ZITA - A Self Learning Tutoring Assistant

Master thesis
Computer Science, Software Technology specialization

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Formal Methods and Tools research group

Supervisors
dr. A. Fehnker
dr. D. Bucur

Abstract

Static analysis tools are often used to quickly and easily verify code for errors or style issues. These tools are used written with a general working environment in mind, so specific types of errors may be left out. Programming education would be an environment which does have specific error types, as each individual programming course has different demands from their students.

Static analysis tools can be used in education, to speed up grading, or to get a general overview of student performance. To include all course-specific errors, however, custom rules must be made to find these errors. This research aims to use machine learning to automatically link code patterns to specific errors, specifically on errors made in the Processing (Java variant) programming language. The biggest problem is how to learn a program in such a way that the internal structure of a program is not lost.

This is achieved by first transforming a program to its corresponding Abstract Syntax Tree (AST). Each sub-tree of this AST is individually analyzed and given a comment about its status: unknown, correct, or faulty code. 7 features are used as input data for machine learning classifiers. A static analysis tool is used to initially identify faulty code, in order to see how well the classifiers are able to learn the rules of such a tool.

In total 287 student-written programs are used to test the performance of two machine learning classifiers: Naïve Bayes and Decision Trees (C4.5). 10-fold cross validation is used to reduce possible noise.

Decision Trees perform the best, in addition to granting the ability to easily look at the reasoning behind determining the class of a program. The overall true positive (TP) rate is 60%, but this includes errors that occur often and cannot easily be found using the chosen features, such as undesirable variable names. Structural errors are able to be found with a precision and recall of over 70%.

Lastly, the practical usability of using machine learning for static analysis is discussed, considering the different types of errors that are able to be reliably caught.

Contents

Chapter 1

Introduction

Static analysis tools are widely used for checking if code is correct. They take a piece of code, look at it line by line, and give a message is something seems to be wrong. These tools can prove very useful, as they can provide valuable feedback without developers themselves having to dedicate time to re-reading their code. These tools are used in nearly all (enterprise) developer environments because of this.

However, these tools are not perfect. For the most part, static analysis have so called rules. When a rule is broken, this means that something in the program is most likely incorrect. Rules are usually hard-coded into a tool: this means that a tool will always give the same output, regardless of the context. But what if there are some rules that are not useful in certain scenarios? What if there is some other common occurring problem that is not caught by any of the rules? It is certainly possible to disable/create new rules, but this takes time. What if this process of the selection of rules automated?

Using automatic rule generation (or automated feedback), teams could have their own personal feedback. If some problem is consistently ignored, this could mean the problem is not a problem after all. Then the tool would also stop giving feedback on that problem.

Especially in education is where automated feedback may prove useful. Professional tools are aimed with experienced code and developers in mind. This means that the feedback given by these tools is not always understandable by novice programmers, devaluing the worth of the tool. This is further discussed in a previous research[10].If some kind of mistake has happened often in the past, and a teacher has annotated this often in some kind of feedback tool, it would help future tutors if these annotations are automatically generated.

However, such a tool does not yet exist. We therefore propose to create Zita, a tool that automatically takes feedback from teachers and existing static analysis tools to generate its own feedback. The feedback will be rated (either positively or negatively), so that Zita can learn from itself.

In this paper, the main focus will be on tutors in programming related courses. Even though automated feedback would be helpful to both tutors and students, a first step for Zita would be to indicate that something is wrong at some part of the code, without actual indication of what exactly is wrong. For students this is not very helpful, as they may not immediately know what is wrong with that piece of code. Tutors, on the other hand, have enough programming experience to know the problem of a piece of code, given knowledge that at least something is wrong.

1.1 Terms

In this proposal, a number of terms and abbreviations will be used. These will be shortly mentioned here.

Tutor
A tutor may be anyone who is in a teaching position inside some programming related course. This can be someone who actually gives lectures (such as professors), or higher year students that help with teaching during the course (teaching assistants).

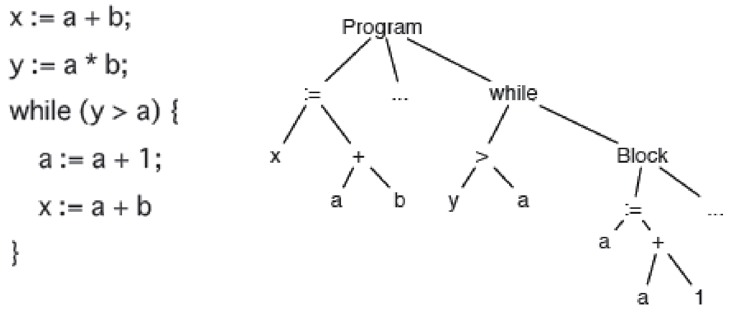
Static (code) analysis
Static analysis is analysis done on code by a tool, without having to run this code. An example of the result of such analysis, is that a piece of code is annotated with 'always true', indicating that a boolean's value is always true.

IDE
An IDE (Integrated Development Environment) is an application that helps a developer to program more efficiently. It accomplishes this by helping the programmer during the writing and testing of code, for example, by using tools that can automatically start your project with one button, or by implementing auto-complete which speeds up the raw typing of code.

Plugin
A plugin is a tool that is an extension to an IDE, which provides additional functionality. The goal of these plugins are similar to what the IDE itself wants to accomplish; making the programmer's life easier. Plugins are often made by third parties, therefore a lot of plugins are available for use.

Feedback
The term "feedback" in this research correspond to comments or annotations about code, given either by a teacher, professor, or teaching assistant. These comments are meant to be read by student who wrote the code, in order to know what they did wrong in some part of the code.

AST
An abstract syntax tree (AST) is a tree containing the data of a piece of code that has been parsed. It contains the structure of a program. An example is



```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

Figure 1.1: An example AST

given in Figure 1.1.

Each node of an AST contains information about what type of node it is. These types can range from simple names of variables, to more complicated ones such as a while statement. The latter has child nodes, in which effectively more information about the parent is stored.

Data point/data set
A *data point* refers to a single collection of information which describes a piece of a larger *data set*. For example, when measuring the average temperature per day over the course of two years: a *data point* would refer to a temperature on a given day. The *data set* would describe all the temperatures over the two years; a collection of all data points.

Available data throughout this research has the same structure; a list of programs, each program being its own self-contained project. This program is converted to an AST, and every sub-tree of this AST is either correct or incorrect. Therefore each sub-tree of an AST is considered a data point.

Classification
This research relies on the concept of classification: a data point should be categorized in a certain class. The process of figuring out the appropriate class is called *classification*. Machine learning algorithms that attempt to match a class to a data point are *classifiers*.

1.2 Example Interactions

To illustrate what role Zita can have in the context of education, this subsections will give a few examples of code found in educational contexts and how Zita can help tutors find faulty code. The first example is a piece of code which contains a bug that is hard to spot. The second example is a fault which is problematic in a specific course's context.

Correlation that is difficult to notice

```java
public class FooBar {
   private static int testNumber = 0;

   public void foo() {
      // code
      testNumber = 10;
      bar();
      // code
   }

   // more functions

   public void bar() {
      int base = 10;
      int result;
      if (testNumber > base) {
         result = base * testNumber;
      } else if (testNumber <= base) {
         result = base - testNumber;
      }
      return result;
   }
}
```

Listing 1.2: Example student code

This piece of code has the same behaviour as Listing 4.8, but has one extra problem: the static class variable *testNumber* is being used as an argument, without actually being an argument.

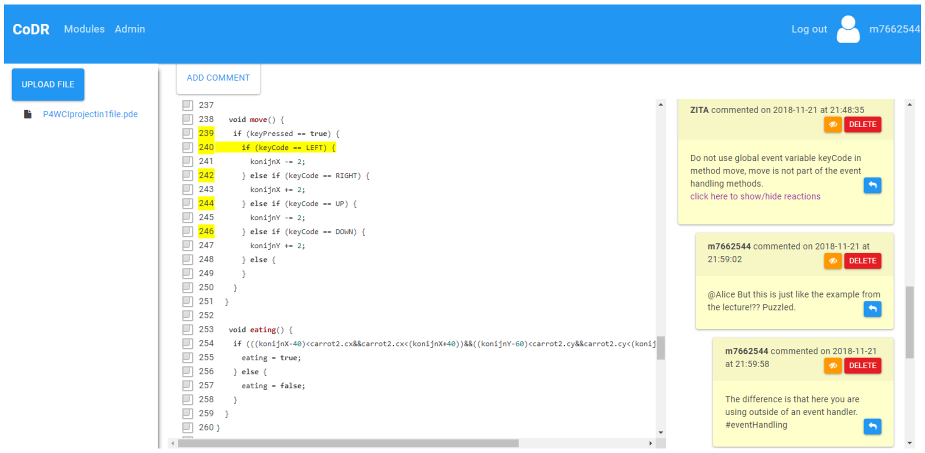
In the code above one can deduce that *testNumber* is being used incorrectly. However, if the code becomes larger (i.e, *foo*() and *bar*() are separated with code in between), this incorrectness will be much harder to spot.

Course-specific error

```java
public class CircleDraw {

   public void draw() {
      // some drawing code
   }

   public void keyPressed() {
      ellipse(56, 46, 55, 55);
   }
```



```
void move() {
  if (keyPressed == true) {
    if (keyCode == LEFT) {
      konijnX -= 2;
    } else if (keyCode == RIGHT) {
      konijnX += 2;
    } else if (keyCode == UP) {
      konijnY -= 2;
    } else if (keyCode == DOWN) {
      konijnY += 2;
    } else {
    }
  }
}

void eating() {
  if (((konijnX-40)<carrot2.cx&&carrot2.cx<(konijnX+40))&&((konijnY-60)<carrot2.cy&&carrot2.cy<(konij
    eating = true;
  } else {
    eating = false;
  }
}
}
```

ZITA commented on 2018-11-21 at 21:48:35

Do not use global event variable keyCode in method move, move is not part of the event handling methods.
click here to show/hide reactions

m7662544 commented on 2018-11-21 at 21:59:02

@Alice But this is just like the example from the lecture!?? Puzzled.

m7662544 commented on 2018-11-21 at 21:59:58

The difference is that here you are using outside of an event handler. #eventHandling

Figure 1.4: Example interaction in CoDR

```
}
```

Listing 1.3: Example student code

The above may seem fine, knowing that keyPressed is a standard *Processing* function that gets called once a key on the keyboard is pressed. However, it is possible that a course teaches students to keep their drawing related methods contained in the draw method. A reason for this would be that it is good practice to keep drawing methods in one place to avoid having drawing methods all over a program, making it difficult to maintain the program later on.

Normal static analysis tools would not look for this, as it is unlikely that a tool has rules specifically made for just one course. Additionally, newer tutors who have less experience with tutoring a course could look over this, as they are not aware of course-specific issues.

With Zita, knowledge of all previous tutors is used to correct a program, so issues like these can be found and the knowledge of them immediately given on to newer tutors.

An example user interface can be seen in Figure 1.4. There are four main components: the possibly faulty code, comments about that code, hiding of comments, and line number that can be clicked in order to create new comments about the code.

From the comments and the block it corresponds to, a tutor can quickly see code that could be problematic. Assuming that the automatically generated comments from Zita are mostly on point, tutors can shift from having to look at all the code equally, and instead focus on smaller blocks of code.

The (un)hiding of comments will help Zita know what comments are good. Zita's comments would start out as hidden, and have to be unhidden (approved) by a tutor before they are shown. When they are unhidden, Zita also gets feedback that indicates that the comment was indeed a correct comment.

Chapter 2

Research Methods and Previous Work

2.1 Related Work

In this section we look over previous work that has been done which can be used for this research. There are three main themes that form the research: education, static analysis of code, and applying machine learning on trees. First, we purely look at automation in programming education, either for benefit of the tutor or for the student. Then we focus on static analysis and how it has been used in an educational environment. Lastly we look at machine learning techniques that can be used on tree structures.

Test-based code verification

Test-based code verification is a simple way to automatically judge the correctness of a student's program. Students write their solution for an assignment, run a test that was made by a tutor beforehand, and see how many of the test cases succeed.

Douce [9] reviews the history of using tests to automatically assess programs. This goes back as early is 1960, by using punched cards and checking if the value from the student's program matches the expected value. There have been three main generations of testing systems: early systems (checking program output against expected output), tool-oriented systems, and web-oriented systems. The last one is user (student) friendly, as they are able to quickly test something without having to run a tool locally.

Autolab [1] is a modern open source tool that automatically grades assignments using tests. It supports many different languages, and includes a scoreboard to add competitiveness into assignments. A perfect score means that all tests pass, with the score getting lower if errors are found.

The disadvantage of using purely test-based feedback, is that the quality of the code is completely ignored. As long as the program passes the tests, it is seen as correct. If a program should contain a simple factorial function, this could be achieved by having a switch statement for numbers 0-99. If the test cases do not go higher than 99, such a program would be seen as correct. For this reason, test-based feedback is less interesting than actually looking at the actual contents of a program.

Adaptive tutoring

Adaptive tutoring is tutoring that changes based off the student's knowledge.

CIMEL ITS [15] has a tool that uses adaptive tutoring for programming concepts. The idea behind this makes a lot of sense: different students require different levels of feedback. A student who has a good understanding of programming will get more use out of advanced feedback, while a student who has trouble understanding the basics requires feedback which focuses on general programming concepts.

This is achieved by storing information about a student: which assignments they have completed, and what their performance was on these assignments. Based off of that information, some conclusions can be drawn about the student's ability in different programming concepts. Then whenever a student makes an error, CIMEL ITS looks at the student's performance in related topics, and automatically gives feedback accordingly.

If a student has proven to be capable, a simple reminder can be given, as the student could have made a typo in their program. On the other hand, if the student has had similar errors previously, more in-depth feedback would be more appropriate, to try to get the student to understand the concept behind the error.

While the concept is interesting, it is out of the scope of this research. The focus in this research is on the catching of errors, not on the feedback. Automatically creating feedback for students would require a lot of extra research to be done.

Feedback assistants

Currently students do not have a convenient way to ask for feedback on their code: the most convenient way is to ask during hours where the students are actually working on the project, but it is difficult for a teacher to actually read the code in-depth enough to see if something may be wrong. Feedback assistants can smoothe the process of feedback, by automating a part of the feedback process.

CoDR is an example of a feedback assistant. It is implemented by having a web-based platform where students can upload code. Teachers can then review this code, and give comments on this code. Comments are able to be hidden by a teacher, and are linked to a program via line numbers. Students are able to see what was commented on their code, in order to improve their coding quality.

Zita takes CoDR as an example of how tutors and students would interact with a web-based environment to assess code.

Another example of feedback assistance is CodeGrade [3]. CodeGrade is similar to CoDR, in the way that a tutor can give feedback in a web appplication. It also provides useful functionality, such as plagiarism detection, testing utilities, and integration with Learning Management Systems such as Canvas and Blackboard.

These feedback assistants, and Autolab as well, have the ability to integrate already existing static analysis tools into their systems. This results in being able to add custom error checkers (or known tools that are known to work well) to these systems. This will yield the tutors more information about the students' code, making tutors able to give more feedback.

Static analysis in education

As mentioned earlier, static analysis tools can immensely help both students and tutors while working in an educational environment. Students can get instant feedback on code that is written, and tutors are able to quickly see problematic code.

An example of a static analysis tool is CheckStyle [2]. CheckStyle looks through source code to spot style issues with coding. This includes issues like improper naming, incorrectly indented code blocks, and missing spaces around statments or expressions. Including this tool for students enforces them to learn proper coding style guidelines, making their code easier to read in the future.

PMD [6] is an open source static analysis tool for a variety of programming languages, such as Java, JavaScript, and PLSQL. Every error-catching code (also called a 'rule') is written manually, and grouped into bigger sets called 'rule sets'. For example, a rule 'SingleCharacterVariableName' could be a rule which checks if a variable consists of a single character. This rule would be part of a bigger rule set called 'Naming'.

There exist plugins for both PMD and CheckStyle for Eclipse, which integrates the utility of these tools into Eclipse by adding markers to the left hand side of code. These markers will display a warning that the current line has broken a rule. Custom rules can easily be integrated in the PMD plugin, as PMD itself provides context of a program in the form of an AST.

In addition to PMD, a tool part of a research presented at CSEDU 2018 [8] is a static analysis tool that was used for research on *Processing* code. This tool currently has no real name, so it will henceforth be referred to as 'CSEDU'.

CSEDU is an extension of PMD with rules specifically made for Processing. These rules were also made with programming education in mind, and use PMD's AST traversal functionality to detect faulty code. This is perfect to compare Zita with: the difference between CSEDU and Zita is only that Zita utilizes ML. Therefore differences in performance can be largely attributed to a ML model's performance.

Extracting features from trees

Phan [12] [13] uses trees and ML to classify programs. Different ML algorithms were checked on 52000 programs written in C. The best results were acquired by using a tree-based convolutional neural network (TBCNN) together with either a k-Nearest Neighbors (kNN) or support vector machines (SVM). Not only the ML algorithms are important, but also the way to interpret ASTs. This has also been accomplished in three different ways.

- The first way is by taking the Levenshtein Distance (LD) between two programs. The LD is the amount of changes required to get to one representation of a program to another by looking at the minimum required additions, deletions, and replacements.
- Similarly, a tree-edit distance (TED) is utilized to calculate the difference between ASTs. This is principally the same as LD, but the minimum three required elements to get from one AST to another are slightly different. An addition is whenever a sub-tree has to be added to an AST. A deletion is similar, but deleting a sub-tree. Replacement is whenever a sub-tree is relabeled.
- Lastly, for the TBCNN, an AST is converted to a vector representation, where similar nodes (such as 'If' and 'While') are mapped to the same identifier. This vector is then converted to a real-value vector.

Wang [14] looks at Java ASTs in particular, which is also what we are interested in. It utilizes deep learning, which can be a powerful tool. The downside, however, is that it is hard to see what is happening 'inside'. To preserve the code semantics, a Deep Belief Network (DBN) is used to automatically learn features from token vectors extracted from the programs' ASTs.

Additionally, sub-tree matching can be used to approximate the similarity between two pieces of code. Similar to Phan's technique, Akutsu [7] utilizes the tree's edit distance. They provide proof for algorithms for both exact and approximate tree edit distance calculations.

2.2 Research Goal

The previous work is missing something: the combination of all three elements (education, machine learning, static analysis). The purpose of this research is to fill this gap, by creating a tool that is able to automatically detect when something is wrong with a piece of code written by a student. This tool should be able to learn from code that has already been annotated with feedback, and apply that knowledge on new unclassified code. To get to this goal, multiple smaller steps have to be achieved first. A general architecture can be found in Figure 2.1.

CoDR will be used as the 'hosting' system of files. As explained earlier, CoDR is a platform that can be used to upload *Processing* code, create comments in the code, and possibly discuss them. In addition, CoDR has a function

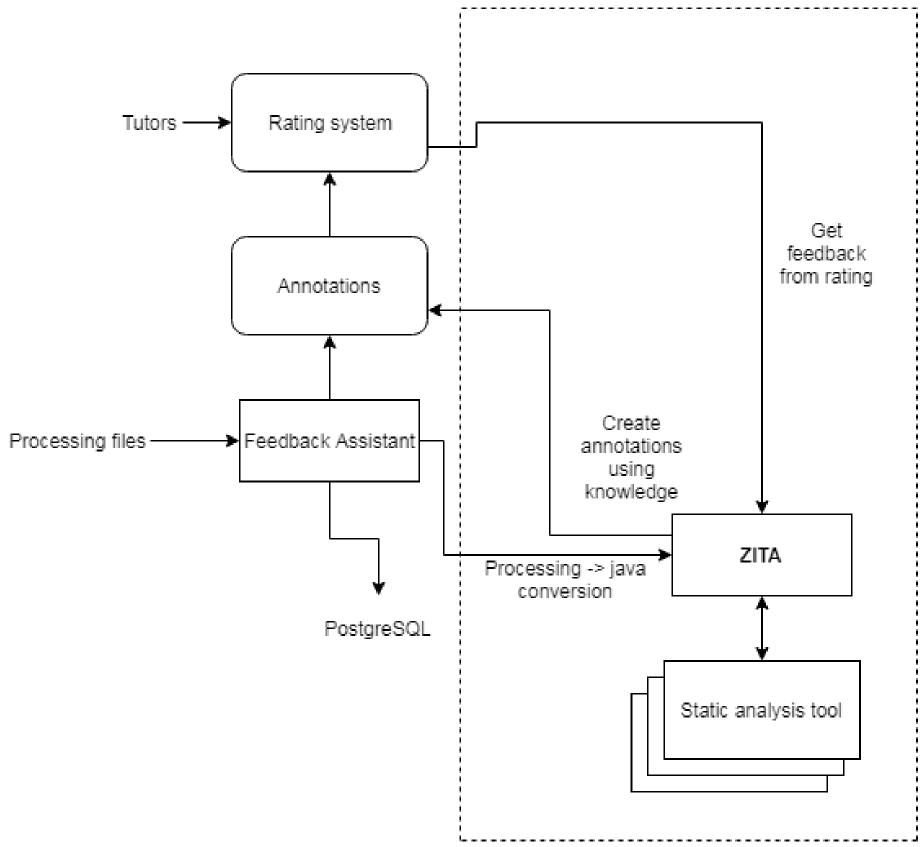


Figure 2.1: General architecture, dotted line representing the focus of this research

to make comments hidden, which can be perfectly used by machine learning to know what is useful and what is not.

The choice has been made to purely use *Processing*-based Java code for now, since there are some tools (mainly CSEDU) that have been used for research on this kind of code. *Processing* files when converted to Java are contained in one file. This means that this will generate one AST for one project.

First a student will upload their *Processing* code to CoDR. Whenever Zita should give feedback (either by request, or after a set amount of time), a feedback iteration starts. The uploaded *Processing* code is converted to a single Java file, as many static analysis tools have been made for Java.

Then the AST with annotations has to be generated from this Java code. Using a tool for code feedback, the feedback can be linked to line numbers, which can then be transfered to nodes in an AST. This will create an annotated abstract syntax tree, which will be the used as input for machine learning.

The combination of the feedback given by teachers on CoDR and the feedback given by tools will form the base of the knowledge of Zita.

Afterwards, the AST has to be transformed to be readable by a machine, without losing its tree-structure properties. For example, an AST's contents could simply be extracted, and concatenated into one big string. This removes the entire point of using a tree, however. The properties of the trees should allow the learner to focus more on structure rather than the contents of a piece of code.

With all the information available about a program contained in an AST, Zita should be able to learn what good feedback would be. At this point the machine learning part comes in: given all the knowledge that is known, what would be the feedback that would be most likely to be given on this AST? Or, if applicable, what current feedback on CoDR might actually be incorrect?

With all the information available about a program contained in an AST, Zita should be able to learn what the reason was that a part of the AST was classified as either correct or incorrect. When given a new unknown program, Zita should be able to know if some part of the program is faulty. At this point the machine learning part comes in: given all the knowledge about previous programs, what part of a new programs are possibly faulty?

The answers to this question should then be returned in the form of comments in CoDR. These can be reviewed by both teachers and students, the former by using the 'hide comment' functionality, and the latter by having some kind of rating system on the comments. This creates a feedback loop: if Zita creates a comment that is not hidden and positively rated, it should know that such a comment is correct and helpful. If a comment is hidden, it has a good chance that the comment is not very useful or even wrong.

2.3 Research Questions and Methods

2.3.1 Is it possible to learn from ASTs without losing their AST-specific properties and apply this in a tutoring environment?

First and foremost, the main research question is to see if it is at all possible to use ML on ASTs, and if so, how well they perform. For example, the expression 'someBool == true' gives a warning when running most tools, as the '== true' part is unnecessary. However, let's say that this expression is not a problem when inside a while statement, but is a problem when inside an if statement.

In this scenario, just checking the expression would not be enough: the context would also have to be taken into account. In the context of ML, the following should hold: given input that e.g. 200 tutors have classified `'someBool == true'` as a problem when inside an if statement, Zita should then mark following occurrences of that same context as problematic.

This will tested by using an artificial benchmark, by manually giving the tool instances where the above problem is marked as problematic as part of a test set. Afterwards, unfamiliar instances will be given with the tool, again with the same error. The unfamiliar instance should not only have the problematic part in it, but also some code that looks like it. If the tool manages to separate the problematic and non-problematic parts, it can be safely said that the AST's properties have not been lost.

Additionally, Zita will be applied to real-life data set, to see how it performs. A normal static analysis tool will be applied to the same real-life data, which will form the base learning data. Then both Zita and the same static analysis tool will be ran through a separate data set, to see how much of the tool has been learned by Zita. Moreover, outputs of machine learning algorithms will be looked at to understand how Zita behaves.

A few sub-questions are brought up in order to help answer the main question.

Sub-question 1: Which AST-specific properties should be preserved?

In order to learn ASTs using ML, the ASTs first have to be represented as a list of attributes (features). There are an infinite amount of features that are available, therefore it is important to choose features that are generic, yet distinctive enough to uniquely classify ASTs into predefined categories.

Sub-question 2: What kind of problematic code is able to be reliably found?

Using the features chosen, what kind of errors are likely to be caught? Since there's many type of errors that can be found, not all of them will be able to be equally likely to be found. For example, when the name of variables is completely ignored, it makes sense that errors that are related to variable names are not often found.

The importance and reasons why some kind of errors are either found or not (reliably) found are discussed after the results of sub-questions 1 are known and applied in ML algorithms.

Sub-question 3: What is the advantage of using machine learning over normal static analysis in a tutoring environment?
The final question is in regards to actual usability of ML over static analysis. Some of the issues tackled in this research also able to be resolved by statically checking, so why would one use ML instead of static checkers? This question is answered at the very end of the research.

Chapter 3

Design and Implementation

This chapter discusses the overall design and implementation of Zita and its components. Firstly the AST's design will be looked at, after which the broader class design is discussed. Following that, the implementation details of the design are explained.

3.1 AST and EXAST

In order to link the AST with data from CoDR, an extended AST (EXAST) is created. This EXAST also includes the information about the comments, and the how the comments are linked to the AST.

```java
public class FooBar {
   public int bar(int testNumber) {
      int base = 10;
      int result;
      if (testNumber > base) {
         result = base * testNumber;
      } else if (testNumber <= base) {
         // the above "else if (..)" could be
         // replaced by just an "else".
         result = base - testNumber;
      }
      return result;
   }
}
```

Listing 3.1: Example Java code for an AST

First, how does the AST from Figure 3.2 get created from Listing 4.8? This is done by using a parser (JavaParser in this research), which reads the program and parses it as Java. A Java program normally starts with a package definition,

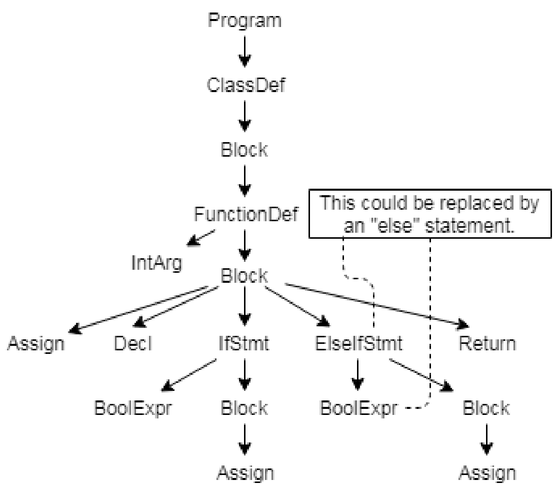


Figure 3.2: Simplified EXAST for the Java code snippet

followed by zero or more ‘import’ statements, followed by a class definition. The exact Java specification can be found in the official Java specification [5]. To keep the AST from being too large, the Java code immediately starts out with a program definition, and code comments are ignored. Additionally, the node names are simplified to again reduce the size of the tree.

While parsing the program according to the Java specification, the AST is built. A new node is created whenever a new part is parsed, which can again contain new nodes. For example, the ”FunctionDef” block corresponds to lines 2 until 13. It is divided into 3 parts: 1 top level node ”public void bar”. This node will then get 2 children: one for the argument ”(int testNumber)” and one for the block between braces. The argument’s node will have no children (also called a ‘leaf’), while the block’s node will contain multiple children. These children correspond to each of the statements in the program.

Initially, the comments and ASTs are completely separated. This is to ensure that the logic of creating comments and ASTs are not overlapping, allowing for easier switching of strategies to actually create the comments and ASTs. If this is ensured, Zita remains extensible in the future, as e.g. a different parser for a different language can replace the current Java parser, without the comment logic having to know this.

The UML diagram for the class structure of ASTs can be found in Figure 3.3.

A node is a single node in an AST. It has references to their children and a parent, each also being a node itself. It is possible for either to not exist, in which case the children will be an empty list, and parent non-existent (*null*, in the case of Java). Attributes is a simple key-value pair, which is used to store the features of a Node. These features correspond to either a string for nominal values, or an integer for numeric values. The content is what the original code

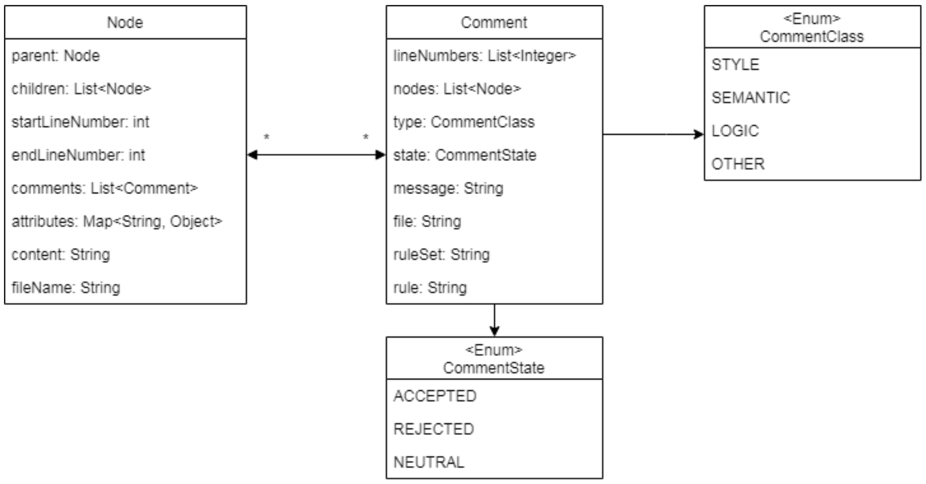


Figure 3.3: Class diagram for the classes relevant for an AST

was that corresponds to the Node and all of its children. This is mostly used for easily knowing what the original code was, without having to decipher the whole tree manually.

The file name and line numbers correspond to the original program. The comments have the same properties, and are used to link a node to a comment. One comment can link to multiple nodes, and one node can link to multiple comments.

A comment is a bit more static compared to a node, as it 'lives' independently from everything else. Aside from the properties mentioned earlier (line numbers and file name), a comment has a little more information which is used to communicate more specific information to whoever is interested.

The message is a simple string: it should describe the error that this comment is linked to. If the comment was originally made by a tutor, this would be the message that the tutor would like to communicate to the student. If the comment originates from a static analysis tool, this would be whatever the tool outputs in case it finds an error.

The rule set is mainly used to store where the comment originated from. If it is tutor-made, this would be "tutor", else the specific tool and rule set combination. The exact rule is only used for tools, to indicate which specific rule found the code problematic.

Lastly, there are two enums to indicate the type and state of a comment. The state is linked to CoDR: this is initially neutral. This is set at the beginning, and at this point the comment should only be visible for tutors. If it has been rejected, neither the tutors nor the students will see it. If it has been accepted, it means the error was successfully found, so it will be shown to both the students and the tutors.

Once all comments of a program are out of their neutral state, the entire

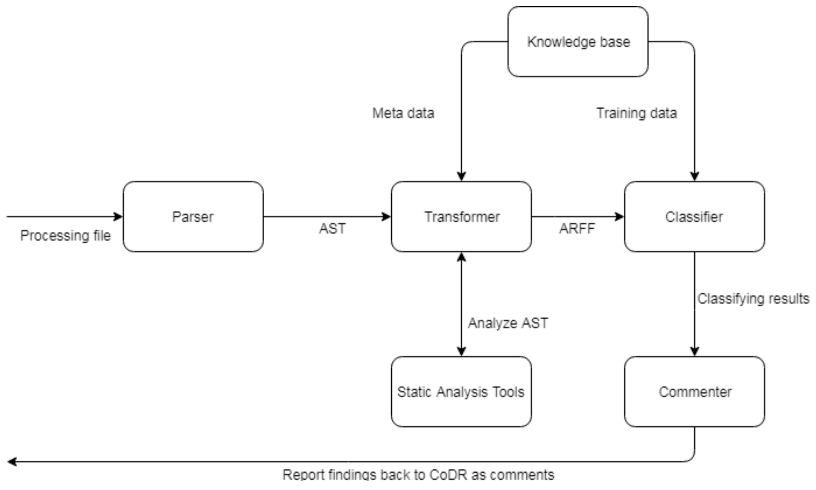


Figure 3.4: Core Zita design

file can be given back to Zita. This will then be part of the training data, with only the accepted comments remaining. This will complete the feedback loop, increasing the knowledge base of Zita.

3.2 Zita Design

This section will take a look at how Zita is structured. Zita is not just a single entity, it consists of multiple elements, as seen in Figure 3.4). The overall picture is a pipeline with a feedback loop inside it, visible in Figure 2.1.

Parsing To start off, the process starts with information from CoDR: The *Processing* code. This is first converted to a Java file, and then parsed to create an AST. This could in theory be replaced by a completely different language (such as C, or even English), but would require the entire parser to be wrapped in an interface to ensure that the same methods that are required are preserved. For the parsing itself, JavaParser [4] is used. This is an easy to use Java library that takes a file and transforms it to its own type of AST (CompilationUnit). This CompilationUnit contains what is needed: contents, line number, and type (variable, method name, etc...), among other data.

Transforming Given the AST, the information can be fed into a classifier. However, before that, the data has to be transformed to the ARFF format that Weka expects. This is done by transforming the AST into features. These features will differ based on the classifier used. For example, for Naive Bayes, the text itself could be only feature. On the other hand, for something such as decision trees, numbered vectors could be used. In this context, this would mean tokenizing the AST (giving each distinct word a number), and then feeding that into the classifier. Refer to Section 4.4 for the implementation of this.

If the AST is not supposed to be classified, but rather used as training data, this is also done via the ARFF format. Using the data from CoDR and static analysis tools, parts of the program are classified 'incorrect' (commented), while the rest is assumed to be 'correct'.

Classifying The Classifier is what will ultimately decide whether or not a piece of code is correct or incorrect. The ARFF files previously used are either used for training of the classifier (by using indicating which part of the code corresponds to which class) or for classification. The classifier itself should easily be able to be changed, to ensure researching the use different classifiers as easy as possible.

Commenter Once an AST has been successfully classified, some sub-trees of the AST may be incorrect, while others are correct. For each block of incorrectness, a comment should be made in CoDR. The Commenter will ensure the original line numbers will be matched with the incorrect AST sub-trees, and combine them together to create a comment in CoDR. In the future the comment may be filled with some information (such as reliability, fault class type, origin of fault, and more), but for now it is simply empty. This also ensures that anyone who sees this comment knows that this comment originated from Zita and therefore may not be entirely accurate.

3.3 Implementation

The next half of this section goes into detail of the implementation of the design. First the language is chosen, after which the pipeline is described in more detail.

Language

Zita is written in Java, an object oriented imperative programming language. It is easy to write and maintain, and is able to handle all data *Processing* that is required. A big advantage of using Java, is that *Processing* uses Java under the hood. Of course only ASTs are used to represent *Processing* programs, but these have to be converted. For this, JavaParser is used, which is a Java parser made in and for Java, including the generation of ASTs. This means that the *Processing* (Java) code can easily be converted into ASTs using JavaParser.

Processing to Java conversion

As the programs are parsed as normal Java, the *Processing* file used as input has to be converted to something parsable as Java. This is relatively easy to do, as the Java version of *Processing* is the same as Java, but with some additional standard functions.

When *Processing* files are exported to a single .pde file, every class is put into one file, ready to be put inside a wrapper class. *Processing* itself uses this to execute the program, but it can easily be used to create a dummy wrapper class using the following structure:

```
 1  + public class Processing {
 2     float currentPosition;
 3
 4     void setup() {
 5        size(400, 600);
 6        currentPosition = 0;
 7     }
 8
 9     void draw() {
10        // draw code...
11     }
12
13     class Circle {
14        // object code...
15     }
16
17  + }
```

Listing 3.5: Example conversion

Only a top-level class and matching bracket have to be added to make it a parsable Java file, as can be seen in lines 1 and 17 in Listing 3.5.

AST and EXAST As said before, Javaparser is a Java library used to parse Java files. The result of this is an AST, in the form of nodes that have zero or more nodes themselves. For the EXAST, a more generic approach is taken. The reason for this, is so that Zita may not only have Java files as input, but any kind of AST as input. When the language is abstracted away, one could insert any language as input: C, Python, or even natural language (to an extend).

This abstraction is achieved by creating an interface for the AST. This interface exposes the necessary getters/setters for common properties (child nodes, line numbers, etc...), and some necessary methods to be able to calculate an AST's features.

3.3.1 Backtracing

Once a file has been converted into a set of data points with the features fully calculated, it is able to be classified using Weka. Weka returns a value which corresponds to the class the classifier has classified the data point as. So given a data point, the class is then known. To give this information back to a student or tutor, it should be linked back to the original program, including the correct line number. This takes some effort, as a lot of information is lost during the transformation to an ARFF data point.

During the transformation process, the order of the data points is stored. If Weka returns that some data point n is classified as an error, this can be immediately be linked to the original n'th node that was transformed. Since the node saved all necessary information, a comment can simply be added. The rule set is set to 'Zita', rule to whatever the classification is, and the default state is set. This process is repeated as often as Weka returns a classification that is not 'correct'.

Also to be considered is the changing of *Processing* to Java, if converted using *Processing*'s own converter, as opposed to the method above where two lines can be added to create parsable Java file. When *Processing* is converted to Java using this method, some setting-related code is relocated (such as the functions 'size') , leaving just an empty line. In the testing set none of these lines contained an error, therefore this behaviour is ignored when looking back for line numbers.

Additionally, *Processing* adds some lines of code at the very beginning and end of a program. These lines are always the same: 12 imports and adding a class definition which extends PApplet, with some new lines in between for readability, totaling 16 added lines. To compensate for this, a simple offset can be used. An error in line 36 in Java code would correspond to line 20 in the same *Processing* code.

Chapter 4

ML Methods

This chapter will explain the different ML methods and techniques used to later classify data. First a list of features is looked at, followed by an explanation how these features and calculated and how Weka is used to do experiments with these features. The classification techniques are handled next, with each an example on how they would behave given a data point and previous knowledge. Lastly, the methods for training and verification are discussed.

4.1 Feature Selection

Before looking at classification techniques, first different features have to be selected, which will function as input for the classification.

For our problem, features will have to be extracted from the AST, keeping in mind that each sub-tree of the AST is its own data point. The root of every sub-tree will henceforth be referred to as 'the node'.

To give an example of how a sub-tree's features would be calculated, two examples are given using a program that has been given as an example earlier (Figure 4.2. In practice the AST would be larger, but this would make the example only more bloated.

We take the following two blocks of code as examples: lines 2 11 starting at the opening brace and ending at the closing brace, and line 7 9 representing the else block (again starting at the opening brace and ending at the closing brace). The "Block" nodes correspond in both cases to the root node of the sub-tree.

For ease of reference, the purple dotted outer box example will be called Example A, and the inner full inner box example Example B.

```
1  public class FooBar {
2     public void bar(int testNumber) {
3        int base = 10;
4        int result;
5        if (testNumber > base) {
6           result = base * testNumber;
```

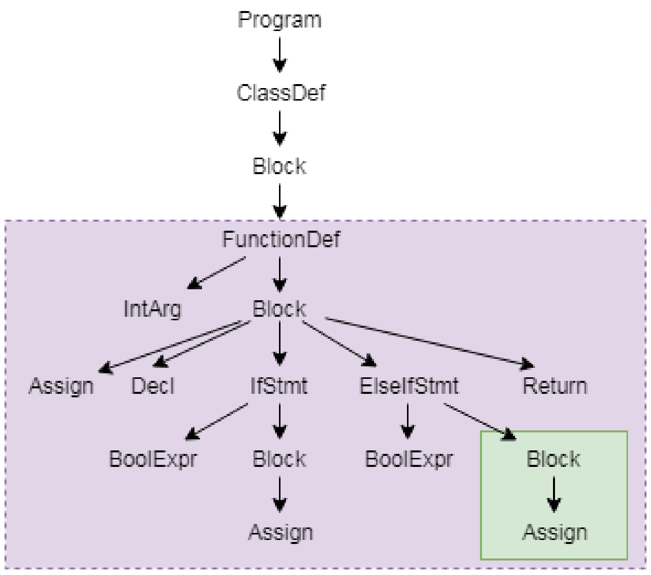


Figure 4.2: Simplified AST for the Java code snippet

```java
        } else if (testNumber <= base) {
            result = base - testNumber;
        }
        return result;
    }
}
```

Listing 4.1: Example Java code for an AST

4.1.1 Numeric Features

Numeric features are features that can be described using a numerical value. In programs, this could correspond to lines of code, characters, or more complex metrics such as cyclomatic complexity or maintainability index. In ARFF, these are represented by decimal numbers (doubles in Java).

Nodes (Node count)
The node count specifies the amount of nodes in the sub-tree of the node. The node count indicates how much code is below the current node in the AST. A high node count may indicate that a function is too long, while a count that is too low can indicate that something important is missing (an empty else statement, for example).

Example A: Node count is 14; all arrows, then plus one for the root node.

Example B: Node count is 2; block and assign

Tree depth
Tree depth is the number of edges it takes to get from the root node of the AST to the current node. A high tree depth can indicate that a program contains too many nested statements, which leads to high program complexity and readability issues.

Example A: It takes 3 steps to get to the root node of the complete AST, so the tree depth is 6.

Example B: Similarly to Example A, it takes 6 steps to get to the root.

The node count and tree depths are loosely related, as any node's parent will always have a higher node count while having a lower depth. The main difference is that the depth describes the structure higher up the tree (length from root to current node), while the node count describes the structure lower in the tree (sum of lengths to leafs). Therefore the two features are distinctive enough to have as separate features.

4.1.2 Nominal Features

Nominal features are features that can be divided into a finite set of classes. Example features could be: used language in a program, or a comment's current state. In ARFF, these are created by creating a list of possible classes at the very top, and then assigning one of those classes as a feature's value to a data point.

Node type
The node type corresponds to the type of node, such as a 'variable declaration', 'if statement', or 'block statement'. The type is the most descriptive feature of a node, as it describes the behaviour that the node has in a program. This on itself cannot be directly linked to an error, but a combination of other features with the node type is a good indication of what the context is at a certain point in the code.

Example A: The node type of the node is 'FunctionDef'.

Example B: The node type of the node is 'Block'.

Containing function
A node may be inside a function. This containing function's name is stored similarly as the used function/variable, as a nominal feature with a threshold. The function name's look-up is done by recursively calling the parent and comparing the node type: if the type is a 'FunctionDef', the name is returned. If

the root node is reached without crossing a 'FunctionDef', then the containing function is $None$: this node is not inside a function.

Example A: As the root node is a function definition of bar itself, the containing function is bar, assuming that bar is used often enough by other programs to warrant its own nominal value. If not, it would be categorized as SelfDefined.

Example B: The function surrounding the current node is bar, so the containing function is either bar or SelfDefined, following the same reasoning as Example A.

Containing node type

Containing node type is comparable to node depth, in the way it describes the structure which is above the node. For this, different block-type statements (if, switch, while, etc...) indicate in which structural part of the program this node is located. The method to obtain this is the same as the containing function, by recursively looking at parent nodes. Do note that actual 'Block' nodes are skipped, since they give no concrete information. They are always part of something that does give concrete information, like an 'IfStmt'.

Example A: The first occurring block statement, is 'ClassDef'

Example B: The first occurring block statement, is the 'ElseIfStmt'

When applying the technique from Section 4.4 (ignoring the function count threshold), an ARFF file would be produced comparable to Listing 4.3. For the nominal classes, we assume that the example was not the only data point in the training set, but a part of one with multiple programs.

Used function/variable

Inside a node, it is possible that a function/variable (or both) are used. This value is stored as a nominal feature, with values that are not often replaced with "SelfDefined". A variable such as 'x', or a function called 'random' may be used by many different people, while program-specific variables and functions such as 'clicksOnRedBoxCounter' or 'drawMouse()'will only be used by one program.

Example A: There are no functions or variables used, so the values for both of these are None.

Example B: The leftmost used variable is a line of code used, which in this case is 'result'. No function is used in this statement, so the value of the used function is None.

```
@relation ast-features

```

```
@attribute nodes numeric
@attribute depth numeric
@attribute used_function {SelfDefined,drawCreature,ellipse,
            random,setup,display,draw,None,main,fill,move}
@attribute used_variable {SelfDefined,x,y,pos,position,...}
@attribute node_type {ObjectCreationExpr,VoidType,
            ClassOrInterfaceDeclaration,StringLiteralExpr,
            BlockExpr,...}
@attribute containing_function {SelfDefined,drawCreature,
            setup,display,draw,None,main,move}
@attribute containing_node_type {SwitchStmt,ElseIfStmt,IfStmt,
            WhileStmt,None,ForStmt}
@attribute classification {correct,ForLoopsMustUseBraces,
            SimplifyBooleanExpressions,IfElseStmtsMustUseBraces,..}

@data
2,6,result,None,BlockExpr,SelfDefined,ElseIfStmt,ElseIfCanBeElse
```

Listing 4.3: ARFF file corresponding to the example

Now that the features are chosen, the first sub-question given in Section 2.3 has been answered. The original question is ‘Which AST-specific properties should be preserved?’. These properties are preserved by calculating the values of the aforementioned features. These features are then able to be given as input for standard machine learning algorithms, either to learn from, or to classify unseen data.

4.2 Weka

Weka [11] is a machine learning workbench. It is widely used, as it is user friendly, is open source, and has a large amount of machine learning algorithms and statistics that can be used. For these reasons, Zita incorporates the Java library of Weka to train and classify.

4.2.1 ARFF Format

The ARFF (Attribute Relation File Format) is the format used by Weka. It is a fairly simple format, which makes any data transformation relatively easy to do. An ARFF file describes a relation, which maps one or multiple attributes (features) to a class. For Zita, the exact features will depend on what classifier is used.

An example ARFF file can be found in Listing 4.4. This file starts off with a relation name ‘golf’. Afterwards a list of features is declared, in the example file there are five features:

- outlook - Can be either sunny, overcast, or rain

- temperature - A real number, ranging from 0 to 100
- humidity - A real number without restriction
- windy - A boolean value
- class - The possible classifications: play or don't play

One 'piece of data' can be represented by these five features. After the feature definitions, a list of data points is given. The features in a data point have the same order as the feature's declaration.

A '?' represents an unknown feature: this will be interpreted by Weka as 'should predict'. In other words, data with only known features is considered training data, and data with some unknown features is testing or general input data.

```
@relation golf
@attribute outlook {sunny, overcast, rain}
@attribute temperature real [0.0, 100.0]
@attribute humidity real
@attribute windy { true, false }
@attribute class { play, dont_play }
% instances of golf games
sunny, 85, 85, false, dont_play
sunny, 80, 90, true, dont_play
overcast, 83, 78, false, play
rain, ?, 96, false, play
```

Listing 4.4: Example ARFF file for training data for whether or not a golf game should be played.

4.3 Weka Library Usage

Weka is used in two ways: as a Java library inside Zita, and as a separate program where experiments can be done.

For Zita, it must be necessary to be able to directly access (multiple) classifiers, to either classify new programs or to retrain using new comments.

Weka's standalone program is used to interactively look at different classifiers and their statistics, classification error rate, and other information regarding classifiers. For example, it is possible to directly look at the decision tree that is made as described in Section 4.5.2. This is perfect to look at the inner workings of a classifier, rather than treating as a black box.

4.4 Transformation

In order to analyze the (EX)ASTs via Weka, they have to be transformed into a format that Weka expects (ARFF, see Section 4.2.1). For every data point, a set of features has to be calculated and then written to a file. The process can be divided into two parts: calculating the feature values per AST, and then combining the AST's information with the features and storing them into a file.

After parsing, the list of ASTs is iterated over, and each AST calculates and stores their own features' values in a Feature - Value map. During this process, a global counter is kept for the amount of used/containing functions in the AST. This is used later when categorizing functions.

Once all calculations are complete, first all attributes are specified according to the ARFF standard. At this point the functions are categorized: either they are self-defined functions, or functions that are often used. For the latter, examples would be $Math.random()$ to generate a random number, or $draw()$ which is a standard function of *Processing*. The functions that are often used have their own nominal class, while self-defined functions have one containing class. The actual categorization is based off a threshold value: if a certain function is used often enough, it will get its own category. This is done to reduce overfitting; it is undesirable to learn the structure of a function which is only used in a single program.

Once categorization is complete, all features can be defined using @*attribute* $<$ *type* $>$. The classification of a data point is simply another feature of this data point. Lastly, below the feature definitions each line will contain the values of a single data point in a comma-separated format. Important to note is that the values have to be in the same order as the @*attribute* definitions. The value for the classification is either a '?' for testing data, or the actual classification class for training data.

4.5 Classification Techniques

4.5.1 Naive Bayes

Naive Bayes is a popular classifier, as it is simple, fast, yet still effective. It uses Bayes' theorem to calculate the probability of an event happening, given previous knowledge about events. It does so by using the following formula:

$P(A|B) = \frac{P(B|A)*P(A)}{P(B)}$

Where:

- $P(A|B)$ is the posterior probability of event A occurring, given event B.
- $P(A)$ is the prior probability of event A occurring.

For example, let's say that there are 100 data points (ASTs). Of these 100, 75 are classified as 'correct' and 25 as 'incorrect'. Of these programs, decide to look specifically at the text '**if (x ¡ 0)**'. Say that 5 of the 'correct' programs contained this text, and 20 of the 'incorrect' programs contained it. If a new program is given, and it contains the text '**if (x ¡ 0)**', what would its classification be?

Using the formula above, the following variables can be made, with 'the text' being '**if (x $\leq$ 0)**':

- $P(A1)$ = the probability that something is correct. $\frac{75}{100} = 75\%$
- $P(A2)$ = the probability that something is incorrect. $\frac{25}{100} = 25\%$
- $P(B)$ = the probability that the text occurs in a program. $\frac{25}{100} = 25\%$
- $P(B|A1)$ = the prior probability of the text occurs in a program, given that the program is correct. $\frac{5}{75} = 6.7\%$
- $P(B|A2)$ = the prior probability of the text occurs in a program, given that the program is incorrect. $\frac{20}{25} = 80\%$

Note that there are two 'As', one for each classification.

So given this information, the values of the formula can be filled in, resulting in the following values.

For A1 (correct): $P(A1|B) = \frac{P(B|A1)*P(A1)}{P(B)} = \frac{0.067\times0.75}{0.25} = 0.20 = 20\%$

For A2 (incorrect): $P(A2|B) = \frac{P(B|A2)*P(A2)}{P(B)} = \frac{0.8\times0.25}{0.25} = 0.80 = 80\%$

Therefore the text '**if (x $<$ 0)**' will have a higher chance of being incorrect, and will be classified as such.

4.5.2 Decision Tree (REPTree)

A decision tree is built by using features as boolean values (such as '$depth > 5$' or '$nodetype =$ IfStmt'), and calculating the highest information gain per feature. The feature with the highest information gain will be put as a first decision to be made. This will split the tree in multiple branches, after which the process repeats, leaving out the previous-highest feature.

In Weka, there are multiple implementations of a decision tree or its derivatives. All implementations had similar results, giving true positive rates +-0.5% compared to eachother. From all the trees that Weka offers, the REPTree is the most convenient and fastest to use, therefore we use the REPTree.

A decision tree is based off of the different features' *information*. If a certain feature is distinguishing feature for some class X, this means that the presence of that feature on a data point correlates to a high chance of that data point being of class X. Such a feature would have a high amount of information.

The other way around is also true: if a certain feature is not distinguishing, the presence of it does not tell much about which class a data point should be. This feature would thus have a low amount of information.

A decision tree is built top-down, by choosing features with the highest information gain (IG) and 'splitting' on them, creating multiple children in the process.

The information gain is calculated by calculating the entropy. Entropy is the opposite of information: a larger entropy of a feature means that this feature is less distinguishing.

Code sample	**BlockType**	**Used X**	**Class**
A	If	true	incorrect
B	While	true	correct
C	If	false	incorrect
D	If	true	incorrect
E	While	true	correct
F	While	false	correct
G	If	true	correct
H	While	true	correct
I	If	false	incorrect
J	While	false	correct

Table 4.5: Example list of programs

Entropy is calculated by the following formula:

$$E(X) = -\sum_{i=1}^{n} P(c_i) log_2 P(c_i),$$

where X is a set of classes class $c_1, c_2, \ldots, c_n$ for some feature, and $P(c_i)$ corresponding to the probability of a certain class occurring.

We use Table 4.5 as an example training set. This set contains a total of 10 programs, with 4 columns of information:

- **Program** An single letter identifier.
- **BlockType** The blocktype, similar to 'containing node type' explained earlier.
- **Variable x** Whether or not a variable called 'x' was used in the code sample.
- **Class** The classification of this code sample, either correct or incorrect.

Using this information, we could calculate the entropy of the Class. There are 4 incorrect and 6 correct code samples. If these values are filled in the entropy of the formula, this results in $E(4, 6) = -(0.4 log_2 0.4) - (0.6 log_2 0.4) = 0.97$.

Similarly, the combination of two features can be calculated using

$$E(X, Y) = \sum_{c \in Y} P(c) E(c),$$

where X and Y are two features, c being a class of Y, $P(c)$ being the probability of c occurring, and $E(c)$ is the entropy of c.

Again using Table 4.5, the entropy of BlockType and Class can be calculated (see Table 4.6 for a frequency table of these two classes).

	correct	**incorrect**	**total**
If	1	4	5
While	5	0	5
total	6	4	10

Table 4.6: Frequency table of Class and BlockType.

$E(C, BT) = P(If)E(1, 4) + P(While)E(5, 0)$
$= 0.5 * 0.72 + 0.5 * 0$
$= 0.36.$

Similarly for Used X and Class,

$E(C, XU) = P(UsedX)E(4, 2) + P(NotusedX)E(2, 2)$
$= 0.6 * 0.92 + 0.4 * 0.5$
$= 0.75.$

So how can this information be used to generate a decision tree? The information gain (IG) is the difference of entropy. Entropy is initialized at 1 in the root of the tree. The following formula is used to calculate the information gain:

$$G(X, Y) = E(X) - E(X, Y),$$

where $E(X)$ is the entropy of an existing node. Combining all these formulas and previously calculated entropies, the information gain can be calculated for both BlockType and Used X.

$G(Root, BT) = E(Root) - E(Root, BT) = 1 - 0.36 = 0.64$ for BlockType and
$G(Root, XU) = E(Root) - E(Root, XU) = 1 - 0.75 = 0.25$ for Used X.

Therefore, BlockType has more information gain than Used X, and the decision tree will split on BlockType rather than Used X (see Figure 4.7).

This process continues until the Entropy of a node reaches 0, after which a decision of classification can be made.

4.5.3 Training and Verification

To compare these different classification techniques, the classifiers are trained with training data, and tested in two different ways.

The first way is using k-fold cross-validation. This involves cutting up the training data into k parts, using $k - 1$ parts for training, and the remaining part for evaluating the effectiveness. The reduces the effect of noise in the data set, as the average is taken of multiple distinct training sets. This works very well for general data, but becomes a problem after the data has been weighted. Weighted data in our data set is equal to duplicating data points. This means

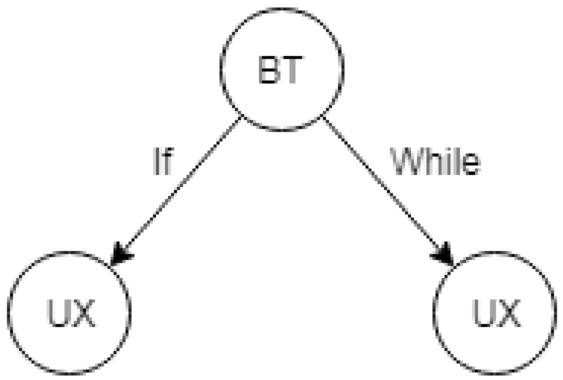


Figure 4.7: Tree after first split

that the same data point can be put in both the training and evaluation part, resulting in verifying the exact data that has just been learned. This does not make the cross-validation irrelevant, however, as it still a useful starting point for comparison.

The second testing way is to have a completely separate data set used as testing. The 2018 set is used as training data, and the 2017 data is used to compare how accurate the classifier is compared to a static tool. This should have no problem with the weights, as there is no overlap between the 2017 and 2018 data sets.

4.5.4 Evaluation metrics

For quantification of the performance, two main attributes will be looked at: precision and recall. Before discussing precision and recall, true/false positives/negatives have to be mentioned.

From the perspective of one class (let's say we call this class X), A 'positive' is when a something is classified as X, a 'negative' is when it is not classified as X. When verifying classifications, a true positive (TP) is when data is correctly classified as X. True negative (TN) is when data is correctly classified as not X. False positives (FP) are incorrectly classified as X, while false negatives (FN) are incorrectly classified as not X.

From these metrics, **precision** can be calculated as follows: $\frac{TP}{TP+FP}$. In words, the precision is the amount of correctly identified X compared to the total amount of identified X.

Recall is calculated as $\frac{TP}{TP+FN}$. Again, in words this corresponds to the amount of correctly identified X to the total amount of actual X.

If one would label all data as X, obviously all positives would be caught (recall of 1), but the precision would be very low. The other way around; i.e. a total of 1 data point is correctly identified as X, would have a precision of 1 (no FNs), but have a low recall due to missing a lot of actual X. Therefore having a high amount of both precision and recall indicates that the classifier is performing well.

4.5.5 Decision Tree Overfitting

Overfitting is a term used to describe that a certain machine learning model is too close to the training data. This is generally a bad sign, since it learns about the very specific details of a class, instead of the more general details. The performance of such a model would be excellent on the training data, but unsatisfactory on any other data.

With the features that are currently given, overfitting is a big concern. For example, let's say that students are not allowed to have nested if statements. Now if one student has made one type of error often in one function in the training set, the wrong behaviour could be learned.

```java
public class Processing {
   public void bar(int testNumber) {
      if (testNumber < 1000) {
         if (testNumber > 5) {
            // some code
         }
         if (testNumber > 10) {
            // ...
         }
         if (testNumber > 20) {
            // ...
         }
         if (testNumber > 30) {
            // ...
         }
         // more if statements
      }
   }
}
```

Listing 4.8: Example Java code for an AST

The expected behaviour is that this type is error is found whenever an 'IfStmnt' statement is found with containing type 'IfStmnt'. However, checking if the depth of an 'IfStmnt' is equal to 3 would result in the same statements being found in this code. However, the former behaviour will still succeed in other programs, while the latter may fail if the nesting is done inside a for loop (increasing the depth by 1).

To reduce overfitting, the tree will be *pruned*. This is done by applying a maximum depth in the tree. This will only make the decisions with the highest information gain appear in the tree, leaving out very specific cases.

Chapter 5

Data

5.1 Data Information

5.1.1 Collection

The two data sets that are used within this research are two separate year's final *Processing* assignments (2017 and 2018). These assignments are taken from the first year's course 'Programming for CreaTe', from the Creative Technology bachelor study.

The assignment is to create a game in *Processing*. The game can be anything the student can think of. Therefore each different program (game) can be completely different in structure, making it important to be able to spot errors in a generalized way.

As a starting point for comment data, CSEDU is run over this data. This results in a CSV file containing all necessary information about a warning that CSEDU gives. An example of such a CSV file can be seen in Table 5.1.

5.1.2 Description

Every *Processing* program can contain multiple classes, which are then exported as one big Java class containing all these classes. So each student's program will be contained in one file.

The CSEDU comments file contains all data regarding a comment: File, priority, line, description, rule set, rule.

- **File**: Describes which file this comment corresponds to.
- **Priority**: The severeness of an error: a naming convention might be less severe than having an unused variable somewhere.
- **Line**: The line of the file which has contains the error.
- **Description**: The description of the error that has been made.

File	**Priority**	**Line**	**Source**	**Class**	**Description**
file1.pde	3	51	Naming	VariableNamingConventions	...
file1.pde	1	94	*Processing* Ruleset	LongMethodRule	...
file2.pde	3	225	Empty Code	EmptyIfStmt	...

Table 5.1: Example comments file, with omitted descriptions

- **Source**: The source of this error. When using a tool for warning generation, the source indicates the name of the bigger set of classes to which this error belongs to. When annotated by a teacher, the source would be 'Teacher'.
- **Class**: The specific class of this error. In our context, this means the specific error type that has been made. This can either be annotated by humans using a tag-like system, or by tools by looking at which part of the tool picked up the error.

An example comments file can be found in table 5.1.

5.1.3 Statistics

In total, 287 *Processing* files are in the two data sets combined. 105 are from 2018, and 182 from 2017. The 2017 data set is further split up into educational (E) and non-educational (N) sets. The educational set's data is gathered from the same source as the 2018 data (final assignment of CreaTe's 2nd module's programming course), but a year earlier.

By running CSEDU's tool on the data from 2018, a list of comments is created. The rule sets are ignored, as the specific rules are more interesting to look at.

5.2 Data Transformation

The data that is present is transformed the way described in Section 3.3. From each file, the AST is built, and the comments from the comments file are connected to the AST, resulting in the EXAST.

Year	**Count**	**Avg # of nodes**	**Avg # of lines of Code**	**Avg # of errors**
2017	206	157.6	73.20	2.3
2018	96	490	233.2	7.4

Table 5.2: Data averages of the three different data sets

Class	**Count**
Correct	31997
VariableNamingConventions	97
ShortVariable	78
IfStmtsMustUseBraces	23
SimplifyBooleanExpressions	14
AvoidFieldNameMatchingMethodName	78
UncommentedEmptyMethodBody	8
TooManyFields	8
FieldDeclarationsShouldBeAtStartOfClass	4
EmptyStatementNotInLoop	3
ExcessiveMethodLength	3
ForLoopMustUseBraces	2
AtLeastOneConstructor	2
UnusedFormalParameter	2
MethodNamingConventions	2
IfElseStmtsMustUseBraces	2
UnusedLocalVariable	2
SingularField	2
ExcessiveParameterList	1
AvoidFieldNameMatchingTypeName	2

Table 5.3: 2017 dataset's annotated data classes

Class/Rule	**Count**
Correct	44155
TooManyFields	174
DecentralizedEventHandlingRule	144
LongMethodRule	126
SimplifyBooleanExpressions	118
ShortVariable	100
DrawingStateChangeRule	92
PixelHardcodeIgnoranceRule	76
AtLeastOneConstructor	71
VariableNamingConventions	35
MethodNamingConventions	26
IfStmtsMustUseBraces	21
LongParameterListRule	20
IfElseStmtsMustUseBraces	18
StatelessClassRule	12
FieldDeclarationsShouldBeAtStartOfClass	12
StdCyclomaticComplexity	12
DecentralizedDrawingRule	12
AvoidFieldNameMatchingMethodName	8
EmptyIfStmt	6
EmptyStatementNotInLoop	5
UnusedLocalVariable	4
UncommentedEmptyConstructor	4
AvoidReassigningParameters	3
AvoidDeeplyNestedIfStmts	2
IdempotentOperations	2
ForLoopsMustUseBraces	2
AvoidFieldNameMatchingTypeName	2
UncommentedEmptyMethodBody	1
UnusedFormalParameter	1

Table 5.4: 2018 dataset's annotated data classes

5.2.1 Cutting off classifications

Looking at Tables 5.3 and 5.4, it is clear that some classifications have a very low occurrence. It is therefore chosen that any class with a combined count lower than 10 will be cut off.

5.2.2 Transformed data statistics

As can be seen from the Table 5.4, the ratio of correct to any incorrect class is unbalanced. To correct this, either the amount of correct data can be reduced by leaving out correct data, or the amount of incorrect data can be increased, by applying weights to the incorrect data nodes. We choose to add weights to the incorrect data nodes. The reason for this is that the current data set is already not that big for incorrect labeled data, and removing data would only make this worse. After adding weights, the data becomes much more balanced.

Note that adding weights is in our case the same as adding the same data multiple times, which can cause the classifier to overfit. For this reason, we have to make sure that the training set is not the same as the testing set. The 2018 data set is chosen as the training set, as this data set also has additional information regarding the comments.

Chapter 6

Results

This section describes the results of different classifiers, using the 2018 data set as training data, and the 2017 data set as testing data. All data preparation has been done as described in Sections 5.2 and 5.1. Throughout all tests, a function threshold value of 1000 has been used for the cutoff between user-defined functions and often-used functions.

6.1 Naive Bayes

The Naive Bayes classifier gives good results in very specific cases; namely when a string of words is directly related to an error. For example '== true' is an error that is perfect for Naive Bayes, as this specific string is often unnecessary to include in boolean expressions.

The most difficult problems for this classifier are the ones where the text is spread out: if e.g. a certain statement should not be made in an if block, it is possible that in actual programs this statement is surrounded by code that is correct. This results in the correlation between the if statement and the statement being lost.

Combining this problem with the fact that the data is naturally unbalanced, the resulting effectiveness of Naive Bayes is still okay to work with, as seen in Listing 6.2.

```
Correctly Classified Instances 89279         58.9335 %
Incorrectly Classified Instances 62212       41.0665 %
Kappa statistic               0.5359
Mean absolute error           0.0538
Root mean squared error       0.1781
Relative absolute error       59.0886 %
Root relative squared error   83.45 %
Total Number of Instances     151491
```

Listing 6.1: Naive Bayes 10-fold cross-validation

When using the raw content as the singular feature, Naive Bayes effectively acts like a spam filter. Regardless, the results of using only the content as feature is slightly better than using the AST's features.

```
Correctly Classified Instances 104972        61.6116 %
Incorrectly Classified Instances 65405       38.3884 %
Kappa statistic              0.5799
Mean absolute error          0.0316
Root mean squared error      0.1437
Relative absolute error      52.1758 %
Root relative squared error   82.5491 %
Total Number of Instances  170377
```

Listing 6.2: Naive Bayes 10-fold cross-validation with raw text

6.2 Decision Tree

The decision tree's results are more interesting to look at, since the decision tree itself can be viewed to see exactly how a sub-tree of an AST will be classified.

About 75% of the instances (data points) are classified correctly. While may look fairly good, this is the result of using k-fold cross-validation on a weighted data set. Part of the training set is also in the data set, which means that these results must be taken with a grain of salt.

```
Correctly Classified Instances   112960      74.5655 %
Incorrectly Classified Instances 38531       25.4345 %
Kappa statistic              0.705
Mean absolute error          0.0357
Root mean squared error      0.1339
Relative absolute error      39.2217 %
Root relative squared error   62.7419 %
Total Number of Instances     151491
```

Listing 6.3: Decision tree 10-fold cross-validation results

When using a completely different data set for training and testing (Listing 6.6), the performance immediately drops. However, this looks worse than it actually is. Listing 6.2 shows a table of statistics of every class. Note that *FieldDeclarationsShouldBeAtStartOfClass* and *StdCyclomaticComplexity* are not in this table, as no instance was classified as either of these. Many of the misclassifications are from classes that appear often, but are purely based on the contents of the code. This is something that is barely represented in the features, resulting in a low precision in these classes.

```
Correctly Classified Instances 85900       60.4563 %
Incorrectly Classified Instances 56186     39.5437 %
Kappa statistic               0.5411
```

TP Rate	FP Rate	Precision	Recall	Class
0.965	0.028	0.753	0.965	a = PixelHardcodeIgnoranceRule
0.898	0.021	0.777	0.898	b = DrawingStateChangeRule
0.758	0.111	0.758	0.758	c = correct
0.750	0.036	0.580	0.750	d = SimplifyBooleanExpressions
0.700	0.003	0.838	0.700	e = IfStmtsMustUseBraces
0.613	0.030	0.710	0.613	f = DecentralizedEventHandlingRule
0.567	0.005	0.694	0.567	g = DecentralizedDrawingRule
0.440	0.004	0.646	0.440	h = MethodNamingConventions
0.381	0.001	0.891	0.381	i = IfElseStmtsMustUseBraces
0.333	0.007	0.162	0.333	j = StatelessClassRule
0.291	0.023	0.592	0.291	k = ShortVariable
0.209	0.017	0.385	0.209	l = VariableNamingConventions
0.193	0.006	0.657	0.193	m = LongMethodRule
0.167	0.018	0.139	0.167	n = AtLeastOneConstructor
0.605	0.049	?	0.605	Weighted Avg.

Table 6.5: Performance of the decision tree on non-trivial classes. Red is unsatisfactory, yellow is okay, green is good, and blue is exceptional

```
Mean absolute error              0.0294
Root mean squared error          0.141
Relative absolute error         49.0227 %
Root relative squared error     81.3211 %
Total Number of Instances   142086
Ignored Class Unknown Instances     100
```

Listing 6.4: Decision tree, max depth=5. 2018 training, 2017 testing results

Table 6.2 shows a confusion matrix of the non-trivial classes, rounded to the whole percent. Additionally, two classes, *AtLeastOneConstructor* and *StatelessClassRule* are removed for readability and being relatively uninteresting, as their error messages are put at the top of the class, making the context irrelevant.

Using Table 6.2 and Table 6.2, we can look through all interesting classes one by one, see how they perform, and reason about why this might be.

a - PixelHardcodeIgnoranceRule Pixel hardcoding is classified very well. In the original static analysis tool, a line is marked as this class whenever a method call is done for which the method is a known drawing function and the arguments are all literals. In the decision tree, something similar is achieved: something is classified as PixelHardcodeIgnoreRule if it has a low amount of nodes in combination with using a drawing function. Since many students use the same drawing functions (like 'ellipse'), this function is put into its own nominal variable, and the model is able to distinguish drawing functions from non-drawing functions.

a	b	c	d	e	f	g	h	i	k	l	m	
97%	1%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	a
6%	**90%**	2%	3%	0%	2%	0%	0%	0%	0%	0%	0%	b
5%	2%	**76%**	2%	0%	3%	2%	1%	0%	2%	1%	2%	c
0%	0%	11%	**75%**	0%	14%	0%	0%	0%	0%	0%	0%	d
0%	0%	13%	3%	**70%**	13%	0%	0%	0%	0%	0%	0%	e
3%	6%	5%	24%	1%	**61%**	0%	0%	0%	0%	0%	0%	f
10%	0%	33%	0%	0%	0%	**57%**	0%	0%	0%	0%	0%	g
0%	0%	8%	0%	0%	0%	0%	**44%**	0%	0%	0%	0%	h
0%	14%	10%	14%	0%	24%	0%	0%	**38%**	0%	0%	0%	i
1%	3%	6%	0%	0%	2%	0%	1%	0%	**30%**	6%	0%	k
0%	0%	0%	0%	0%	0%	0%	0%	0%	22%	**21%**	0%	l
0%	0%	21%	0%	0%	0%	0%	0%	0%	0%	0%	**19%**	m

Table 6.6: Confusion matrix of interesting classes

b - DrawingStateChangeRule The DrawingStateChangeRule is violated whenever a student tries to change the state of the program while in the 'draw()' method. This is normally checked by looking at the call stack whenever a non-local variable is changed. If this call stack contains the 'draw()' function, this means that the state is being changed while in the 'draw()' function.

The model achieves a similar thing by look at purely expression statements (high probability of being an assignment), and checking if the containing function is 'draw()'. It is not perfect, however, as it also looks for usages self-defined functions. Still, given that most of the code is done in the drawing method, a large part of the classified instances are indirectly used in 'draw()', even if the model doesn't know about this.

c - correct Simply saying, something is correct whenever it is not an error. If no pattern can be found, it is deemed correct. Given that 97% of the original code is correct and only 75% is found, we can conclude that the model is quite aggressive in its error labeling behaviour. This makes sense, as code that is erroneous has a large weight attached to it. In the context of Zita, this is a good thing, as it is more important that errors are found than that correct code is marked as correct.

d - SimplifyBooleanExpressions The rule that says that boolean expressions can be simplified, e.g. 'keyPressed == 'K' == true' can be simplified to 'keyPressed == 'K', performs fairly well. In PMD, this is implemented by checking if some boolean expression is followed by a boolean literal. In the model, a broader approach is taken, where the node count is checked, in combination with what kind of block the current statement is in. Interesting to see is that it is at times confused with f: DecentralizedEventHandlingRule. The reason for this is that event handling often makes use of boolean values, like the example above which utilizes *keyPressed*. Since the model is only able to classify an instance once, it is possible that some overlap exists between these two rules.

e - IfStmtsMustUseBraces This rule performs better than expected with the features available. It only occurs 18 times in total, so the amount of available examples is limited to learn from. The first check in the model is one for the node type (an IfStmt check), followed by a node count check. This is fairly accurate, even though there can be some overlap between having a large statement without braces, and a smaller statement with braces. The node count would be similar in this case.

f - DecentralizedEventHandlingRule This class has a relatively low recall, indicating that it is either strict in its classification, or that another class takes priority over this one. The largest amount of classifications are done via looking at block statements and checking if it has a large 'if' part. This is completely different from what is done in the normal static analyzer, which checks if event-related properties are being used outside of their event-related methods. Still, using the heuristic that a large if block correlates to probably some event handling being done is an interesting and decent way to look at it.

g - DecentralizedDrawingRule The decentralized drawing rule occurs very sparsely in the data set (12 times). The model attempts to link the usage of a specific drawing function (ellipse) and it being used in a self-defined method to a violation of this rule. This is not very accurate, but is still able to catch some cases regardless.

h, i, k, l, m - Contextually-unaware errors Everything after this point is considered unsatisfactory performance in terms of recall. The classes **MethodNamingConventions**, *ShortVariable*, and *VariableNamingConventions*, and are all nearly impossible to verify by purely looking at the context. All the rules have to do with the actual string contents of a data point, which is something that is not represented using the chosen features. For this reason, it makes sense that the model is not very capable at finding these classes. It tries its best by assuming that students will have wrong variable names in places where variables often occur (top of the class, method arguments, see Listing 6.8), but that is all that is possible.

IfElseStmtsMustUseBraces and *LongMethodRule* both have very low recall, while having good precision. Similar to *DecentralizedEventHandlingRule*, this means that these rules are most likely too strict or that another class takes priority. For the former class, the amount of data is fairly low (18 instances), so it most likely matched on features very similar to these 18 instances, resulting in a strict but precise filter.

For the *LongMethodRule*, the issue is that a high amount of nodes is for most features not the most distinguishing feature. Instead, first a combination of other features are selected (node type, containing function), after which the actual node count is placed. This results in the specific combination of high node count with these select features giving a correct result, but a high node count with other combinations not being found.

This section looks at parts of the resulting decision trees, where some interesting properties can be seen.

```
|  containing_block = IfStmt
|  |  used_function = None
|  |  |  nodes < 2.5 : EmptyIfStmt (344/73) [159/30]
```

Listing 6.7: Simple structure check

Listing 6.7 shows a very simple pattern that was found for the rule which checks if an if statement is empty. If a node is inside an if statement, but has 2 or less nodes, it is classified as an empty if statement. When looking back at the original code, this makes sense: the original error is given on the line where the if is defined. This line has two children, namely the condition of the statement and a block statement without anything in it.

There are some misclassified instances, however. This can be explained by students not always using braces. If an if statement has a condition and immediately following a singular statement, this behaves the same as if the singular statement would be in a block statement. The node count would still be equal to 2 regardless.

```
node_type = FieldDeclaration
|  depth < 2.5
|  |  nodes < 2.5 : VariableNamingConventions (588/393)
    [287/182]
|  |  nodes >= 2.5
|  |  |  nodes < 6.5 : ShortVariable (1013/276) [523/160]
|  |  |  nodes >= 6.5 : VariableNamingConventions (242/48)
    [133/27]
```

Listing 6.8: Student behaviour guessing

Listing 6.8 shows an example of where Zita takes seemingly irrelevant information, but still performs decently. In this example of the created decision tree, it is assumed that the student will make some kind of naming error when creating a field. Depth ¡ 2.5 indicates this is an instance variable, and that amount of nodes indicates the complexity of the statement. A simple ‘int number;’ has 2 nodes, while something more complex such as ‘Triangle triangles[] = new Triangle[10]’ would have more nodes.

With these results, research sub-question 2: ‘What kind of problematic code is able to be reliably found?’ can be answered. There are two points that are important in order to allow Zita to reliably find errors:

- There has to be enough data

- The pattern of an error is able to be represented using (a subset of) features.

If there is a lack of data, the ML algorithms is unable to find a correct pattern that represents an error in general. If the chosen features are not able to represent an error, then the algorithm has no way of actually recognizing faulty code.

This can be verified by looking at the results: error types that were easily found satisfy the two points. Error types that performed badly either had a very low occurrence or were untraceable using the chosen features.

Chapter 7

Discussion

This final chapter will shortly discuss what can be concluded from the results, and how further research can be done, using this research as a starting point.

7.1 Usability in Real Environments

This section aims to answer the final sub-question: What is the advantage of using machine learning over normal static analysis in a tutoring environment? Any of the rules discovered by the ML algorithm is also able to be found using static analysis. So why would one use Zita over already existing tools such as PMD?

The main advantage is being able to find very specific errors that occur often, but only in the context of the course. In a tutoring environment (such as assignments during tutorials) the code students write has a lot of overlap. One could write custom rules for static analysis, but this costs a lot of time and work to implement. Especially the errors that are currently caught require context awareness, which would take more time to implement than simpler checks than variable name checks. Variable name checks, such as names being too short can be found using a simple regular expression, so the value of Zita for these kind of errors is not very high.

7.2 Conclusion

So in conclusion it is indeed possible to learn from trees. The most important technique when using decision trees is to apply proper feature extraction and selection, since this is the key to maintaining the tree's structure for the machine to learn. We've learned that contextually relevant features result in well performing classifications on contextually relevant classes, and that context-ignoring features such as naming errors are found through guessing of behaviour. In the end, to properly use machine learning in an educational environment, more data will have to be gathered and explored in order to fully enable the potential of machine learning.

7.3 Future Work

This research has laid the ground work of making an ML-based automatic assessment tool. However, it is not yet ready to be fully used. Future work is necessary to not only create an actual connection between this tool and the tutor, but also increase the performance of the classification part of the tool.

- Zita could be integrated in some kind of online programming environment, such as CoDR (discussed earlier). Zita would be able to directly gather data from CoDR, and integrate it with its own knowledge base. This increases the amount of available data to learn from, and therefore increase performance.
- Currently, Zita mainly learns from static analysis tools, and tries to imitate them. Ultimately, Zita should create rules based on the feedback of tutors, not tools. For this, a large amount of data would have to be annotated by a tutor, which costs a lot of time.
- Alternatively, different static analysis tools (Sonarcube, Findbugs) could be added to increase the amount of detected errors. A downside would be that different tools have different opinions about code, which can prove to be problematic.
- Looking more to the ML side of things, one could look to improve or alter the features that are created in order to learn from ASTs. Only the AST itself is currently taken into account, but related data structures such as control flow graphs could be created, in order to get more contextual information about a given node.
- Additionally, different ML methods can be explored to look at performance. A good example is clustering, where each class would be its own 'cluster' of trees that look like each other.
- Lastly, more data in general can be added to see if the model is able to increase its performance. 300 programs is okay, but some errors still only managed to appear less than 10 times.

Bibliography

[1] Autolab project. http://www.autolabproject.com/. Accessed: 2019-11-04.

[2] Checkstyle project. http://checkstyle.sourceforge.net/. Accessed: 2019-10-04.

[3] Codegrade. https://codegra.de/. Accessed: 2019-10-04.

[4] Javaparser. http://javaparser.org/. Accessed: 2018-10-02.

[5] The Java® language specification. https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf. Accessed: 2019-03-19.

[6] Pmd website. https://pmd.github.io/. Accessed: 2019-04-03.

[7] T. Akutsu, D. Fukagawa, M. M. Halldórsson, A. Takasu, and K. Tanaka. Approximation and parameterized algorithms for common subtrees and edit distance between unordered trees. *Theoretical Computer Science*, 470:10–22, jan 2013.

[8] R. de Man and A. Fehnker. The smell of processing. In *Proceedings of the 10th International Conference on Computer Supported Education - Volume 2: CSEDU,*, pages 420–431. INSTICC, SciTePress, 2018.

[9] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming. *Journal on Educational Resources in Computing*, 5(3):4–es, sep 2005.

[10] A. Fehnker and T. Blok. Automated program analysis for novice programmers. In *Proceedings of the 3rd International Conference on Higher Education Advances*. Universitat Politècnica València, jun 2017.

[11] G. Holmes, A. Donkin, and I. Witten. Weka: A machine learning workbench. In *Proc Second Australia and New Zealand Conference on Intelligent Information Systems*, Brisbane, Australia, 1994.

[12] A. V. Phan, P. N. Chau, M. L. Nguyen, and L. T. Bui. Automatically classifying source code using tree-based approaches. *Data & Knowledge Engineering*, 114:12–25, mar 2018.

[13] V. A. Phan, N. P. Chau, and M. L. Nguyen. Exploiting tree structures for classifying programs by functionalities. In *2016 Eighth International Conference on Knowledge and Systems Engineering (KSE)*. IEEE, oct 2016.

[14] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering - ICSE 16*. ACM Press, 2016.

[15] F. Wei, S. H. Moritz, S. M. Parvez, and G. D. Blank. A student model for object-oriented design and programming. *J. Comput. Sci. Coll.*, 20(5):260–273, May 2005.