NTNU
Norwegian University of
Science and Technology

Automatic Grading of Programming Exams

Jørgen Sirhaug

Master of Science in Informatics
Submission date: June 2018
Supervisor: Trond Aalberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

The fields of Computer Science and IT are more needed than ever before, and the number of students enrolled in programming courses is rising. An increase in students leads to an increased demand for more teachers and student assistants, and for the final exam, more graders.

To resolve this need, this thesis explored the possibility of an automated grading system, which would learn grading patterns from human graders by extracting features from students' exam submissions and using them to train a machine learning classification system.

A variety of source code evaluation strategies, classification algorithms, parameter ranges and feature sets were assessed, and an experiment was conducted using the continuation exam of 2017 for the course *TDT4100 – Object-Oriented Programming* as a dataset. The experiment lays the foundation for further research and development within the fields of source code analysis and quality checks, while results were inconclusive because of what we believe to be a sub-optimal dataset.

Keywords: Machine learning, classification, Linear SVC, Naive Bayes, feature extraction, scikit-learn

Sammendrag

Studier innenfor informasjonsteknologi og informatikk er mer trengende enn noensinne, og antall studenter som tar programmeringsfag er på vei opp. En slik økning fører til en voksende behov etter flere lærere og studentassistenter, og mot eksamen trengs det flere sensorer.

For å løse dette behovet tar denne masteroppgaven og utforsker muligheten for å lage et automatisk graderingssystem, som skal lære rettemønstere fra menneskelige sensorer og bruke disse til å trene opp et klassifiseringssystem.

En utvalg av evalueringsmetodikker, klassifiseringsalgoritmer, parametersett og kodeegenskaper (eng: feature sets) ble vurdert, og et eksperiment ble utført på et datasett bestående av besvarelser fra *TDT4100 – Objektorientert Programmering* sin utsatte eksamen i 2017. Eksperimentet har lagt grunnlaget for videre forskning og utvikling innen analyse av kildekode og kvalitetskontroll. Derimot er resultatene mangelfulle på bakgrunn av det vi vurderer som et ufullstendig datasett.

Nøkkelord: Maskinlæring, klassifisering, Linear SVC, Naive Bayes, egenskapsutvinning, scikit-learn

Preface

This Master thesis is written as a final part of the Master's degree in Informatics, submitted to the Department of Computer Science at the Norwegian University of Science and Technology. It is completed under the supervision of Associate Professor Trond Aalberg.

I would like to say thank you to everyone who has helped me during my studies. To family, friends, acquaintances, strangers and everyone in between, no matter how great a favour or how small an explanation: You have my warmest of thanks, because this would have been impossible to do alone.

Table of Contents

Chapter 1

Introduction

1.1 Problem description

Computer Science is a growing field, and for programming courses this means that more and more students are taking exams each year. Assessment and grading of these exams is a time-consuming task, and with very large courses and exam sets this increases the chance of grade inconsistencies; two fairly similar exam submissions getting two different grades.

Given that digital exams are becoming the norm then this opens up the possibility of automating the grading process. For multiple-choice exams this is a trivial issue, as automatic grading exists even today for analogue, pen-and-paper multiple-choice exams. For exams where the answers must be source code this is not as simple. This thesis will describe one such non-trivial grading system, and the processes behind it.

1.2 Motivation

Time.

Today, hours upon hours are being spent manually reading through and evaluating code. Not just in large scale production, but also for educational purposes, where professors, teachers and assistants are reading heaps of student-produced code so that the students may end up with a working program, some feedback and eventually a grade.

Grades can be given out in any way the teachers deem fit, and in many cases this boils down to asking questions and somehow evaluate the answers. The easiest questions and answers to evaluate are those that have *one* clear and defined solution, for example multiple-choice. Such exams can be fed into a computer and evaluated automatically, and the final grade can be given out to the students that very same day. On the other hand we have exams where both questions and answers may be somewhat open and ambiguous while still being correct. Here the teacher(s) may have to create their own interpretation of what the student actually meant, and give an explanation and a grade which might be completely different from what the student thought the end result would be.

And programming lays somewhere in the middle.

The logical and technical side of programming suggests that each problem presented by the teacher may only have *one* correct solution, while the students are required to answer with free-hand code. By creating a tool which is able to change the way programming exams are evaluated, it may be possible to reduce the time spent by teachers evaluating source code from exams, and bring more consistent and predictable evaluations to the students as well.

1.3 Project goal

The main goal for this project can be summarized as follows: Let teachers spend less time grading, while giving students equal grades and better feedback.

To accomplish this, three questions were asked:

- *What kinds of source code evaluation techniques exists today, and how can they be used for machine learning?*
- *Which features and metrics are viable for an objective evaluation of source code?*
- *How would a machine learning classifier perform when grading programming exams?*

1.4 Machine learning classification

The problem was to create an evaluation tool which could be used on a smaller set of exam submissions for a programming exam, and return a grade for each submission. This problem description fits the scope of *classification*, a type of supervised machine learning where a set of known data is fed into a learning algorithm, and returns its best estimation for which of the pre-defined categories the data matches best.

The automatic grading system described and used in this thesis and its experiment is built with *scikit-learn*, a set of machine learning tools made for the programming language Python.

1.5 Thesis contents and order

In addition to the thesis introduction, which was chapter 1, this thesis consists of the following:

Chapter 2 introduces the concept of *code quality*, before presenting various methods for evaluating source code and how they are used today, both in education and within the industry.

Chapter 3 describes subjects relevant to grading and how exam submissions and source code can be represented as objective features.

Chapter 4 presents this project's use of machine learning; which algorithms were used and how they were implemented.

Chapter 5 features the experiment and its results.

Chapter 6 rounds off this thesis with a discussion, conclusion and recommendations for future work.

Chapter 2

Theory

2.1 Code quality

The ability to analyze and validate the quality of source code is important for every professional or aspiring programmer, but it comes with a few limitations: what does it mean for something to have *high quality*? The Oxford English Dictionary defines "quality" as follows: "*The standard of something as measured against other things of a similar kind; the degree of excellence of something.*"[1] In short, something of a high quality is simply better than other, comparable things around. So what do we use to measure the quality of our code? What is our baseline?

2.1.1 Code quality in the industry

Just as with spoken languages there exists a long list of available programming languages, each with their own strengths and quirks. But is any definition of "good code" restricted to only one language? In his book *Clean Code* [25], Robert C. Martin describes how he thinks code should be written by imbuing a set of rules for how to write classes, methods, comments, tests and everything in between. The general take is that code should be efficient, elegant, readable, simple and non-repeating, but there are also more specific rules. For instance, two such rules for methods is that they must do something and return something, and that the optimal number of parameters is zero, after one and two.

While very detailed and quite helpful, it is no secret that this book was written with the industry in mind. Just as there is a difference between writing text for a school assignment and writing a best-selling series of novels, there is a distinction between answering a programming exam and writing code for a huge project spanning

[1]Definition taken from the Oxford Dictionaries' website, `https://en.oxforddictionaries.com/definition/quality` at the 20th of June, 2018.

multiple years and dozens of developers. Still, while project scope and size might have an impact on how to design your program, and by extension how to write your code, there is one thing that *Clean Code* holds above everything else: source code is created for humans, by humans and should be understood by humans.

This statement, that readable and understandable code (or ”clean code”, if you will) must be the goal of every programmer is also backed up by other sources. A 2009 study [6] concluded that there was a significant connection between the quality of identifier names and the number of bugs and warnings found when analyzing the source code; an older study from 1982 [11] showed that breaking a large program up into smaller functions (not changing functionality, but focusing on readability) had a positive impact on maintainability, and reduced the overall number of bugs. By looking at source code features which may indicate a level of readability, it could be possible to assess whether it could be deemed as good code or not. These features will be discussed later in chapter 3.

2.1.2 Testing code for quality in education

Codecademy and Treehouse are two of many websites where users can take online programming courses. However, these courses are created and maintained by the sites themselves, and are not necessarily meant to work as supplements to courses taken at a university or college. One of the main differences between a university course like *TDT4100 – Object-Oriented Programming* at NTNU and a course at Codecademy or Treehouse is the scale: *TDT4100* had just over 700 students enrolled in spring 2017 who have to follow a pre-planned semester, while Codecademy has 45 million registered users and Treehouse 180 000 users[2], who are able to take whichever course they want at whichever time that suits them the best. This means that their tasks and assignments must be evaluated quickly and precisely.

Even though the scale and scope of traditional universities and these online learning sites might be different, the evaluation techniques seems to be the same. There are a lot of different evaluation tools available, for example Web-CAT and AutoGradr for education, DOMjudge and Mooshak for programming competitions, HackerRank for recrutation and even *TDT4100*'s own tool, JExercise. But all of these tools have one thing in common: they generally use unit tests to evaluate their tasks.

One possible reason for there being so many evaluation options for assignments and

[2]Numbers taken from NTNU's official grade statistics, Codecademy's and Treehouse's websites, respectively (15th of May, 2018).

single tasks, but not for exams could be that assignments are meant to *teach* people in how to do programming, while exams are meant to *test* learned knowledge and skills. By reusing tasks for teaching, those tasks could be incrementally improved over time; the opposite can happen with exams, where tasks are remade, changed and/or randomized each year to keep the tests from becoming too predictable. In other words, creating tasks and tests for education may happen once, while creating tasks and tests for exams may have to be done anew each semester.

2.2 State of the art in code evaluation

2.2.1 Testing

Generally speaking, the term *testing* refers to any one of a number of different practices, including system testing, acceptance testing, integration testing and user testing. However, for the purpose of this thesis, testing is referring to unit testing, which tests the functionality of different *units* of the program. A unit might be a function or a class, and unit tests ensures that these functions and classes behaves as intended.

In quite a few scenarios, testing code is a good way to check if your program behaves at it should. By writing test cases, a developer is able to define which types of results that should be approved and which should be declined. Test-driven development (TDD)[3] has shown to be of great help both in large scale production [16] and when used in education [12].

Tests are both acceptable and reasonable when the output is the most important part of the program and when incremental development is employed, but not in an extremely restricted time frame where the code itself is important, which it is during an exam. If, during an exam, a student writes code which would not compile nor give the correct answer, the student might very well get partial or even a high score because the code itself was written according to the specified task.

In addition to this, there is another problem. With production code, test cases are not only used to test the current state of development, but they also serve as a simple way of checking that existing functionality isn't hampered by new additions or improvements [16]. The tests are as much a part of the program as the rest of the

[3]TDD is a programming and development strategy where a solution is defined by a series of test-cases (unit tests). These test-cases are made first, while the actual solution is made as a response, aiming to pass the tests.

code, which indicates that the test cases are written *specifically for each program.* This is not great news for exams, as exams tend to change from year to year. Yes, some tasks may be changed, updated and reused, but this means that in addition to updating all of the task descriptions and example code, all test cases must also be rewritten and updated.

It seems like test cases, accompanied by a (quick) manual review of the written code, can be a great way of evaluating homework or assignments during the semester; not so much as a reliable way of scoring programming exams.

As per December 2017, a quick search on Google for the phrase "*unit test*" lists just shy of 36 million results, including numerous blogposts and articles about the subject; Wikipedia has a comprehensive list of unit test frameworks for about 80 separate programming languages and development systems [47]; Microsoft has its own guide for unit tests in Visual Studio [27], and the official Python documentation has its own section for unit testing [34], based on the popular JUnit framework for Java [41].

In education, websites such as Codecademy [8] and Treehouse [42] are sites where you can take online programming courses. This includes solving programming tasks, and getting direct feedback on whether you solved the task or not. In the case of Codecademy, the feedback is a combination of different messages related to the current task, console output and a set of red or green checkboxes, indicating which subtasks which has been cleared. A senior web developer from Treehouse answered a question on Quora [26] and confirmed that they are using unit testing to validate the users' code. Others who also answered this question said the same for both Codecademy and Treehouse. At NTNU, the course *TDT4100 – Object-Oriented Programming* utilizes JExercise [43], an extension of JUnit, to present and validate the students' course assignments.

Applied to real-world projects, unit testing (as well as TDD) is used in companies of all sizes, from Fortune 500 [36] to those smaller and less known [9, 46]. Several studies also shows that unit testing can improve productivity in a project, and reduce time it takes to fix a problem [4, 28, 36, 40].

2.2.2 Parsing

Parsing, in general, is referring to the process of analyzing written languages, be it natural or programming languages, as sequences of symbols. These analyses must follow a strict, formal grammar. When parsing source code, the result can be an Abstract Syntax Tree; a tree structure representing code segments as nodes, and body/condition splits as edges.

As an evaluation tool for programming tasks and exams, parsing (and by extension, the AST) can be used to look for conceptual similarities between the students' answers and the actual solution. This can be done by building an AST for the solution, searching the tree for important concepts, and comparing these concepts to what is found in the students' syntax trees. To know which concepts are more important than others, a manual review and ranking could be made before the automatic analyses are run.

An advantage to this type of evaluation is that no compiling is necessary. The code is processed as a string, or a sequence of symbols; the grammatical rules for that language, in addition to concept mining, is what is needed to evaluate the code.

A disadvantage is that even though the code doesn't need to compile to be evaluated, the parser must still be able to decipher what the different symbols are, and what they represent. Code with (a lot of) grammatical errors (including missing parenthesis, semi-colons and indentations) could make the parser return an error, or wrongly analyze the code.

The result of parsing code, the Abstract Syntax Tree, have many different applications, and can be utilized wherever a structural overview of source code is needed. For instance, when dealing with the concept of code similarity there has been research into how one can use ASTs to discover structural similarities between two samples of source code, both in terms of *clones* and *plagiarism* [14, 20].

2.2.3 Language processing

It is also possible to evaluate a whole programming language as a natural language. Source code could then be analyzed the same way one would written text, like a newspaper article.

There are many challenges with Natural Language Processing, one of them being dealing with context and knowledge which comes from *outside* the written document, i.e. "worldly knowledge". In a sense, this can be related to the problem of defining how a computer program has to *do something*. A bit of code can be grammatically and syntactically correct, but also be absolutely useless for any practical purpose.

The work of Marcus and Maletic [24] dives into the realm of using information retrieval techniques on source code. By processing both ”sentences” and individual words/tokens of source code, they have shown that it is possible to identify domain concepts, and using these to detect what they call *high-level concept clones*. Similar work is done by Kuhn, Ducasse and Gîrba [21] where they use some of the same techniques (latent semantic indexing) to detect *source code topics* and to “*reveal the intention of the code*”.

As with parsing, language processing tools are also used to detect plagiarism, both within the industry and academia. Agrawal and Sharma [1] did a review of different plagiarism detection tools, where natural language processing techniques where used in one of those tools.

2.2.4 Code idioms

In natural languages, an idiom is a set of words, or a *phrase*, where the words have an implied meaning greater than the individual words themselves. Some idioms are more transparent than others, and can be (directly) translated to other languages; other idioms can't be translated, because the implied meaning would have been lost in translation.

Code idioms they are similar to those in natural languages: bits of code which have an implied meaning, or specific usage, which is only really useful (or possible) in the particular language the idiom is written in. One very clear example is the usage of *list comprehensions* in Python. Most, if not all programming languages have `for`-loops, which can be used to fill a list with increasing numbers. In Python, such a `for`-loop would look like this:

```python
# 1) Define a new list
# 2) Using a Pythonic for—loop, append
#    each integer from 0 to 9 to the list
my_list = []
for i in range(10):
    my_list.append(i)
```

An idiomatic way to do this in Python would be

```
# Using Pythonic list comprehension, unpack
# all integers from 0 to 9 into a new list
my_list = [i for i in range(10)]
```

An argument could be made that a programmer who fully understands the usage of code idioms for a given programming language is fluent in that language, and is capable of utilizing said language beyond that which is similar to other languages. If the purpose of a programming course is to teach the student the ways of a specific languages, then evaluating their usage of code idioms could provide an indicator to whether they fully understand that language or not.

Analyzing and utilizing code idioms is also a much wider field than first anticipated, and have also been studied for much longer. For instance, *Programming with idioms in APL* by Perlis and Rugabe discusses the usage of code idioms in a symbol-rich language like APL, and was published in 1979 [33]; *Analyzing and compressing assembly code* by Fraser, Myers and Wendt, which was published in 1984, mentions the use of code idioms as a part of code compression [13]. Today, examples of code idiom usage include bug detection [17] and building code from templates [7], in addition to tools which can detect and extract code idioms shared between multiple projects [2].

The argument about code idioms being representative for a programmer's knowledge of a certain language is supported by Buschmann et al., who states that "*(...) we can say that idioms demonstrate competent use of programming language features. Idioms can therefore also support the teaching of a programming language*" [5]. This is put into practice with the tool *Pencil Code*, where common snippets and idioms are presented as graphical blocks to make it easier to learn the language in use [3].

2.2.5 Program synthesis

Program synthesis is the task of creating (the source code of) a computer program by utilizing a high-level program specification or template. This concept is explored within the field of Model Driven Development, where computer programs are created by a combination of high level graphical models and pre-written snippets of functional source code. In short, rules and models that fulfill certain requirements are created, which can be combined to describe and solve more complex problems.

While the aim of evaluating solutions to programming exams doesn't quite fall into the realm of "building new programs", there are aspects here which can be put to good use. If the professor or teacher writes source code as a solution to the programming task, this code can be used as a reference solution, while the students' answers will serve as the templates for program synthesis. By trying to build the *reference programs* with a student's solution as a basis, it is then possible to mark the number of changes needed for the template to be as *functionally equivalent* as possible to the reference solution.

Program synthesis is not wildly used as a form of code evaluation, but a 2013 paper from MIT [39] presented research into this field. Here the authors created a system to find and correct common errors made by students when solving assignment problems. While this tool is not created to evaluate or grade assignments it is still a base-line for other, potential projects. For example: rating the severity of the corrections made in assignment submissions, and deducting points away from a full score. The authors themselves concluded that "*We believe this technique can provide a basis for providing automated feedback to hundreds of thousands of students learning from online introductory programming courses that are being taught by MITx, Coursera, and Udacity,*" so there might be more research into this field in the coming years.

2.2.6 Benchmark testing

Another way to measure quality is to look at the program's performance when it is running. The term "benchmarking" is really an umbrella statement, not describing one specific way of measuring a program, but rather stating a goal of making the best and most efficient program possible. Benchmarking is not restricted to just software, and can also refer to hardware testing.

Some comparisons can be made between benchmarking and unit testing. Both utilize tests to check for very specific behaviour within the program, and the results from those tests are used as a basis for further improvements. The difference is that the result from a unit test can only be right or wrong, the developers know in advance which exact values the different tests should yield, while for a benchmark test the results are a performance indicator.

While high benchmark scores can be useful for validating the performance and efficiency of a program, it is not indicative of good source code.

2.3 Comparing evaluation strategies

Below is a quick summary and comparison between the different code evaluation strategies described in the above sections. All of these strategies have their own strengths, weaknesses and uses, also when seen through the scope of machine learning. Every one can be used to convert source code into a set of values, which in turn can be used when training the classifier. For example, unit testing can return scores for each function and method in a program, while the experimental software from MIT [39] can count the number of differences or similarities between a student program and the teacher's solution.

Name	**Industrial use**	**Educational use**
Unit testing	Design compliance	Scoring
Parsing	Compilers and interpreters	Plagiarism
NLP	Intent and meaning	Plagiarism
Idioms	Templates and patterns	Graphical code
Program synthesis	Templates and patterns	Experimental
Benchmarks	Performance testing	None

Table 2.1
A quick comparison between various code evaluation strategies.

For the purpose of this thesis we will focus on *one* of these strategies. Because of how exams are conducted, with limited time to ensure that your written code is runnable, it is unreliable to rely on output from any submissions, which rules out unit and benchmark testing. In addition to this, program synthesis is an experimental field, while NLP and idiom analysis seems to be, in our opinion, a bit too contextual. This leaves us with one option.

For the purpose of this thesis we will focus on source code parsing and the use of ASTs, which will be described in the next chapter. We will also be using benchmarks and performance testing, not directly in relation to the exam submissions, but as a method of finding the best possible parameters for our classification algorithms. This is covered in section 4.4.

Chapter 3

Grading and Features

3.1 The grading process

The way that grading is done today is a simple process, if not an easy one: After the students have completed their exams, the submissions are sent to the assessors (which could be the teachers) for grading, and after a while the grades are released to the students. Figure 3.1 illustrates this process. Every single exam submission has to be evaluated and graded manually, and, as an example, for NTNU's course *TDT4100 – Object-Oriented Programming* that is around 600 submissions each time.

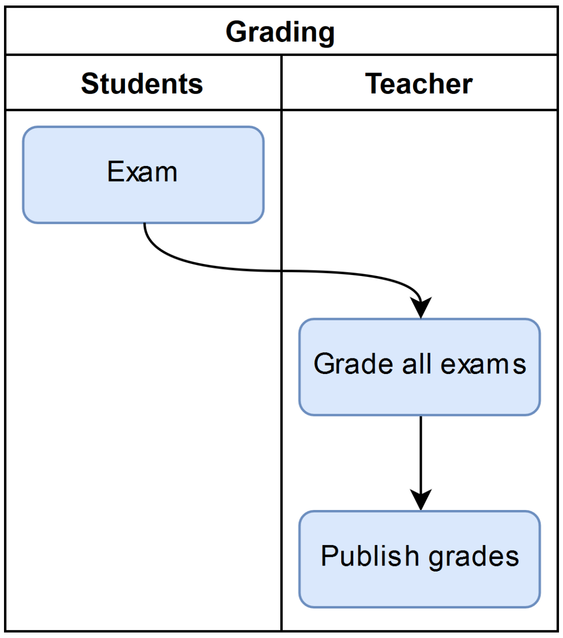

Figure 3.1
Today's grading process, done manually.

A new grading tool, designed to make this job easier is needed. By using this tool and handing over most of the work to it, only a fraction of the submissions has to be graded by the teacher. The concept is that each grader may use distinct grading patterns which might be so ingrained that they become difficult to explain to others, but which can be learned. The grading tool can pick up this pattern by analyzing a subset of manually graded exam submissions, and take over the job from there. Figure 3.2 shows this new process.

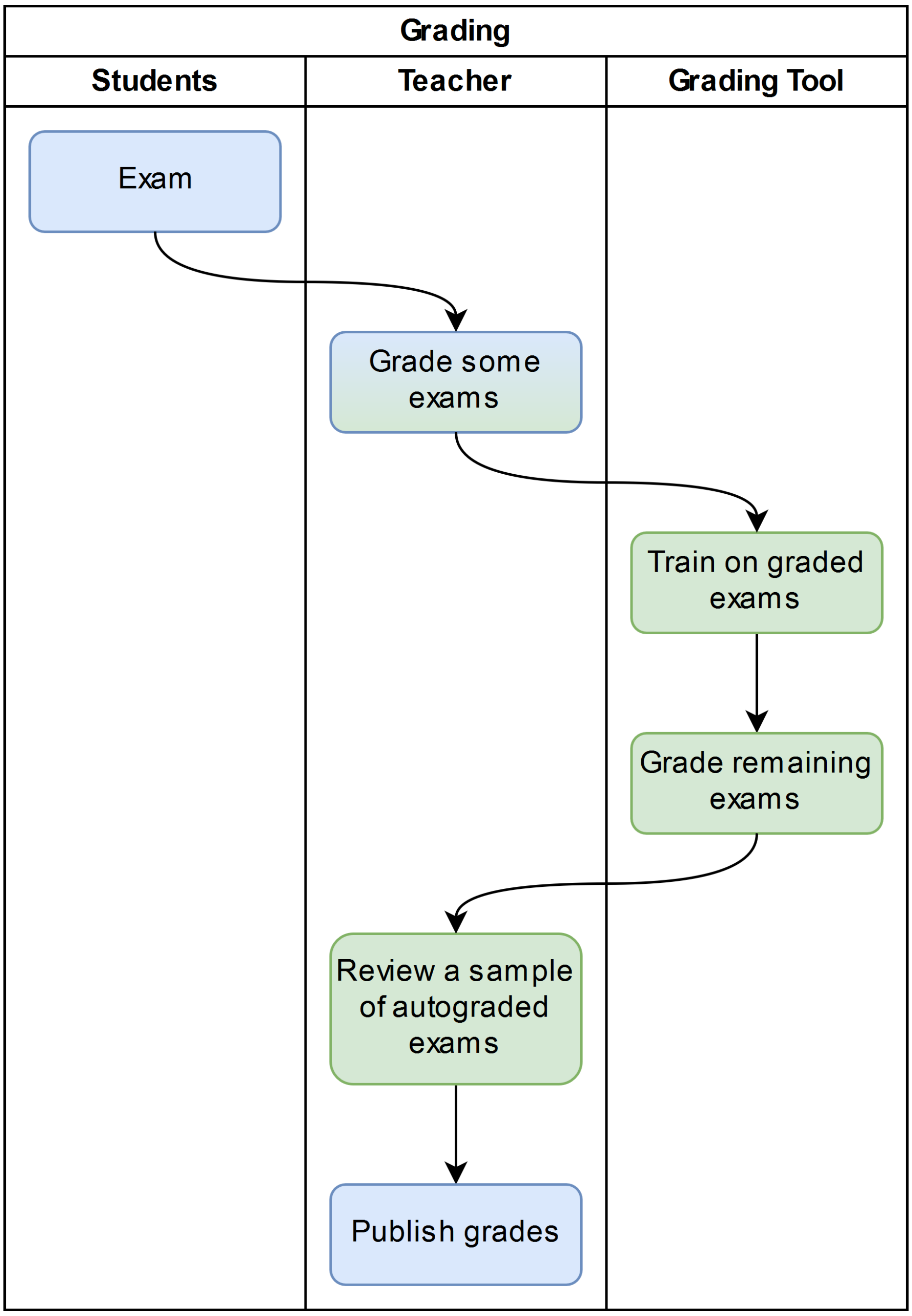

Figure 3.2
New steps are introduced (marked in green), letting the grading tool take over the majority of the work.

After a subset of exams has been manually graded, it is fed into the grading tool as a training set which will act as a template for how to grade the remaining exams. It is the teachers' responsibility to ensure that the training set is (fairly) evenly distributed, so that none of the grades are mis- or overrepresented in the training. Based on this training set the grading tool will give a grade to each of the remaining submissions. At last, to ensure the quality of the automated grading, a review needs to be done on a sample of the autograded submissions.

The rest of this chapter will go through the steps to design such a grading tool.

3.2 Feature extraction

The concept of machine learning and how that is implemented in this project will be discussed in further detail in chapter 4, but a short explanation will be given for context: To let the machine learning tool be able to give correct grades to exam submissions we need to tell the tool what to look for in the code. In other words, the *raw code* needs to be transformed into a set of *numerical features* so that out new grading tool can grasp what it is analyzing. The chosen approach was to find a set of features which could reflect two goals when analyzing code.

First off: *How complex is the code?* Subsection 2.1.1 mentioned how the industry seem to value readable code, so by bringing the concept of readability over to exam assessment we try to add something new to a tried and true process. Readability is linked with simplicity, and measuring simplicity is just another way of measuring complexity. So to get an idea of the readability and complexity of the code, we have chosen the Halstead complexity measures.

Second: *How similar is the exam submission to one of the teacher's solutions?* If we accept that the teacher's solution is the ideal answer, then anything that is similar to the solution should also be considered a good answer. Here we have chosen to determine similarity by calculating the cosine distance between the submissions' and solution's *term-frequency vectors* and *feature set vectors.*

Figure 3.3 shows how these features will be extracted, and the rest of this chapter will be used to describe the different features and parts of this figure.

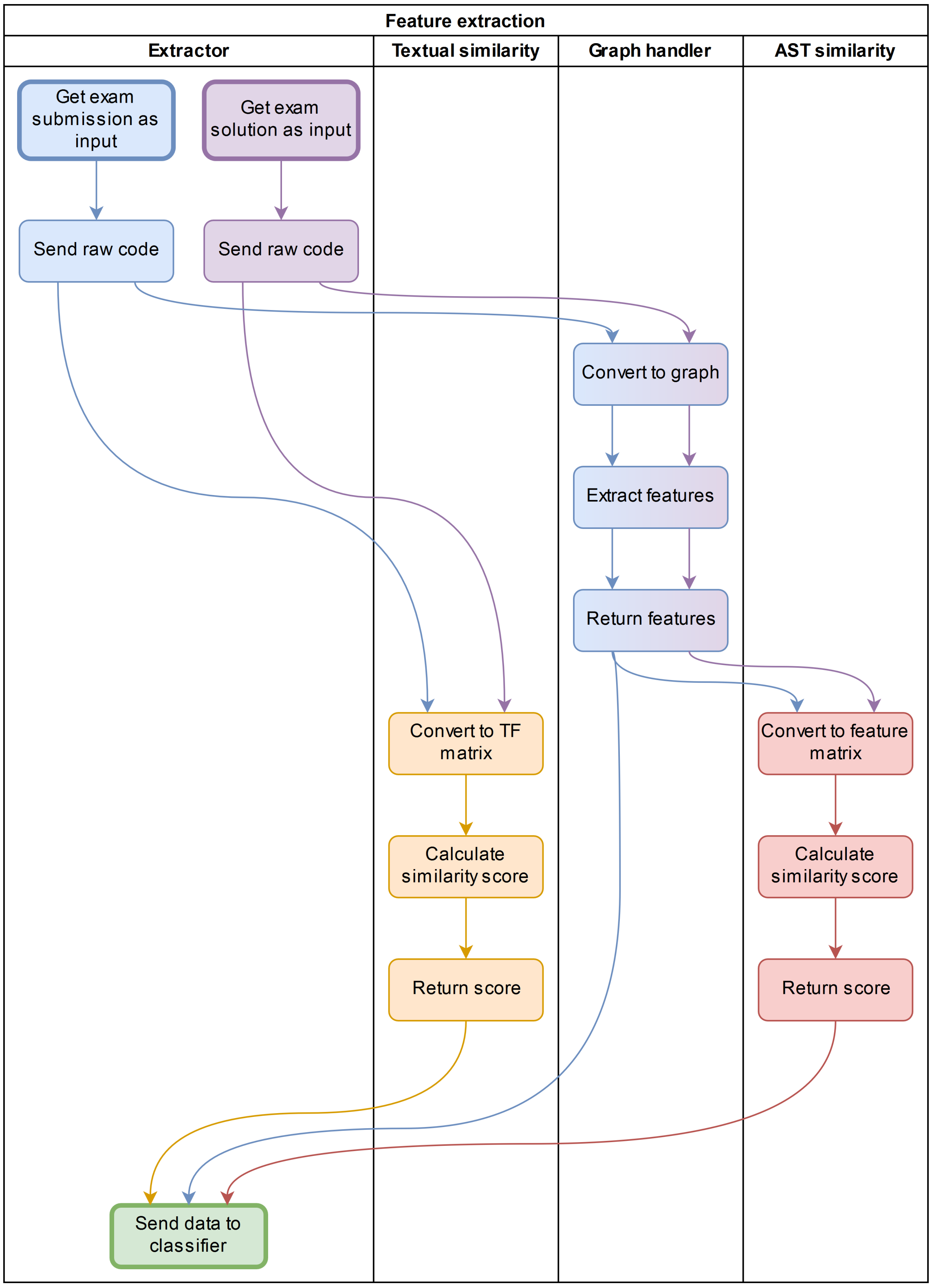

Figure 3.3
Flowchart showing the process of extracting features from exam deliveries and solutions. The complete feature set is a set of Halstead complexity measures and two similarity scores.

3.3 Parsing exam deliveries

If the first step was to collect data, then the second is to read through the students' code in such a way that features can be extracted. This could be achieved by a few lines of standard Python code if only the number of lines, words or characters were to be counted, but here we needed something to differentiate between variables, literal values, methods, keywords, etc. One way to accomplish this was to parse the students' code to create an *Abstract Syntax Tree* (AST), and search the AST for wanted patterns of nodes; this is represented by the third column (*Graph handler*) of figure 3.3. The tool chosen to generate and search through such a tree was ANTLR 4 [31].

Using ANTLR, every snippet of code would first be parsed with an already existing set of Java 8 grammars[4], and if that failed, parsed with a custom grammar[5]. This custom grammar was a lot more forgiving regarding missing semicolons and parenthesis, as well as code structure, while still creating an AST which could be used for feature extraction. Figure 3.5 shows an AST created with the standard Java 8 grammar, and it also shows that there are two minor errors in the code. These would stop the code from compiling and running, but they are not severe enough that ANTLR can't make a guess of what the rest of the AST should look like. If the errors were worse, the fallback custom grammar would have been applied instead.

After an AST is made it is subject to a depth-first search where each node (or subtree[6]) is analyzed, and relevant data is recorded. This data is used for calculating Halstead complexity measures, the AST similarity score and four extra features, all described in the following sections.

[4]The Java 8 grammar for ANTLR 4 was downloaded and used "as is" from the ANTLR GitHub page (`https://github.com/antlr/grammars-v4/tree/master/java8`) at the 12th of February, 2018.

[5]The custom grammar for ANTLR 4 was made by the author, with guidance and inspiration from T. Parr's book *The Definitive ANTLR 4 Reference* [30].

[6]In ANTLR 4, each node is either of type `ParseTree` or `TerminalNode`, indicating whether that node has children or not.

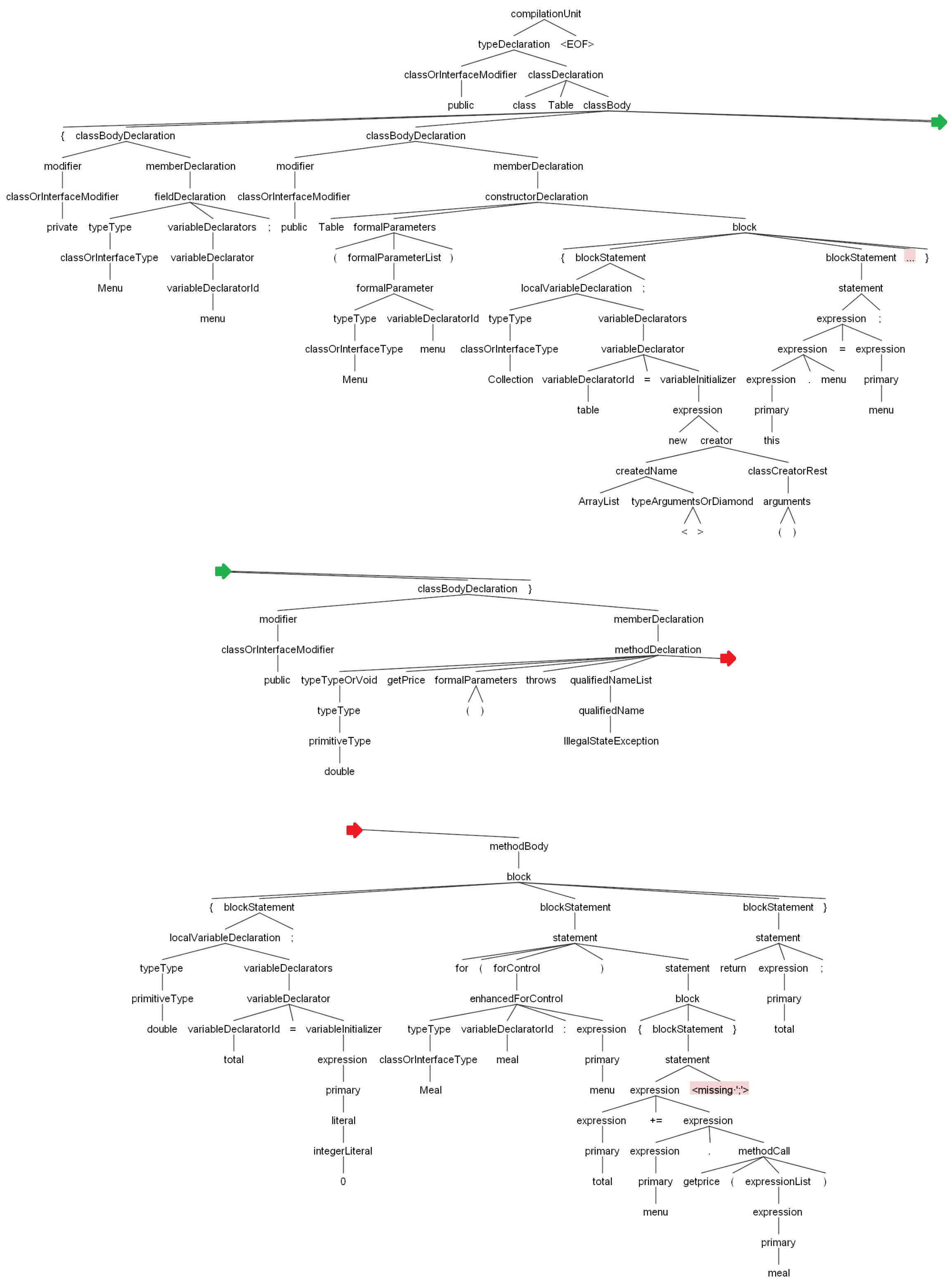
compilationUnit
typeDeclaration <EOF>
classOrInterfaceModifier classDeclaration
public class Table classBody
{ classBodyDeclaration classBodyDeclaration
modifier memberDeclaration modifier memberDeclaration
classOrInterfaceModifier fieldDeclaration classOrInterfaceModifier constructorDeclaration
private typeType variableDeclarators ; public Table formalParameters block
classOrInterfaceType variableDeclarator (formalParameterList) { blockStatement blockStatement ... }
Menu variableDeclaratorId formalParameter localVariableDeclaration ; statement
menu typeType variableDeclaratorId typeType variableDeclarators expression ;
classOrInterfaceType menu classOrInterfaceType variableDeclarator expression = expression
Menu Collection variableDeclaratorId = variableInitializer expression . menu primary
table expression primary menu
new creator this
createdName classCreatorRest
ArrayList typeArgumentsOrDiamond arguments
< > ( )
classBodyDeclaration }
modifier memberDeclaration
classOrInterfaceModifier methodDeclaration
public typeTypeOrVoid getPrice formalParameters throws qualifiedNameList
typeType ( ) qualifiedName
primitiveType IllegalStateException
double
methodBody
block
{ blockStatement blockStatement blockStatement }
localVariableDeclaration ; statement statement
typeType variableDeclarators for (forControl) statement return expression ;
primitiveType variableDeclarator enhancedForControl block primary
double variableDeclaratorId = variableInitializer typeType variableDeclaratorId : expression { blockStatement } total
total expression classOrInterfaceType meal primary statement
primary Meal menu expression <missing ';'>
literal expression += expression
integerLiteral primary expression . methodCall
0 total primary getprice (expressionList)
menu expression
primary
meal

Figure 3.4
Example of an AST generated by ANTLR. Split into three parts for readability. Colored arrows indicate where the tree is connected.

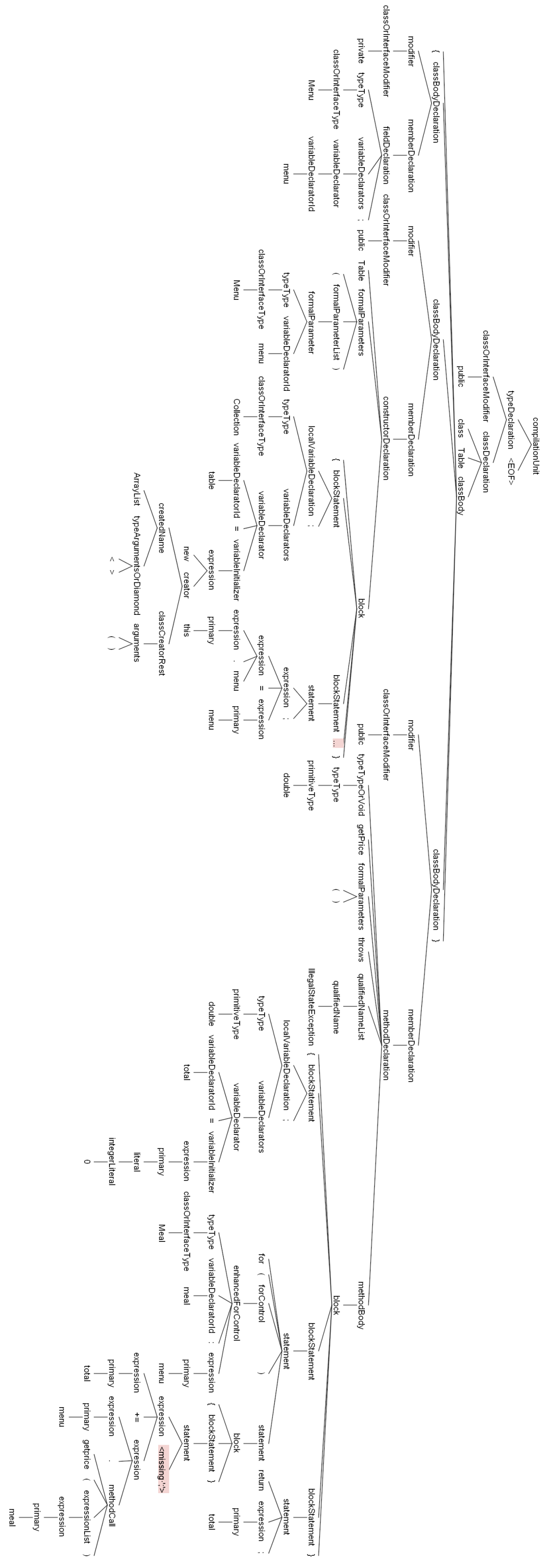

Figure 3.5
Example of an AST generated by ANTLR.

3.4 Halstead complexity measures

The *Halstead complexity measures* [15] is a set of software metrics used to measure properties of source code. While there is some debate about the usefulness of the Halstead measures, there are other researchers who continue to use them [23, 48]; Zhao et.al. said in their footnote about Halstead measures that *"We are aware that some researchers do not support the use of Halstead metrics. Our work to date has shown them to assist greatly in maintainability predictions."*; Leach [22] used Halstead measures as one of the methods to discover plagiarism within his students' code deliveries by comparing extremely similar measures. Also, the company Virtual Machinery [45] are using the *volume* and *effort* measures (described below) as part of their JHawk metrics tool for Java.

There are four key numbers used for these metrics:

η_1	$\rightarrow$	the number of distinct operators
η_2	$\rightarrow$	the number of distinct operands
N_1	$\rightarrow$	the total number of operators
N_2	$\rightarrow$	the total number of operands

The definition of what constitutes an operator and operand may vary, but here we are using the following: An *operand* is defined as an *identifier*, a *numerical literal* or a *string literal*, while an *operator* is everything else. The reasoning for such a broad definition of an operator is that because we are not dealing with production code, but simplified exam answers from an introductory programming course, we can assume that each answer contains no more information than necessary: Package declarations are almost never written, class definitions are not always present either, and methods are usually short and non-complex.

From η_1, η_2, N_1 and N_2 we calculate four additional metrics:

$$\begin{aligned}
\eta &= \eta_1 + \eta_2 && \rightarrow \text{program vocabulary} \\
N &= N_1 + N_2 && \rightarrow \text{program length} \\
\widehat{N} &= \eta_1 log_2 \eta_1 + \eta_2 log_2 \eta_2 && \rightarrow \text{calculated program length} \\
V &= N \times log_2 \eta && \rightarrow \text{program volume}
\end{aligned}$$

In addition to these eight complexity measures there are others which can be derived, but are not used as features in this project:

$$\begin{aligned}
D &= \tfrac{\eta_1}{2} \times \tfrac{N_1}{\eta_2} && \rightarrow \text{difficulty} \\
E &= D \times V && \rightarrow \text{effort required} \\
T &= E \,/\, 18 && \rightarrow \text{time to program (in seconds)} \\
B &= E^{(2/3)} \,/\, 3000 && \rightarrow \text{expected number of bugs}
\end{aligned}$$

These four measures are used to explain how likely errors might occur, how much effort and time (in seconds) it takes to implement a program, and how many bugs you are likely to find, respectively. The reason why these measures were dropped is that we view these as not intrinsic to the code itself, but as ways to measure *the programmer*. In the case of an exam, the difficulty, effort and time required to solve a problem are preset by the teacher (the last questions of an exam are often more difficult than the first; the time required to complete an exam usually is four hours), and the expected number of bugs is by extension not needed, as the measure from which it is derived (effort) is not used.

3.5 Code similarity

As mentioned in section 3.2 we will focus on two similarity measures. Both are calculated as the cosine distance between two vectors, where one represents a student's exam submission while the other represents a solution. The difference between these two similarity measures lies in how the vectors are created.

The first similarity measure is based on a statistic from information retrieval, Term

Frequency / Inverse Document Frequency, or TF-IDF. For the uninitiated, TF-IDF is a measure of words, symbols or numbers (collectively called tokens) in a document and collection of documents (corpus), where TF counts every word in any isolated piece of text (document) and IDF weights that count towards the the total number of occurrences in the corpus.

In this context, every answer to a subtask is a document, and the collection of answers from all students for a subtask is a corpus, i.e. there is one corpus per subtask. The TF for each document will be represented as a vector, or row in a table. The top row is the token names, while the rows below are the token counts for each document.

	for	**while**	***variable_1***	***variable_2***	**...**
document 1	2	1	3	4	...
document 2	1	2	5	1	...
document 3	0	3	2	2	...
...	...	...	...	...	...

Table 3.1
Example of token counts in a corpus

While IDF weighting is useful for finding identifying features in a document by giving less weight to words which appear in multiple documents in a corpus, the vocabulary of a programming language is rather restricted. This would essentially give no weight to all common Java keywords (and potentially no weight to intuitive variable names) while giving more weight to obscure variable and method names. With this in mind it was decided to drop the IDF weighting.

But having the TF matrix was not the end; a score needed to be given to each document (subtask) as to indicate its similarity with the solution document. First, the solution document was added to the TF matrix:

	for	**while**	***variable_1***	***variable_2***	**...**
solution	2	2	3	3	...
document 1	2	1	3	4	...
document 2	1	2	5	1	...
document 3	0	3	2	2	...
...	...	...	...	...	...

Table 3.2
Example of token counts in a corpus containing a solution document

Second, each document's row, including the solution's, was converted into a vector consisting of the term counts:

$$\begin{array}{rcl} \textbf{solution} & \rightarrow & [2, 2, 3, 3, \ldots] \\ \textbf{document 1} & \rightarrow & [2, 1, 3, 4, \ldots] \\ \textbf{document 2} & \rightarrow & [1, 2, 5, 1, \ldots] \\ \textbf{document 3} & \rightarrow & [0, 3, 2, 2, \ldots] \\ & \ldots & \end{array}$$

Figure 3.6
TF matrix as count vectors

And finally, each document was given a similarity score, calculated as the cosine similarity between the solution vector and that document's vector. If u and v are vectors, and n is the length of those vectors, then $distance$ is equal to

$$\frac{\sum_{i=1}^{n} u_i v_i}{\sum_{i=1}^{n} u_i^2 \sum_{i=1}^{n} v_i^2}$$

Because $distance$ is a value between 0.0 and 1.0, where a higher $distance$ means *less similar vectors*, then $similarity$ is simply the inverse of $distance$, or

$$similarity = 1 - distance$$

There is no direct advantage to using $similarity$ over $distance$, except for making the terminology a bit easier; a $similarity$ of 1 is, at least for this project, more intuitive than a $distance$ (or *dissimilarity*) of 0. If the cosine similarity between two vectors are 1, then those vectors, and documents, are identical. For simplicity, only the three example values are used in the calculation. When compared to the exam solution's vector, the result looks like this:

Document	**Vector**	**Similarity score**
document 1	[2, 1, 3, 4]	0.96676
document 2	[1, 2, 5, 1]	0.84536
document 3	[0, 3, 2, 2]	0.85617
...	...	...

Figure 3.7
Calculated similarity between the documents and the solution

3.6 AST similarity

The second similarity measure is a combination of the Halstead complexity measures, the cosine similarity and a few other measures extracted from the AST. To calculate this similarity score, a process was used similar to that which has already been described in this chapter:

1. Create abstract syntax trees for each subtask solution, as described in section 3.3. ASTs for the submissions are reused.
2. Extract Halstead measures from the solutions' ASTs (section 3.4).
3. Extract an extra three measures from *all* ASTs (submissions *and* solutions), indicating whether the code contains `for`-loops, `while`-loops and/or `if`-statements.
4. Create a vector for each subtask solution and submission, consisting of the Halstead measures and the three extra measures, ending up with a table similar to table 3.2 for each subtask.
5. Calculate the cosine similarity between the solution's and the deliveries' measures as described in section 3.5.

3.7 Extra features

There are four features in addition to those described previously in this chapter. These last features marks very specific attributes of the submitted code based on the fact that we are dealing with introductory level programming exams, and that some of the questions are designed to check for very specific types of programming knowledge and skill.

For example, there are questions which might ask the student to write a piece of code that iterates over a list of four integers and add 1 to each number. While in this case the correct answer would be to write a `for`-loop, the student might submit something like this:

```
void addOneToNumbers(int[] numbers) {
    numbers[0] += 1;
    numbers[1] += 1;
    numbers[2] += 1;
    numbers[3] += 1;
}
```

So to check for possible mistakes similar to this one, features measuring if for-loops, `while`-loops, `if`-statements are present, as well as the number of methods are introduced.

3.8 Data collection

3.8.1 Continuation exam

The purpose of a continuation exam is to give those who did not complete the standard exam a second chance before a new school year starts. Those eligible to take a continuation exam are usually students who withdrew from the original exam, could not attend because of medical issues or failed their previous attempt.

The dataset used in this project was from *TDT4100*'s continuation exam of 2017, which was at the time of writing the only digital exam available for this course. When having digital exams at NTNU, a system called Inspera Assessment [18] (shortened to "Inspera") is used. Inspera presents questions and accepts answers on various forms: scanned PDF documents, raw text, code, mathematical symbols, multiple choice answers, drag-and-drop as well as interactive tools (drawings, hotspot, etc). The continuation exam had answers delivered as code and raw text. The answer set (provided by Inspera) was a single JSON file which contained, in addition to some metadata, a list of all candidates and their respective submissions. Each submission, one per candidate, came with a final grade as well as a score and a max score for each subtask. A total of 45 students handed in their deliveries.

The exam structure was similar to exams from the past few years, consisting of 4 main tasks divided into a total of 17 subtasks. Of those 17, 11 subtasks required some form of code delivery, which meant that with 45 students taking this exam we had a potential 495 code responses in our dataset. As it were, only one of the 495 subtasks could not be preprocessed and parsed properly, so we ended up with 494 subtask deliveries ready to be used as training and testing data.

It is not unusual for the average grade after a continuation exam to be lower than its original. As the number of code responses for each grade was skewed towards F, there was an inherent risk of *overfitting* the training algorithms, that is, training the machine learning algorithm so well on a particular class or category that it becomes unable to recognize new or other patterns. This will be discussed in greater detail in chapter 6.

3.8.2 Privacy

The dataset provided by Inspera's contact person only contains one type of private information: the candidate number (CN) for each student taking the exam. No other information about the students is known. This CN is not related to NTNU's official student number, and no information is available which would allow a linking between the CN and other student details.

After registering this project with the Norwegian Centre for Research Data [29] (NSD) they concluded that even though the CN is sensitive and private information, the data collection could be conducted given that we were not able to link the CN to any additional information, and that the CNs should be anonymized after the project's end.

Chapter 4

Training and Prediction

4.1 Machine learning

Machine learning describes a system which is capable of teaching itself how to perform a task, without the programmers explicitly instructing the system how to do so. The term was first used by Arthur L. Samuel in *Some Studies in Machine Learning Using the Game of Checkers*[35], where he also said that "*Enough work has been done to verify the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program.*" This behaviour is extremely useful for pattern recognition; completing tasks where we know what kind of answer we want, but not exactly how to reach those answers.

4.2 scikit-learn

For this project, everything related to machine learning is handled by *scikit-learn* [32, 38], a set of tools for the Python programming language used for machine learning, data mining and analysis. One of the code examples in this chapter (specifically, the one in subsection 4.4.1) will show some of the functions from scikit-learn. Most notable are the functions `train_test_split`, `fit` and `predict`.

`train_test_split` takes lists as input and splits them up into new lists used for training and testing, meaning that a list of 12 elements will be returned as two lists with 9 and 3 elements, used for training and testing the classifier, respectively.

`fit` is the function used to train a classifier, where the training sets from `train_test_split` are used as input.

`predict` is used after training the classifier, and will predict classes for new input (predict grades for new exam submissions).

Similar descriptions are also written as comments within the appropriate code example.

4.3 Classification

The field of machine learning is a big one, and just like with human problem solving there is no ”one size fits all” solution. Each type of problem requires its own process, or class of algorithms, to work, and finding the best algorithm for your problem may simply be the case of trial and error. Most of the effort regarding machine learning goes into something other than actually coding the ”machine learning program”; experimenting with feature sets, trying different learning algorithms and tweaking the parameters of those algorithms is an important part of the job.

To help with the grading of code exams a kind of machine learning called *classification* was used. Classification is a type of problem solving which tries to answer questions like “*If all men wear hoodies and are generally taller then women, while all women wear pink clothing, what is the probability that a short person wearing a pink hoodie is a woman?*” This is a gross oversimplification, but it serves to illustrate the point: Given a set of known data and a finite group of categories, or *classes*, in which class does a new data-entry fit in with the highest probability?

The end result is perfect for this project, classifying each exam submission into 1 of 6 classes representing the grades A to F. However, there are some guidelines which can help to set a more clear path, and one of those is a flowchart from scikit-learn called the *scikit-learn algorithm cheat sheet* [37] (shown in full as figure 4.2 on page 30). The purpose of this flowchart is to give everyone a starting point based on which kind of dataset is being used, and what the output or result of the learning algorithm should be.

Based on the bottom line within the classifier category of figure 4.1, showing the top-left section of figure 4.2, we got two separate types of algorithms to use: *Linear Support Vector Classification* (Linear SVC) and *Naive Bayes*. The latter has three different implementations (or *decision rules*) in scikit-learn: Bernoulli, Gaussian and Multinominal[7]. Bernoulli Naive Bayes assumes binary features and will therefore not be used. In the end, three viable algorithms were used to classify exam submissions: Linear SVC, Gaussian Naive Bayes and Multinominal Naive Bayes.

[7]Descriptions of the different algorithms and implementations can be found at scikit-learn’s website [38].

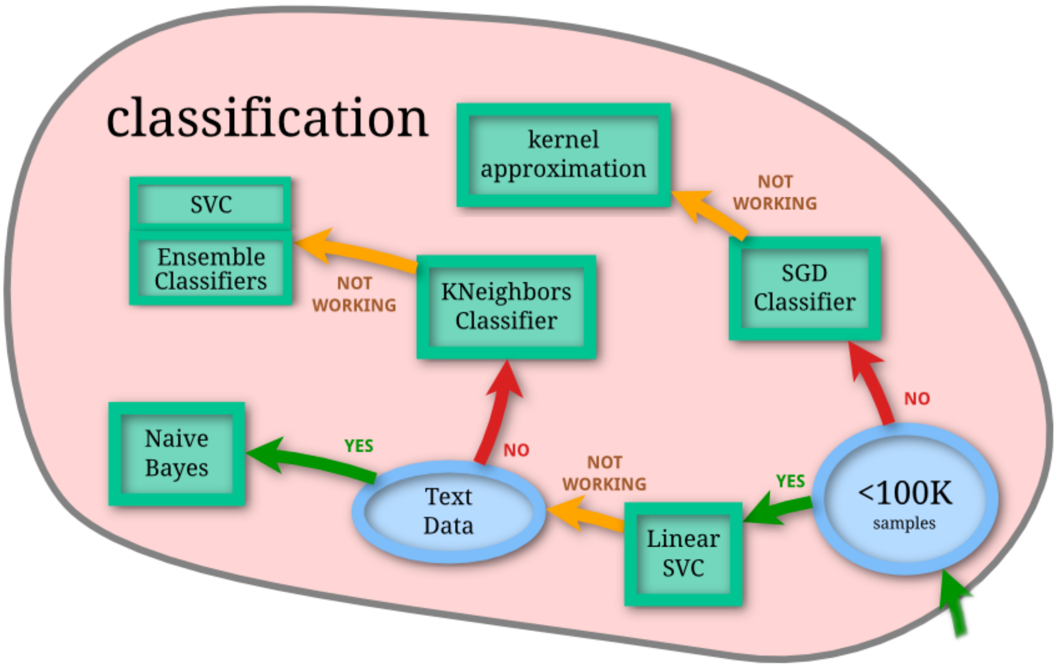

Figure 4.1
A section of figure 4.2, to help users specify a type of classification algorithm.

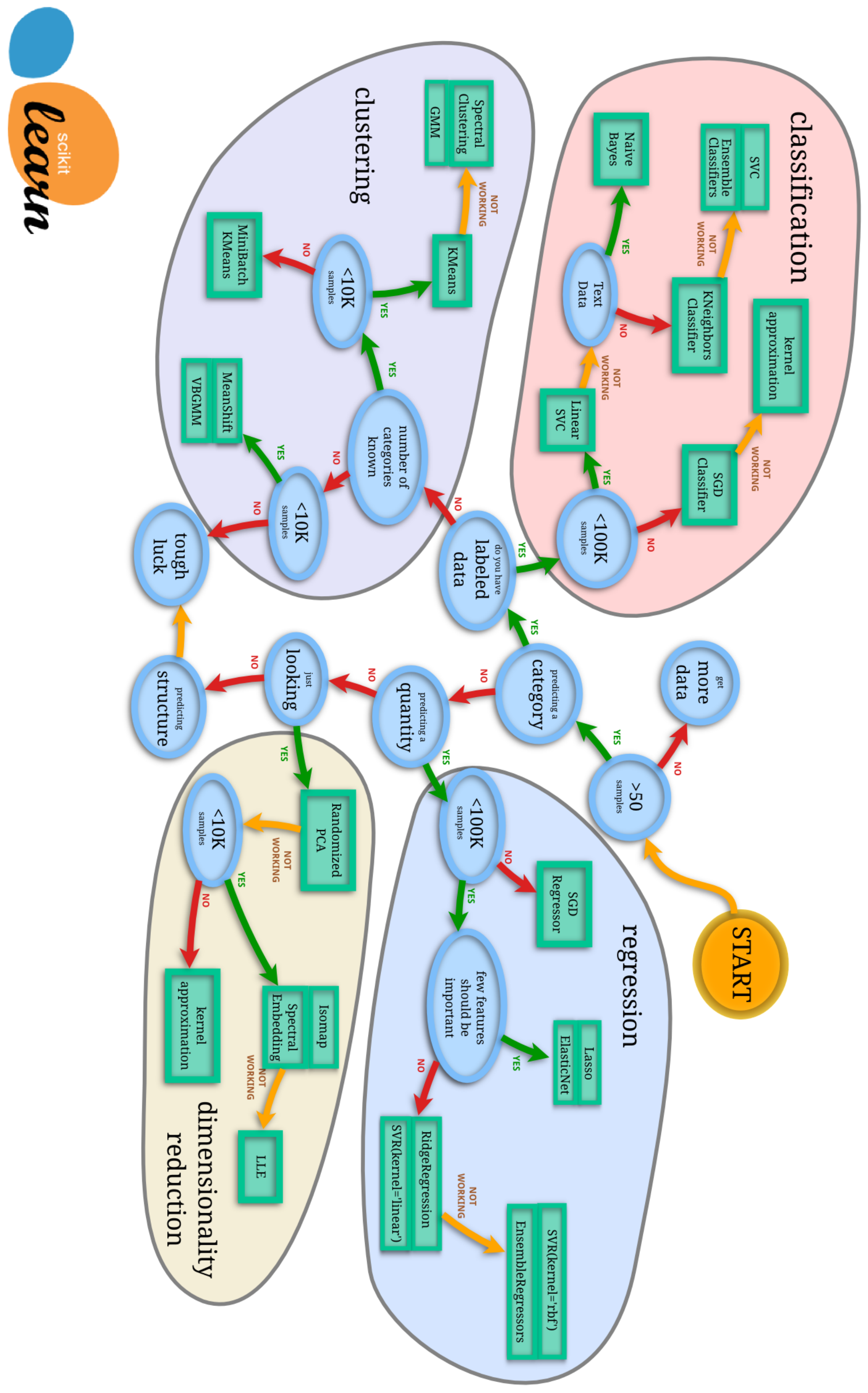

Figure 4.2
A flowchart from scikit-learn, to help users choose which type of algorithm they should use for their projects.

4.3.1 Training and testing

Back in chapter 3 it was illustrated through figure 3.3 how the relevant features was extracted from exam submissions and solutions, and it is in this chapter that the usage of these features will be shown. The most important thing to keep in mind is that while the text will refer to "training" and "testing", what lays behind those words are relatively simple concepts, which will be explained here. Another thing is that after the feature extraction process, the exam submissions (and solutions) no longer looks like code, but are lists of numerical values. As a short example: Given the simple code snippet 'HelloWorld.java' and an extraction process which finds the number of methods, variable references and comments, as well as the number of lines in the code, the code would go from this

```java
public class HelloWorld {
    public static void main(String args[]) {
        String message = "Hello World!";
        /* Print a message to the console. */
        System.out.println(message);
    }
}
```

to this

```
[1, 2, 1, 7]
```

Training, within the context of this thesis, is simply the act of giving a bunch of such lists to an algorithm, together with the correct grade for each list, and asking "*Hey! These lists represents some exams which were given these grades. Would you please try to find some pattern between all feature-lists with similar grade? Thanks!*" This leaves us with a classifier which hopefully was able to sense a pattern between all *A*s, *B*s, and so on.

Testing is when the trained classifier is given a new set of feature lists and asked "*Based on what you've already seen, could you please guess which grade these exams should have?*" The result from the algorithm's guessing (or *prediction*) is checked against a list of grades which are known to be correct, which in turn gives us the classifier's *prediction accuracy*.

4.3.2 Tweaking algorithms

One correction needs to be made from the previous section: Nothing is simple when it comes to machine learning. No matter how much data preparation has been done up until this point, there is at least one more thing that has to be done; tweaking the parameters of machine learning algorithms is more often than not essential to be able to get them to perform optimally, and a lot of time goes into finding the best set of algorithm parameters. Figure 4.1 shows a quick overview of the four algorithms, including the number of parameters that can be adjusted for each of them.

	Linear SVC	**Naive Bayes**		
		Bernoulli	**Gaussian**	**Multinominal**
Specialty	Loss and penalties	Boolean features	Distributed data	Text data
Computation	Vector distance	Probabilistic	Probabilistic	Probabilistic
Parameters	11	4	1	3
Used	✓	✗	✓	✓

Table 4.1
A quick overview of the four evaluated algorithms.

Linear SVC is the algorithm with the highest number of parameters, and thus the highest number of possible configurations. Many of these parameters have binary options (i.e. choose between two separate modes), and some are even mutually exclusive. In addition to this, only one of the options for the parameter `multi_class` will be used, as the other option is "*interesting from a theoretical perspective as it is consistent, [but] is seldom used in practice as it rarely leads to better accuracy and is more expensive to compute*"[38]. This means that the actual number of configurations is in reality not that high, and mostly depends on the granularity of the floating point parameters.

Both **Gaussian** and **Multinominal Naive Bayes** center around the parameter `priors` (called `class_prior` in MNB), which takes in a set of pre-defined probabilities for each class. If these probabilities are not given, as is the case here, they will be automatically generated based on the input data.

4.4 Grid-search and scoring

No matter which type of software is being made, the goal is always to end up with a tool that can outperform all previous iterations in some way or another. In the case of ”regular” software development a new iteration is complete when a list of specifications is implemented, and it is up to the developers to figure out how to implement every new specification. However, when creating a machine learning tool the goals will be clear, but the path to them are not so. As shown in table 4.1 there can be a multitude of different options to experiment with, given the right (or wrong) algorithm. To avoid having to do everything manually, a *grid-search* was implemented.

4.4.1 Grid-search

An example: Let us imagine an algorithm `A` which takes 3 parameters, `x`, `y` and `z`, which is a floating point number, a boolean and a string keyword, respectively; `x` can theoretically take any imaginable floating point number, `y` is restricted to the values `True` and `False` while `z` accepts one out of three keywords, `'alpha'`, `'beta'` and `'gamma'`. To make the grid-search possible we limit `x` to $[0.0, 0.2, 0.4, 0.6, 0.8, 1.0]$, which gives us a total of $6 \times 2 \times 3 = 36$ possible parameter combinations. Instead of manually testing every combination of parameters, we define a grid of all parameter names and their possible values. This grid is then passed to a special kind of classifier, `GridSearchCV`, which also takes in an algorithm to run the parameters with, in this case `A`.

```
# 'train_test_split' is a scikit-learn function which splits up a list
    of features (X) and a list of classes (y) into respective training
    and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y)

# Acceptable values for each parameter
grid_of_parameters = {
    'x': [0.0, 0.2, 0.4, 0.6, 0.8, 1.0],
    'y': [True, False],
    'z': ['alpha', 'beta', 'gamma']
}

# Passing an estimator plus our grid of parameters
classifier = GridSearchCV(
    estimator=A(),
    param_grid=grid_of_parameters
)
```

```
# Train the classifier on all possible parameter combinations
classifier.fit(X_train, y_train)

# Get the best results, and test on our training set
predicted_ys = classifier.predict(X_test)
```

`GridSearchCV`'s `fit`-function will then run through all parameter combinations, and train itself with the parameter set which gives the highest *score*.

4.4.2 Scoring

When rating the performance of a machine learning algorithm it is necessary to get some form of measure from the learning and prediction process. The score is based on the predictions the algorithm makes after its training, and scikit-learn comes with a variety of different scoring functions, as well as the ability to create such a scoring function from scratch. For this thesis both one pre-made and one custo, scoring function is used.

The first scoring function is the one which is default for Linear SVC and Naive Bayes, and is a simple score of accuracy. The scoring function counts the number of correct predictions and the total number of predictions, and returns the fraction. For example, given a prediction table for an arbitrary algorithm and parameter set (table 4.2) we see that 3 out of 5 predictions are correct, which gives this particular algorithm/parameter combination a score of 0.6, or 60 %.

ID	**Predicted class**	**Actual class**
0	'a'	'a'
1	'a'	'b'
2	'b'	'b'
3	'a'	'a'
4	'b'	'a'

Table 4.2
Example predictions for an arbitrary algorithm with random parameters

The second scoring function is a custom variety of the accuracy scoring, made to compensate for the fact that grades may not be evenly distributed, and that too many elements within a few number of classes may overfit the algorithm to only recognize those classes. This works as follows:

1. The algorithm is trained and tested.
2. A score is given for each class according to the accuracy scoring function.
   - The grades A through F are counted and scored according to the prediction accuracy for each grade.
3. Each class' score is weighted based on the disproportion of total class elements.
   - i.e. if there are fewer elements graded B than C, then B will be weighted higher than C.
4. At last, the final score is calculated as the sum of the weighted accuracy scores for all classes.

This score is only used internally for evaluating the effectiveness of the algorithm/parameter combinations, as it does not directly correlate with any percentage of accuracy. For example, given the predictions from table 4.2 the accuracy is 60 %, but with this weighted scoring function the internal score would be $2.36\overline{11}$; evaluating this as a percentage would give an accuracy of 236.11 %, which makes no sense. So to reiterate: The scoring function is a function used internally by `GridSearchCV` to rate the different combinations of algorithms and parameters, and should be seen as separate from the accuracy results presented in chapter 5.

Chapter 5

Experiment and Results

5.1 The experiment

All of the different components of the experiment has been described throughout this thesis, and will in this section be presented as one connected process. The goal of the experiment was to check how well the different classification algorithms, parameters and scoring functions would perform with our chosen dataset.

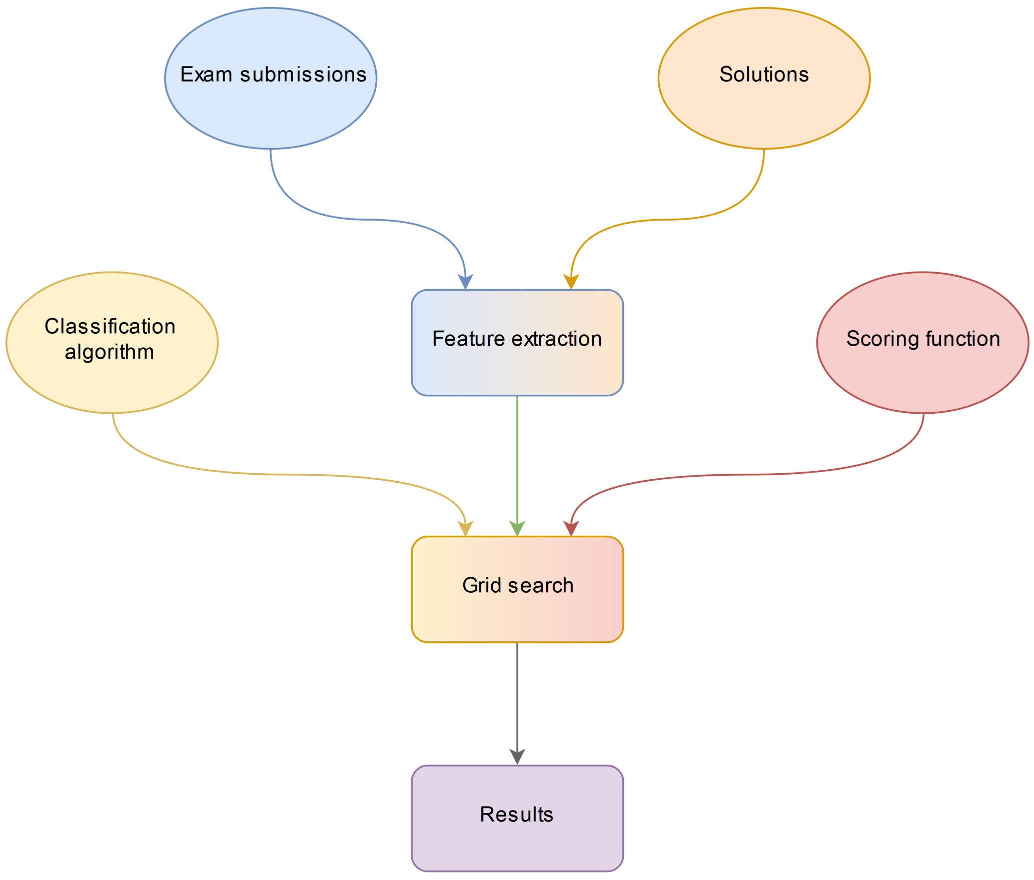

Figure 5.1
A depiction of all the necessary parts for the experiment.

Part 1 – Feature extraction: Using the methods from chapter 3 we extract a set of 10 features, which are responsible for representing each piece of code in a way the classifier can understand.

Feature name	**Feature type**	**Description**
vocabulary	int	The sum of the distinct number of operators and operands. The total number of distinct symbols or tokens in a program.
length	float	The sum of all operators and operands. Equivalent to counting all words, commas and periods in a paragraph of text.
calculated_length	float	An estimation of program length based on the distinct number of operators or operands.
volume	float	Measures the amount of information contained within the program. How much does a reader have to understand to comprehend the program's meaning.
code_similarity	float	Measures the similarity between a program submission and its solution, based on raw text.
ast_similarity	float	Measures the similarity between a program submission and its solution, based on measures from their Abstract Syntax trees.
num_methods	int	Counts the number of distinct methods in the program.
has_for_loop	boolean	Marks if the program contains `for`-loops or not.
has_while_loop	boolean	Marks if the program contains `while`-loops or not.
has_if_statement	boolean	Marks if the program contains `if`-statements or not.

Table 5.1
All features used for the experiment, and a description of them.

Part 2 – Grid-search: Each grid-search combined the exam submissions (represented as lists of the above features) with a scoring function and a classification algorithm (together with its related parameters).

Part 3 – Results: The results from each grid-search were tables containing a set of parameter values which the classifier deemed as optimal, and a set of prediction scores.

5.2 Table walkthrough

Section 5.3 shows tables of input ranges for each of the algorithms' parameters, where the possible values are different for each parameter: Square brackets are used around the allowed values for that parameter; the parameters accepting floating point numbers were all tested with the same range of numbers, shown in tables 5.2 and 5.4 as *Float range*, and is defined as $[10^x/2, 10^x]$ where $x \in [-4, 4]$ (in other words, *Float range* is the set $[0.00005, 0.0001, 0.0005, \ldots, 1000, 5000, 10000]$); parameters with no specified range is marked with *None*, meaning that the algorithm is using its default behaviour for that parameter; a single value indicates that it will be used for every combination of parameters.

The three tables in section 5.4 are each composed of four sub-tables, where the ones on the left represents parameter and scoring output for the standard scoring function, while the ones on the right are equivalent tables for the weighted scoring function. Details on how the scoring functions are implemented can be found in section 4.4. For each table, sub-tables *(a)* and *(b)* contains the optimal parameters for each combination of algorithm and scoring function, while sub-tables *(c)* and *(d)* shows the prediction results.

The results in *(c)* and *(d)* are shown as two values: *Accurate* and *One off*. The former is the ratio of correct predictions, while the latter is where we accept a misprediction by one grade in either direction, meaning that a prediction of C will be marked as correct if the real grade is B, C or D. The columns *In test set* and *Total* shows how many code submissions for each grade there were in the test set and whole dataset, respectively; they are identical for all tables, but are included for quick reference.

5.3 Input

Linear SVC

Parameter name	**Value ranges**
C	Float range
class_weight	None
dual	[True, False]
fit_intercept	[True, False]
intercept_scaling	Float range
loss	['hinge', 'squared_hinge']
max_iter	[100, 1000, 10000]
multi_class	['ovr', 'crammer_singer']
penalty	['l1', 'l2']
random_state	751
tol	Float range

Table 5.2
Value ranges for Linear SVC parameters

Gaussian Naive Bayes

Parameter name	**Value ranges**
priors	None

Table 5.3
Value ranges for Gaussian Naive Bayes parameters

Multinominal Naive Bayes

Parameter name	**Value ranges**
alpha	Float range
fit_prior	[True, False]
class_prior	None

Table 5.4
Value ranges for Multinominal Naive Bayes parameters

5.4 Output

Linear SVC

Parameter name	**Value**
C	0.0005
class_weight	None
dual	True
fit_intercept	True
intercept_scaling	10.0
loss	'squared_hinge'
max_iter	1000
multi_class	'ovr'
penalty	'l2'
random_state	751
tol	0.0001

(a) Parameters for standard scoring

Parameter name	**Value**
C	500
class_weight	None
dual	True
fit_intercept	True
intercept_scaling	5.0
loss	'hinge'
max_iter	10000
multi_class	'ovr'
penalty	'l2'
random_state	751
tol	0.0001

(b) Parameters for weighted scoring

Grade	**Correct**	**In test set**	**Total**
A	0	6	14
B	0	8	25
C	1	6	14
D	0	8	30
E	0	5	16
F	86	91	271
Accurate	70.16 %		
One off	75.00 %		

(c) Results for standard scoring

Grade	**Correct**	**In test set**	**Total**
A	0	6	14
B	0	8	25
C	2	6	14
D	1	8	30
E	0	5	16
F	77	91	271
Accurate	64.52 %		
One off	70.97 %		

(d) Results for weighted scoring

Table 5.5
Parameters and results of Linear SVC grid-search

Gaussian Naive Bayes

Parameter name	**Value**
priors	None

(a) Parameters for standard scoring

Parameter name	**Value**
priors	None

(b) Parameters for weighted scoring

Grade	**Correct**	**In test set**	**Total**
A	0	6	14
B	0	8	25
C	2	6	14
D	6	8	30
E	1	5	16
F	47	91	271
Accurate	45.16 %		
One off	55.65 %		

(c) Results for standard scoring

Grade	**Correct**	**In test set**	**Total**
A	0	6	14
B	0	8	25
C	2	6	14
D	6	8	30
E	1	5	16
F	47	91	271
Accurate	45.16 %		
One off	55.65 %		

(d) Results for weighted scoring

Table 5.6
Parameters and results of Gaussian Naive Bayes grid-search

Multinominal Naive Bayes

Parameter name	**Value**
alpha	5000.0
class_prior	None
fit_prior	True

(a) Parameters for standard scoring

Parameter name	**Value**
alpha	0.00005
class_prior	None
fit_prior	False

(b) Parameters for weighted scoring

Grade	**Correct**	**In test set**	**Total**
A	0	6	14
B	0	8	25
C	0	6	14
D	0	8	30
E	0	5	16
F	91	91	271
Accurate	73.39 %		
One off	77.42 %		

(c) Results for standard scoring

Grade	**Correct**	**In test set**	**Total**
A	1	6	14
B	0	8	25
C	2	6	14
D	2	8	30
E	1	5	16
F	79	91	271
Accurate	68.55 %		
One off	79.03 %		

(d) Results for weighted scoring

Table 5.7
Parameters and results of Multinominal Bayes grid-search

5.5 Result summary

The set of continuation exams used for this project's experiment consisted of 494 parseable code snippets from 45 different students. These were run through 3 different machine learning classification algorithms with 2 separate scoring functions and a multitude of parameter combinations. The best results came from the algorithm *Multinominal Naive Bayes* (table 5.7) where the most accurate prediction gave a hit-rate of 73.39 % using the standard scoring function; MNB with weighted scoring had a prediction accuracy of 79.03 % when including ”one off” results.

However, when looking at the distribution of grade predictions wee see that they are skewed in favour of F, which contains the most code snippets by far; the number of snippets marked F is almost 2.75 times larger than the set of A to E combined.

The two results with the lowest and highest accurate prediction values are interesting for a couple of different reasons. First off is the *Gaussian Naive Bayes* algorithm (table 5.6), which achieved the lowest scores for both ”accurate” and ”one off”, 45.16 % and 55.65 %, respectively. GNB also yielded the same results for both the standard and the weighted scoring function, the only algorithm of the three to do so. If we compare the distribution of results to the other two algorithms, we see that GNB appears to have done some sort of trade-off between D and F. While F seems to have been overfitted for the other algorithms, it only achieved a prediction accuracy of 51.65 %. D, on the other hand, has its highest score of any other grade (except for F) in any other result table: 75 %.

Second is the MNB with standard scoring, which had the highest accurate prediction rate of 73.39 %. This run is the only one with a 100 % match for F, and also with a 0 % match for every other grade. If we add the predictions for E and F together, and calculate for ”one off”, we get

$$\begin{aligned}(5 + 91) \ / \ (6 + 8 + 6 + 8 + 5 + 91) \\ = 96 \ / \ 124 \\ \approx 0.7741935 \\ \approx 77.42\%\end{aligned}$$

which is exactly what the ”one off” results are. This points to the algorithm guessing F for every member of D, which, by extension, could point to a case of overfitting, and possibly a prediction of F for every code snippet in this run.

Chapter 6

Discussion, Conclusion and Future Work

6.1 The experiment

Both of the Naive Bayes algorithms presented some points of interest. In the case of GNB, the results from tables 5.6c and 5.6d shows that there are some values for `priors` which will work better for grades other than F. It would be interesting to see if repeating this experiment with a focus on statistical pre-analysis could have any viable impact on prediction results, if the score would be the same but with a more balanced spread, or if the scores could become better overall. The same could be said for MNB; we relied on scikit-learn's ability to find a good statistical spread for `class_prior`, as with GNB's `priors`.

Another possible variant of Naive Bayes classification would be to split up the grading prediction, making six distinct classifiers where each would deal with a binary classification, i.e. *grade_A_classifier* would check if a submission matched for an A or not, *grade_B_classifier* would check for a B, etc. Training a classifier solely to predict for example A-graded code could potentially lower the negative effects of a skewed dataset, as each subgrader would be expected to recognize a smaller (an more manageable) set of patterns. It could also be easier to identify outliers, since that would be the submissions not recognized by any of the classifiers.

The runs conducted with Linear SVC were overall the most average. There were some expectations for these grid-searches to perform better than the two other algorithms, simply because of the sheer number of parameter combinations in addition to there being no special tweaking for the `priors` or `class_prior` parameters. Again, it would be interesting to see how this would perform with another dataset.

6.2 The dataset

Multiple options for which dataset to use was considered, and three alternatives stood out as plausible candidates; standard exams, continuation exams and assignment submissions. As became evident when these datasets were examined they all had their own good and bad sides, which presented three scenarios:

1. **Dataset:** Standard exams.
   Advantage: Large dataset, standard grade distribution.
   Disadvantage: Analogue; written with pen and paper.
   Solution: Manually re-writing every exam submission to a digital format.

2. **Dataset:** Continuation exams.
   Advantage: Digital.
   Disadvantage: Small dataset with a skewed grade distribution.
   Solution: No immediate solution possible.

3. **Dataset:** Assignments.
   Advantage: Very large dataset.
   Disadvantage: Not graded, and tasks are different from a standard exam.
   Solution: Manually grading assignment submissions.

Option 1 was heavily considered as it would, if completed, give the most reliable dataset; option 3 was also considered, because even though the assignment deliveries were different than the code from an ordinary exam, this code base was larger than the smaller-than-hoped set of continuation exams. But the main issue with these two options was time. These tasks, even doing just one of them, would be too time-consuming to do by ourselves, and were eventually dropped as it proved too difficult to find additional resources and assistance for re-writing or grading. In the end, the dataset of continuation exams was chosen as the basis for training and testing data because it was ready to use.

6.3 Conclusion

This Master thesis has explored the possibility of creating an automatic grading tool to make the process of grading programming exams more efficient and reliable. Three questions were asked at the beginning, and now those have been answered:

- *What kinds of source code evaluation techniques exists today, and how can they be used for machine learning?*

There are a variety of different methods used for evaluating source code, from assessing the code's structure and syntax, to only focusing on the compiled program's performance or output, all of which have their own interpretation of what constitutes "good code" and how to make code better. We have mentioned 5 such methods (6, including benchmarks) in this thesis, reported their uses for industry, education and research, as well as given our own view in regards to how and if they could be used for machine learning and exam evaluation.

- *Which features and metrics are viable for an objective evaluation of source code?*

The purpose of an exam is to give as objective an assessment as possible, and to accomplish this we selected a set of objective features to use as a basis for the evaluation process. These features consisted of Halstead measures, two separate similarity scores and four metrics meant to check for discrepancies between a task and a student's submitted solution.

- *How would a machine learning classifier perform when grading programming exams?*

To answer this question an experiment using the described system was conducted on a dataset consisting of exam submissions from the *TDT4100* continuation exam of 2017. The results from this experiment were inconclusive, which we believe to come from the fact that the dataset was suboptimal, and ended up overfitting the classifier.

6.4 Future work

Automatic evaluation, assessment and feedback for source code is a growing field, with multiple projects trying to expand its usefulness in different ways. The purpose of this Master Thesis was to look into the possibility of using machine learning on university level programming exam submissions, and train a classification system to give out grades for said exams.

More work needs to be put into studying how and when to use machine learning as an evaluation tool. Not only within the context of education and exams, but also how we wish that the finished product should be able to operate. There are still many unexplored options for how to use different software evaluation strategies for systems driven by machine learning.

The features used in this thesis for evaluation and classification were selected based on objective metrics, revealing no information about the code's quality. While this was done deliberately, giving all responsibility of distinguishing between ”good” and ”bad” to the classifier, it would be interesting to see how using *subjective* measures would impact such a system.

And, most importantly, to reach more conclusive results for this experiment it should be rerun with a new set of exam submissions, preferably when a digital version of the standard exam is available.

References

[1] M. Agrawal and D. K. Sharma, "A state of art on source code plagiarism detection," in *Next Generation Computing Technologies (NGCT), 2016 2nd International Conference on.* IEEE, 2016, pp. 236–241.

[2] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 472–483.

[3] D. Bau, D. A. Bau, M. Dawson, and C. Pickens, "Pencil code: block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children.* ACM, 2015, pp. 445–448.

[4] M. Bloch, S. Blumberg, and J. Laartz, "Delivering large-scale it projects on time, on budget, and on value," *Harvard Business Review*, 2012.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, A System of Patterns*, ser. Pattern-Oriented Software Architecture. Wiley, 2013, ch. 4, pp. 345–346. [Online]. Available: https://books.google.no/books?id=j_ahu_BS3hAC

[6] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, pp. 31–35.

[7] R. Clifton-Everest, T. L. McDonell, M. M. Chakravarty, and G. Keller, "Embedding foreign code," in *International Symposium on Practical Aspects of Declarative Languages.* Springer, 2014, pp. 136–151.

[8] Codecademy. (2011–) Codecademy. https://www.codecademy.com/.

[9] E. Dietrich. (2013) Employers: Put your money where your mouth is. https://www.daedtech.com/employers-put-your-money-where-your-mouth-is/.

[10] Eclipse Foundation. (2001–) Eclipse ide. https://www.eclipse.org/ide/.

[11] J. L. Elshoff and M. Marcotty, "Improving computer program readability to aid modification," *Commun. ACM*, vol. 25, no. 8, pp. 512–521, Aug. 1982. [Online]. Available: http://doi.acm.org/10.1145/358589.358596

[12] H. Erdogmus, M. Morisio, and M. Torchiano, “On the effectiveness of the test-first approach to programming,” *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, March 2005.

[13] C. W. Fraser, E. W. Myers, and A. L. Wendt, *Analyzing and compressing assembly code.* ACM, 1984, vol. 19, no. 6.

[14] D. Gitchell and N. Tran, “Sim: a utility for detecting similarity in computer programs,” in *ACM SIGCSE Bulletin*, vol. 31, no. 1. ACM, 1999, pp. 266–270.

[15] M. H. Halstead, *Elements of software science.* Elsevier New York, 1977, vol. 7.

[16] P. Hamill, *Unit test frameworks: tools for high-quality software development.* ”O’Reilly Media, Inc.”, 2004.

[17] D. Hovemeyer and W. Pugh, “Finding concurrency bugs in java,” in *Proc. of PODC*, vol. 4, 2004.

[18] Inspera. (2016–) Inspera assessment. http://www.inspera.com/.

[19] JBoss Forge. (2013–) Roaster. https://github.com/forge/roaster.

[20] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 96–105.

[21] A. Kuhn, S. Ducasse, and T. Gîrba, “Semantic clustering: Identifying topics in source code,” *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[22] R. J. Leach, “Using metrics to evaluate student programs,” *SIGCSE Bull.*, vol. 27, no. 2, pp. 41–43, Jun. 1995. [Online]. Available: http://doi.acm.org/10.1145/201998.202010

[23] M. Madhan, I. Dhivakar, T. Anbuarasan, and C. Thirumalai, “Analyzing complexity nature inspired optimization algorithms using halstead metrics,” in *Trends in Electronics and Informatics (ICEI), 2017 International Conference on.* IEEE, 2017, pp. 1077–1081.

[24] A. Marcus and J. I. Maletic, “Identification of high-level concept clones in source code,” in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on.* IEEE, 2001, pp. 107–114.

References

[25] R. C. Martin, *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[26] K. Meyer. (2016) How do websites such as codecademy.com and teamtreehouse.com evaluate user entered code? – Answered by Kyle Meyer. https://www.quora.com/How-do-websites-such-as-codecademy-com-and-teamtreehouse-com-evaluate-user-entered-code.

[27] Microsoft. (2016–) Unit test basics. https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics.

[28] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Software Engineering*, vol. 13, no. 3, pp. 289–302, 2008.

[29] Norsk senter for forskningsdata. (1971) NSD – Norsk senter for forskningsdata. http://www.nsd.uib.no/.

[30] T. Parr, *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2013. [Online]. Available: https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference

[31] ——. (2010–) Antlr. http://www.antlr.org/.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] A. J. Perlis and S. Rugaber, "Programming with idioms in APL," in *ACM SIGAPL APL Quote Quad*, vol. 9, no. 4. ACM, 1979, pp. 232–235.

[34] Python Software Foundation. (2001–) Unit testing framework. https://docs.python.org/3/library/unittest.html.

[35] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, July 1959.

[36] R. S. Sangwan and P. A. Laplante, "Test-driven development in large projects," *IT Professional*, vol. 8, no. 5, pp. 25–29, 2006.

References

[37] scikit learn. (2007–) Choosing the right estimator. http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html.

[38] ——. (2007–) scikit-learn: Machine learning in python. http://scikit-learn.org.

[39] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 15–26, 2013.

[40] D. Talby, A. Keren, O. Hazzan, and Y. Dubinsky, "Agile software testing in a large-scale project," *IEEE software*, vol. 23, no. 4, pp. 30–37, 2006.

[41] The JUnit Team. (2000–) Junit. http://junit.org/.

[42] Treehouse Island, Inc. (2011–) Treehouse. https://teamtreehouse.com/.

[43] H. Trætteberg. (2010–) Jexercise - test-based java exercise. https://github.com/hallvard/jexercise.

[44] D. van Bruggen. (2011–) Javaparser. http://javaparser.org/.

[45] Virtual Machinery. (2017–) Virtual machinery. http://www.virtualmachinery.com.

[46] WeDoTDD, LLC. (2016) We do tdd. http://www.wedotdd.com/about.

[47] Wikimedia Foundation. (2017) List of unit testing frameworks. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java.

[48] L. Zhao and J. Hayes, "Predicting classes in need of refactoring: an application of static metrics," in *Proceedings of the 2nd International PROMISE Workshop, Philadelphia, Pennsylvania USA*, 2006.

Appendix A

Alternatives

This appendix is meant as a supplement describing two alternatives which were used or considered at some point during this project, and is not vital for any potential replication. However, the following sections can be used in part as an inspiration or a guideline for later projects.

A.1 Data collection

In addition to using exams it was also considered basing the auto-grading on assignment submissions. The evaluated assignment set consisted of tasks labeled 00 to 10, where each task was divided into several subtasks. The tasks were given out to the students on a weekly basis, where 00 was the introductory assignment, and 10 was the last assignment, incorporating some earlier topics into that of game development. 5 out of these 11 assignments were deemed suitable for this project based on their topics; these 5 assignments covered the majority of the curriculum, and was comparable to tasks given at various *TDT4100* exams:

- 01 – Objects and classes, state and behaviour
- 02 – Encapsulation and validation
- 05 – The structure of objects
- 08 – Observer-observed and delegation
- 09 – Inheritance and abstract classes

A.1.1 Comparing datasets

The structure of the **continuation exam** was described in section 3.8: "*[The continuation exam of 2017 consisted] of 4 main tasks divided into a total of 17 subtasks. Of those 17, 11 subtasks required some form of code delivery, which meant that with 45* *students taking this exam we had a potential 495 code responses in our dataset. As it were, only one of the 495 subtasks could not be preprocessed and parsed properly, so we ended up with 494 subtask deliveries ready to be used as training and testing data.*"

A portion of the **hand written exams** from 2017 were made available, with more on the ready if needed. Even though the responses on this exam would have been preferable over the responses on the continuation exam, the fact that everything was hand written made them very hard and time consuming to work with: Every subtask from every exam delivery would have had to be re-written as a digital copy for them to be useful.

The **assignments** for *TDT4100 – Object-Oriented Programming* is set up in such a way that there are more subtasks available than required to be eligible for taking the exam. For example, in each of the 11 assignments there might be 5 subtasks to complete, but only one or two are required to pass the assignment. Because of this, some students have delivered a lot more code than others. Out of 474 students (with at least one submission):

- 153 students delivered at least one subtask per task
- 54 students delivered at least two subtasks per task
- 10 students delivered at least three subtasks per task
- 321 students had one or more tasks where no subtasks were delivered

No exact count was made of how many subtasks in total were delivered, but the dataset contained 6702 separate files. In some cases there were multiple files per delivered subtask, but there were 6702 code snippets nonetheless. This dataset was over 13.5 times larger than the one from Inspera, but it was missing a very important feature: it was not graded. Instead, the students' code had to go through a series of unit tests to be marked as "Passed" (in theory each submission should have been marked with a score from 0 to 100, which could have been converted to a grade; in practice most were marked as a flat 0 or a flat 100). As mentioned in subsection 2.2.1, the course *TDT4100 – Object-Oriented Programming* utilizes an in-house plugin for the IDE Eclipse [10] called JExercise which, in addition to running the students' code through the unit tests, presents the students with more detailed information about the assignment, each task and the tests for that task. Because of this major difference in code validation between these assignments and an exam, a classifier would most likely not yield the most accurate results for exam grading if trained with this dataset.

A.2 Parsing exam deliveries

There are many tools available for parsing source code, and while JavaParser [44] and Roaster [19] seemed like good and solid options for parsing Java code, the choice fell on ANTLR [31] for one simple reason: ANTLR is not restricted to just Java.

While not explicitly stated in the thesis, the original intent was to use digitally re-written pen-and-paper exams. When writing source code on a computer, using a full IDE or modified text editor, many of the small errors become clear immediately after making them, for example a missing parentheses or spelling errors. These luxurious correction features are not available at an exam, which means that it is common for code deliveries to contain some amount of errors. This would also trip up any parser trying to parse the code as ”proper Java”; if a statement in Java doesn't end with a semicolon, then the entire statement is wrong. This could be circumvented by writing custom error handlers for the parser, letting a special-case-parser handle the incorrect code segments or writing a more forgiving Java grammar to be used by ANTLR.

The idea behind writing our own Java grammar was that a good deal of customization would have to be done in either case, and a new grammar would allow for creating a more relaxed set of rules for Java code, where for instance

- parenthesis, brackets and square brackets are (mostly) optional,
- semicolons are ignored, and
- methods and variables does not have to be written inside a class.

But, as described in section 6.2, the chosen dataset ended up not being the one consisting of pen-and-paper exams, but one where the code was written digitally. After a review of the dataset it became clear that most of the code could be parsed by a standard parser, which in this case ended up being ANTLR with a pre-written grammar for Java 8. The special case grammar was still kept as a backup, as mentioned in section 3.3.

Appendix B

Acronyms and Abbreviations

AST	Abstract Syntax Tree
CLF	Classifier
CN	Candidate Number
DFS	Depth-First Search
GNB	Gaussian Naive Bayes
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LSVC	Linear Support Vector Classification
MNB	Multinomial Naive Bayes
NLP	Natural Language Processing
NSD	Norsk senter for forskningsdata
NTNU	Norges Teknisk-Naturvitenskaplige Universitet
TFIDF	Term Frequency / Inverse Document Frequency