that included 1983 filenames, 3954 method names, and 49 exception types.

According to Table 5.9, the maximum precision of 94% with *Filename-Exception Type*

as features for the Top-10 stack depth and the Naive Bayes as algorithm. In contrast, the lowest precision of 26% was found to be with the *Filename* as a feature of the Top-1 stack, with Naive Bayes as the machine learning algorithm. When using *Filename* as a feature and Top-1 stack frame, the highest recall of 98% is recorded for the Naive Bayes algorithm and the lowest recall of 3% is recorded with Logistic Regression as the machine learning algorithm.

Table 5.9: Evaluation metrics for the Eclipse Platform UI Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	0.26	**0.98**	0.42	0.33	0.27	0.94	0.42	0.37	0.35	0.98	0.52	0.56	0.35	0.94	0.51	0.57
	Logistic Regression	0.71	0.03	0.06	0.76	0.55	0.04	0.07	0.76	0.66	0.23	0.35	**0.78**	0.61	0.3	0.41	**0.78**
	SVM	0.64	0.07	0.13	0.77	0.43	0.04	0.07	0.75	0.76	0.14	0.24	**0.78**	0.64	0.09	0.16	0.77
Top 3	Naive Bayes	0.47	0.97	0.63	0.49	0.47	0.95	0.63	0.5	0.49	0.96	0.65	0.54	0.49	0.95	0.65	0.53
	Logistic Regression	0.62	0.21	0.31	0.59	0.58	0.32	0.41	0.59	0.61	0.75	0.67	0.67	0.61	0.72	0.66	0.67
	SVM	0.63	0.2	0.31	0.59	0.6	0.23	0.33	0.59	0.6	0.79	0.68	0.67	0.59	0.71	0.64	0.65
Top 5	Naive Bayes	0.84	0.12	0.21	0.5	0.8	0.15	0.25	0.51	0.9	0.16	0.28	0.53	0.85	0.19	0.31	0.53
	Logistic Regression	0.6 7	0.86	0.71	0.61	0.6	0.81	0.69	0.6	0.64	0.85	0.73	0.65	0.64	0.82	0.72	0.65
	SVM	0.6	0.88	0.71	0.61	0.59	0.86	0.7	0.59	0.63	0.9	0.74	0.65	0.61	0.9	0.73	0.63
Top 10	Naive Bayes	0.91	0.11	0.2	0.45	0.84	0.15	0.25	0.46	**0.94**	0.17	0.29	0.49	0.89	0.2	0.32	0.49
	Logistic Regression	0.65	0.93	0.76	0.65	0.65	0.9	0.76	0.64	0.68	0.93	0.79	0.69	0.68	0.91	0.78	0.68
	SVM	0.65	0.93	0.77	0.65	0.64	0.94	0.76	0.63	0.67	0.94	0.79	0.69	0.66	0.94	0.78	0.67

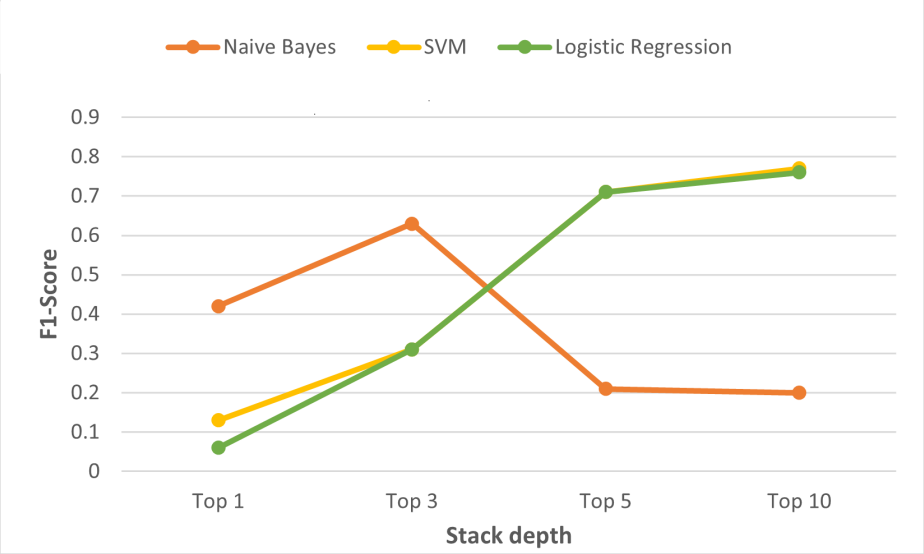

Figure 5.9: F1-score Measure for Top-N stack depth for Eclipse Platform UI Project with Filename as feature.

In terms of accuracy, the Top-1 stack frame and *Filename-Exception Type*, *Filename-Method Name-Exception Type* as feature set in combination with the Logistic Regression and *Filename-Exception Type* with SVM algorithm produced the best result (78%). Using the Top-1 and *Filename* feature set along with the Naive Bayes algorithm, the worst

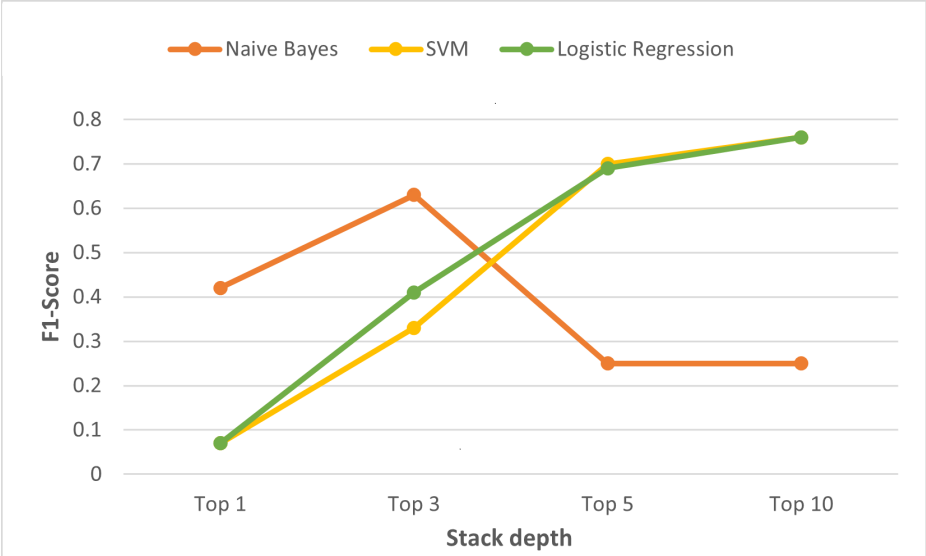

Figure 5.10: F1-score Measure for Top-N stack depth for Eclipse Platform UI Project with File name, Method name as features.

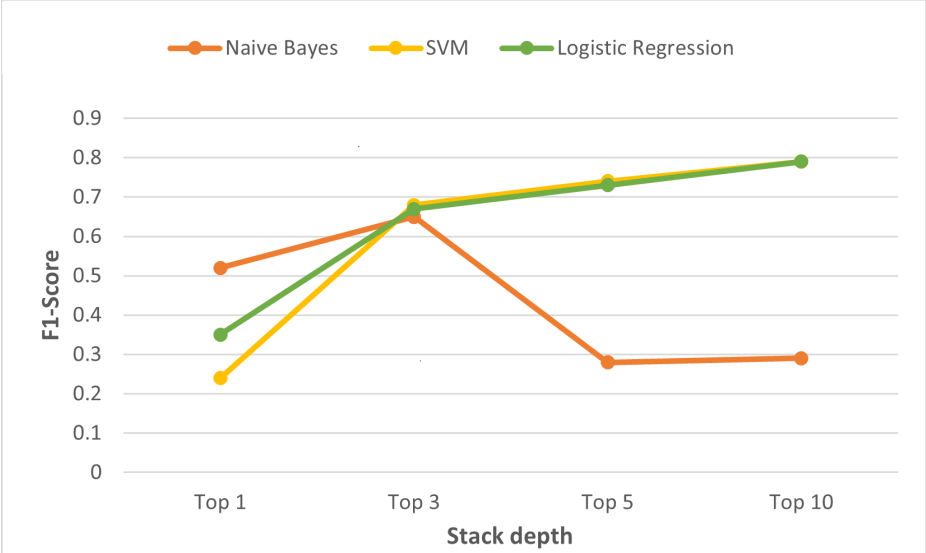

Figure 5.11: F1-score Measure for Top-N stack depth for Eclipse Platform UI Project with File name, Exception as features.

outcome of 33% was discovered.

The F1-score for SVM started at 7% for the Top 1 stack frame using the *Filename* and *Filename-Method Name* feature combinations as shown in Figures 5.9 and 5.10. In case of the Top-3 stack depth, F1-score for the three machine learning algorithm ranged between 64% and 68% for both *Filename-Exception Type* and *Filename-Method Name-*

Exception Type features as shown in Figures 5.11 and 5.12. Regardless of features, all machine learning algorithms performed similarly for the Top-5 and Top-10 stack depths. The highest F1-score (79%) is reported for the features *Filename-Exception Type* in the Top-10 stack frames as shown in Figure 5.12.

5.4.4 JDT Project

As shown in Table 5.1, we obtained 458 bug reports with stack traces from the JDT project, including 1,773 filenames, 3,309 method names, and 60 exception types.

The highest precision of 90% was found with *Filename-Exception Type* as features in the Top-10 stack frame and the Naive Bayes algorithm, according to Table 5.10. In contrast, the Top-1 stack frames with *Filename* feature and the Naive Bayes as the algorithm have the lowest precision at 24%. When using *Filename* as a feature and Top-1 stack frame, the highest recall of 97% is recorded for the Naive Bayes algorithm and the lowest recall of 3% is recorded with Logistic Regression as the machine learning algorithm.

With regard to both *Filename-Exception Type* and *Filename-Method Name-Exception Type* as features and Logistic Regression as a machine learning algorithm, Top-1 has the

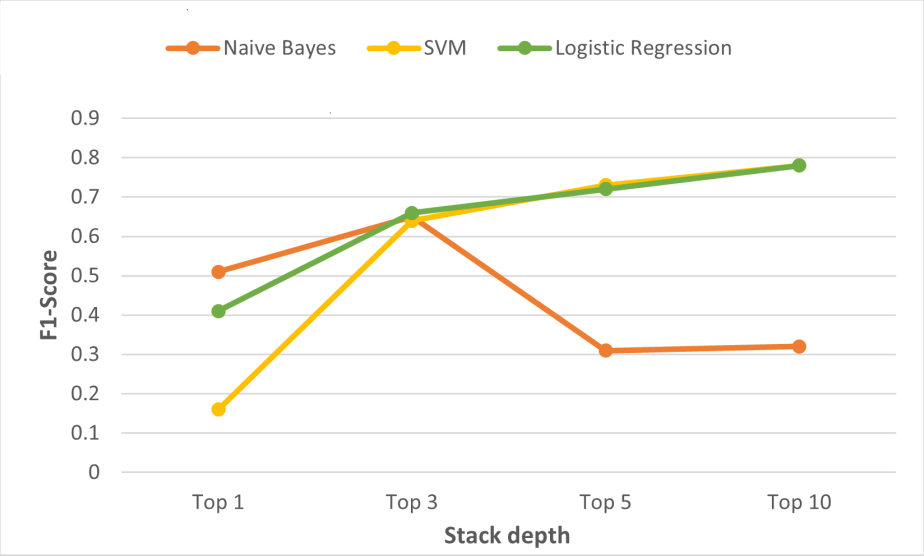

Figure 5.12: F1-score Measure for Top-N stack depth for Eclipse Platform UI Project with File name, Method name, Exception as features.

maximum accuracy (80%). The lowest accuracy (32%) was found using the Top-1 and *Filename* feature set with the Naive Bayes algorithm.

The various machine learning methods performed consistently across all stack depths and feature combinations. For the Top-5 and Top-10 stack frames, the F1-score for the Naive Bayes algorithm is almost similar for all the features as shown in Figures 5.13,5.14,5.15, and 5.16. The highest F1-score (77%) is reported for the Top-10 stack frame with Logistic Regression and *Filename-Exception Type*, *Filename-Method Name-Exception Type* as features and also in case of SVM as algorithm and *Filename-Method Name-Exception Type* as feature.

Table 5.10: Evaluation metrics for the JDT Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	0.24	**0.96**	0.39	0.32	0.25	0.93	0.39	0.36	0.34	0.96	0.5	0.57	0.35	0.93	0.51	0.59
	Logistic Regression	0.6	0.03	0.06	0.78	0.54	0.04	0.07	0.78	0.62	0.3	0.4	**0.8**	0.6	0.31	0.41	**0.8**
	SVM	0.59	0.09	0.16	0.78	0.44	0.04	0.08	0.77	0.65	0.11	0.19	0.79	0.52	0.05	0.09	0.78
Top 3	Naive Bayes	0.49	0.97	0.65	0.52	0.5	0.95	0.65	0.53	0.51	0.97	0.67	0.55	0.52	0.95	0.67	0.56
	Logistic Regression	0.58	0.51	0.54	0.6	0.58	0.5	0.54	0.6	0.67	0.75	0.71	0.71	0.68	0.71	0.71	0.71
	SVM	0.61	0.39	0.48	0.6	0.59	0.39	0.47	0.59	0.67	0.75	0.7	0.71	0.65	0.71	0.68	0.69
Top 5	Naive Bayes	0.81	0.13	0.22	0.49	0.81	0.15	0.25	0.5	0.84	0.15	0.25	0.51	0.84	0.17	0.28	0.52
	Logistic Regression	0.63	0.85	0.72	0.64	0.64	0.82	0.72	0.64	0.71	0.78	0.74	0.7	0.71	0.78	0.74	0.7
	SVM	0.63	0.85	0.73	0.64	0.62	0.84	0.72	0.63	0.7	0.8	0.75	0.7	0.69	0.79	0.74	0.68
Top 10	Naive Bayes	0.88	0.12	0.22	0.46	0.84	0.15	0.25	0.47	**0.9**	0.15	0.25	0.48	0.87	0.17	0.28	0.48
	Logistic Regression	0.65	0.9	0.76	0.65	0.66	0.88	0.76	0.66	0.71	0.83	**0.77**	0.7	0.71	0.83	**0.77**	0.7
	SVM	0.66	0.89	0.76	0.66	0.66	0.88	0.75	0.65	0.71	0.83	0.76	0.69	0.69	0.87	**0.77**	0.68

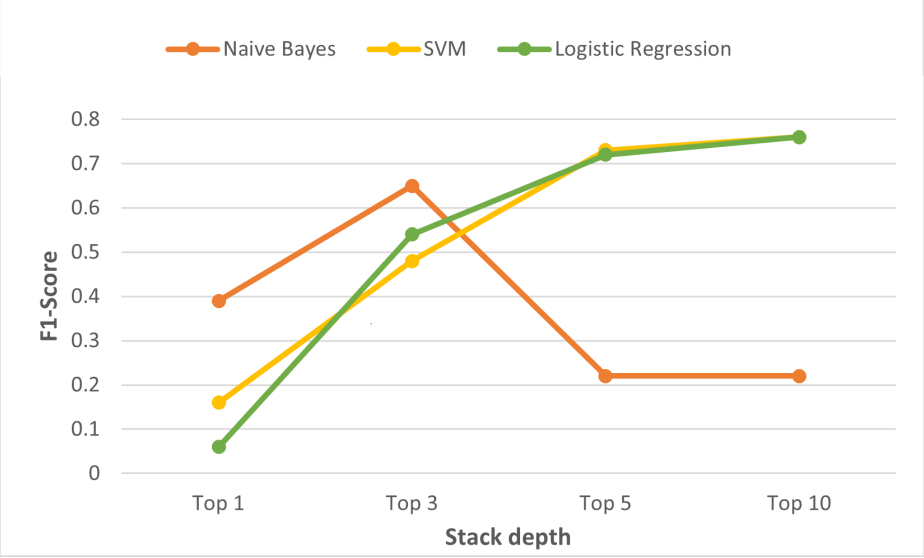

Figure 5.13: F1-score Measure for Top-N stack depth for JDT Project with Filename as feature.

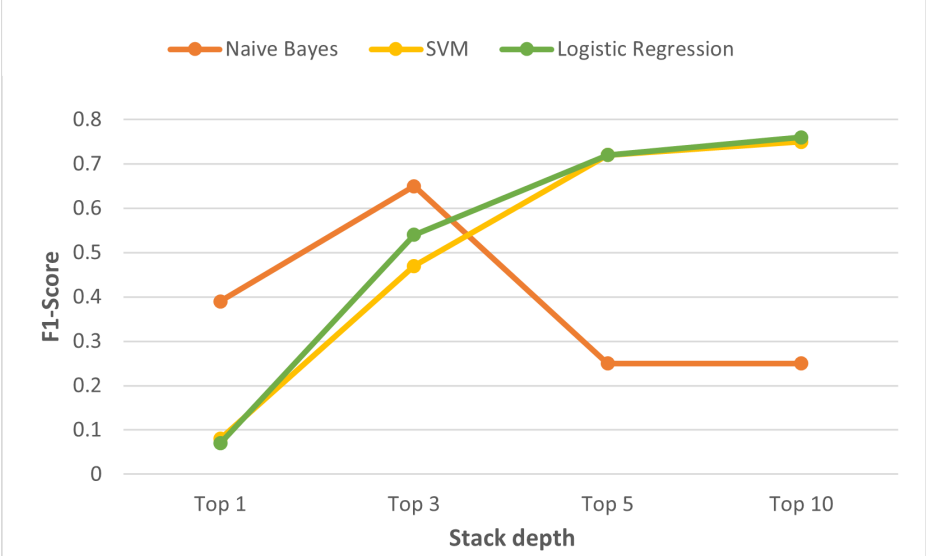

Figure 5.14: F1-score Measure for Top-N stack depth for JDT Project with File name, Method name as features.

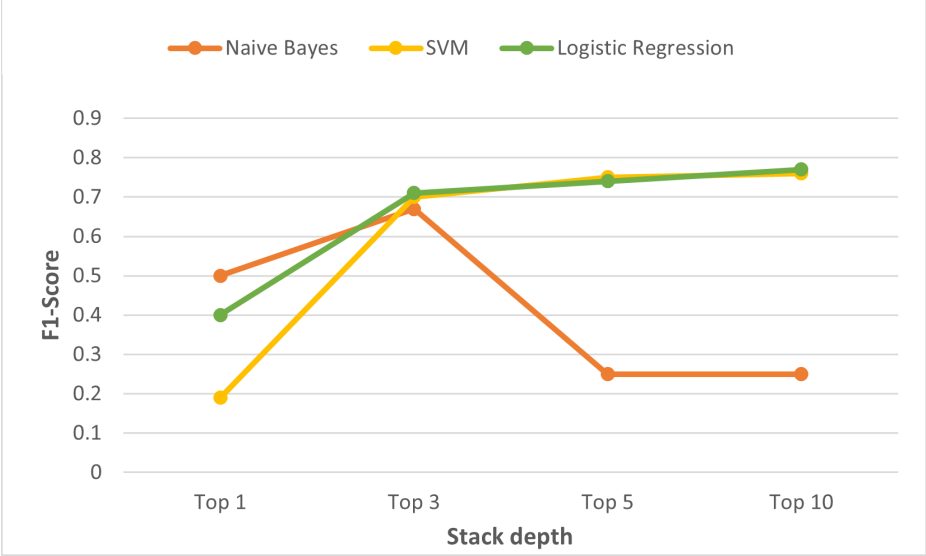

Figure 5.15: F1-score Measure for Top-N stack depth for JDT Project with File name, Exception as features.

5.4.5 Cassandra Project

The Cassandra recommenders were trained on 903 bug reports with stack traces and contained 1,702 file names, 3,272 method names, and 104 exception types as shown in the Table 5.2.

Precision (89%) is highest for the Naive Bayes and accuracy (79%) is highest for the

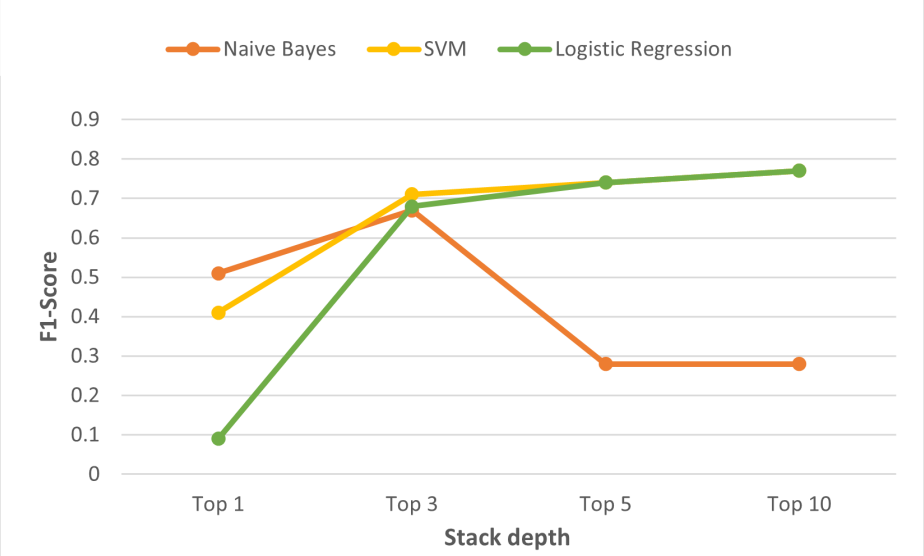

Figure 5.16: F1-score Measure for Top-N stack depth for JDT Project with File name, Method name, Exception as features.

SVM with *Filename-Exception Type* as features for Top-10 stack frames, as demonstrated in Table 5.11. Lowest Precision (26%) is reported when using the Naive Bayes as algorithm and *Filename*, *Filename-Method Name* as features in the Top-1 stack frame. The lowest accuracy (36%) was found in the Top-1 stack frame and *Filename* feature set with the Naive Bayes algorithm.

Using *Filename-Exception Type* as the feature, the Naive Bayes model has the maximum recall (94%) with Top-3 stack frames. The worst recall (3%) occurred when training the Logistic Regression algorithm with the Top-1 stack frames and the *Filename* feature set.

Table 5.11: Evaluation metrics for the Cassandra Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	0.26	0.89	0.4	0.36	0.26	0.79	0.39	0.42	0.29	0.9	0.44	0.46	0.3	0.81	0.43	0.5
	Logistic Regression	0.67	0.03	0.06	0.77	0.51	0.05	0.08	0.76	0.65	0.16	0.26	0.78	0.63	0.19	0.29	0.78
	SVM	0.55	0.1	0.16	0.77	0.44	0.07	0.12	0.76	0.68	0.18	0.28	**0.79**	0.55	0.12	0.2	0.77
Top 3	Naive Bayes	0.45	0.93	0.61	0.49	0.46	0.89	0.6	0.49	0.49	**0.94**	0.64	0.54	0.48	0.9	0.63	0.54
	Logistic Regression	0.62	0.32	0.42	0.62	0.6	0.35	0.44	0.62	0.63	0.46	0.53	0.65	0.62	0.5	0.55	0.65
	SVM	0.62	0.32	0.42	0.62	0.58	0.25	0.35	0.6	0.66	0.37	0.47	0.64	0.62	0.3	0.41	0.62
Top 5	Naive Bayes	0.71	0.17	0.28	0.52	0.72	0.19	0.3	0.53	0.72	0.39	0.51	0.6	0.75	0.24	0.36	0.55
	Logistic Regression	0.6	0.66	0.63	0.58	0.61	0.66	0.63	0.59	0.68	0.7	0.69	0.66	0.67	0.7	0.69	0.66
	SVM	0.6	0.66	0.63	0.58	0.59	0.71	0.64	0.58	0.66	0.76	0.71	0.66	0.63	0.74	0.68	0.63
Top 10	Naive Bayes	0.83	0.12	0.2	0.44	0.77	0.2	0.31	0.47	**0.89**	0.18	0.31	0.48	0.82	0.24	0.38	0.5
	Logistic Regression	0.66	0.87	0.75	0.64	0.65	0.85	0.74	0.63	0.72	0.86	**0.79**	0.71	0.72	0.85	0.78	0.7
	SVM	0.65	0.89	0.75	0.64	0.64	0.88	0.74	0.63	0.7	0.9	0.78	0.69	0.67	0.89	0.77	0.67

For higher Top-N stack depths, the F1-score for Naive Bayes dramatically drops. However, when the stack depth increases, the F1-score for SVM and Logistic Regression also

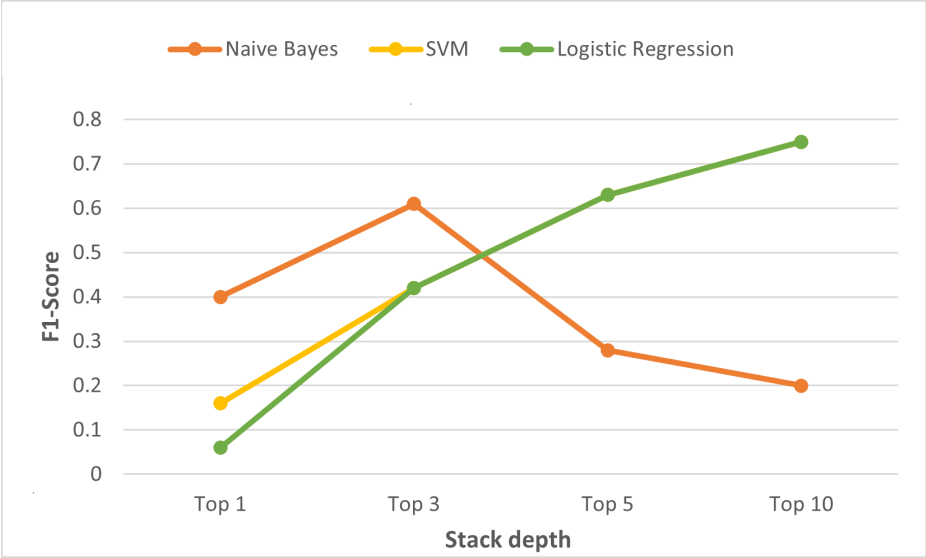

Figure 5.17: F1-score Measure for Top-N stack depth for Cassandra Project with Filename as feature.

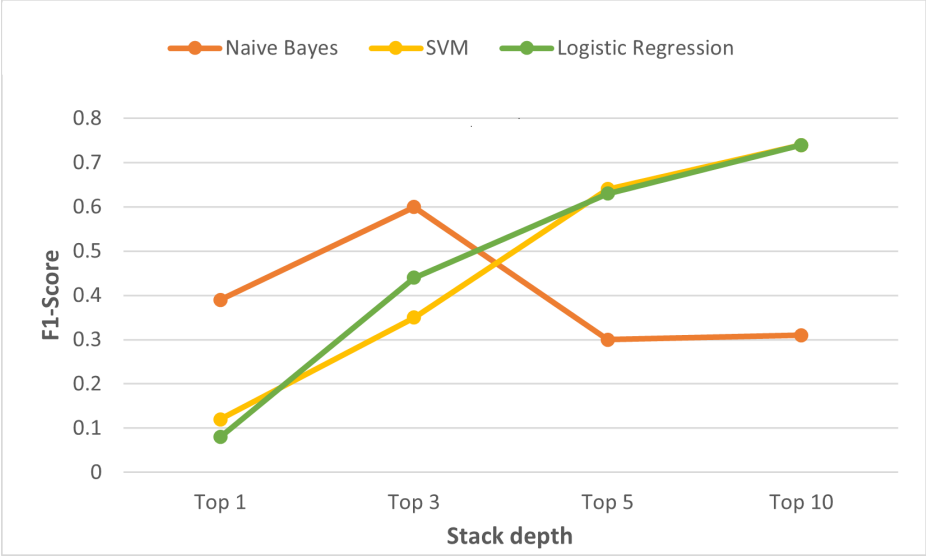

Figure 5.18: F1-score Measure for Top-N stack depth for Cassandra Project with File name, Method name as features.

increases. Logistic Regression and SVM had F1-Scores between 74% and 79% in the Top-10 stack frame, while Naive Bayes had F1-Scores between 20% and 38% as shown in Figures 5.17, 5.18, 5.19 and 5.20.The highest F1-score (79%) is reported for the Top-10 stack frame with Logistic Regression and *Filename-Exception Type* as features, and the lowest F1-score of 5% is found in the Top-1 stack frame when using Logistic Regression as

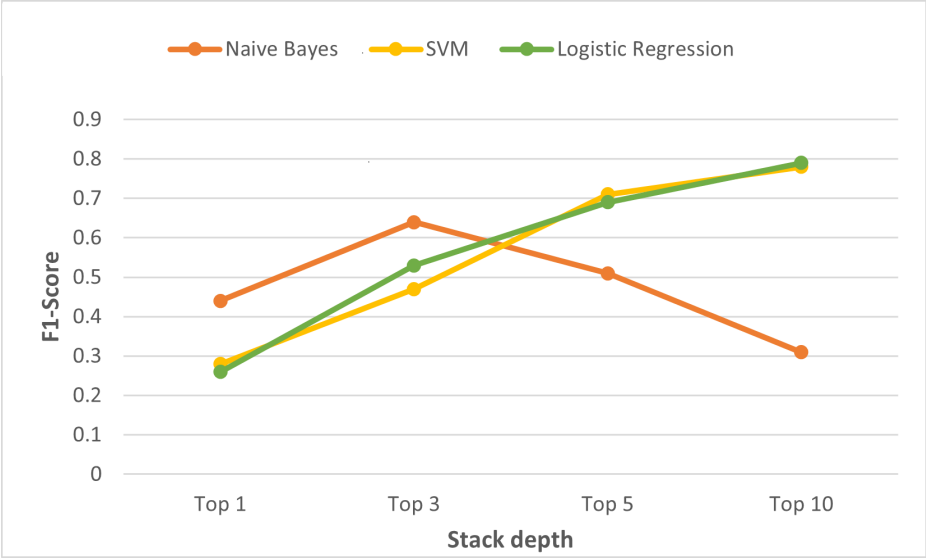

Figure 5.19: F1-score Measure for Top-N stack depth for Cassandra Project with File name, Exception as features.

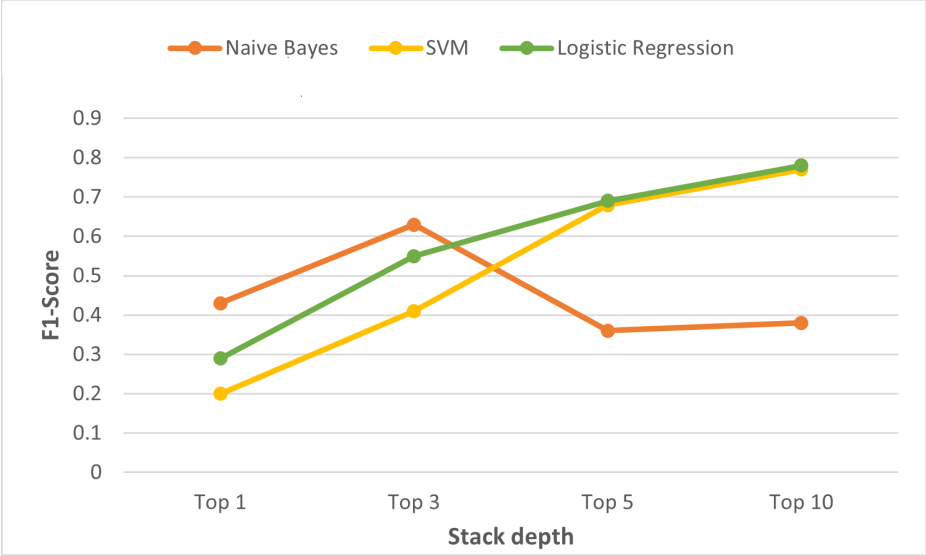

Figure 5.20: F1-score Measure for Top-N stack depth for Cassandra Project with File name, Method name, Exception as features.

machine learning algorithm and *Filename-Method Name* as features. The worst F1-score of 6% was found using the Top-1 and *Filename* feature set with the Logistic Regression algorithm.

5.4.6 Hadoop Project

As shown in Table 5.2, we retrieved only 36 bug reports containing stack traces from a total of 2,395 bug reports from the Hadoop project. These stack traces contains 202 filenames, 311 method names, and 18 exception types.

The highest accuracy of 98% was found using *Filename-Method Name-Exception Type* features in Top-1 stack frames when using the Naive Bayes algorithm, as shown in Table 5.12. The lowest accuracy was 56% with the *Filename* as a feature and the Naive Bayes as the machine learning algorithm for the Top-3 stack depth. The Logistic Regression model records the minimal recall (17%) with Top-1 stack frame for *Filename* as a feature.

Table 5.12: Evaluation metrics for the Hadoop Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	0.51	1	0.68	0.73	0.68	0.97	0.8	0.86	0.85	1	0.92	0.95	0.94	0.97	**0.96**	**0.98**
	Logistic Regression	0.86	0.17	0.29	0.76	1	0.34	0.51	0.81	1	0.77	0.87	0.93	1	0.8	0.89	0.94
	SVM	0.79	0.54	0.64	0.83	0.92	0.34	0.5	0.8	0.9	0.77	0.83	0.91	1	0.54	0.7	0.87
Top 3	Naive Bayes	0.49	0.92	0.64	0.56	0.61	0.88	0.72	0.71	0.75	0.92	0.83	0.84	0.82	0.88	0.85	0.87
	Logistic Regression	0.75	0.23	0.35	0.64	0.87	0.38	0.53	0.72	0.98	0.87	0.92	0.93	0.96	0.85	0.9	0.92
	SVM	0.74	0.44	0.55	0.7	0.81	0.33	0.47	0.68	0.86	0.71	0.78	0.83	0.89	0.48	0.62	0.76
Top 5	Naive Bayes	0.58	0.85	0.69	0.59	0.63	0.92	0.75	0.67	0.95	0.88	0.91	0.91	0.86	0.95	0.91	0.89
	Logistic Regression	0.71	0.42	0.52	0.6	0.8	0.62	0.7	0.72	1	0.89	0.94	0.94	1	0.89	0.94	0.94
	SVM	0.71	0.42	0.52	0.6	0.84	0.55	0.67	0.71	0.9	0.83	0.86	0.86	0.93	0.85	0.89	0.89
Top 10	Naive Bayes	0.81	0.41	0.54	0.59	0.9	0.51	0.66	0.67	1	0.74	0.85	0.85	0.98	0.74	0.85	0.84
	Logistic Regression	0.69	0.82	0.75	0.67	0.77	0.76	0.76	0.72	0.99	0.93	**0.96**	0.95	0.95	0.93	0.94	0.93
	SVM	0.77	0.49	0.6	0.6	0.83	0.54	0.66	0.66	0.91	0.8	0.85	0.83	0.92	0.8	0.86	0.84

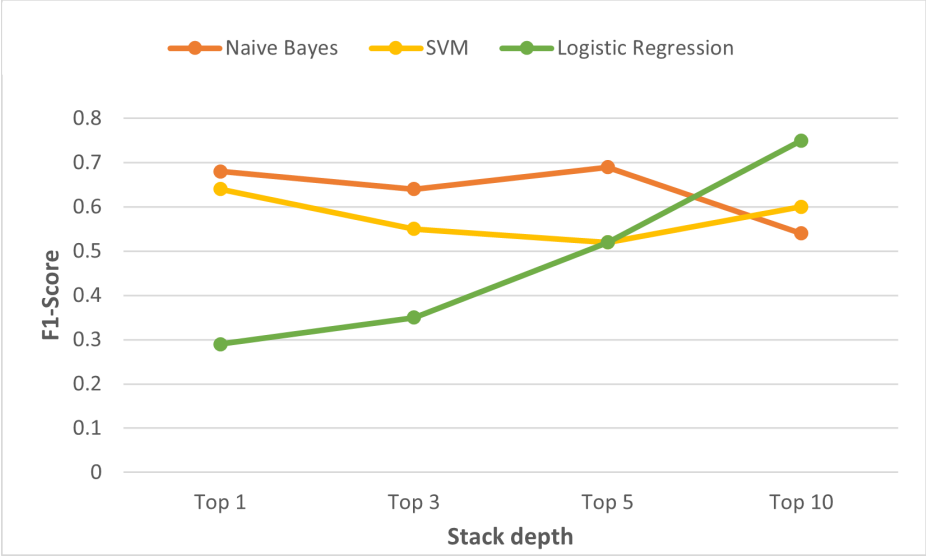

Figure 5.21: F1-score Measure for Top-N stack depth for Hadoop Project with Filename as feature.

When using the *Filename* as a feature and the Top 1 stack depth, the F1-score for Lo-

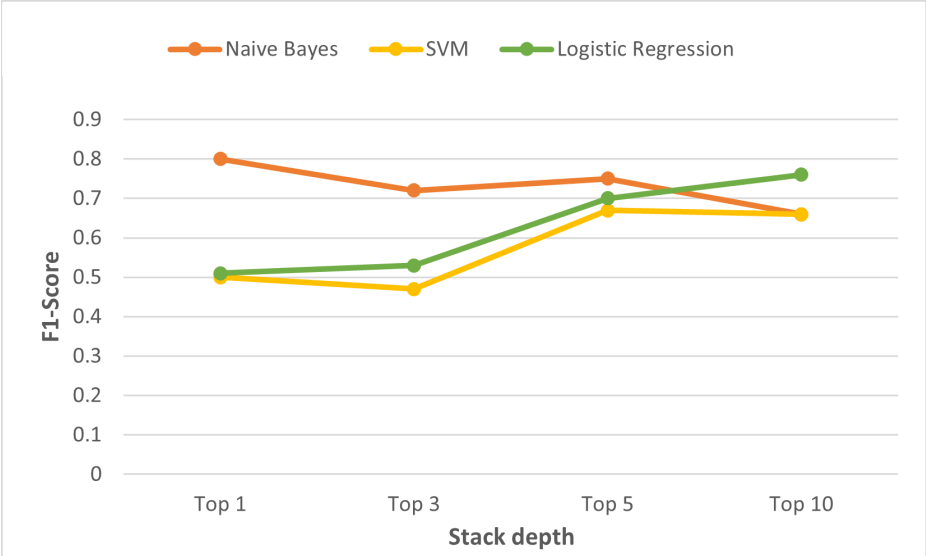

Figure 5.22: F1-score Measure for Top-N stack depth for Hadoop Project with File name, Method name as features.

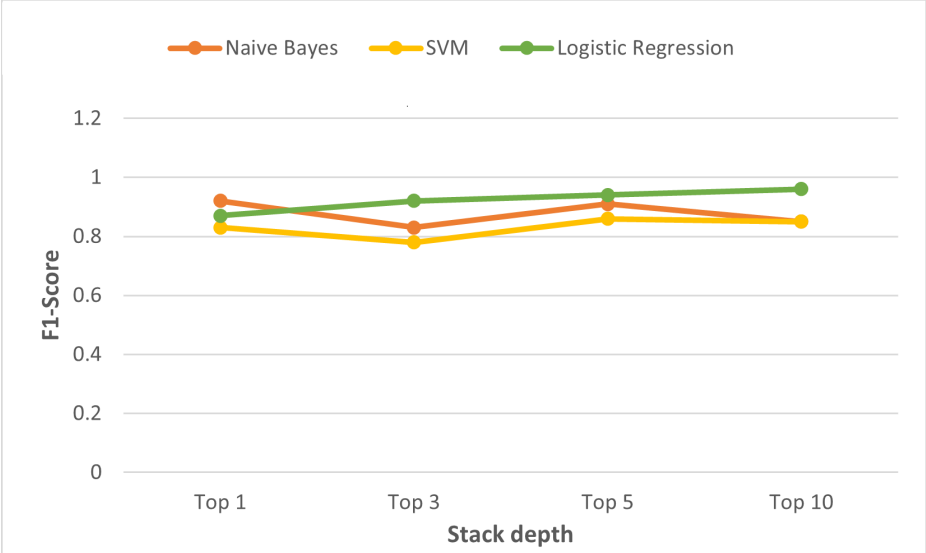

Figure 5.23: F1-score Measure for Top-N stack depth for Hadoop Project with File name, Exception as features.

gistic Regression started at 29% and increased to 75% in the Top 10 stack depth as shown in Figure 5.21. For all stack depths and algorithms, the F1-score for *Filename-Exception Type* as features ranged from 78% to 96% as shown in Figure 5.23. Logistic Regression performed well in all stack depths and *Filename-Method Name-Exception Type* , as shown in Figure 5.24. The highest F1-score (96%) is reported for the Top-10 stack frame with

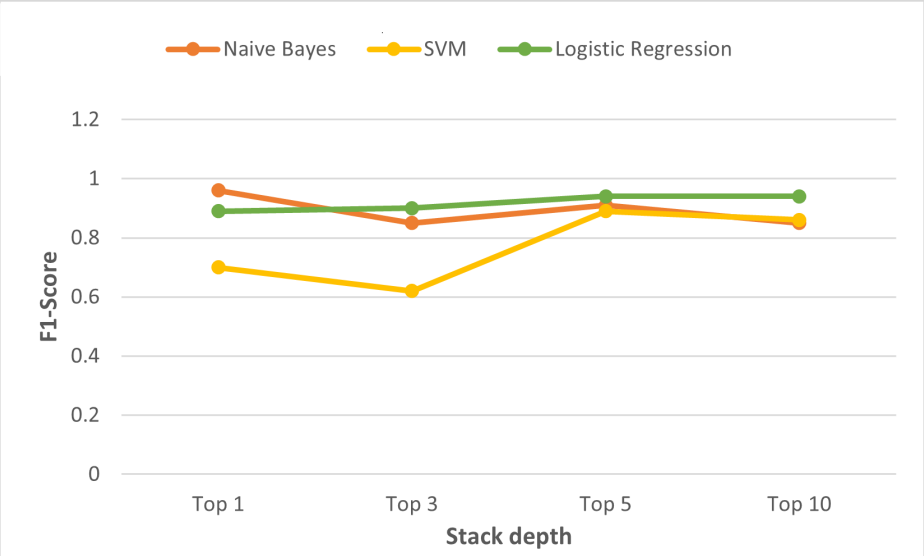

Figure 5.24: F1-score Measure for Top-N stack depth for Hadoop Project with File name, Method name, Exception as features.

Logistic Regression and *Filename-Exception Type*, as features and also in case of Naive Bayes as algorithm and *Filename-Method Name-Exception Type* as feature for the Top-1 stack frame.The lowest F1-score (29%) was found using the Top-1 and *Filename* feature set with the Logistic Regression algorithm.

5.4.7 Hbase Project

The Hbase project has 1,264 bug reports with stack traces, as seen in 5.2, with which we trained the machine learning algorithms. These stack traces contains 1,931 filenames, 3,076 method names, and 96 exception types.

Logistic Regression had the highest accuracy with 84% using Top-1 Stack depth, and Naive Bayes had the highest precision with 94% using Top-10 stack depth with *File name-Exception Type* features. In contrast, the Naive Bayes has the lowest precision with 20% and the lowest accuracy with 32% using Top-1 stack depth with *File name* features. With *Filename* as feature, Logistic Regression has the lowest recall of 1% for Top-1 stack frames, while Naive Bayes has the best recall of 96% for Top-3 stack frames.

As illustrated in Figure 5.25, the F1-score for Logistic Regression and SVM began very

Table 5.13: Evaluation metrics for the Hbase Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	0.2	0.93	0.33	0.32	0.21	0.83	0.34	0.4	0.24	0.93	0.38	0.46	0.25	0.85	0.39	0.52
	Logistic Regression	0.54	0.01	0.02	0.82	0.43	0.02	0.05	0.82	0.7	0.16	0.26	**0.84**	0.66	0.17	0.27	0.83
	SVM	0.47	0.05	0.1	0.82	0.37	0.05	0.09	0.81	0.72	0.14	0.24	0.83	0.63	0.12	0.21	0.83
Top 3	Naive Bayes	0.46	0.96	0.62	0.49	0.47	0.92	0.62	0.51	0.5	**0.96**	0.66	0.57	0.51	0.92	0.65	0.58
	Logistic Regression	0.6	0.38	0.46	0.63	0.61	0.4	0.48	0.63	0.72	0.69	0.7	0.75	0.71	0.68	0.69	0.74
	SVM	0.62	0.35	0.45	0.63	0.63	0.29	0.4	0.62	0.68	0.7	0.69	0.73	0.7	0.57	0.63	0.71
Top 5	Naive Bayes	0.58	0.93	0.71	0.6	0.73	0.22	0.33	0.54	0.62	0.94	0.75	0.66	0.81	0.32	0.46	0.59
	Logistic Regression	0.66	0.66	0.66	0.63	0.65	0.72	0.68	0.64	0.75	0.78	0.76	0.74	0.75	0.78	0.76	0.74
	SVM	0.64	0.74	0.69	0.63	0.63	0.78	0.69	0.63	0.72	0.79	0.75	0.72	0.71	0.78	0.74	0.71
Top 10	Naive Bayes	0.89	0.12	0.21	0.41	0.83	0.19	0.31	0.44	**0.94**	0.24	0.38	0.49	0.88	0.29	0.44	0.51
	Logistic Regression	0.72	0.92	0.8	0.7	0.71	0.9	0.8	0.7	0.79	0.89	**0.84**	0.77	0.79	0.89	**0.84**	0.77
	SVM	0.71	0.93	0.81	0.7	0.7	0.92	0.8	0.69	0.78	0.92	**0.84**	0.77	0.75	0.92	0.82	0.74

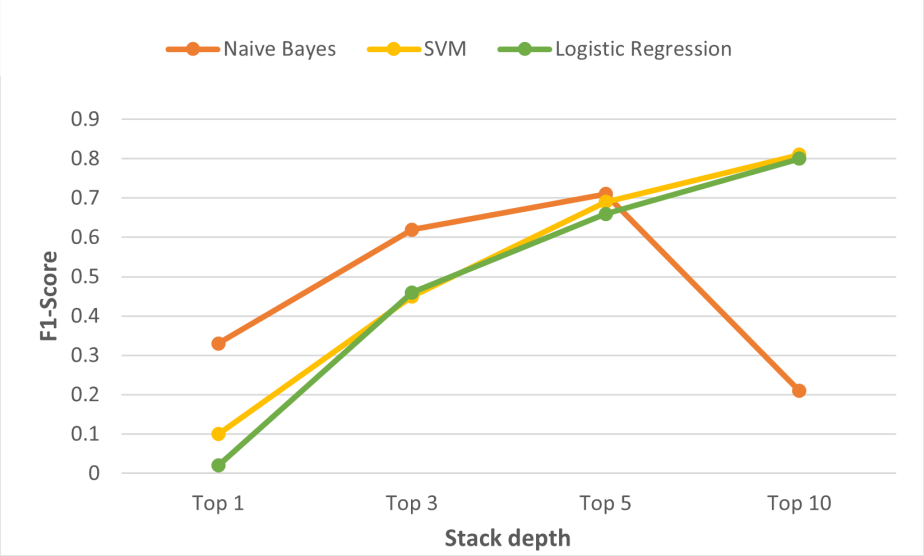

Figure 5.25: F1-score Measure for Top-N stack depth for Hbase Project with Filename as feature.

low, at 2% and 1%, respectively, for the Top-1 stack frames using the filename as a feature. Compared to the other two machine learning algorithms in the Top-3, Naive Bayes did well with 66%. Logistic Regression and SVM are 76% and 74% with *Filename-Method Name-Exception Type* as features for Top-5 stack depth and were 84% and 82% in the Top-10 stack depth respectively as shown in Figure 5.28.

The F1-score for the Naive Bayes falls from Top-5 stack frame for *Filename-Method Name* feature set as shown in Figure 5.26, whereas it falls in Top-10 stack depth for *Filename-Exception Type* features as shown in Figure 5.27. The highest F1-score (84%) is reported for Logistic Regression and SVM for Top-10 stack depth and both *Filename-Exception Type*, *Filename-Method Name-Exception Type* as features.

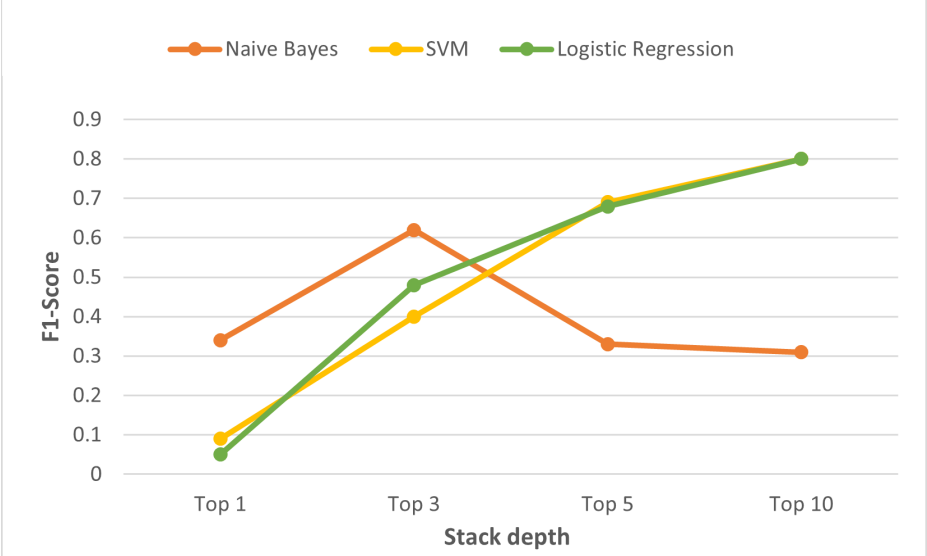

Figure 5.26: F1-score Measure for Top-N stack depth for Hbase Project with File name, Method name as features.

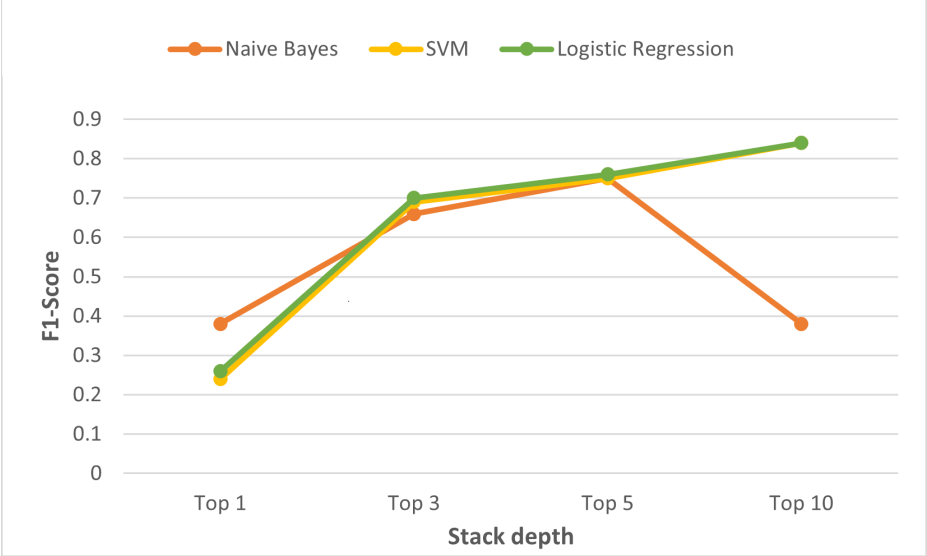

Figure 5.27: F1-score Measure for Top-N stack depth for Hbase Project with File name, Exception as features.

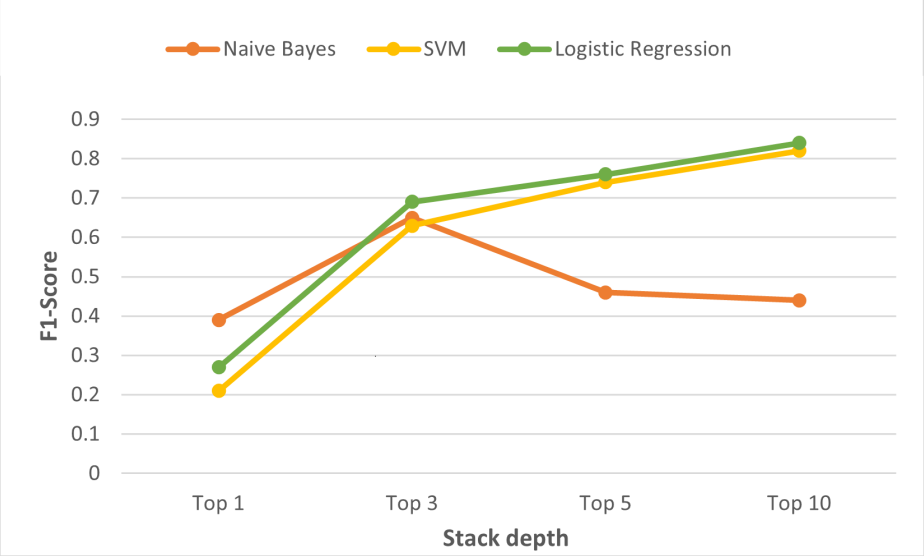

Figure 5.28: F1-score Measure for Top-N stack depth for Hbase Project with File name, Method name, Exception as features.

5.4.8 Spring Project

As shown in Table 5.2, we retrieved 431 bug reports with stack traces containing 1,931 filenames, 3,076 method names, and 96 exception types from the Spring project.

According to Table 5.14, the highest precision of 87% was found using *Filename-Exception Type* as features in Top-10 stack depth, while the lowest precision of 19% was found using the *Filename* as a feature in Top-1 stack frames using the Naive Bayes as an algorithm. The highest recall of 96% is found for the Naive Bayes algorithm when using *Filename*, *Filename-Exception Type* as features in Top-3 stack depth and also for Top-5 stack depth with *Filename-Exception Type* as feature set. When using *Filename-Method Name* as a feature set and Logistic Regression as a machine learning algorithm, the lowest recall (2%) was recorded in Top-1 stack frames.

Accuracy is highest in the case of Top-1 stack frames with 88%, in both *Filename-Exception Type* and *Filename-Method Name-Exception Type* as features using the Logistic Regression as a machine learning algorithm.

Table 5.14: Evaluation metrics for the Spring Top-N Stack depths.

Stack depth	ML Algorithm	Filename				Filename, Method name				Filename, Exception				Filename,Method name, Exception			
		Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy	Precision	Recall	F1-score	Accuracy
Top 1	Naive Bayes	**0.19**	0.88	0.31	0.38	0.2	0.78	0.32	0.46	0.27	0.92	0.42	0.6	0.28	0.82	0.42	0.64
	Logistic Regression	0.78	0.02	0.04	0.84	0.54	0.02	0.04	0.84	0.9	0.29	0.44	**0.88**	0.82	0.29	0.43	**0.88**
	SVM	0.5	0.1	0.17	0.84	0.3	0.04	0.07	0.83	0.77	0.3	0.43	0.87	0.7	0.24	0.36	0.86
Top 3	Naive Bayes	0.4	**0.96**	0.56	0.47	0.41	0.91	0.56	0.5	0.45	**0.96**	0.62	0.58	0.46	0.92	0.61	0.59
	Logistic Regression	0.58	0.28	0.37	0.67	0.59	0.39	0.47	0.69	0.73	0.6	0.66	0.78	0.74	0.62	0.67	0.79
	SVM	0.63	0.34	0.44	0.7	0.57	0.3	0.39	0.67	0.77	0.49	0.6	0.77	0.73	0.4	0.52	0.74
Top 5	Naive Bayes	0.48	0.95	0.64	0.53	0.5	0.9	0.64	0.56	0.55	**0.96**	0.7	0.63	0.56	0.92	0.7	0.65
	Logistic Regression	0.67	0.49	0.56	0.66	0.64	0.54	0.59	0.66	0.74	0.72	0.73	0.76	0.75	0.72	0.73	0.77
	SVM	0.67	0.45	0.54	0.66	0.68	0.5	0.57	0.67	0.77	0.6	0.67	0.74	0.74	0.61	0.67	0.73
Top 10	Naive Bayes	0.8	0.29	0.42	0.61	0.76	0.29	0.42	0.6	**0.87**	0.44	0.58	0.69	0.84	0.43	0.57	0.68
	Logistic Regression	0.62	0.74	0.67	0.65	0.63	0.7	0.66	0.65	0.75	0.77	**0.76**	0.76	0.75	0.78	**0.76**	0.76
	SVM	0.66	0.57	0.61	0.64	0.65	0.62	0.63	0.64	0.76	0.75	0.75	0.76	0.74	0.72	0.73	0.73

As demonstrated in Figure 5.29 and 5.30, the Naive Bayes classifier dominated the other two classifiers up to Top-5 stack depths where as Logistic Regression dominated in all stack depths cases as shown in the Figures 5.31 and 5.32.The Logistic Regression has the highest F1 scores of 76% using both *Filename-Exception Type*, *Filename-Method Name-Exception Type* as features for the Top-10 stack depth.In contrast, the lowest F1-score (4%) is reported for the Logistic Regression algorithm for Top-1 stack depth when using both *Filename* and *Filename-Method Name* as features.

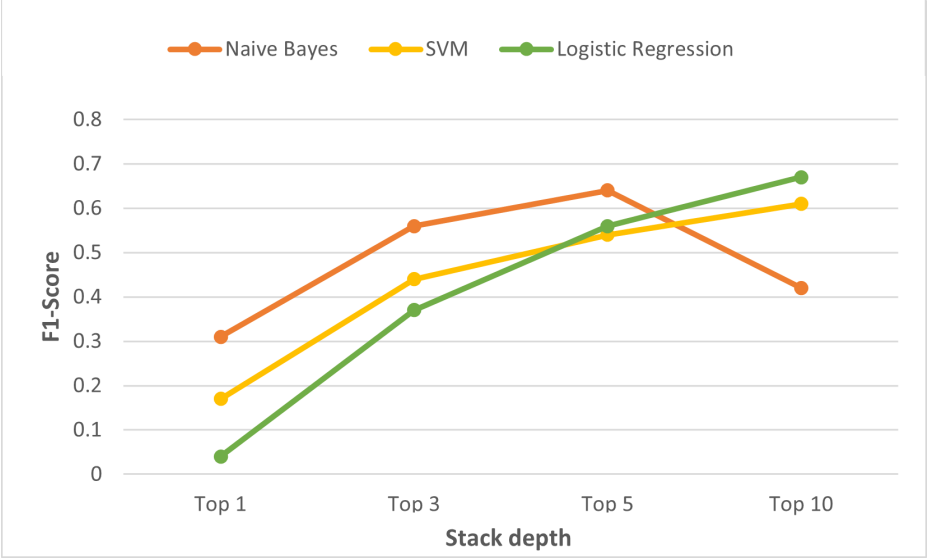

Figure 5.29: F1-score Measure for Top-N stack depth for Spring Project with Filename as feature.

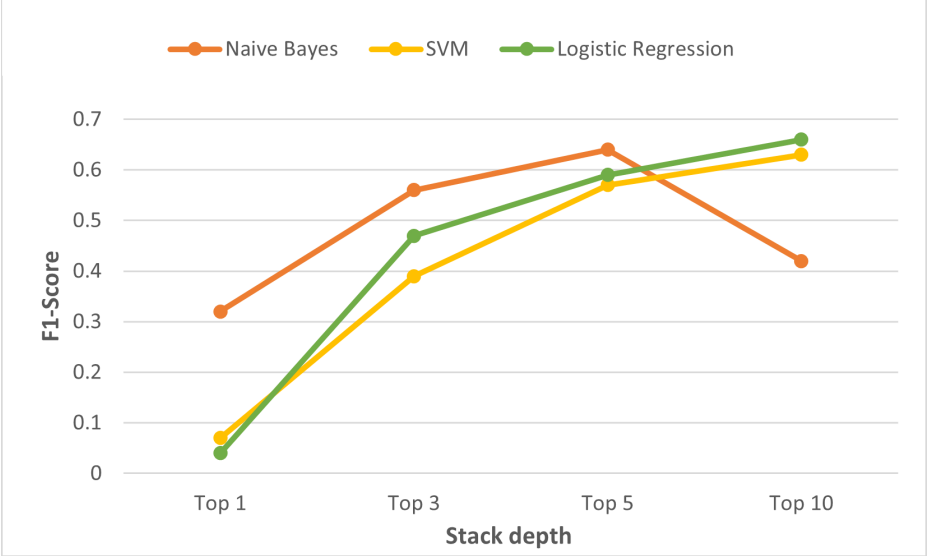

Figure 5.30: F1-score Measure for Top-N stack depth for Spring Project with File name, Method name as features.

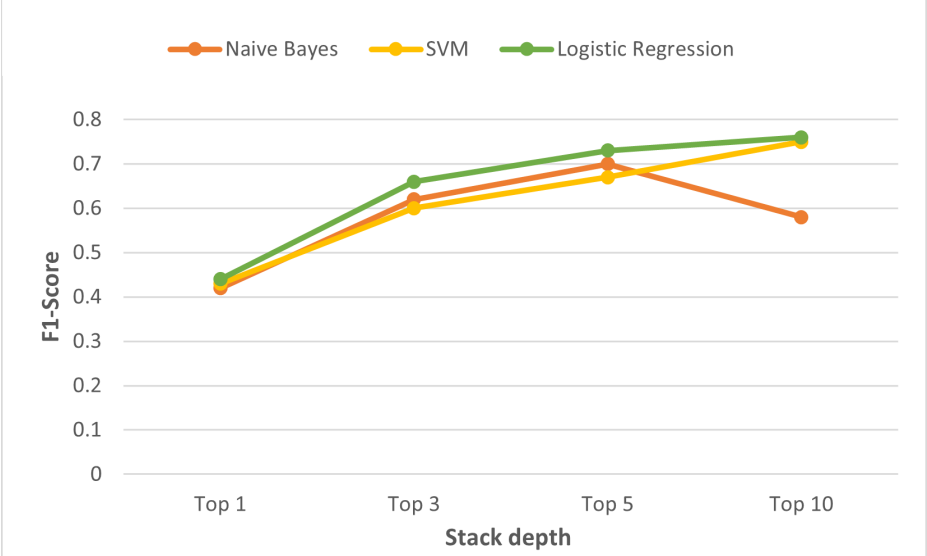

Figure 5.31: F1-score Measure for Top-N stack depth for Spring Project with File name, Exception as features.

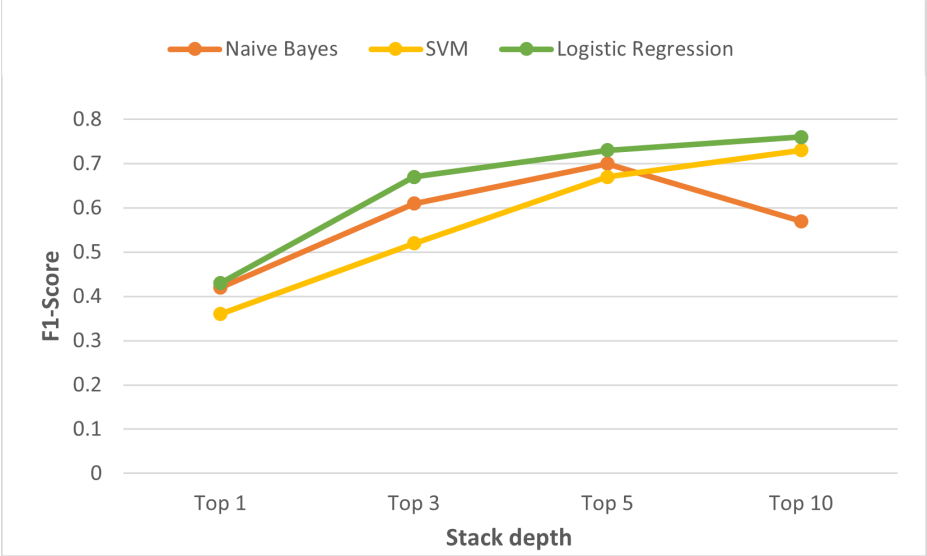

Figure 5.32: F1-score Measure for Top-N stack depth for Spring Project with File name, Method name, Exception as features.

5.5 Summary

According to the research question #1 results, the number of inaccurate stack traces decreased as the number of stack frames increased. In the Eclipse data set, we observed 72% inaccurate stack traces in the Top-1, 51% in the Top-3 stack depth, 43% in the Top-5 and 39% in the Top-10 stack depth. However, in the Apache data sets, we observed 69% and 54% inaccurate stack traces for the Top-1 and Top-3 stack depths, respectively. We discovered 26% and 14% for the Top-5 and Top-10 stack depths, respectively.

From the research question #2 results, the most frequent exception in inaccurate stack traces across all stack depths was the *NullPointerException* with the mean percentage of 35% and 15.6% in the corresponding data sets from Eclipse and Apache. Another frequent occurrence, with 9.6% in the Eclipse and 11.4% in the Apache data set in both the data sets, is *IlleagalArgumentException*.

Finally, analysing the research question #3 results, in the Eclipse data sets, the F1-score for the Naive Bayes classifier improved as the stack depth increased for the Birt project. Still, it decreased or stayed within a similar range for the rest of the projects after the Top-3 stack frame in all the feature combinations. In contrast, the F1-score for Logistic Regression and SVM were different for Top-1 and Top-3 stack depths, while they were identical for Top-5 and Top-10 stack frames regardless of feature combination.

Three projects (AspectJ, Eclipse Platform UI, and JDT) claimed the highest precision for the feature *Filename-Exception Type* and the Naive Bayes as machine learning algorithm, while the Birt project reported the highest precision for the Logistic Regression. Similarly, we found that the Naive Bayes classifier achieves the maximum precision and recall values when using the *Filename-Exception Type* as features for the projects in the Apache data set. Therefore, *Filename-Exception Type* is the best feature combination for both the Eclipse and Apache projects.

On the other hand, across all feature combinations for the Apache data sets, the F1-score for the Naive Bayes classifier either declined significantly or maintained close range after

the Top-3 stack frames. Except for the Hadoop project, all feature combinations used with Logistic Regression and SVM showed improved F1 scores as the stack depth increased, unlike the Eclipse projects. The highest F1-score for all the Apache projects varied from 76% to 96%, and accuracy was between 79% and 98%. While for the Eclipse data sets, the highest accuracy was obtained using the Logistic Regression algorithm and was between 78% and 82%.

For the Eclipse data sets, the F1-score ranges from 6-75% for the Top-1 stack depth, 28-85% for the Top-3 stack depth, 21- 85% for the Top-5 stack depth and 20- 87% for the Top-10 stack depth. Whereas, the F1-score for the Apache data sets falls between 2- 96% for the Top-1 stack depth, 35- 92% for the Top-3 stack depth, 28- 94% for the Top-5 stack depth and 20- 96% for the Top-10 stack depth. Therefore, when training a stack trace accuracy recommender, the Top-5 frames will creating a sufficiently accurate recommender.

Evaluating the machine learning algorithms across on both the data sets, the best F1-score was found using the Logistic Regression algorithm. We observed that the F1-score for the Logistic Regression ranged between 4% and 96% , Naive Bayes between 20% and 91%, and SVM fell between 8% and 77%.

Chapter 6

Discussion

This chapter discusses the different feature combinations, which stack trace depth is most beneficial, and the lessons we learned from using machine learning algorithms to recommend inaccurate stack traces.

6.1 How deep should stack traces be in bug reports?

In research question #1, we investigated detecting accurate/inaccurate stack traces by considering the filenames included in up to 10 stack frames. The results indicate that it makes little difference whether you utilize the Top-5 or Top-10 stack depths. The Eclipse data set shows a 2% to 5.5% increase in accurate stack traces, whereas the Apache data set has a 6% to 14% improvement in accurate stack traces from the Top-5 to the Top-10 stack frames. These relatively minor increases when moving from considering the Top-5 to considering the Top-10 stack frames demonstrates that developers who contribute stack traces of only up to the Top-5 stack frames in bug reports is usually sufficient to assist developers in bug fixing.

According to Schröter *et al.* [13], they examined small data set from the Eclipse projects and found 90% of bug reports are resolved for the filenames in the Top-10 stack frames. However, according to our research, the percentage of bug reports that are fixed for the Top-10 stack depth for the Eclipse data set is significantly lower (AspectJ (70.25%), Birt (48.03%), Eclipse Platform UI (63.15%), and JDT (66.13%)). In comparison, only the H-base project outperformed the 90% benchmark for the Apache data set, with a bug report

fix percentage of 92.48% for the Top-10 stack depth, while the others were less (Cassandra (82.83%), Hadoop (75%), and Spring (74.24%)).

Furthermore, for both the data sets, we experimented with using whole stack traces to determine accurate/inaccurate labels. When we compared accurate stack traces frequency to the Top-10 Stack depth, we discovered a 6-8% increase in Eclipse projects and a 2-7% increase in Apache projects.Although the increase in accuracy seems attractive, this improvement comes at higher computational costs that may outweigh the benefits.

6.2 Which exceptions should developers be most suspicious?

In research question #2, we found that the *NullPointerException (NPE)* occurred the most in inaccurate stack traces compared to other exceptions for both the Eclipse and Apache data sets as shown in Tables 5.5 and 5.6. Since *NullPointerException* is a common result of many types of faults, regardless of application domain, we do not consider this to be useful in warning developers about a potentially inaccurate stack trace. Furthermore, for AspectJ Top-10 stack depth, we used the Naive Bayes algorithm to calculate the probability of NPE for both accurate and inaccurate labels, and discovered that 33% and 11% of the NPE probability fall under accurate and inaccurate labels, respectively. These findings indicate that the likelihood of NPE occurring in an accurate stack trace is three times greater than that of the inaccurate label.

Table 6.1: The percentage of the exception types in inaccurate stack traces (Top-10 stack depth) for each project in the Eclipse data set.

AspectJ		**Birt**		**Eclipse Platform UI**		**JDT**	
Exception Type	**Percentage**	**Exception Type**	**Percentage**	**Exception Type**	**Percentage**	**Exception Type**	**Percentage**
NullPointerException	31.2	NullPointerException	34.9	NullPointerException	48.4	NullPointerException	30.2
BCException	12.8	ClassCastException	7.1	SWTException	9.2	AssertionFailedException	15.9
IllegalStateException	8.5	IllegalArgumentException	6.7	AssertionFailedError	8.7	IllegalArgumentException	13.4
ArrayIndexOutOfBoundsException	6.4	DataException	6.7	IllegalArgumentException	6.1	ArrayIndexOutOfBoundsException	5.6
VerifyError	6.4	ReportServiceException	4.2	ClassCastException	5.6	SWTException	3.9

However, when comparing the occurrence of NPE in the two data sets, we noted a difference. According to the Tables 6.1 and 6.2, NPE is at the top of the lists for the projects in the Eclipse data set but this exception falls to second or third place for the

Table 6.2: The percentage of the exception types in inaccurate stack traces(Top-10 stack depth) for each project in the Apache data set.

Cassandra		Hadoop		H-Base		Spring	
Exception Type	**Percentage**	**Exception Type**	**Percentage**	**Exception Type**	**Percentage**	**Exception Type**	**Percentage**
AssertionError	38.1	ClassNotFoundException	33.3	IOException	62.1	IllegalArgumentException	22.5
NullPointerException	27.1	AssertionFailedError	22.2	AssertionError	33.7	NullPointerException	16.2
IOException	16.1	IllegalArgumentException	22.2	NullPointerException	32.6	IllegalStateException	15.3
RuntimeException	14.8	NoSuchMethodException	22.2	IllegalArgumentException	27.4	NoSuchBeanDefinitionException	7.2
IllegalArgumentException	12.2	FileNotFoundException	11.1	ClassNotFoundException	16.8	NoSuchMethodError	5.4

projects in the Apache data sets. Looking beyond the NPE exception, we find that the next most freqently occuring exception varies between Eclipse projects. The next two most occurring exceptions are *AssertionFailedException* occurring in JDT at a rate of 15.9%, and *BCException* occurring in the AspectJ project at a rate of 12.8%. As these percentages are occur in less than 20% of inaccurate stack traces, it makes make it challenging to use the occurrence of these exceptions as warning signs of inaccurate stack traces for the Eclipse projects.

In contrast, inaccurate stack traces in the Apache data sets show that *IOException* occurs in the H-base project at a rate of 62.1%. As Hbase provides real-time read/write access to Big Data, this *IOException* occurring most of the time makes it significant because it is caused by failed or interrupted Input-Output processes.In other words, it is not surprising that *IOException* would occur commonly in stack traces for this project. The *AssertionError* in the Cassandra project occurred at a rate of 38.1%. When any of the program's assumptions are violated, an *AssertionError* is thrown. Cassandra, as we know, is a nosql database that caches and replicates data across multiple nodes within the same cluster. Typically, the Cassandra configuration includes assumptions such as disk storage and replication strategy. If the configuration does not satisfy the assumptions or if any key node runs out of cache size while operating, the system may crash, throwing an assertion error. This makes it reasonable that *AssertionError* is the most frequent in the Cassandra project.

In a batch of stack traces from the Hadoop project, *ClassNotFoundException* is found at a rate of 33.3%. Finally, we noticed that *IllegalArgumentException* occurred at a rate of 22.5 % in the Spring project. When illegal or wrong arguments are passed to a function, the

IllegalArgumentException is issued. This Java-based framework supports Dependency Injection, which allows the Spring container to inject objects into other instances. This could be one of the reasons why *IllegalArgumentException* occurs more frequently. The frequencies for these exceptions may mean that identifying exceptions occurring in inaccurate stack traces may be more useful for developers of Apache projects.

6.3 Which feature combination and machine learning algorithm should be used?

In research question #3, we experimented with different combinations of features (filename, method names, and exception type) with different machine learning algorithms to create an accurate/inaccurate stack recommender. We found a maximum increase in accuracy of 5% for the projects in the Eclipse data set and a maximum increase of 12% for the projects in the Apache data set when using only the filename as a feature versus using the filename paired with the method name. The inclusion of the method name to the features did not result in a significant increase in accuracy for either data set. We believe this is due to similar method names (viz. *init, show, run, build*) occurring in both accurate and inaccurate labels across all the classes, which did not assist machine learning models in understanding the patterns.

On the other hand, we discovered a maximum accuracy raise of 25% for the projects in the Eclipse data set and a maximum accuracy raise of 34% for the projects in the Apache data set when comparing the use of filename and exception type with using the filename alone. Compared to using the filename and exception type combination, the accuracy when using all features (filename, method name and exception type) varies by -1% to 2%. As a result, we conclude that using the filename and exception type as features is the best feature combination for creating an accurate/inaccurate stack trace recommender.

When we evaluate the performance of all three machine learning algorithms, we discover that Logistic Regression and SVM performed similarly and better than the Naive

Bayes algorithm. When the features are correlated, the Logistic Regression Algorithm separates its features linearly and employs linear classification for data analysis, while SVM uses non-linear kernels (linear, polynomial, etc.) to partially understand the features' relationship. This, we believe, is one of the reasons why the two algorithms produce similar results. In contrast, in Naive Bayes, the features are assumed to be conditionally independent, which is rarely true in real-world data sets. Therefore, classification will not occur as anticipated if any of the features are correlated. Thus, we assume that Naive Bayes considering features as independent, when in fact they are not in these data sets, is the reason for its poor performance.

We believe that training machine learning algorithms on imbalanced data is another factor that affects these algorithms performance in this domain. We can see from Tables 5.1 and 5.2 that in the Top-1 and Top-3 stack frames, inaccurate stack traces are more numerous than accurate stack traces, whereas accurate stack traces occur more in the Top-5 and Top-10 stack frames. This imbalanced data might affect the performance of machine learning algorithms.

6.4 Summary

We conclude that users merely providing the top five stack frames in bug reports can typically help developers with bug fixes. Also, information about the frequency of exceptions occurring in the Apache projects will likely be more beneficial for developers to identify the inaccurate stack traces than in the Eclipse projects. Finally, the Logistic Regression and the *Filename-Exception type* were found to be the best for producing an accurate/inaccurate stack trace recommender.

Chapter 7

Conclusion

In bug reports, providing stack traces can be helpful since they assist developers in debugging. However, certain stack traces may be inaccurate, leading to longer debugging times and increases to the project cost due to longer bug fixing times. This work investigates the prevalence of inaccurate stack traces, identifies exceptions commonly occurring in inaccurate stack traces and explores the use of machine learning to create recommenders for identifying inaccurate stack traces.

First, we classified stack traces as "Accurate" or "Inaccurate" by comparing the file names in the commit history for a particular bug report with the filenames in the provided stack traces. We observed that inaccurate stack traces occurred more often if only the Top-1 stack frame is considered and fell as more stack frames are considered. We found that in most cases, only considering the Top-5 stack frames is sufficient. We also found that the Eclipse projects in our data set have a higher frequency of inaccurate stack traces than those of the projects in our Apache data set.

Second, we calculated the most common exception types occurred in inaccurate stack traces up to the Top-10 stack frames in both the data sets. We found that *NullPointerException* is the most common exception. However, after the NPE, the most frequently occurring exception become more project-dependent with the next most commonly occurring exception being *IllegalArgumentException*.

Finally, for creating an accurate/inaccurate stack trace recommender, we explored training three machine learning algorithms (Naive Bayes, Logistic Regression and Support Vec-

tor Machines) with different combinations of features from stack traces. We observed that the Logistic Regression performed better than the other two machine learning algorithms for all the stack depths, although SVM had similar performance. In contrast, the Naive Bayes algorithm performed poorly on these data sets. Also, we found that the best feature combination for training an accurate/inaccurate stack trace recommender is *Filename-Exception Type*.

7.1 Limitations

We identified the following limitations to our approach:

1. Our approach is only applicable to programming languages that have built-in support for retrieving or generating a stack trace when a program crashes. Therefore, our approach may not be applicable for languages, which don't produce a stack trace when the program crashes.

2. As a software project evolves, the accurate/inaccurate recommender must be updated with the changed code or user-specific exceptions. Specifically, the recommender must be trained on newly obtained stack traces from bug reports in order to be reliable.

7.2 Future Work

The results we obtained provide empirical evidence of the inaccurate stack traces occurrences in bug reports. We identified some more future improvements based on the results of this work as follows:

1. Expanding our investigation by studying additional projects with more bug reports that include stack traces.

2. Increasing the potentiality of our investigation by training recommender on balanced data.

3. Investigating how deep learning networks can be used to create inaccurate stack trace recommenders instead of machine learning algorithms.

4. Creating a tool which can be integrated with bug repositories to warn developers by showing the stack trace inaccuracy level based on either stack trace or the frequent exception types.

Bibliography

[1] A. Bergel and I. Slater Muñoz. Automated test generation for stack-trace reproduction using genetic algorithms. *In Proceedings of the 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021.

[2] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? *In Proceedings of FSE. ACM, 2008*, page 308–318, 2008.

[3] B. Boehm and V. Basili. Software defect reduction top 10 list. *Computer*, pages 135–137, 2001.

[4] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng. Assessing the quality of the steps to reproduce in bug reports. *In Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19),*, 2019.

[5] R.Marc D. Steven. What's in a bug report? *In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2014.

[6] M.J. Harrold H. Shah, C. Görg. Why do developers neglect exception handling? *In Proceedings of the 4th International Workshop on Exception Handling*, pages 62–68, 2008.

[7] M. Hamdaqa A. Hamou-Lhadj K. Sabor. Automatic prediction of the severity of bugs using stack traces and categorical features. *Published by Elsevier B.V.*, 2019.

[8] E. Middleton. Npe in workbenchpage. *https://bugs.eclipse.org/bugs/show_bug.cgi ?id=397872*, 01 2013.

[9] S. K. Ramalingam. Large_message connection allocates heap buffer when bufferpool exhausted. *https://issues.apache.org/jira/browse/CASSANDRA-15358*, 10 2019.

[10] S. Russell and Peter. N. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[11] F. Khomh S. Wang and Y. Zou. Improving bug management using correlations in crash reports. *In Proceedings of the Empirical Software Engineering*, pages 337–367, 2016.

[12] K. Sabor, A. Hamou-Lhadj, A. Trabelsi, and J. Hassine. Predicting bug report fields using stack traces and categorical attributes. *In Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, 2019.

[13] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? *In Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

[14] J. Sebes V. Paila. Automating crash report analysis using exception-based patterns and reference assembly mapping. *In Proceedings of the 8th India Software Engineering Conference*, pages 70–79, 2015.

[15] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 689–699, 2014.