UNIVERSITEIT UTRECHT

SOGETI

COMPUTING SCIENCE

MASTER THESIS RESEARCH

Automated Grading of Java Assignments

Author
Nadia Boudewijn
Student Number
3700607

First Supervisor
prof. dr. Johan Jeuring
Second Supervisor
dr. Ad Feelders
Daily Supervisor
Erwin de Gier

July 14, 2016

UTRECHT UNIVERSITY

Abstract

Faculty of Science
Graduate School of Natural Sciences

Master of Science

AUTOMATED GRADING OF JAVA ASSIGNMENTS

by Nadia Boudewijn

This study explores the influence of Java code features on the prediction accuracy of manual grades. In particular, we follow the high level definition of the proposed feature grammar by Aggarwal and Srikant and provide a new low level interpretation[45]. Through a series of experiments we explore the influence of changing the feature granularity. In order to utilize predictive models the source code of Java solutions has to be converted into suitable input. In this regard we develop a feature generation tool for Java code, named JFEX. The grading process of the solutions follows a distinctive grading rubric which is more solution oriented than the originally proposed rubric.

We empirically test if the algorithm-oriented features are able to capture these different grade distinctions and verify if this improves the grading accuracy in comparison to test case predictions. In the end this work did not provide significant proof that automated test case based grading is improved by feature modeling. However, we did demonstrate some encouraging evidence that shows the features have the *capacity* to improve test case accuracy. A subset of the selected features seemed highly relevant to the problem at hand. Classification modeling with attention for the ordinal ordering between the grade levels presented itself as the best candidate to realize the potential of the features.

Contents

Chapter 1

Introduction

1.1 Origin

When starting the process of writing my master's thesis the wide range of fields covered by Computing Science was a blessing in disguise. At first I was overwhelmed by the seemingly infinite number of directions I could take. After attending over thirty colloquium presentations I still had not seen a subject that truly captured my interests. I pondered this problem for quite some time before it dawned on me that my study allows me to shape my thesis to my own interests!

Somehow this not-so-bright insight opened a door for me to concretize my ideas for a research thesis. The one subject that never ceases to amaze me is Artificial Intelligence. Having a bachelor degree in AI I realize it is quite idiotic to refer to Artificial Intelligence as a single subject. When narrowing down I arrived at the concept of learning or predicting from data and/or environmental feedback. Although interesting, this line of subjects was not quite cutting it for me. I switched from AI to Computing Science for several reasons; one of the first being my new-found enjoyment of object oriented programming in my first year of study. No need to ponder anymore: this is it. I want to combine machine learning with object oriented programming.

As luck would have it, there was a vacancy for an internship at Sogeti titled "automated developer hiring test". The very general description certainly left room for personal interpretation. Mind boggled by questions as "what defines programming skill" and "how can one measure programming skill" I spent quite some time researching the existing literature. From knowledge modeling with Bayesian networks to tracing patterns in the programming processes of humans, I considered it all. Through some sort of natural work flow I arrived at my current research niche which I will describe and motivate in detail in this paper.

1.2 Collaboration

For almost all of my research ideas, including the current one, data from eligible test candidates has to be collected. This introduces a very high risk factor of failing to gather enough data needed for a decent analysis within the time span of a master's thesis. This risk has been eliminated by a collaboration with Codility. The next two subsections introduce Sogeti and Codility, and their relation to this project. The motivation and details of

this thesis will be explicated in detail in the remainder of this chapter and those to follow.

1.2.1 Sogeti

With a focus on craftsmanship and involvement Sogeti has managed to become one of the top ten IT-services in the Netherlands. Sogeti designs, builds, implements, tests, and manages IT solutions. On their quest to provide tomorrow's solutions for today's problems, innovation has become a core value within Sogeti.

One of the ways this innovative attitude shines through are the offered internships, graduation projects, and traineeships. This allowed me to transform a Java graduation project at Sogeti into a machine learning experiment. The functional and creative support of Sogeti has truly raised this project to a higher level.

1.2.2 Codility

Striving to make the hiring process of programmers easier, Codility provides an on line recruiting platform with a specialized programmer testing solution. Recruiters are able to create or select programming tests and invite candidates to do this test online. Upon completion, a detailed test report is instantly available to be compared with other invited programmers or the pool of previous Codility assessments. Saving them time and money, the recruiters invite only the best candidates to a follow-up interview.

Today, more than 1200 companies in over 120 countries have used Codility, resulting in more than 2,5 million test assessments and counting. One of the contributing factors to this success has to be the innovative approach of the company. Besides being a commercial recruitment platform, Codility offers free training, lessons, and challenges for the programming community.

The interest in supporting knowledge and learning at Codility arises from a scientific background and close bonds with the Warsaw academia. As a result, Codility is very open to all research activities around their company leading to our current collaboration in this study. Currently, 16 lessons and around 70 training tasks are offered for free on the Codility website. The Java solutions of a subset of these tasks constitute the dataset used in this study.

1.2.3 Aspiring Minds

Aspiring Minds is one of the world's leading assessment companies that helps organizations, governments and institutions measure and identify talent. The Indian division of Aspiring Minds published the research results that sparked the development of this thesis. In addition, the authors Varun Aggarwal and Shashank Srikant kindly shared a subset of their data with me for a comparative analysis.

1.3 Motivation

Today, both in academic and commercial settings, computers play a vital role in almost every organization. Nearly all aspects of modern life are touched by information technology (IT). This global digitalization introduces many job openings in computing careers as well as a strong demand for various computing skills in other fields of work. As such, for most of the past 20 years, employment in IT related services has grown rapidly. Projections indicate that the current growth of computing careers continues at least through 2020 [16]. It is therefore not surprising that the demand for skilled programmers is increasing on a global scale.

For students or professionals who wish to benefit from these IT career opportunities it almost becomes a necessity to acquire the skill of computer programming. However, for most people, developing programming skills requires a fair amount of effort. World-wide, on average one third of students fail their introductory programming course [50]. This stimulates governments throughout the world to bring programming into the classroom environment, addressing the current problem of preparing students for future labor demands. However, introducing new basic skills into educational programs is a process that requires a lot of consideration. Teachers are concerned that they lack the right skills to deliver new computing curriculum's and that they will not have enough time to acquire these new skills [15].

When teachers are successfully prepared for the task of teaching programming, another issue comes into play: scalability. There is an old saying *"practice makes perfect"* and although it may not apply to every situation, it certainly applies to learning how to program. Having been through this process myself, I certainly agree with the idea that reading programming books and studying code examples will only get you to a certain level. In order to be successful you need to solve programming exercises and learn from feedback.

It is the assessment and feedback generation of programming exercises that introduces the scalability issue. While computer and communication technologies have provided effective means to scale up many aspects of education, the submission and grading of assessments such as homework assignments and tests remains a weak link [32]. Automated assessment tools have the ability to alleviate the burden on teachers, increase the consistency of markings, improve automated feedback to learners and to expand the candidate pool considered for hiring. As such, automated program assessment tools are being actively researched (a short overview of closely related work in this field is provided in chapter two).

Now imagine you are in charge of hiring new Java programmers for a large software development company. The company just landed a major project so you do not only need good programmers, you need them fast. Budgets are tight (not so hard to imagine), so while you cannot afford to hire the wrong person you are also not in the position to invest a lot of time in the hiring process.

Naturally, you begin to sweat because there are a lot of "programmers" who know Java syntax and libraries but are not able to implement simple

algorithms or come up with a decent design. Surely you do not want to hire such a programmer! A quick browse on the Internet seems to indicate that the perfect solution is just a mouse-click away. Modern commercial tests allow you to invite candidates to take an online programming test. Candidates are confronted with a set of questions from a pool of programming tests. You receive a result report that displays the time the candidate used per task, the results of numerous test cases, and an evaluation of time and space complexity used by the solution.

At first sight, this seems like a great solution for your problems. And if even Google and Facebook use it to pre screen candidates, why not you? But when this approach is used to evaluate a large number of candidates you will not be able to go through the detailed result report of every candidate. The natural evaluation method for this type of automated assessment is then to look at the subset of candidates that passed the highest number of test cases. At most companies, passage of a certain number of test cases is a hard requirement for submissions to even be considered.

There are (at least) two pitfalls associated with this approach. One is your problem: programs that pass many test cases may be written with bad programming practices (such as hard coding). Because the source code at any point in time is included in the result report it is likely that after manual inspection you do not wish to invite this particular candidate. However, you did waste your time on it! The other pitfall affects both you and the candidate. Programs that pass a low number of test-cases could be very efficient or quite close to the correct solution. As these programs are not manually evaluated, their potential will not be noticed. This causes you to miss out on a possibly great candidate and denies the candidate the job interview that he or she was hoping for (and quite possibly deserves).

This thesis is dedicated to improving the discussed automated hiring processes by exploring the added value of semantic source code features. This approach was proposed in a recent paper by Aggarwal and Srikant[45], which introduces a feature-grammar to tag Java source code. Solutions to programming assignments are represented as a vector of features and used as input for machine learning models. The idea is that the algorithmic structure of the code can be captured with the features, which allows the models to detect similarity with other solutions. This similarity is expected to indicate the quality of the code more accurately than test case scores. The relevance and goal of my research is motivated in the following section.

1.4 Research Goals

The goal of this paper is to further analyze the approach proposed by Aggarwal and Srikant to grade programming assignments with machine learning techniques[45]. Their work concludes that the machine learning approach provides much better grading than test case based grading. Four of the five problems Aggarwal and Srikant analyze benefit from the proposed grading method compared to test case rating. The grading of the fifth problem shows no improvement or deterioration. As these results are promising, this thesis makes an attempt to verify the reported performance

for different datasets. In addition, experiments are constructed to demonstrate the influence of altering assumptions on which the approach is based.

One of the points of distinction of this research is the modeling and definition of the grades. The grading process proposed by Aggarwal and Srikant follows a five-level rubric based on algorithmic structures[45]. My grading style is more solution oriented, attempting to more closely resemble real-life applications. It will be interesting to see if the algorithm oriented features are able to capture these different grade distinctions. To model program grades, Aggarwal and Srikant use simple feature selection followed by a regression with 3-fold cross validation. However, as the grades fall into 5 discrete categories, regression modeling is not the obvious choice. I will therefore cast the method in both a regression and classification setting.

In short, an experiment is conducted to validate claims made by the original authors as well as the influence of adjustments made in this research. In order to utilize a machine learning framework the source code of Java solutions has to be converted into suitable input for a machine learning algorithm. Thus, another goal of this project is the development of a feature generation tool for Java code. This application is called JFEX (an acronym for Java Feature Extraction) and is introduced in chapter three.

1.4.1 Research Questions

The main research question this study aspires to answer is:

Can we improve automated test-case based grading of programming assignments with semantic source code features?

To be able to answer this question I developed the JFEX application to derive semantic features from Java source code. Next, we empirically test the theory by modeling the human grading process. In a supervised learning setting we test the performance of various classifiers and regression models on our dataset.

In order to derive source code features we follow the high level definitions of the categories defining a semantic grammar introduced by Aggarwal and Srikant[45]. These high level definitions allow many low level interpretations. That is, a single solution has many possible semantic feature sets. Specific decisions regarding the feature extraction process are documented and we compare the impact of different interpretations.

In order to answer the research question my thesis will address the following specific problems:

- Are dependency and control flow features better grade indicators than basic keyword counts?
- Does the granularity of semantic feature definitions affect the performance of prediction accuracy with respect to human grading?
- Can we improve grading accuracy by modeling the problem with classification instead of regression?

- What are the most influential source code features when classifying a solution program?

1.4.2 Research Relevance

The added value of this paper is twofold: an academic contribution and practical results. For the academic world it is important to replicate studies and validate results in an independent fashion [43]. As is mentioned by Shull et al.[42], the main goal of replication does not need to be limited to statistical hypothesis testing and p-values. Testing that a given result is reproducible and understanding the sources of variability that influence the results are also valid motivators for replication studies. This study shares both goals; on the one hand we intend to gain knowledge from a more thorough analysis of previously made claims. On the other hand we are exploring possible influences of the used modeling methods and feature definitions.

Two categories of replication studies can be distinguished[42]: *exact replications*, in which the procedures of an experiment are followed as closely as possible; and *conceptual replications*, in which the same research question is evaluated by using a different experimental procedure. This study can be classified as a *conceptual replication* where independent research attempts to replicate a published study using its own experimental design. If the study succeeds in reproducing results of the original study, this provides a high degree of confidence that the result is real and significant. However, when the results of the replication contradict those of the original study there will be no solid basis for hypothesizing about the cause of the discrepancy.

In terms of practical value we see a direct connection to every day practice. In our global and digital society people are already being judged with test-case based grading to see if they deserve a shot on a job interview. Having numerous candidates and limited time, recruiters have no other option but to use this imperfect assessment technique. Small mistakes can be punished extremely hard leading to good programmers being falsely classified as unworthy of an interview. On the other hand, candidates who write imperfect solutions that manage to produce the right results for specific test cases might undeservedly be classified as top candidates.

In academic settings the number of students enrolled in universities at standard and on-line programming courses is rapidly increasing. MOOC's (Massive Open Online Courses) open a whole new world of learning possibilities, with accurate assessment being the biggest drawback at the moment. Methods for objective and reliable grading that can also provide substantial and comprehensible feedback are essential for learning. As such, automated assessment and feedback generation are currently keen topics of interest [2], [21], [29], [30].

Besides the direct application of our research to generate more accurate automated grading, analyzing the influential features in our models could help to automatically discover important logic elements needed in a correct solution of a problem. This may support feedback/hints to students on what constructs to use whilst solving a problem.

Chapter 2

Related Work

This chapter discusses some of the work on automated program assessment and the closely related field of programming tutoring systems. The current body of research related to automatic diagnosis of programs involves many different aspects, including: program representation, intelligent programming tutoring systems, automated program assessment, visual feedback systems, peer review systems, and problem solving strategies. It is by no means the goal of this chapter to provide a complete overview of these topics, but rather to outline the framework which embodies my research.

2.1 Programming Skill Assessment

The foregoing chapter of this thesis mentioned that one of the motivations for this research is to improve an approach to assess someone's level of programming skill. In companies, one usually wants to assess someone's programming skills because they believe it to be an indication of a person's performance on a programming job. This can be done trough inspection of of their education and experience on CVs and through interviews. Sometimes standardized tests of intelligence, knowledge, and personality are also used. Even though such methods may indicate an individual's level of skill, they do not measure programming skill per se.

In academic institutions a similar problem arises when teachers want to know: what skills do students have and what is their level? With programming the problem is that there is not a simple distribution of ability; some students noticeably outperform others. A strongly bimodal distribution of marks in the first programming course is frequently reported anecdotally and corresponds to experiences in academic institutions [10]. Teachers try to find ways to help students by analyzing their learning process and leading them to better results.

The standard practice in research and industry for assessing programming skills is mostly to use proxy variables of skill such as education, experience, and multiple-choice knowledge tests. There is not yet a broadly accepted valid and efficient way to measure programming skill. In the work by Bergersen et al.[9], an instrument is developed to measure programming skill by inferring skill directly from the performance on programming tasks. Over two days, 65 professional developers from various countries solved 19 Java programming tasks. Other attempts exist but this work distinguishes itself with a combination of theory-driven research and a strict definition of measurement. This enables rigorous empirical testing of the validity of the instrument, which was found to have satisfactory internal psychometric properties and correlated with external variables in compliance with

theoretical expectations. The implicit assumption made is that the level of performance a programmer can reliably show across many tasks is a good indication of his programming skill level. This is not a unique idea, the latest practice in industry is already applying this concept for a couple of years. The problem is that these test results are expected to predict programming skill, but this hasn't been properly assessed in an attempt such as the one by Bergersen et al[9]. It is important to address when and how performance on a combined set of programming tasks can be regarded as a valid measure of programming skill.

So how does one go about assessing programming skill? If we forget for a moment about the programming aspect, we are left with one construct: skill assessment. Skill can be defined theoretically as a psychological variable. Together with motivation and knowledge, skill defines the three direct influences on the performance of an individual[12]. Other aspects such as experience, education, and personality, have an indirect influence on individual performance through their influence on motivation, knowledge, and skill. There is a large body of work done on the theory of skill, especially in the field of psychology. Over the past few decades different theories emerged on how humans acquire skills.

2.1.1 Early Work

Much research was done in the 1980's regarding this subject and also regarding programming skills. While most of the earlier studies focused on motor skills, Anderson et al. devoted much attention to the research on cognitive skills in general during the 1980s[4], and programming skills in particular[5]. Anderson noted that the errors associated with solving one set of programming problems was the best predictor of the number of errors on other programming problems. Another early attempt to predict programming success can be traced back to 1985 when Adelson and Soloway reported that familiarity with the problem domain helps novices to solve programming problems[1].

Besides the search for predictors of success there was also the more pragmatic approach of developing techniques or tools to make programming easier for all novices. Ranging from IDE's to designing whole new programming languages such as LOGO which was developed in the mid 1960s[22]. Today the language is remembered mainly for its use of "turtle graphics", in which commands for movement and drawing produced line graphics either on screen or with a small robot called a "turtle". The language was originally conceived to teach concepts of programming related to Lisp and only later to enable what Papert, one of the creators, called "body-syntonic reasoning" where students could understand (and predict and reason about) the turtle's motion by imagining what they would do if they were the turtle. Despite all efforts there is no evidence that this has had any significant impact on the success rate among novices.

2.1.2 Recent Work

Many studies aim to explore and reveal profiling patterns in the measurement of cognitive and non cognitive characteristics of undergraduate students' programming performances[3]. These studies explore many indicative variables that might affect programming performance including spatial skills, working memory, academic grade point average scores mathematics and science achievement, prerequisite knowledge, success/failure attribution, perceived programming self-efficacy, encouragement, comfort level, working style preference, prior experience of programming, prior experience of computers except for programming, intelligence, computer attitude, cognitive style, learning style, mother tongue, cognitive development, socio-economic status, creativity, problem-solving skill, and gender.

Studies that report exciting results on new predictors for programming skills should be handled with caution. *"The camel has two humps"* by Bornat and Dehnadi is a popular paper that appeared to have discovered an exciting new predictor of success in a first programming course[17]. However, after six experiments involving more than 500 students at six institutions in three countries, the predictive effect of the test has failed to live up to that early promise[10].

2.2 Automated Program Assessment

Automated program assessment has been researched for more than fifty years. The review by Douce et al. provides a comprehensive overview of the history of the field where automated test-based assessment systems are classified into three generations[18]. These generations range from early assessment systems to distributed structures with command-line or GUI clients and finally to the modern assessment platforms with web-based interfaces. A more detailed review of modern assessment systems is provided by Ihantola et al.[29]. One of the key problems identified in this survey is that most assessment tools are developed for the purposes of one class or assignment. Even though some generalized tools are available, many programming instructors feel that they have to write their own frameworks to realize the functionality they require for their courses.

An example of a framework that was developed in-house to fulfill the need of automatic grading is CodeAsessor [13],[53]. Teachers of CS1 courses (introductory C/C++) at the University of Tennessee created CodeAssessor to grade student's coding problems on exams as they could not find a satisfactory existing solution. The grading component compares the student's output on test cases against the instructor's output and awards credit when the output matches. This emphasis on program behavior is consistent with the findings of Ihantola et al.[29], which state that most of the discussed assessment tools are assessing functionality rather assessing style or performance.

In this research we also focus on automated assessment from the viewpoint of program behavior. We try to assess if the programmer understands the grammar of the programming language and is able to apply common algorithms and different data structures into a working program. This approach does not capture the full range of somebody's programming skills as

programming is a complex creative skill which requires a lot of specialized knowledge[40]. It is therefore important to realize that there are many more indicators for a good programmer such as: documentation quality, coding style, testing/debugging skills, and effective use of available programming tools and libraries.

In the most well-known survey of the field of automated programming assessment[2], Ala-Mutka describes which features of programming assignments are automatically assessed by different assessment tools. The features are organized according to whether they need static of dynamic evaluation. The difference is that dynamic evaluation needs program execution while static evaluation can be performed directly on the source code. The remainder of this section gives an overview of the properties of these two approaches.

2.2.1 Dynamic Assessment

Dynamic assessment methods are widely used to assess program functionality, which is often viewed as one of the most important evaluation aspects. The assessment of functionality depends on a method's ability to recognize and discriminate between correct and incorrect program behavior. Many programming assignments can be formulated in such a way that key aspects of required behavior can indeed be recognized and assessed by automated methods.

The main problem with dynamic assessment is that it is very sensitive to all kinds of program errors. Programs containing syntax or contextual errors cannot be tested. Programs with logic errors may be testable but the results might be unpredictable and if a logic error causes excessive memory allocation, or an endless loop, the testing of the program will fail. Features of programming assignments that can be automatically assessed using dynamic assessment are:

- **Functionality**: does the program function according to the given requirements? In general, the answer to this problem is known to be undecidable. In practical applications this question depends critically upon two things: providing test data to the running program, and evaluating the accuracy of the resulting outputs. Preferably both without human involvement. Output accuracy is often tested using comparison against several test data sets for which the desired output is known. Other approaches rely on unit testing in combination with scripts or other testing frameworks.

- **Efficiency**: Measures the program's behavior (time and space complexity) during execution. Considering the literature, the measuring of efficiency in automated assessment systems has focused almost exclusively on the time dimension. The analysis of the dynamic memory footprint of programs has received far less attention[36].

- **Testing Skills**: Is a programmer able to demonstrate the correctness and validity of his own program? This is assessed with test coverage analysis, where programmers not only submit their solution programs but also a set of formalized test cases. An example of a framework that relies on this principle is Web-CAT, the Web-based Center

for Automated Testing[20]. Web-CAT encourages students to write software tests for their own work. The benefit of these test-driven learning approaches is that they address the issue of creating an appropriate test suite that validates code behavior[19].

- **Special Features**: For example dynamic memory management with C++.

Automated Test Generation A problem with many dynamic testing tools for code functionality, ranging from scripting and output comparison to using testing frameworks like XUnit, is that an instructor must write a suite of tests to validate the behavior of a program. This does not only cost a significant amount of time but the tests might also fail to take into account oddities of programs. To solve these problems, especially the time consuming problem, a large amount of work in the area of automated test generation is done.

When developing automated test generation programs the main criterion is to create test suites that maximize code coverage. This is generally achieved with some kind of input space exploration, in order to find inputs which cause particular program-flow paths to be executed.

Many approaches that try to solve the problem of creating test suites rely on random tests (black-box tests)[37]. Random testing can eliminate subjectiveness in constructing test data and increase the diversity of test data. The difficulty encountered is the creation of test oracles that decide test results. Avila and Cheon propose an automated testing approach for Java programs based on random testing and assertions[14]. The approach uses OCL (Object Constraint Language) constraints as test oracles by translating them to runtime checks written in AspectJ. Experiments and case studies show that the approach can detect errors in both implementations and OCL constraints.

More guided approaches take the structure of a program into account when generating test input (white-box testing). These approaches come in many forms, of which the majority focuses on the specification of contracts for methods or objects which can be used to generate tests [11]. An example of a white-box approach to test generation in Java is the thesis by Bell[8], which presents JSymTester. JSymTester utilizes automated test generation techniques, based on the symbolic execution framework of the Java PathFinder, to develop a suite of inputs which test student code fully. The instructor code is used as a test oracle. Its performance was tested on small assignments for an introductory computer science course and was similar to existing, more traditional (manual test generation) approaches of unit testing and output comparison.

Another category of white-box test-generation tools use a technique called evolutionary testing. Evolutionary algorithms are used to evaluate the "fitness" of test inputs based off of particular criteria such as statement coverage, branch coverage, or size of generated tests. These inputs are then mutated in an attempt to derive a more "fit" set of inputs. This technique can be used either on method level to explore method inputs or at the class level to find method call sequences[7], [23].

Despite the large body of work in this area there is still progress to be made if we want to optimize automated testing. It is unsure how much progress can be made, as Forisek demonstrates that for some programming

tasks it is impossible to design good test inputs[21]. The next section discusses static assessment, which is recently rediscovered as a tool for assessing functionality.

2.2.2 Static Assessment

Static source code analysis is a common feature in automated grading and tutoring systems for programming exercises. Static evaluation benefits from the fact that it can be carried out even when there are problems in the dynamic behavior of the program. Different approaches and tools exist, each with individual benefits and drawbacks. An overview of different principal approaches and tools for static analysis of Java code are presented in a review by Goedicke and Striewe[46].

An important feature of a Java code analysis tool is whether the tool operates on source code or byte code. Source code is written directly by the programmer whereas byte code is generated from the source code by a compiler. Although for many tasks these two approaches can perform an equivalent analysis there are some differences.

Byte code analysis cannot be carried out completely when a program contains compiler errors. It also has an influence regarding the granularity of the analysis. For example, all loop statements are represented by GOTO-statements in byte-code. Although all loop constructs in Java result in typical byte code patterns, analysis of these patterns is not trivial in all cases. In source code analysis, this problem does not exist, since every statement can be recognized from the source code directly. This is one of the main reasons why we have chosen to work with source code analysis in this study.

Features of programming assignments that can be automatically assessed using static assessment are:

- **Coding Style**: The compiler checks for correct syntax, a customized tool checks for style requirements. In 1982, the groundwork for assessing coding style was laid by Rees[41]. He identified ten simple metrics through which the quality of Pascal programs could be judged: line length, comment density, indentation, blank lines, embedded space, program decomposition, reserved words, variety of identifiers, and labels & gotos. These style metrics are useful across a wide variety of programming languages.

- **Programming Errors** Find bugs or "errors" that are syntactically correct but indicate misunderstood concepts (for example, when in Java an if-declaration is ended directly with a semicolon instead of a statement or body). As finding and fixing bugs is an important part of software development, there is a lot of work done in this area.The approaches range from coding rules, code review and testing to tool supported error detection. The application of machine learning to static analysis for program error detection is proposed in the paper by Hannes et al.[48]. Source code from various open source projects was studied to find the relevant features to classify an instance of code as faulty or correct. The results show that this method is a possible approach for future static analysis tools but the authors note that much research remains to be done before this area has been sufficiently studied.

- **Software Metrics**: general measurements that characterize a computer program. An example of a software metric that has proven especially useful in teaching environments is the Cyclomatic Complexity measure[35], which estimates the complexity of program control flow.
- **Design:** conform to given interface or structural requirements.
- **Special features:** For example a plagiarism check.

As the above features show, static assessment has traditionally been used to assess quality, not correctness. However, this paradigm is shifting. New techniques that focus on using static methods to assess program functionality are being developed. These techniques may reveal functionality issues that have been left unnoticed (or punished too harshly) by the limited test case suites. It is exactly this potential that motivated our research to explore the possibilities of functionality assessment with static techniques.

- **Functionality:** New methods for assessing functionality shift their focus from the dynamic to static methods. We can distinguish three main approaches to static program assessment[52]. They differ with respect to the ways in which knowledge of correct student programs are requested and handled.

  Source-to-Specification matches against a specification that is a high-level description of the program's goals. This approach is complicated due to the many possible variations. **Specification-to-Specification** uses formal specification matching of example and candidate programs. This is infeasible because formal specifications of programs cannot always be derived automatically. **Source-to-Source** matches a program against a model program stored in the system.

  An example of research in the **Source-to-Source** category is a semantic matching based automatic scoring method [33] [49]. Programs are converted to the intermediate representation of a system dependence graph and a semantic equivalence conversion is carried out according to a series of standardization rules. The matching degree of the standardized system dependence graph scores the programs according to the matching result and scoring rule. At the moment, it is only applied to simple introductory programs as the number of possible solutions grow rapidly with increased program size and complexity. For the introductory problems the method gets a high grading precision (90-100 percent) for most cases but sometimes scores very poorly due to the lacking of appropriate model programs.

2.3 Programming Tutoring Systems

A robust technology that is able to determine whether programs coded by a student are correct is essential for programming tutoring systems. When a student's program is incorrect the system should be able to pinpoint errors in the program as well as explain and correct the errors. This is at a deeper level than the program assessment this paper focuses on. Due to the difficulties that arise on this deeper level of evaluation, no existing system performs this task entirely satisfactorily. The recent paper of Jeuring

et al. provides a systematic review of automated feedback generation for programming exercises[30]. It analyses and categorizes the feedback generation in 69 tools for learning programming. This feedback helps students to improve their work and is an important factor in learning.

A student program may contain two kinds of errors: syntax errors and semantic errors. In the context of programming tutoring systems, the problem of detecting syntax errors in a student program has been solved, making use of compilers. However, the problem of detecting semantic errors in a student program has not been solved satisfactorily yet.

2.4 Unsupervised Grade Modeling

When the main goal for automated assessment is not to grade each assignment on a specific scale but to simply separate good programs from inferior ones, more options become available. Instead of manual labels, which require a lot of effort, one could use a set of codes with high test scores for the purpose of prediction. Aggarwal and Srikant published the following preliminary investigation of this idea [45].

The threshold for "good programs" is set to be those that pass 80 percent or more of the test-cases, have the right time complexity and follow programming best practices. Out of five problems this resulted in 443 out of the 999 programs that were automatically detected to be "good" (ranging from 27 up to 151 per individual programming task). Of these 443 "good" programs, 432 were marked as good (grade 4 or 5) by the experts. So the automatically detected "good" set is indeed of high quality. This is in line with the findings presented in chapter three.

A simple one-sided distance metric, which penalizes having less of a feature but ignores having more of a feature, was defined to define the distance from the good set. This is based on the intuition of the authors that having more of a particular feature is generally not indicative of an incorrect solution. Due to the small size of the available datasets all features kept the same weight. The underlying assumption is: the larger the distance between a feature vector representation of a program and (a subset of) the good set, the lower the score.

Experiments were carried out twice, once for just the basic feature distances and once for all semantic features including test-case scores. Distances to all programs in the good set are summed after which the following measures were considered:

- **the mean of these distances**, which might be noisy given that the good set would have codes implementing different strategies.
- **the minimum of these distances** which is noisy given the presence of outlier codes.
- **the mean of the least 25 percent of the distances** which seems to be a good trade-off of first two approaches. The hypothesis is that this measure tries to identify the cluster of good programs that the current program belongs to.

As hypothesized above, taking the mean of the least 25 percent of the distances to good solutions produced the best results.

A moderate correlation is reported between the expert assigned scores and the distances. Scores based on all features outperformed those that are only determined based on basic features. Unfortunately this is not such a powerful statement because the "all features" set uses test-cases whereas basic features does not. There are no results based on the semantic features or test-cases alone, which may have given more insight in the added value of semantic features in the obtained correlations of 0.58 to 0.83 between the expert assigned scores and the distances in the feature space.

In their latest work, Aggarwal et al. do provide a more detailed analysis of the results from the unsupervised models [44]. Contrasted against the baseline of test-case score predictions the model outperforms it on all 19 problems in the dataset. An important aspect of this research is that all predictions for the different tasks were made by the same model. The concept behind this model is explained in the final section.

2.5 One Size Fits All?

The latest research of Aggarwal et al. regarding grade predictions uses a bigger dataset and attempts to tackle the problem of having to train a model for every question separately[44]. The so-called question independent model that is developed is able to assess an ungraded response of an unseen question. The main assumption is that the distance from a solution to the good set in the feature space is an invariant feature (although it might need normalization) across questions. To demonstrate this a model is trained using labeled examples for some questions, learning the distances from a solution to good solutions that correspond to grade categories. Next, for a new question to be graded one only needs data for a good set, which can be obtained automatically through test-cases if enough samples are available. The predetermined distance thresholds are used to predict grades.

For each of 19 questions a specific supervised model is learned from the feature distances and its performance is compared to the performance of a question-independent model trained on a subset of the questions in the dataset. The programming languages used for the solutions in the dataset are c, c++ and Java with an average of 285 responses per question. When compared to question specific models the question-independent model provides comparable performance in correlation values of the predicted grades but shows a higher bias and average mean error. The authors see this as an indication that the normalization technique is not able to scale the distances for the questions perfectly.

Chapter 3

Application: Feature Derivation

This chapter begins with an overview of the different types of features to be derived from Java solutions. Next, the Spoon library is introduced which is the main building block of the program that derives the features from solutions.

3.1 Program Features

To be able to capture program semantics automatically I developed a tool that extracts a potentially large number of program features from Java code. The tool is called *JFEX*, which is an acronym for *Java Feature Extraction*. In the current setting, the Java code to be analyzed consists of single-class solutions to various programming problems. For a specific problem, JFEX dynamically generates all features corresponding to each individual solution. Then each solution is tagged with the number of occurrences of its features.

The feature derivation rules are based on two concepts:

- Features should be able to capture algorithmic design.
- Features should be accurate and easy to compute.

Srikant and Aggarwal suggest the following five general categories for Java code features[45]:

- **Basic Features:** counts of various keyword and token occurrences. Basic features indicate if the right constructs appear in the code.
- **Expression Features:** counts of the occurrences of all particular expressions that appear in a program. Arithmetic and relational operations occurring in the program are captured by Expression features.
- **Control Context Features:** associate Basic and Expression features with the context of the control flow structures (such as if conditions of loops in Java) they appear within.
- **Dependency Features:** counts of Expression features that are dependent on other particular Expression features. A dependency from expression a on expression b indicates that the value of a variable in a is influenced by the evaluation of expression b.

- **Dependency Features in Control Context:** associate Dependency features with the control flow structures they appear within.

The five categories mentioned above are very general and allow for multiple interpretations. Per language and application use, decisions have to be made as to how to define these features. These decisions have to enforce the capacity of the features to capture the algorithmic design of the program. In my application, the formal language Java defines which constructs are eligible for certain features. However, I have to make choices regarding the most meaningful scope of the features. The next section describes the decisions that were made regarding the feature categories and the accompanying extraction rules.

3.2 Feature Definitions

I: Basic Features Basic features represent simple counts of the following code constructs:

- **Keywords:** break, continue, return, case* (*see **switches**).
- **Constants:** Java constants are captured based on their explicit value. For example, when a program contains two strings *"hello"* and *"world"*, they are represented as one occurrence of *"hello"* and one occurrence of *"world"* instead of one representation for two String occurrences.
- **Variables:** variables are represented in a double fashion. One set of variable count Basic features capture the class of variables that occur (i.e.: Integer, Double, String, Stack, ArrayList etc.). Another type of variable count Basic features capture the type of the variables: local variables, parameter variables, or field variables.
- **Loops:** loop occurrences are counted based on their type: for, foreach, do, or while.
- **Conditionals**: conditional occurrences are also counted based on their type: if or ternary.
- **Switches:** simply the number of switch occurrences (the number of cases is represented in a keyword count).
- **Binary Operators:** logical and bit and, bit or, bit xor, division, equality, greater or equal comparison, greater than comparison, InstanceOf (OO specific), lower or equal comparison, lower than comparison, subtraction, multiplication, inequality, logical or, addition, shift left, shift right, unsigned shift right.
- **Unary Operators:** binary complement, negation, logical inversion, positivation, decrementation post assignment, incrementation post-assignment, decrementation pre-assignment, incrementation pre-ass.
- **Operator Assignments:** Java allows x = x + 5 to be declared as x += 5. To stress the equality of both definitions we categorize them identically as PLUS ASS.

- **Method calls:** method call occurrences are uniquely defined by the name of the method that is called.
- **Recursive method calls:** when a method is called from within it's own body this is categorized as a recursive call defined by the method name (instead of an ordinary method call).

II: Expression Features Expression features capture the expressions that occur within all statements in the program. A statement represents a single expression in the source code of the program. In JFEX, Expression features are obtained by analyzing all statements that include: binary operators, unary operators, assignments, declarations, or method calls. In addition to the operators that occur in the statement, all variables and constants are represented in the expression.

- **Keywords:** the counts of keywords that occur within a statement are represented per keyword type (break, continue, case) in the Expression feature.
- **Constants:** all constants that occur in a statement are captured in a single count for the Expression feature. There is no distinction made based on explicit values or types.
- **Variables:** variable occurrences in an Expression feature are counted based on types: local variables, parameter variables, or field variables.
- **Loops, Conditionals, Switches:** are not part of Expression features.
- **Binary Operators, Unary Operators, Operator Assignments**: every operator occurrence in a statement is represented in the Expression feature that describes the statement. If an operator occurs multiple times in a statement this results correspondingly in multiple counts of the operator in the Expression feature. Operator combinations are always represented in the same lexical order, regardless of the order they appear in a statement.
- **(Recursive) Method Calls:** the counts of specific method calls (based on the method name) within a statement are part of an Expression feature.

III: Dependency Features The flow of data within a method can be captured by data dependencies. A data dependence A to B exists if statement B references a variable which is defined or modified in statement A. Dependency features represent the most recent dependency of one variable. So when an Expression feature contains two variables this may result in two dependency feature counts, one expressing the most recent dependency for each variable influence.

IV: Control Features Control features map out control and data dependencies between the statements of a Java program. A control dependence from statement A to B exists, if the execution of a statement B relies on the execution of statement A. Because it is common practice in Java to nest control structures, JFEX considers the complete nested depth of a feature. Control features can consist of Basic Features, Expression features, or Dependency features, appended with the context options listed below. In essence,

Control features themselves can also be appended with their context because we assess the complete nested context depth.

- **Loops:** loop context is defined by loop-type and occurrence of the feature within a loop. We differentiate between a loop-body and a loop condition. For-loop conditions are a special case where we distinguish features based on if they occur in the initialization condition, expression condition or update condition.
- **Conditionals:** conditional context is defined by occurrence in the if-condition, else-condition or then-condition of an if or ternary statement.
- **Method context:** when a call to a method defined in the program under analysis occurs within control context, the control context of the method call statement is appended to all statements within the called method. Whenever a method is called multiple times but from within different control contexts, alls statements in the called method are turned into multiple Control features: one for every surrounding control context of the method call. The idea behind this is that it reduces feature differences between programs with equal functionality. The next section demonstrates an example of the features that are generated by JFEX for a method that is called twice from different control contexts.

3.3 Feature Examples

Figure 3.1 on the next page is an example solution for the Fish task. The current section introduces some of the features that JFEX derived from this solution in order to illustrate the feature categories described in the previous section. The uppercase characters surrounded by dollar signs indicate the feature category. Next, short-hand notations for feature aspects are followed by a colon and a digit that indicates the number of times this feature occurs. Finally, a short description and/or the line numbers of the occurrences are provided in parentheses.

I: Basic Features

- **Keywords:** Bbreak:1 (line 38), Breturn:1 (line 30).
- **Constants:** Bc:0:3 (line 10,13,17).
- **Variables:** Bfv:4 (field variables: line 4,5,6,7), Blv:2 (local variables: line 13, 14), Bpv:3 (parameter variables: line 9, 33), Bvt:Array:2 (variable usage of type Array: line 14, 15), Bvt:Stack:8 (variable usage of type Stack: line 11, 16, etc.).
- **Loops:** Bwhile:1 (line 34), $B$for:1 (line 13).
- **Conditionals**: Bif:5 (line 15, 16, 17, 22, 35).
- **Binary Operators:**BEQ:1 (line 15), BGT:2 (line 17, 35), BLT:1 (line 13), BPLUS:1 (line 30).

- **Unary Operators:** BNOT:1 (line 34), BPOSTINC:3 (line 13, 18, 23)
- **Method calls:** Bmc:battle:1 (line 21), Bmc:empty:3 (line 16, 22, 34), Bmc:peek:1 (line 35), Bmc:pop:1 (line 36), Bmc:push:1 (line 27), Bmc:size:1 (line 30).

```
1  import java.util.Stack;
2
3  public class Solution {
4      static final int UPSTREAM=0;
5      static final int DOWNSTREAM=1;
6      int upstreamSurvivors;
7      Stack<Integer> downstreamSurvivors;
8
9      public int solution(int[] a, int[] b) {
10         upstreamSurvivors = 0;
11         downstreamSurvivors = new Stack<Integer>();
12
13         for (int i = 0; i < a.length; i++) {
14             int currentFishSize = a[i];
15             if (b[i] == UPSTREAM) {
16                 if (downstreamSurvivors.empty()) {
17                     if (currentFishSize>0) {
18                         upstreamSurvivors++;
19                     }
20                 } else {
21                     battle(currentFishSize);
22                     if (downstreamSurvivors.empty()) {
23                         upstreamSurvivors++;
24                     }
25                 }
26             } else {
27                 downstreamSurvivors.push(currentFishSize);
28             }
29         }
30         return upstreamSurvivors + downstreamSurvivors.size();
31     }
32
33     private void battle(int currentFishSize) {
34         while (!downstreamSurvivors.empty()) {
35             if (currentFishSize > downstreamSurvivors.peek()) {
36                 downstreamSurvivors.pop();
37             } else {
38                 break;
39             }
40         }
41     }
42 }
```

FIGURE 3.1: Fish solution: test case score 100%, grade: 5.

II: Expression Features

- **Keywords:** Efv:2_return_op:PLUSmc:size$:1 (a return statement, 2 field variables, addition operator and a method call to size(): line 30).
- **Constants:** Elv:1_c:1_op:GT$:1 (a local variable, 1 constant and a greater than operator: line 17).
- **Variables:** Elv:1pv:1fv:1_op:EQ$:1 (a local variable, parameter variable, field variable and an equality operator: line 15).
- **Binary Operators, Unary Operators, Operator Assignments**: Efv:1_op:POSTINC$:2 (post increment unary operator on a field variable: line 18, 23), Elv:1pv:1_op:LT$:1 (local variable, parameter variable and a less than operator: line 13).
- **(Recursive) Method Calls:** Elv:1_op:mc:battle$:1 (method call to battle() with a local variable: line 21), Epv:1fv:1_op:GTmc:peek$:1 (a parameter and field variable, greater than operator and a peek() call: line 35).

III: Dependency Features

- DE$lv:1_c:1_op:GT$_@_Elv:2pv:1_op:ASS$:1 (a local variable, constant and greater than operator with a dependency to 2 local variables, 1 parameter variable and an assignment operator: line 17 dependency to line 14).
- DE$lv:2pv:1_op:ASS$_@_Elv:1_op:POSTINC$:1 (2 local variables, a parameter variable and assignment operator with a dependency to a local variable and post increment operator: line 14 dependency to line 13).

IV: Control Features

- **Basic features in control context:** BC_mc:battle_con:ForIf_bIf_e$:1 (a battle() call inside an ifelse condition, nested in an if body, nested in a for-loop body: line 21), BCop:while_con:ForIf_bIf_e_$:1 (a while operator inside an ifelse condition, nested in an if body, nested in a for-loop: line 34). This last feature is an example of context that is added due to the context of the method. Because the battle() method call comes from within ForIf_bIf_e, this context is added to all features of the battle method.
- **Expression features in control context:** EClv:1_op:POSTINC_con:For_upd$:1 (a local variable and post increment operator, that occur as the update expression of a for loop: line 13), ECpv:1fv:1_op:GTmc:peek_con:ForIf_bIf_e_While_bIf_c$:1 (a parameter and field variable, greater than operator and a peek() call, that occur in the condition of an if-statement, inside a while loop, inside the context of the method call battle: line 35).
- **Dependency features in control context:** DEC$lv:1_c:1_op:GT_con:ForIf_bIf_bIf_c$_@_$EC$lv:2pv:1_op: ASS_con:For$:1 (a local variable, 1 constant and greater than operator

inside the condition of an if that is nested inside two other ifs and a for loop, that has a dependency to 2 local variables and a parameter variable assignment that occur in the body of a for loop: line 17 with a dependency to line 14).

3.4 Spoon Code Analysis

The JFEX feature derivation program which derives features of Java source code is developed with Spoon. Spoon is a library for the analysis and transformation of Java source code [39]. Based on compile-time reflection, Spoon enables developers to analyze every single program element and allows full intercession up to the statements and expressions of the language. It can be used for domain specific analyses, written in plain Java. Spoon achieves this with:

- A Java meta model for representing Java abstract syntax trees (ASTs), which allows for both reading and writing.
- Queries and processors for traversing the program under analysis.

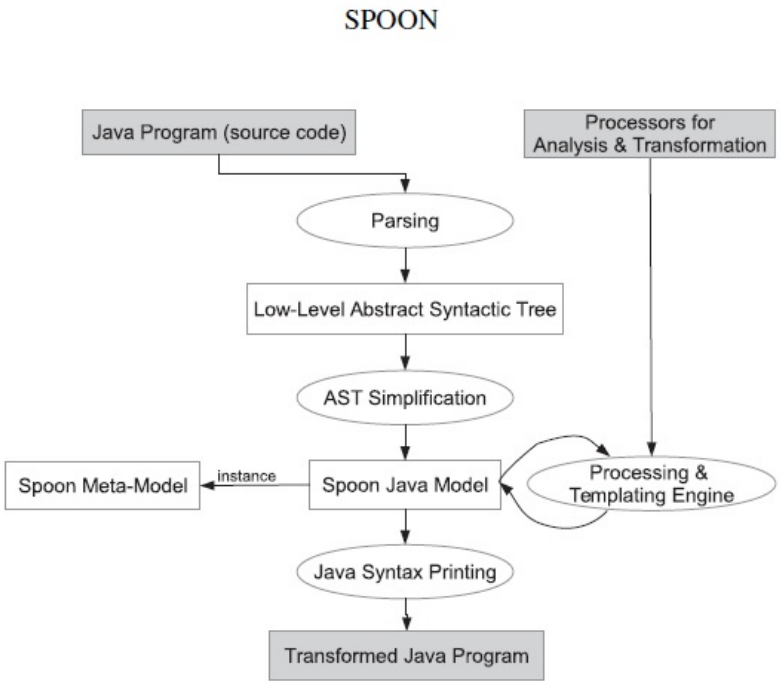

FIGURE 3.2: Overview of Spoon: Java programs are transformed and analyzed as instances of the Spoon Java model. (Image source: [39]-Figure 1.)

Fig 3.2 on provides an overview of the Spoon model. After an Abstract Syntax Tree (AST) is generated by a default Java compiler, Spoon simplifies the AST. This simplified AST is a compile-time instance of the Spoon meta

model. The analysis and transformation of programs are written as *program processors* and *templates* which are applied to the Java model by the engine. A visitor pattern implements the processing. The visitor scans each visited program element and can apply user-defined processing jobs (processors) [38]. The Spoon meta model contains all the required information to derive compilable and executable Java programs and supports Java 8.

The Spoon meta model can be split in three parts. All names are prefixed by CT which means compile time..

- **The structural part**:contains the declarations of the program elements, such as interface, class, variable, method, annotation, and enum declarations.
- **The code part**: contains executable Java code, such as the one found in method bodies. There are two main kinds of code elements: ct-statements and ct-expressions. Ct-statements are untyped top-level instructions that can be used directly in a block of code. Ct-expressions are used inside statements. Some code elements implement the interface Ct-invocation and are both statements and expressions.
- **The reference part**: models the references to program elements such as a reference to a type. The reference part of the meta model expresses the fact that the program references elements that are not part of the meta model, they may for example belong to third party libraries.

3.4.1 Querying Source Code Elements

The information that can be queried is that of a well-formed typed AST. For this, a query API is available that is based on the notion of "Filter". A filter defines a predicate of the form of a matches method that returns true if an element is part of the filter.

```
// collecting all assignments of a method body
list1 = methodBody.getElements(new TypeFilter(CtAssignment.class));

// collecting all deprecated classes
list2 = rootPackage.getElements(new AnnotationFilter(Deprecated.class));

// creating a custom filter to select all public fields
list3 = rootPackage.getElements(
  new AbstractFilter<CtField>(CtField.class) {
    @Override
    public boolean matches(CtField field) {
      return field.getModifiers.contains(ModifierKind.PUBLIC);
    }
  }
);
```

FIGURE 3.3: Spoon code example on the usage of filters. (Image source: *http://spoon.gforge.inria.fr/filter.html)*

A filter is given as a parameter to a depth-first search algorithm. During AST traversal, the elements satisfying the matching predicate are given to the developer for subsequent treatment. Spoon has multiple built-in filters.

3.4.2 Processing Source Code Elements

Program analysis is a combination of a query and analysis code. This is combined in a processor, which is a class that focuses on the analysis of one kind of program elements. The elements to be analyzed are given by generic typing: the programmer declares the AST node type under analysis as class generics. The process method takes the requested element as input en does the analysis. Multiple processors can be used at the same time. The launcher applies them in the order they have been declared. Processors are implemented with a visitor design pattern.

3.4.3 Source Code Transformation

Spoon is designed to facilitate source code transformation. Source code transformation is a program transformation at the source code level, as opposed to program transformation performed on binary code. JFEX doesn't alter the source code that it analyses so the transformation mechanisms of Spoon fall outside the scope of this research. The interested reader is referred to the work of Pawlak et al. [39].

Chapter 4

Data

In this chapter an overview is presented of the data that is used in this research. We begin by exploring the characteristics of the Fish dataset, which will be used in a supervised learning setting to predict manual grades. Next, the five problems in the dataset from Aspiring Minds are introduced. The properties of the problem sets are also discussed in some detail as this may provide clues regarding the differences in comparison with the Fish dataset properties.

The Aspiring Minds dataset serves to gain insight in the predictive power of our features compared to those derived by Aspiring Minds. The last two sections of this chapter are dedicated to a comparative analysis of the feature sets.

4.1 Codility Dataset

As an online provider of specialized programmer tests, Codility offers training material on their website. Codility provided us Java solutions for the Fish assignment, for which I hand-labeled a subset of the solutions with grades ranging from one to five. The task description is provided below. The solutions in the dataset are produced by people all over the world. Most of these people are novices but more experienced programmers may also have tried this training task in order to prepare themselves for a real assessment.

4.1.1 Fish

You are given two non-empty zero-indexed arrays A and B consisting of N integers. Arrays A and B represent N voracious fish in a river, ordered downstream along the flow of the river. The fish are numbered from 0 to N. If P and Q are two fish and P $<$ Q, then fish P is initially upstream of fish Q. Initially, each fish has a unique position.

Fish number P is represented by A[P] and B[P]. Array A contains the sizes of the fish. All its elements are unique. Array B contains the directions of the fish. It contains only 0s and/or 1s, where:

- 0 represents a fish flowing upstream
- 1 represents a fish flowing downstream

If two fish move in opposite directions and there are no other (living) fish between them, they will eventually meet each other. Then only one

fish can stay alive: the larger fish eats the smaller one. More precisely, we say that two fish P and Q meet each other when P $<$ Q, B$[P] = 1$ and $B[Q] =$ 0, and there are no living fish between them. After they meet:

- If $A[P] > A[Q]$ then P eats Q, and P will still be flowing downstream
- If $A[Q] > A[P]$ then Q eats P, and Q will still be flowing upstream.

We assume that all the fish are flowing at the same speed. That is, fish moving in the same direction never meet. The goal is to calculate the number of fish that will stay alive. The assignment is to write a function:

class Solution { public int solution(int[] A, int[] B); }

that, given two non-empty zero-indexed arrays A and B consisting of N integers, returns the number of fish that will stay alive.

Assume that:

- N is an integer within the range $[1..100,000]$;
- each element of array A is an integer within the range $[0..1,000,000,000]$;
- each element of array B is an integer that can have one of the following values: 0, 1;
- the elements of A are all distinct.

Complexity:

- expected worst-case time complexity is O(N)
- expected worst-case space complexity is O(N), beyond input storage (not counting the storage required for input arguments)

When a person starts the task he or she has 120 minutes to complete it. The task can be developed in the provided web editor, or in any IDE after which the code can be copy-pasted it into the Codility editor. The code can be run as many times as necessary with custom defined test cases. Whenever the solution is submitted the code cannot be changed anymore and the candidate is forwarded to the scoring report.

4.1.2 Automated Scoring: Test-Cases

The total score ranges from 0 to 100 percent. It is represented as two equally weighed components: Correctness and Performance. The score of these components is determined from the results of the following test-cases:

- **Example:** tests the example that is provided in the task description.
- **Extreme one (Extreme small):** only one fish.
- **Extreme two (Extreme small):** two fishes.
- **Simple one:** simple order test.
- **Simple two:** simple order test.

- **Small random:** average length: 100 fishes.
- **Medium random:** average length: 5.000 fishes.
- **Large random:** average length: 100.000 fishes.
- **Extreme range one:** all except one fish flowing in the same direction.
- **Extreme range two:** all fishes flowing in the same direction.

All ten test-case results have the following output levels: OK, RUNTIME ERROR, WRONG ANSWER, and TIMEOUT ERROR. These outcome gradations have been transformed to dummy variables in the test case features sets.

4.1.3 Grading Criteria

Assigning grades from categories 1 to 5 to solutions for the Fish task is not trivial. Grading programming solutions is a complex process driven mainly by subjective evaluation criteria of a given assessor. This is mainly due to the fact that software quality is not a unitary concept. To answer which of two different solutions are of higher quality, one must know which quality factors should be optimized given the purpose of the task. Furthermore, each assessor is somehow biased; meaning that the assessor is not completely strict in assigning grades to solutions.

There are also some other factors contributing to bias in grades, for example: mistakes or a too rough/fine grained grading scale. In order to obtain more objective scores it is common practice to use two assessors to score solutions and let them come to a consensus whenever they disagree. Unfortunately such resources were not available to this research so all of the manual labeling was done by me.

It turned out that distinguishing really bad solutions from perfect solutions is quite straightforward but there are many levels in between for which there is no golden rule. In order to establish the most realistic grading criteria possible I imagined that I truly am a recruiter. People have to make this one Fish task for me, and I have to assign a score from categories 1-5. In this score it is reflected that each level up, a person is better than all grades below.

Grades in categories 4 and 5 indicate a decent solution, and people with this score are eligible for hiring. Grades in categories 1 - 3 identify solutions with severe flaws. What are severe flaws? What determines if a solution scores category 3 or category 4? Well, the answer is: me. The "recruiter" or "assessor" in the scenario is the one who determines the acceptable mistake ranges.

The levels have been defined as follows:

- **Grade 5:** Perfect solution, conform complexity requirements.
- **Grade 4:** Very good solution, but is not CAT 5 because a minor mistake causes test-cases to fail.
- **Grade 3:** A good algorithmic approach is present but complexity requirements are violated. Or a solution that contains a mistake which indicates a lower level of understanding of the Java language or data

structures. The presence of this mistake disrupts the otherwise possibly correct flow of the algorithm. The fact that the mistake remains in the submitted code also indicates that the programmer was not able to properly test his code.

- **Grade 2:** The solution contains errors that indicate a misunderstanding of the problem. Any problem having to do with the ordering of the fish or the arrangement and survival of fights between fish causes a solution to score no higher than CAT 2.

- **Grade 1:** Barely a solution. Incomplete logic or a missing algorithm structure that could resemble a correct solution approach. It could also be a decent starting algorithm structure but flooded with so many CAT 2 mistakes that the solution can no longer compete with other CAT 2 solutions.

For the sake of brevity only the general category descriptions are provided here. However, to enforce consistency a full list of encountered issues has been maintained during the grading process. An overview is presented in the appendix. Coding style does not influence any of the categories unless the code is truly chaotic or barely readable. For example a solution with a test case score of 100% scores no higher than category 4 if the code is really messy and hard to understand.

4.1.4 Data Analysis

The graded dataset consists of 230 graded Fish solutions. Figure 4.1 displays the distribution of the test scores in the dataset. The colored levels indicate the distribution of grades for solutions with a specific test case score. The darkest level indicates solutions that fall in scoring category 1. Grade 1 occurs in solutions scoring between 12% and 55% on the test cases. Grade 2 stretches a bit further, from 11% to 77%. Grade 3 covers almost the complete range: 12% to 100%. But only one solution in category 3 scored 100%, most of the category 3 solutions scored 75% to 88%. A very similar pattern is displayed for grade category 4. Finally, grade level 5 consists almost solely of solutions that scored 100%.

The overlapping distribution of grades over the test cases cores indicates that test scores alone are not a perfect mapping to a solution's "true" score. The concentration of almost all category 5 solutions in the 100% bar indicates that there are very few solutions (3, to be precise) in our dataset that are undeservedly recognized as good by the test cases (a hypothesis was that this might occur due to for example hard coding). Inspection of these solution shows that the reason for not scoring 5 is usually code that is unnecessary complicated, or contains entire pieces of wrong code that aren't executed because they are shielded by logic conditions that are never set to true.

Even though the mapping is not perfect, there does seem to be a recognizable pattern where lower scoring solutions on the test cases are more likely to fall in a low grade category, and vice versa. This is a desirable observation because test cases are an established scoring metric. Although it is a rough metric, the general results it produces are widely recognized as indicative of program quality. If my grades would have shown no (or even

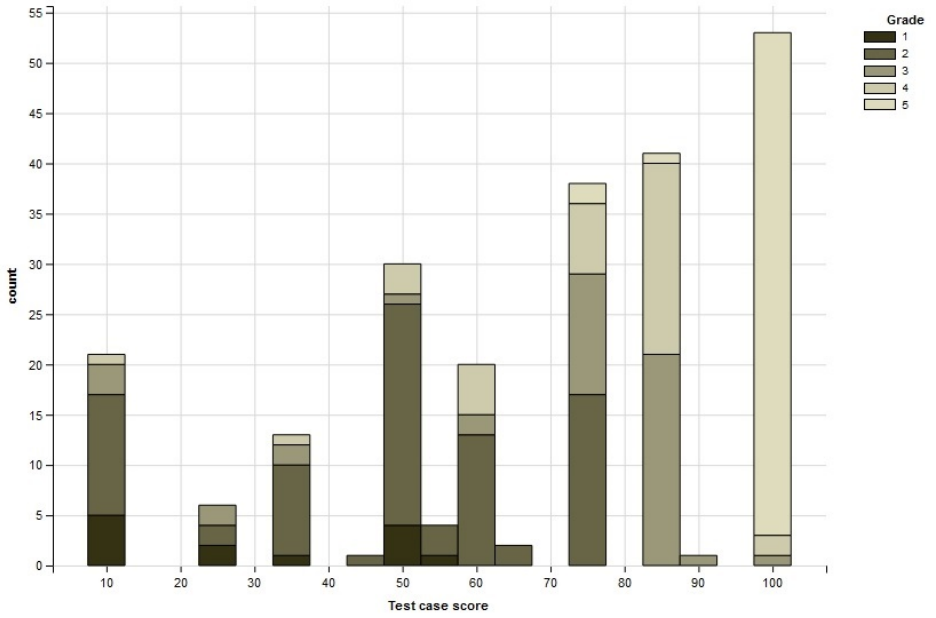

FIGURE 4.1: Bar plot of the test case scores on Fish solutions. Colored levels indicate the distribution of grades per test case score.

negative) correlation with test case scores this would have been difficult to defend.

This test-case correlation is an important baseline for us to compare to the results of our models trained on semantic features. To make the best predictions from test-cases cores, we simply assign a solution to the most likely grade category. So we simply define thresholds for the test-scores that map to grading categories based on the observations in our dataset. To be able to asses the accuracy of such a prediction technique I randomly assigned 161 observations (70% of the data) to a training set. Based on the labeled solutions in the dataset the optimal thresholds resulted in:

- Grade 1: test score $\leq 25\%$
- Grade 2: test score $\leq 62\%$
- Grade 3: test score $\leq 88\%$
- Grade 4: leave out, do not predict.
- Grade 5: test score $\leq 100\%$

The Pearson Correlation value *r* for the true grades and scores in the training set is 0.75 . The correlation in the training set for the true grades and the predicted scores according to the thresholds mentioned above is 0.83. This are not the results we are going to use as a baseline for comparison. For that we use the test set holding the 69 other observations. These observations were no part of the definition of the thresholds above. The correlation in the test set between the scores and true grades is 0.72, which is comparable to the training set (which is to be expected from a random

subset). When we predict the test set scores with the thresholds the correlation of the predicted grades with the true grades is 0.80. This is similar to results reported in [45] where the validation *r* for test-case results was 0.54, 0.80, 0.64, 0.80, and 0.84 for five different problems.

Fish task: validation set – test score predictions					
True Scores	Predicted Scores				
	1	2	3	4	5
1	1	1	0	0	0
2	7	14	6	0	0
3	4	3	6	0	0
4	0	2	11	0	0
5	0	0	1	0	13

FIGURE 4.2: 49% of the predictions have the correct grade, 90% of the predictions fall within a one grade shift and 10% have shifted 2 grade categories.

Predictions	Fish : N=69 simple test-case predictions	Encrypt N = 33 Best semantic model predictions	List Primes, N = 92 Best semantic model predictions
Correct	49%	58%	77,2%
1-grade shift	41%	36%	7,4%
2-grade shift	10%	6%	4,3%
3-grade shift	-	-	1,1%

FIGURE 4.3: Illustration of prediction accuracy reported in [45] versus prediction accuracy of simple test-case score predictions on our own validation set.

Figure 4.3 illustrates the possible gain in prediction accuracy when using semantic models instead of test-case results. This is only an indication as the performance on different problems might not be comparable. But it seems reasonable to hypothesize that our predictions could be improved by using semantic features in addition to test-cases as this increased the prediction accuracy for two other problems. To check if the hypothesis also holds with the Fish task Chapter 6 compares the best model predictions based on all individual test case information against model predictions using either just semantic features or a combination with test case information.

4.2 Aspiring Minds Dataset

This section will discuss the tasks that are included in the dataset from Aspiring Minds. I received a complete dataset: solutions that are tagged with their feature vectors and labeled with the grade consensus of two experts and a test case score based on Aspiring Mind's test case suites. Inspection of the dataset provides a good setting for comparison with the properties found in the Fish dataset that is tagged and labeled by me. The final part of this section assesses the influence of the difference in feature granularity in comparison to our own features.

4.2.1 EliminateVowel

Task description: Given a string *"string"*, write a program to eliminate all the vowels (lower or upper cases) from it. The list of vowels (a,e,i,o,u) is provided. The input to the the method **eliminateVowelString** of class **EliminateVowel** shall consist of a string that will only contain letters from A to Z in upper or lower cases. The method should return the string without vowels. Two hints are given:

- The *length( )* method returns the length of a String in Integer format. Usage: *int len = string.length();*
- The *toCharArray()* method converts a String to a Character array. Usage: *char[ ] str = string.toCharArray();*

Dataset properties: There are 182 graded solutions for the EliminateVowel problem. Figure 4.4 provides an overview of the distribution of grades per test cases core. There is a very sharp distinction noticeable in the test case results: all solutions score either 100% or below 20%. All solutions that score 100% score category 5, so the test cases correctly identified good solutions. However, the lower scoring subset contains many solutions that scored 4 or 5, even when test-cases indicated a score of 0% percent. This has to do with the nature of the assignment: characters and strings are surrounded by single or double quotes. Solutions that precede these quotes with backslashes trigger a compiler error, which prevents test cases from assessing code quality.

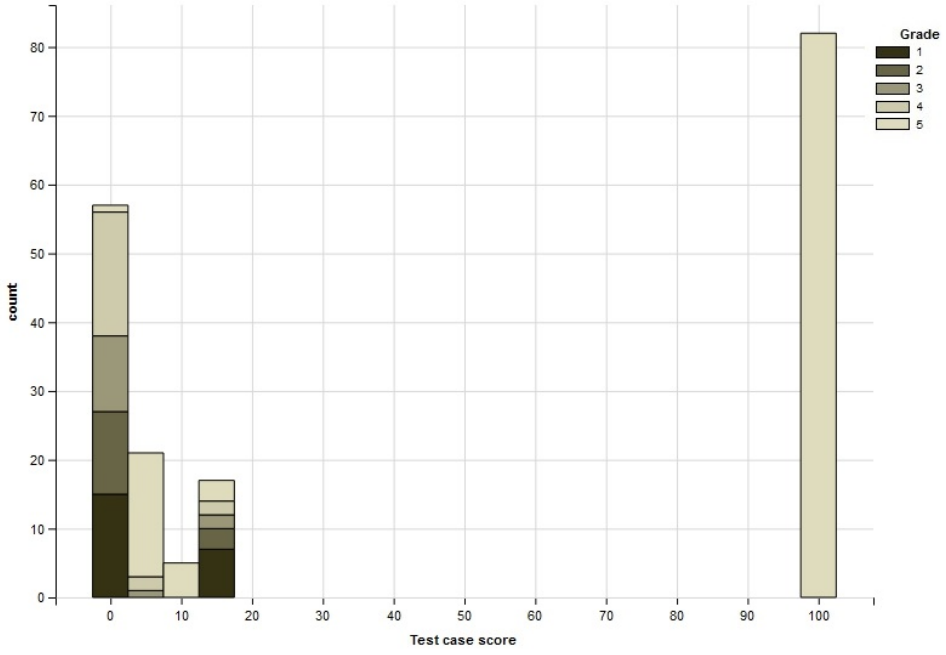

FIGURE 4.4: Bar plot of the test case scores on EliminateVowel solutions. Colored levels indicate the distribution of grades per test case score.

4.2.2 IsTree

Task description: Given an undirected connected graph in adjacency matrix form, determine if it is a tree or not. In a connected graph, there exists a path from every node to every other node in it. This path does not need to be an edge directly connecting the nodes. An adjacency matrix for a graph with n vertices is a n x n two-dimensional matrix with i,j entry as 1 if there is an edge from ith vertex to jth vertex and 0 otherwise. The matrix contains 0s and 1s only. The input to the method **isTree** of class **GraphTree** shall consist of an undirected connected graph represented by adjacency matrix grid. The method should return 1 if the graph is a tree otherwise it returns 0. An example input is provided for which the method should return 1:

0101
1010
0100
1000

Dataset properties: The IsTree dataset contains 124 graded solutions. As indicated by Figure 4.5, the test scores are spread over the range between zero and a hundred, with a very large concentration of solutions that scored 0% on the test cases. The grade distribution depicts that almost three quarters of the 0%-scoring solutions are indeed lousy programs. However, some of the programs have a score of category 4 or 5.

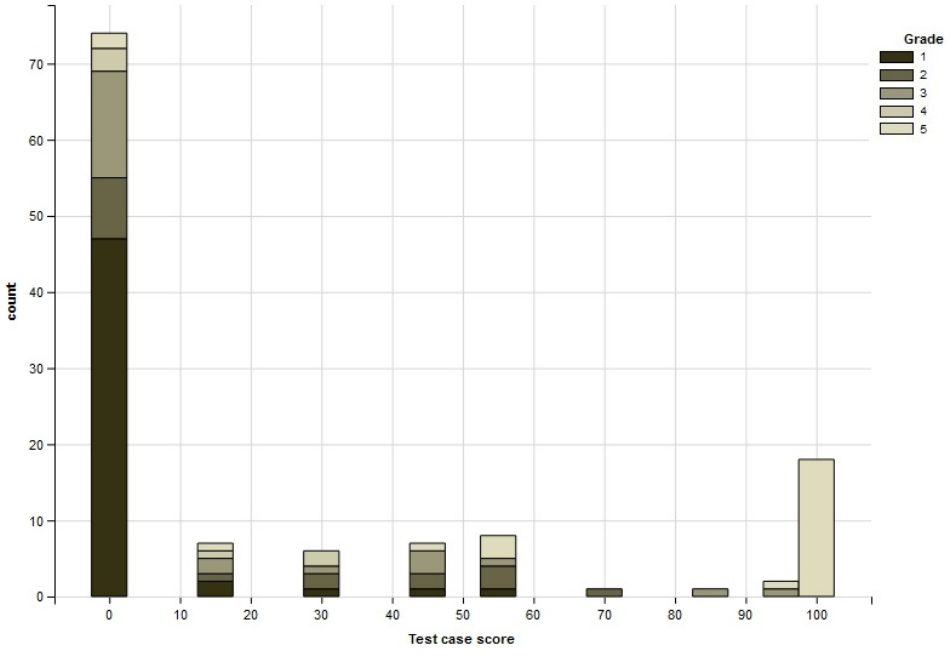

FIGURE 4.5: Barplot of the test case scores on IsTree solutions. Colored levels indicate the distribution of grades per test case score.

4.2.3 PatternPrint

Task description: Given an integer N, print N lines in the following manner. For e.g., if N=6:

1111112
3222222
3333334
5444444
5555556
7666666

The input to method **patternPrint** of class **NumberPattern** shall consist of an integer $1 \leq N \leq 100$ representing the number of lines to printed. Do not return anything from the method but print the required pattern using *System.out.println()* or *System.out.print()*.

Dataset properties: There are 176 graded solutions in the PatternPrint dataset. Figure 4.6 displays the grade levels of the test case score distribution. Most solutions score less than 10%, and the grade levels indicate that for the majority of these solutions this score is justified. A small subset of the solutions that should have scored higher have grade 4 or 5, the rest has grade 3. Once again, solutions that score100% have earned this score. A few solutions score in the midrange section of the test case predictions. There does not seem to be a clear pattern in the grade distribution amongst them.

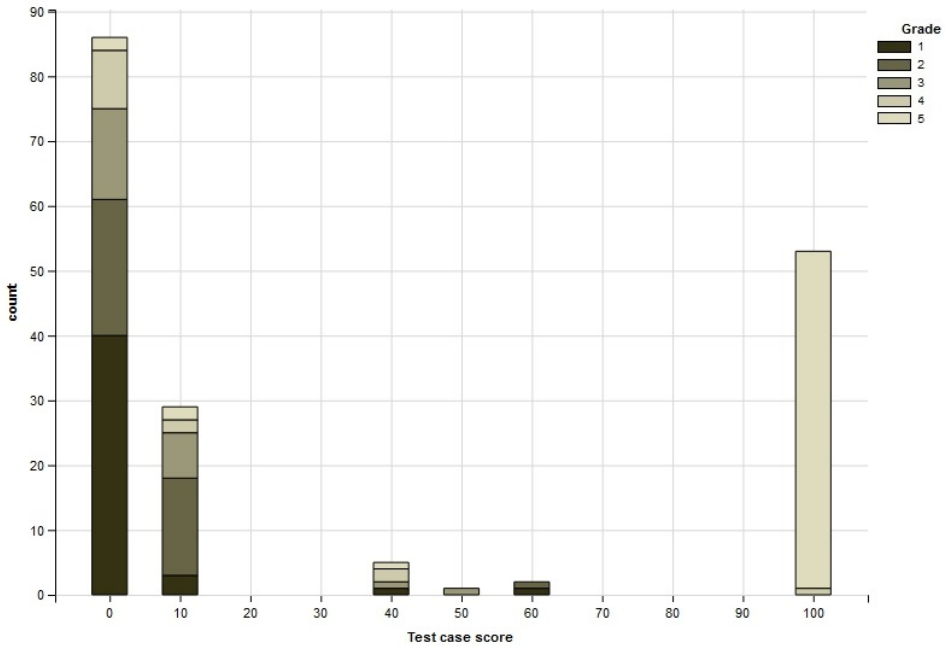

FIGURE 4.6: Bar plot of the test case scores on PatternPrint solutions. Colored levels indicate the distribution of grades per test case score.

4.2.4 GrayCheck

Task description: Given two bytes as input, your task is to find out if they can be placed successively in a gray code sequence. If they can be placed, return 1 else return 0. In a gray code sequence, two successive values differ in only one bit. Input to the method **GrayCheck** of class **GrayCheck** shall consist of two bytes *term1* and *term2* and should return an integer. The following hints are given:

- ˆ operator is a bitwise XOR operator, & is a bitwise AND operator and | is a bitwise OR operator
- << and >> are shift left and shift right operators
- 0x is used to represent data in hexadecimal form.
- To assign hex code to a byte use: *byte ch = (byte)0x03*;

Dataset properties: The GrayCheck dataset consists of 175 graded solutions. With an exception for a small subset that scores around 60%, all of the category 1 programs have been scored less than 40% by the test case suite. It also performed well for the good programs: all solutions that scored above 70% are category 4 or 5 solutions. However, a substantial amount of the category 4 solutions have been assigned a score below 70%, and a few category 5 solutions scored 0%.

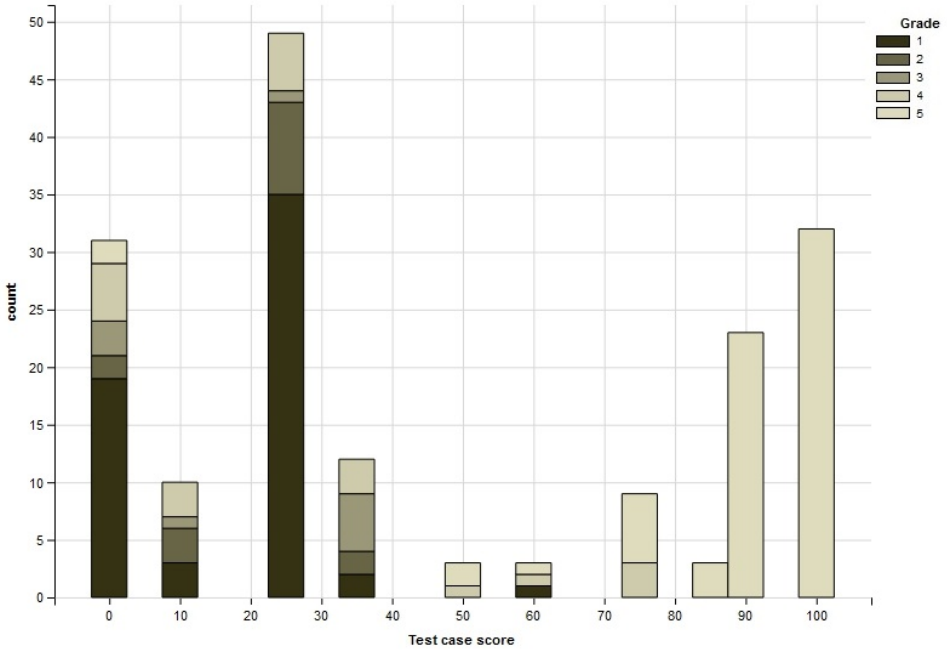

FIGURE 4.7: Barplot of the test case scores on GrayCheck solutions. Colored levels indicate the distribution of grades per test case score.

4.2.5 TransposeMatrix

Task description: Given values m and n as the dimensions of an increment matrix and an initial value s, multiply the original increment matrix with its transpose. An increment matrix is the matrix whose elements are the incremented values of the initial values s. The input of the method **transposeMultMatrix** of class **TransposeMult** shall consist of the initial value s and the dimensions of the increment matrix m and n (all values are positive integers). The method should return a 2-dimensional matrix representing the multiplication matrix. For example, if the initial values are $s = 1, m = 3, n = 3$:

increment matrix	**transpose matrix**	**multiplication matrix**
123	147	14 32 50
456	258	32 77 122
789	369	50 22 94

The following hint is provided: to declare a matrix of dimension m and n use: $int[][]matrix = newint[m][n];$

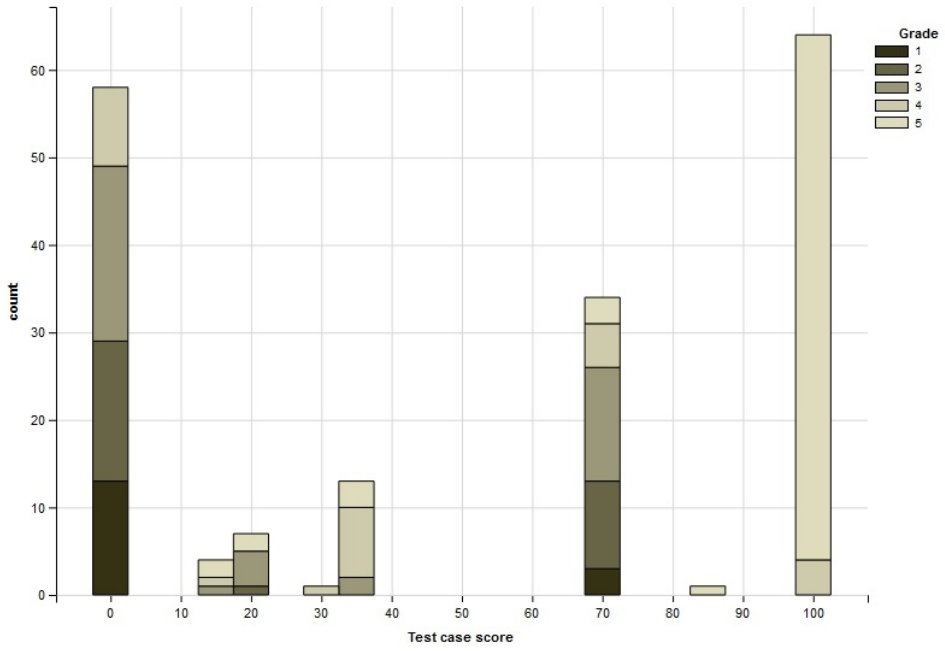

FIGURE 4.8: Barplot of the test case scores on TransposeMatrix solutions. Colored levels indicate the distribution of grades per test case score.

Dataset properties: There are 182 graded solutions in the TransposeMatrix dataset. Solutions scoring above 80% are correctly identified as good solutions. Solutions scoring below 70% have scattered grades among the test case scores. Except for half of the 0% scoring solutions there doesn't seem to be much agreement between the test case scores and assigned grades.

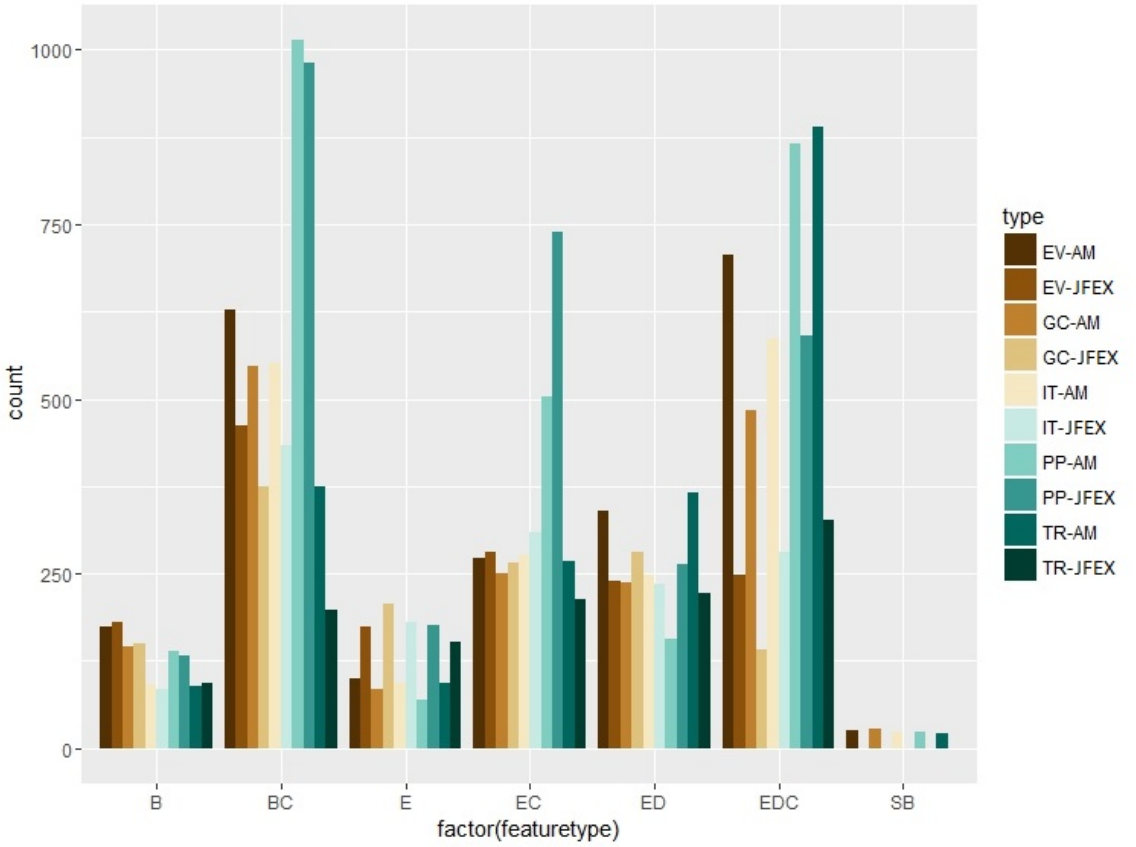

FIGURE 4.9: Counts of unique features in the Aspiring Mind (AM) feature set and our own feature set (JFEX), per feature category for all problems.

4.2.6 Feature Granularity Comparison

Figure 4.9 provides some insight into the differences between the features generated by Aspiring Minds and our own features. The unique generated features are displayed per feature category for all five problems in the Aspiring Mind dataset. We see that both approaches generate roughly the same amount Basic features (B). For the PatternPrint (PP) problem the number of Basic Context features (BC) is nearly equal as well. However, for the other four problems the AM approach generates 158 more unique Basic Context features. We see an opposite pattern for Expression features (E), where JFEX produces on average 90 more unique features than the AM approach.

We have seen that the addition of context to basic features caused the AM dataset to generate more Basic Context features than JFEX. An identical influence seems to affect the Expression features when context is added, because the number of Expression Context features (EC) in both feature sets are approximately equal. The only exception is once again the PatternPrint problem, which is consistent with the observations in the Basic Context feature category.

The difference between the two feature sets for Expression Dependency features (ED) is not as clear as in the previous categories. The GrayCheck

and IsTree problems generate roughly the same amount of Expression Dependency features. EliminateVowel and TransposeMatrix trigger more Expression Dependency features in our feature set than in the feature set of AM. The opposite is the case for the PatternPrint problem. Once again we see that the addition of context to a feature category generates more features in the AM feature set than in our own feature set: for all problems, including PatternPrint, the AM Expression Dependency Context features outnumber their counterparts in our feature set.

Super Basic features (SB) is a class that is not specially defined in our feature set. The super basic features in the AM dataset have some overlap with our basic features. The following section provides some clarification on the exact differences between the feature sets for all encountered feature categories. Chapter 7 discusses the influence of the feature differences on prediction accuracy.

4.2.7 Concrete Differences

Figure 4.10 shows the source code of a solution to the EliminateVowel assignment. I have added some indentation to make the code more readable, as the original solution contained all if conditions right below each other. However, I did not alter the indentation correctly: the placement of the else statement seems to suggest that it belongs to the first if statement. This was probably the intuition of the programmer. However, Java connects an else clause to the closest if statement, which in this example is $if(string.charAt(i) ==' U')$. But as the test case score of 0 might indicate, this is not the only issue. The code will always return the memory address of the empty character vector s.

The Aspiring Mind feature approach generates 167 features for this particular solution, and JFEX generates 122 features. We will now discuss the most notable differences.

Basic features: The AM feature set contains 11 Basic features that describe nested control structures: BASIC@Loop_IF:2, all the way up to BASIC@Loop_IF_IF_IF_IF_IF_IF_IF_IF_IF:2. The JFEX Basic feature set does not include information about nested control flow structures. The intuition behind this is that these features are redundant because all their information is captured in the Context features. We also notice that all features that occur within a loop structure have double feature counts. This pertains to a loop-unrolling step done in the Aspiring Minds approach. JFEX skips this step as the result seems to be a constant increase of feature counts.

The Basic features from JFEX that count the variable types (for example String or int) and variable categories (such as local or parameter variable) are not present in the AM set.

Context features: The AM features do not differentiate between the position within a for loop condition, whereas JFEX features do capture these differences. Another difference is that AM distinguishes the method body as a context type, which is not the case in JFEX. So the return statement on line 26 in Figure 4.10 is not part of the JFEX Basic Control features, but it is part of the AM Basic Control feature set as BASIC_CNTRL@return()@m:1. Our decision not to include the method body as context is once again motivated by the wish to reduce the number of features that capture the same

information. This was also our motivation to not include loop unrolling, which introduces many extra features in the AM set, such as BASIC_CNTRL@Loop_IF@m_Loopb1:2 up to BASIC_CNTRL@Loop_IF_IF_IF_IF_IF_IF_IF_IF_IF@m_Loopb1_IFb1_IFb1_IFb1_IFb1_IFb1_IFb1_IFb1_IFb1_IFb1:2. Which is basically just the whole control sequence in the assignment, in the context of itself. Notice the numbers that are mentioned in the context part of the AM feature. These numbers indicate the position of the control structure. There is only one loop so Loopb1 doesn't need any explaining. However, the number of ifs are all referred to as 1 which doesn't seem to very informative. In JFEX there is no such enumeration for control structures.

Expression features: JFEX generates more expression features as it differentiates between variable categories whereas the AM featureset regards all variables as type VAR. For the operators JFEX is also more expressive, specific types of operators are mentioned in the features whereas the AM features separate features based on their type (such as a relational or arithmetic operator). This is also the case for method calls: they are explicitly captured by name in JFEX, and only as type FNCALL in AM.

We have now seen a concrete example of the feature differences. The feature sets demonstrate decisions to include more or less details were made based on different opinions. The effect of these decisions is discussed in Chapter 7.

```java
import java.util.*;
public class ev7t
{
 // METHOD SIGNATURE BEGINS, THIS METHOD IS REQUIRED
  public static String eliminateVowelString(String string)
  {
    char[] str=string.toCharArray();
    char[] s=new char[string.length()];

    for(int i=0;i<str.length;i++)
    {
     if(string.charAt(i)==\'a\')
        if(string.charAt(i)==\'e\')
            if(string.charAt(i)==\'i\')
                if(string.charAt(i)==\'o\')
                    if(string.charAt(i)==\'u\')
                        if(string.charAt(i)==\'A\')
                            if(string.charAt(i)==\'E\')
                                if(string.charAt(i)==\'I\')
                                    if(string.charAt(i)==\'O\')
                                            if(string.charAt(i)==\'U\')
                                                break;
      else
        s[i]=string.charAt(i);
    }
    return s.toString();
  }


  public static void main(String[] args)
  {
    // PLEASE DO NOT MODIFY THIS FUNCTION
    // YOUR FUNCTION SHALL BE AUTOMATICALLY CALLED
    String string=\"abcde\";
    String result;
    // ASSUME INPUTS HAVE ALREADY BEEN TAKEN
    result=eliminateVowelString(string);
    System.out.println(result);
  }
}
```

FIGURE 4.10: Solution to the EliminateVowel problem from the Aspiring Minds dataset. Test case score: 0%. Grade: 2.

Chapter 5

Predictive Modeling

This chapter discusses the different modeling techniques that have been applied to the Fish dataset. We begin with an explanation of the overall setup and a motivation for the chosen predictive modeling techniques. Next, we pay attention to preprocessing steps for the data. The nature of the dataset has been explored in the previous chapter, and the high sparsity of the features may benefit from preprocessing. In the final part we discuss the predictive modeling techniques that are used in the experiments to model the Fish grades. The results of the modeling process are presented in the chapter seven.

5.1 Learning Setting

The data for our supervised experiments consists of a set of Java solutions for the Fish assignment. Each solution is represented as a vector of program features X_p: the predictor variables. We want to obtain the relationship between these predictor variables and the response variable Y, the grading category. This relationship is unknown to us. Therefore, our best option is to approximate this relationship and try to determine the quality of the relationship that we defined. This is not as straightforward as it sounds, because how does one determine the quality of an approximation when the original is unknown? This is a question that has been faced many times by researchers and as a result a mature but ever improving body of work on statistical learning and validation has arised.

Before we do any modeling of the relationship between the outcome and predictor variables we have to decide if we analyze our problem as a regression or classification problem. For regression problems, the outcome variable Y is quantitative. In classification problems Y takes values from a finite set.

The levels of the Fish grading rubric are coded as consecutive integers from 1 to 5. We are dealing with **ordinal variables**: we can rank the categories but we can not assume the distance between the categories to be equal. Unlike real-valued regression labels, ordinal class labels do not carry metric information. Ordinal labels are also different from the labels of multiple unordered classes due to the existence of ordering information. Ordinal classification deals with these kind of problems by trying to exploit the monotone relationships between the ordinal levels (see section 5.7).

The grading process can thus be defined as: assigning an ordered class label to an unlabeled observation. Based on our training data we want to:

- Accurately predict the grade categories of unseen solutions.
- Understand which inputs affect the prediction and how.
- Assess the uncertainty in each prediction.

Traditional methods for modeling an ordinal response (such as best subsets, forward selection, and backward elimination procedures) often assume independence among the predictor variables and require that the number of samples (N) exceeds the number of predictors (P) included in the model. This is obviously not the case four the Fish dataset where $N = 230$ and $p = 5705$. These thousands of feature predictors make our data *high-dimensional*.

Penalized models are known to have excellent performance for high-dimensional datasets in fitting linear and logistic models. We will therefore apply Ridge and Lasso regression to the data. As noted before, modeling five discrete classes with a regression approach has certain shortcomings. First of all, as the predicted value are real values, one has to decide if a prediction of 2.45631 belongs to category 2 or category 3. A cutoff point at 2.5 seems a reasonable suggestion. However, the ordinal labels carry no metric information so a cutoff point at 2.5 is completely arbitrary. Fitting the cutoff points to levels where they best predict the training set might make the model vulnerable to overfitting. Secondly, there could be multiple sets of cutoff points that generate the same prediction accuracy, in which case one would still have to randomly select one. In addition, penalized methods have not been fully extended to the ordinal response setting.

To benefit from possible information that is captured by the ordering of the classes we also apply ordinal classification (also known as ordinal regression) to the data. In order to assess if there is indeed hidden information in the label ordering we contrast the results with the predictions of a multinomial classification model. Multinomial classification makes no assumptions on the ordering of labels.

We now first discuss the options for preprocessing the data, as this may improve the prediction accuracy of our models. In the remainder of the chapter the modeling techniques for the regression and classification settings are explained.

5.2 Data Preprocessing

Data preprocessing refers to adding, deleting or transforming the data in the training set. Data preprocessing can profoundly influence the prediction accuracy of models. Some models might be more sensitive to different types of predictors than other models. It may also be of interest how the predictors enter the model.

A straightforward first step in the preprocessing of the predictors is to center and scale them, two techniques that may improve the numerical stability of some model calculations. Centering shifts a predictor's variable to

a zero mean by subtracting the average predictor from all the values. For scaled predictors, each variable is divided by its standard deviation which results in a common standard deviation of one.

Many of the modeling algorithms applied in this research handle the centering and scaling of predictors for us. In addition, given our large set of predictors, we will focus on feature selection and extraction techniques that generate a smaller set of predictors.

5.3 Feature Selection

The process of feature or variable selection aims to identify a subset of features that are relevant with respect to a given task. In regression and classification tasks this usually comes down to the subset of variables with the highest predictive power. Motivations for performing feature selection are: improving performance of the predictive model, avoid the cost associated with measuring all features and provide a better understanding of the predictive model.

For the Fish dataset we have 230 observations of 5705 features. This typically means that we will be able to find a function that can classify the examples in the training set pretty well, without this necessarily meaning that it will have good performance for the test set. The underlying problem is overfitting. A simple guarding rule against overfitting is to choose a simple function over complicated functions.

High dimensional data introduces what is known as *the curse of dimensionality*: the volume of the feature space increases so fast that the available data becomes sparse, and it gets very hard to find reliable clusters. The concept of distance becomes less precise as the number of dimensions grows. There might also be irrelevant features which may obscure the effect of the relevant ones. Furthermore, given a large number of features, it is likely that some of them are correlated. This is known as (multi)collinearity, which increases the variance of all coefficients and degrades the predictability of the model.

A distinction that is often made in feature selection techniques is between scoring/filtering and wrapper methods. Scoring/filtering methods involve ranking the features by a given criterion. Each feature is scored by itself, and the selection of features does not depend on other features. We apply filtering by selecting features that occur in at least 21 and at least 77 solutions, resulting in two different feature sets to be used for modeling. Wrapper methods pick new features by how much they impact the classifier given the features already selected.

5.3.1 Filtering near-zero Variance Predictors

Features that only occur in a minority of solutions are unlikely to have a large positive impact on the predictive power of a model. When resampling techniques are used is is very likely that the resampled sets do not contain any solutions with this particular feature. To identify these features we look at the frequency of their occurrences. The problem is indicated by severely disproportionate frequency of the feature occurrences. To filter

these features from our predictor set we follow the rule of thumb for detecting near-zero variance predictors proposed by Johnson and Kuhn[31]:

- The fraction of unique values over the sample size is low (say 10%)
- The ratio of the most common frequency to the second most common frequency is large (say around 20)

As this is an heuristic method, there is no golden standard for the levels that are suggested. For our application we have to raise these levels considerably. When analyzing the frequency ratio of the Fish predictors we observe the following distribution:

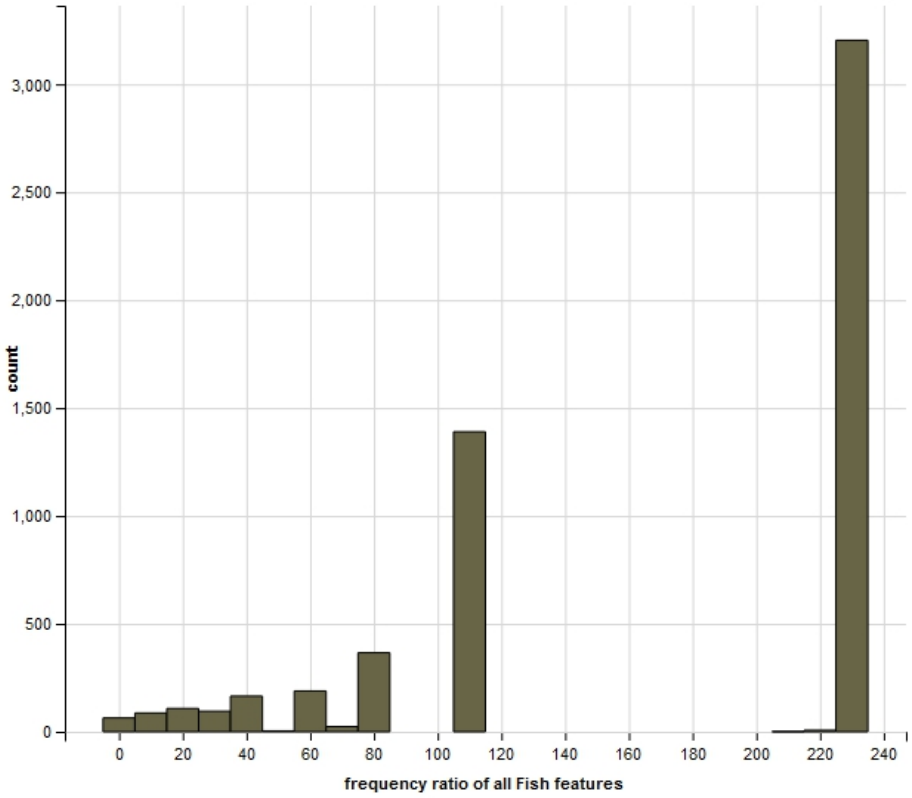

FIGURE 5.1: Histogram of the frequency ratios of all Fish features.

The clear split in Figure 5.1 makes it easier to decide on the discriminating frequency ratio level. A total of 3212 features have a frequency ratio of 214 or higher. The highest ratio of 229 is observed for 3085 features. A ratio of 229 most likely indicates in our situation that the feature occurs in one solution only (technically it is also possible that the feature occurs in all except one solution). Removing these features from the fish dataset results in 2.493 remaining features.

5.3.2 Between-Predictor Correlations

When a pair of predictors have a are substantially correlated this is known as collinearity. Collinearity affects the variance of all coefficients and degrades the predictability of the model. To detect highly correlated pairs of

predictors we extracted them from the correlation matrix of the predictors. This was done for the cutoff point of a between-predictor correlation of 0.80, 0.85. 0.90, and 0.99.

5.4 Feature Extraction

Feature extraction is the process of combining multiple predictors in single variables. Dimensionality reduction is an alternative path to reduce the number of features. A method such as Principal Component Analysis (PCA) is applied to the entire dataset matrix. The original matrix is transformed into a new, low dimensional matrix with the same number of rows but a reduced number of columns. Predictors are combined in linear combinations (called the components), that capture the most variability of all possible components. Then, subsequent components are derived such that these linear combinations capture the most remaining variability while also being uncorrelated with all previous PC's. It is not guaranteed to improve results, partially because most reduction techniques ignore class labels in their criteria. This may cause the PCA approach to be misled by high but non-systematic variance. Partial least squares (PLS) methods are basically supervised versions of PCA, where the components are in the direction of the highest covariance with the outcome.

5.5 Penalized Regression

5.5.1 Ridge Regression

The concept of ridge regression was introduced by Hoerl and Kennard[26]. Ridge regression involves constraints on the coefficients. The Ridge solution is chosen to minimize the penalized sum of squares (Equation 5.1). The shrinkage penalty $\lambda \sum_{j=1}^{P} \beta_j^2$ is small when the coefficient values are close to zero, as the total equation is minimized this has the effect of shrinking the coefficient estimates towards 0. The value of the tuning parameter λ is chosen by computing the estimators for a range of λ values and plotting the results against λ. This continuous process yields statistical models having coefficients with non-zero estimates for important covariates, while many coefficients are shrunken towards zero. It doesn't provide variable selection as the penalty enforcement does not force coefficients to be exactly zero.

$$\sum_{i=1}^{n}(Y_i - \alpha - \sum_{j=1}^{P} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{P} \beta_j^2 \tag{5.1}$$

Ridge regression is capable of reducing the variability and improving the accuracy of ordinary least squares models. These gains are the largest in the presence of multicollinearity.

5.5.2 LASSO

Lasso (Least Absolute Shrinkage and Selection Operator) is similar to ridge regression in it's regularization process but in addition it also performs variable selection. It was introduced in 1996 by Tibshirani and is now a widely

used method to generate a compact model for high-dimensional data[47]. Lasso penalizes the sum of the absolute value of the unknown regression parameters (Equation 5.2). A tuning parameter λ is included as a constraint in the least-squares estimates. Lasso yields sparse models which only include a subset of the variables. This causes Lasso to generate better results than Ridge regression in terms of model interpretation and model size.

$$\sum_{i=1}^{n}(Y_i - \alpha - \sum_{j=1}^{P} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{P} |\beta_j| \tag{5.2}$$

5.5.3 Implementation: glmnet

For the implementation of the Ridge and Lasso models I used the glmnet package for R[24]. The glmnet package uses the elastic net family of penalties: ridge and lasso and hybrids in between and solves the penalized residual sum of squares. The regularization path for the lasso is computed at a grid of values for the regularization parameter lambda. It solves the lasso problem by coordinate descent: optimize each parameter separately, holding all the others fixed. This cycles around until coefficients stabilize. It does this on a grid of lambda values, from max to min (uniform on log scale), using warm starts. Lambda max is now represented by the smallest value of lambda for which all coefficients are still non-zero. When lambda min gets close to zero we get close to the unrestricted fit of the model. Glmnet can repeat the fitting process with a variety of loss functions and additive penalties.

5.6 Multinomial Classification

We mentioned before that there are some downsides to using regression predictions to predict categorical class labels. With some adjustments, regression techniques can easily be used for classification. We could perform a regression for each label, altering the output to 1 for training predictions that get this label and to 0 if they do not get this label. The result is a linear expression for the label. Then, given a test example with an unknown label, we calculate the value of each linear expression and select the one that is largest. This scheme is known as multi-response linear regression. Multi-response linear regression often yields good results in practice. But unfortunately the membership values it produces are not proper probabilities because they can fall outside the range 0 to 1.

A related statistical technique called logistic regression does not suffer from these problems. Logistic regression facilitates linear regression to model a binary response. The binary response is transformed to a continues value via a link function. For logistic regression the link function is the logarithm of the odds ratio between the responses (the log-odds or logit). The motivation for modeling the log-odds is the outcomes between [0:1] can directly be interpreted as probabilities. The model parameters are chosen to maximize the log-likelihood. A threshold can be applied to discretize the modeled log-odds to class predictions. There are several methods for solving this maximization problem. A simple one is to iteratively solve a sequence of weighted least-squares regression problems until the

log-likelihood converges to a maximum, which usually happens in a few iterations[51].

Several ways have been defined to generalize logistic regression to >2 classes. One possibility is to proceed in the way described above for multi-response linear regression by performing logistic regression independently for each class. Unfortunately, the resulting probability estimates will not sum to 1. To obtain proper probabilities it is necessary to couple the individual models for each class. Direct comparison of the classes is replaced by a set of binary comparisons. This yields a joint optimization problem, and there are efficient solution methods for this.

Multinomial logistic regression is a generalization approach that assumes no ordering between the class labels. One of the classes is selected to be the reference class and the log-odds of all other classes to that reference class are calculated. These log-odds are then modeled with logistic regression. The coefficients are iteratively calculated to maximize the log-likelihood.

5.7 Ordinal Classification

The encoding from category 1 to 5 makes it tempting to analyze the ordinal outcomes with a linear regression model (LRM). But because an ordinal dependent variable carries no metric information it violates the assumptions of the LRM. Therefore, this approach may lead to incorrect conclusions. It is more appropriate to use models that avoid the assumption that the distances between categories are equal.

The ordinal information carried in the discrete classes seem to introduce two properties[34]:

- **Closeness in rank space:** the cost for assigning the wrong label depends on the "closeness" of the prediction. Accordingly, most cost vectors are v-shaped: increasing equally on both sides as one gets further from the original class label. This condition can be made stronger with convex cost vectors: pay increasingly more if the prediction gets further from the real class.

- **Structure in feature space:** the total order within the predictor variable and the target function introduces a total order in the feature space. Compared to nominal classification, the order between class labels makes that two different class label observations can always be compared using the defining order relation.

As indicated in Figure 5.2, there are several approaches to ordinal regression problems in the domain of machine learning:

- **Naive approaches:** make assumptions in order to simplify classification tasks into other standard problems. An example is mapping the labels to real values and then use standard regression techniques. Other approaches assign different misclassification costs according to the ordering in the classes to predict. The cost matrix is usually related to the absolute difference between true and predicted classes. When more training data is available it is also possible to use nominal classification, thereby ignoring any ordering information.

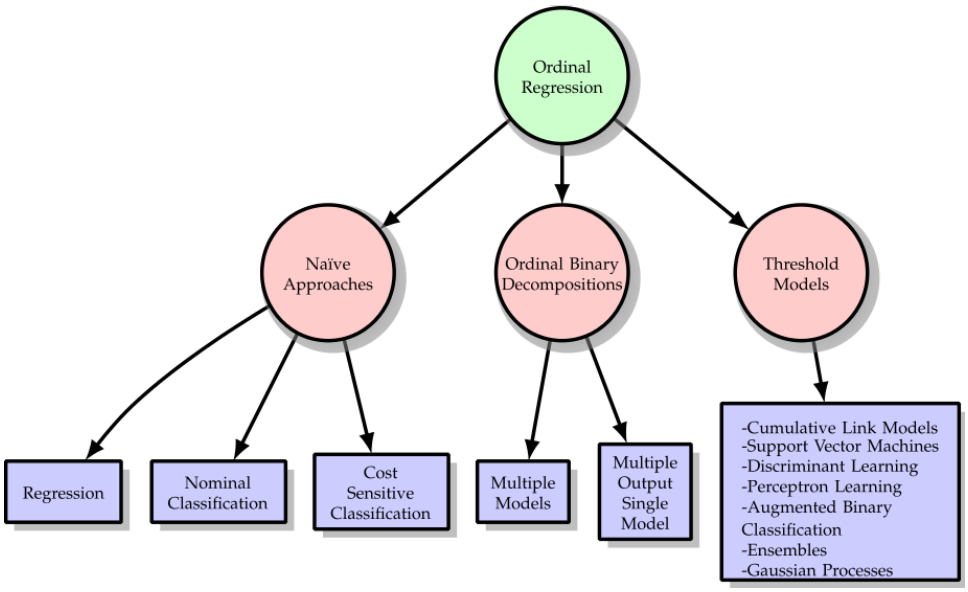

FIGURE 5.2: Proposed taxonomy of ordinal regression methods in [25], image source: *http://www.uco.es/grupos/ayrna/index.php/orreview*

- **Ordinal binary decompositions:** decompose the ordinal classification task into several binary classification subtasks. This is in a similar fashion to multi-class classification problems which are decomposed into a set of binary tasks using One-vs-One or One-vs-All schemes. Ordering information can be incorporated in these decompositions, for example by stating that if an observation belongs to category 3 for example, it also belongs to category 1 and 2.

- **Threshold models:** are based on the assumption that all ordinal class labels originate from an unobservable latent variable. A mapping is generated together with a set of thresholds that divide the projection into intervals that each represent a class.

There are more complex classifiers such as nonlinear SVM's or artificial neural networks which take interactions between features into account. This corresponds to being able to have a nonlinear decision surface. They do not always provide a significant advantage in practical performance and in this study the datasets are too small to learn complicated relationships between features.

Traditional ordinal response models are usually simpler to interpret than multinomial models. Multinomial models generate different sets of slope coefficients for the log-odds of each prediction. Most ordinal response models assume proportional odds and therefore only have one set of slope coefficients regardless of the prediction levels.

An experimental study by Huhn and Hullermeier [28], explores to what extent existing techniques and learning algorithms for ordinal classification are able to exploit order information, and which properties of these techniques are important in this regard. They found that learning techniques

specifically designed for ordinal classification are indeed able to exploit order information about classes and that the less flexible the learner is, the more it benefits from the ordinal structure of the data.

5.7.1 Implementation: ordinalGMFS

The ordinalgmifs R package can fit various ordinal response models when the number of predictors exceeds the number of observations. It extends the penalized approach we have seen before for Ridge regression and LASSO to an ordinal response setting.

The package is based on an adapted version of the Incremental Forward Stagewise (IFS) algorithm. IFS can be used to obtain solutions for LASSO and elastic net penalized models. It is a penalized strategy that enforces monotonicity in a regression setting. The adaptation of IFS is called GMIFS, which is an acronym for Generalized Monotone Incremental Forward Stagewise method[6].

The ordinalgmifs function can be used to fit traditional and penalized cumulative link, forward continuation ratio, and backward continuation ratio models using either a logit, probit, or complementary log-log link. It can also be used to fit adjacent category and stereotype logit models. The ordinal.gmifs function allows the user to specify a model formula, identify the matrix of covariates to be penalized and specify the model type and link function.

Chapter 6

Experiments

This chapter presents the setup of the experiments. We start by discussing in what way the experiments provide answers to the research questions. Next, the set-up of the individual experiments is explained. Section 1.2 of this report states the goals of this project. The research value of the proposed project is motivated in the current chapter.

6.1 Answering the Research Questions

Once the features are defined and extracted for each program we are able to continue searching for the answer to our research question. As explained in Chapter 1, we will guide this process by answering the following sub questions:

- **S1:** Are dependency and control flow features better grade indicators than basic keyword counts ?
- **S2:** Does the granularity of semantic feature definitions affect the performance of prediction accuracy with respect to human grading?
- **S3:** Can we improve grading accuracy by modeling the problem with classification instead of regression?
- **S4:** What are the most influential source code features for classifying when classifying a solution program?

Relevant source code features (S4) are determined by the feature selection step for the predictive models and can be analyzed manually. To determine if dependency and control flow features add value over basic keyword counts (S1), we need to check the ratio of these selected advanced features to the selected basic- and test-case features. The influence of feature granularity (S2) will be assessed by comparing predictions for five problems using two feature sets that differ in granularity. The difference in grading accuracy between classification and regression (S3) can be measured if we discretize the regression predictions. The experiments to answer these questions as well as the main research question are described in the rest of this chapter.

6.2 Research Methodology

We want to test if regression against expert grades can provide better grading than test-case based grading on its own. We also research if a classification approach, which seems to suit the domain, outperforms a regression

approach. Experiments will be used to validate the proposed measures. The experiments will be done on four sets of features:

- **Basic Features (B)**: Basic + Expression Features
- **Complete Features (C)**: B + Control Context Features + Dependency Features + Dependency Features in Control Context
- **Test Case Features (T)**: The percentage of test-cases passed.
- **All Features (CT)**: C + T

These feature sets make it possible to reason about the main research question as well as **S1**. The two different feature sets needed to answer **S2** are established for five different programs. One feature set follows our feature definitions whereas the other feature set is generated by Srikant and Aggarwal [45]. The differences between these feature sets are highlighted in chapter four.

Comparison of the performance of different learning algorithms is necessary to find one that suits this task best. Performance comparison of learning algorithms is done based on grading accuracy relative to manual grades. Following the GQM template for goal definition by Basili [27] we can describe the research goal as:

Analyze test-case based and semantic feature based metrics
for the purpose of evaluation
with respect to grade prediction accuracy
from the point of view of the human grader
in the context of professional and student programmers performing an online programming task.

The independent variables are the counts of feature-occurrences and the dependent variable is the grade. To answer the research question we contrast the performance of two different treatments: test case features (T) and all features including test case features (CT). The performance measure is an experimental estimation of prediction accuracy.

It is important to identify the sources of variation that must be controlled by each experiment. We can distinguish four important sources of variation:

- Random variation in the **selection of the test data** that is used to evaluate learning algorithms. On any particular randomly drawn test data set one classifier may outperform another even though on the whole population the two classifiers would perform identically. This is a particularly pressing problem for small test data sets.
- The second source of random variation results from the **selection of the training data**. On any particular randomly drawn training set one algorithm may outperform another even though, on the average, the two algorithms have the same accuracy. Even small changes to the training set (such as adding or deleting a few data points) may cause large changes in the classifier produced by a learning algorithm.

- A third source of variance can be **internal randomness** in the learning algorithm. When an algorithm is initialized with a set of random weights on which it then improves, the resulting learned weights depend critically on the random starting state. In this case, even if the training data are not changed, the algorithm is likely to produce a different hypothesis if it is executed again from a different random starting state.
- The last source of random variation we address is **random classification error**. If a fixed fraction of the test data points is randomly mislabeled, the learning algorithm can achieve an error rate of less than this fraction.

To account for test data variation and the possibility of random classification error, the statistical procedure must consider the size of the test set and the consequences of changes in it. To account for training data variation and internal randomness the learning algorithm will be trained multiple times on different training sets. The variation in accuracy of the resulting classifiers is an indicator of the influence of training data variation. Whether we have to take into account internal randomness will depend on the chosen learning algorithm.

The main hypothesis can be defined as:

- **H0**: For a randomly drawn training set R of fixed size, learning algorithms based on test-case features or semantic- and test-case features will have *the same error rate* on a randomly drawn unseen test example, where all random draws are made according to the dataset distribution.
- **HA**: For a randomly drawn training set R of fixed size, learning algorithms based on test-case features or semantic- and test-case features will *not* have the same error rate on a randomly drawn unseen test example, where all random draws are made according to the dataset distribution.

The two obvious goals are:

- **Model selection:** estimating the performance of different models in order to choose the best one.
- **Model assessment:** having chosen a final model, estimate its prediction error (generalization error) on new data. Unfortunately I do not posses enough labeled data to estimate prediction error on a separate validation set. To obtain an indication of the generalization error I will apply cross-validation.

A third goal, introduced in S3 is:

- **Learning method selection:** determine whether the estimated performance of regression models differs from the estimated performance of classification models. Comparing classification estimates (discrete) vs. regression estimates (continuous) is not completely straightforward. One approach would be to discretize the continuous regression outcomes to classification categories.

Chapter 7

Results

This chapter provides an overview and discussion of the experiment results. The first section discusses the feature granularity experiment results. The second part of this chapter is focused on the results of the different modeling approaches for the Fish task. Insights in the most important features are discussed for every model.

7.1 Impact of Feature Granularity

To investigate the influence of different feature generation rules I analyzed the five problems in the dataset from Aspiring Minds using Ridge and Lasso models. In the work of Aggarwal and Srikant the predictive power of ridge models is assessed with the correlation coefficient of the predicted regression grades and the original ordinal label[45]. To clarify what this entails, I follow the same approach when assessing models based on the Aspiring Mind feature set and our own feature set generated by the JFEX program.

	All features without test case scores, Ridge results							
Task	#Feat. AM	Train *r* AM	Test *r* AM	Test r SD AM	#Feat. JFEX	Train *r* JFEX	Test *r* JFEX	Test *r* SD JFEX
EV	2243	0.92	**0.67**	0.00	1582	0.79	**0.61**	0.02
IT	1867	0.95	**0.74**	0.00	1525	0.96	**0.81**	0.00
GC	1773	0.96	**0.88**	0.00	1415	0.96	**0.88**	0.00
PP	2772	0.95	**0.64**	0.00	2883	0.96	**0.62**	0.00
TM	2097	0.95	**0.76**	0.00	1203	0.92	**0.70**	0.00

	All features including test case scores, Ridge results							
Task	#Feat. AM	Train *r* AM	Test *r* AM	Test r SD AM	#Feat. JFEX	Train *r* JFEX	Test *r* JFEX	Test *r* SD JFEX
EV	2243	0.91	**0.68**	0.00	1582	0.80	**0.63**	0.02
IT	1867	0.95	**0.74**	0.00	1525	0.96	**0.81**	0.00
GC	1773	0.96	**0.89**	0.00	1415	0.96	**0.88**	0.00
PP	2772	0.93	**0.66**	0.00	2883	0.96	**0.64**	0.00
TM	2097	0.95	**0.77**	0.00	1203	0.92	**0.72**	0.00

FIGURE 7.1: Averaged results based on one hundred 10-fold cross validation runs.

Figure 7.1 reports the correlation values for the train and test sets as well as the standard deviations for all five problems. The left-hand side of the

tables show the results of the Aspiring Minds (AM) features, the right-hand side reports the results of the JFEX featuers. The cross validation approach used on the training set uses random folds which may introduce some variance in the resulting λ parameter estimate. To reduce this effect the reported results are the average of 100 models for which the lambda parameter was determined through cross validation on the training set. The *r* values seem to indicate that the different feature sets have roughly the same predictive power.

To ensure that the quality of the predictions reflects the influence of the feature set there are no test-case results included in the models in the upper table. The bottom results were generated by models that used all features and the test cases cores. There is no substantial increase in performance gained from the test case scores. The test *r* reported by Aggarwal and Srikant for Ridge regression models using all features but no test case scores ranges from 0.56 to 0.90, which is comparable to the 0.61-0.88 range reported in Fig. 7.1. The question is what information we gain from these results. A linear correlation between the predicted and the true grade is certainly expected in good predictions. But different datasets can have the same *r*, of which the most straightforward example is when two datasets differ by a constant value for each observation.

It does not seem natural to assess the correlation between quantitative predictions and ordinal labels. What is the correct interpretation of a predicted score of 2.734? Is it grade category 2 or 3? There is no correct answer to this question as ordinal levels contain no distance metrics. One could pick a threshold at any arbitrary point between two and three. A more guided approach used by Aggarwal and Srikant is to determine the thresholds by approximating the grade distribution in the training set as close as possible. However, as the test set is selected taking care that the grade distribution remains equal this approach might contribute to overfitting.

Overfitting is very well disguised by the reported *r* values. There are over a factor hundred more variables than observations in the datasets. Many of these variables uniquely belong to one or two single solutions. Fitting a model with so many variables to a small training set is not very hard: it can easily account for all variance.

A commonly used measure to gain some more insight in the differences between the predicted and observed values is the root-mean-square error (RMSE). The RMSD (Equation 7.1) represents the sample standard deviation of the prediction errors.

$$RMSE = \sqrt{\frac{1}{n}\left[\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2\right]} \tag{7.1}$$

Fig. 7.2 on the next page reports the RMSE for the models whose correlations are depicted in Fig. 7.1. The RMSE values of the two different feature sets do not display severe diversions. There is no indication that the two feature sets differ substantially in their potential predictive power. In fact, both feature sets are not able to produce very accurate results: the RMSE's on the test sets range between 0.95 and 1.35 (on a five point grading scale).

	All features without test case scores, Ridge results							
Task	Train rmse AM	CV rmse AM	Test rmse AM	Test rmse SD AM	Train rmse JFEX	CV rmse JFEX	Test rmse JFEX	Test rmse SD JFEX
EV	0.77	1.15	**1.12**	0.04	1.29	1.40	**1.35**	0.07
IT	0.73	1.15	**1.12**	0.02	0.75	1.23	**1.07**	0.03
GC	0.70	1.00	**0.93**	0.01	0.73	1.07	**0.95**	0.01
PP	0.89	1.32	**1.35**	0.02	0.68	1.26	**1.32**	0.02
TM	0.59	0.91	**0.93**	0.01	0.73	0.99	**1.00**	0.01

	All features including test case scores, Ridge results							
Task	Train rmse AM	CV rmse AM	Test rmse AM	Test rmse SD AM	Train rmse JFEX	CV rmse JFEX	Test rmse JFEX	Test rmse SD JFEX
EV	0.83	1.17	**1.13**	0.04	1.28	1.40	**1.34**	0.05
IT	0.72	**1.14**	1.11	0.02	0.76	**1.22**	1.07	0.03
GC	0.70	0.99	**0.91**	0.01	0.77	1.07	**0.96**	0.01
PP	0.86	1.29	**1.33**	0.01	0.72	**1.25**	**1.31**	0.02
TM	0.57	0.90	**0.92**	0.01	0.71	0.97	**0.98**	0.01

FIGURE 7.2: Averaged results based on one hundred 10-fold cross validation runs.

7.1.1 Most Important Predictors

We have discussed the results of the predictions on the AM dataset produced by Ridge regression. Modeling the grades with LASSO did not yield any improvements. However, LASSO has a big advantage over Ridge regression in terms of model interpretability. We will now inspect the predictors with non-zero coefficient values in the LASSO models as this may provide more insight into the consequences of the feature differences.

For the TM problem, the AM set has a lower train RMSE but as Fig 7.3 on the next page indicates, it uses more predictors to obtain this result. This seems to be a case of overfitting as the test RMSE is not as good. For the PP problem we see the same pattern, but now for the JFEX features. For the IT problem JFEX uses 31 features versus only only an intercept on the AM dataset. Many of the selected features in the JFEX model have a context that refers to a specific for-condition expression, something that is not captured by the AM dataset.

For the EV problem we see that some code properties are captured by AM features but not by JFEX features. This results in a higher accuracy for the AM features. For both problems sets the most contributing feature is a Basic feature for the string *"bcdfgh"*. This constant is used in hard coded solutions that return a string with the vowels removed by hand. This is not scalable to other inputs and explains the big negative coefficient value. The selected AM coefficients also contain an expression dependency feature that states that a variable and one function call are dependent on an input parameter. Most of the assignments with this particular feature declared a variable to refer to *input.size()* or *input.toCharArrary()*. These code properties are captured by more specific features in JFEX, which are not selected in the model. Instead, a number of basic features that describe constants or simple expressions are selected.

Problem	Without test cases cores				With test case scores			
	AM		JFEX		AM		JFEX	
	intercept	#pred	#pred	intercept	intercept	#pred	#pred	intercept
EliminateVowel	3.93	5	10	4.01	3.44	5	3	3.38
GrayCheck	1.44	16	12	1.58	1.16	17	25	0.70
IsTree	2.48	0	31	1.40	1.61	11	1	2.17
PatternPrint	2.94	2	115	1.87	1.96	30	20	2.02
TransposeMultiMatrix	1.32	27	8	2.27	1.69	18	10	1.99

FIGURE 7.3: Number of non-zero coefficients in the LASSO models for the Aspiring Minds problems.

7.2 Supervised Modeling

This section discusses the results of predicting the manually assigned grades of the Codility Fish task. First the division of the data into a training and test set is motivated. Next, the performance of multiple modeling approaches is discussed.

7.2.1 Train-Test Split

The overall grade distribution is as follows:

- Grade 1: 6%
- Grade 2: 35%
- Grade 3: 20%
- Grade 4: 16%
- Grade 5: 23%

The data is divided into two random subsets, ensuring the grade distributions in both sets to remain equal. The distribution of the grades has two outlier classes: grade category 2 dominates the dataset whereas grade category 1 is significantly smaller than the other grade categories. The training set holds 154 (67%) of the observations and the test set contains 76 observations (33%).

7.2.2 Regression Models

The results for two penalized regression approaches are shown in Fig. 7.4. Top-down the results are displayed for the four different feature sets: all features without test cases, all features including test cases, only the basic features without test cases and only test case results. The standard deviations are omitted because they amounted to 0.00 rounded at two digits. Such small deviances are not of interest for grade prediction on a scale of 1 to 5. The LASSO and Ridge model performance is comparable. However, for the complete feature set including test cases LASSO clearly outperforms Ridge regression on the test set. In addition, the LASSO models provide a better interpretation of the predictors. Therefore we will focus on the LASSO model properties.

Variable selection: Feature Ratio					
Dataset	Train *r*	Test *r*	Train rmse	Test rmse	CV rmse
Fish, all features no TC, Ridge results	0.83	0.24	1.01	1.23	1.20
Fish, all features no TC, Lasso results	0.58	0.18	1.17	1.25	1.23
Fish, all features + TC , Ridge results	0.92	0.43	0.71	1.15	1.07
Fish, all features + TC, Lasso results	0.92	0.82	0.54	0.73	0.74
Fish, basic features + TC, Ridge results	0.72	0.20	1.15	1.24	1.24
Fish, basic features + TC, Lasso results	0.49	0.07	1.21	1.27	1.25
Fish, only TC, Ridge results	0.85	0.80	0.70	0.77	0.76
Fish, only TC, Lasso results	0.85	0.81	0.70	0.75	0.76

FIGURE 7.4: Averaged results based on one hundred 10-fold cross validation runs.

All features without test cases achieve an average test RMSE of 1.25. This can be regarded as a high error, especially when compared to the error of 0.75 from test case predictions only. Combined, the features together with the test case results produce an average test error of 0.73. Is there information captured by the additional features that could no be caught by test cases alone? Fig. 7.6 shows prediction plots that may help answer this question.

The first plot for all features without test cases is clearly less precise than the two plots that include test case results. The distinction between test case only and all features combined with test cases is less obvious. When features are added to the test case results we see that the prediction range widens from 1.35-4.22 to a more accurate 1.12-4.90. The predicted values for each grade categories also become more clustered towards their true values. For grade category 5 , the prediction range is now 3.73-4.90 instead of 4.22 for every prediction based on test cases only. Even though the range has become wider for category 5, the average prediction error has gone down from 0.79 to 0.75.

The included predictors from the features without test cases are:

```
1  "(Intercept)"                                                                  " 3.1678728433"
2  "X$B$NE"                                                                       "-0.1676407344"
3  "X$B$vt.Stack"                                                                 " 0.0091073608"
4  "X$BC$op.NE_con.ForIf_c$"                                                      "-0.0213749944"
5  "X$BC$op.while_con.ForIf_e$"                                                   " 0.3500822883"
6  "X$D$E$lv.2_op.PLUSmc.size$_@_$E$lv.1_op.ASS$"                                 " 0.3537394021"
7  "X$D$E$lv.2pv.1_op.LT$_@_$E$lv.1_op.POSTINC$"                                  "-0.3610886580"
8  "X$D$E$lv.2pv.1_op.LT$_@_$E$lv.2pv.1_op.ASS$"                                  "-0.0022237625"
9  "X$D$EC$lv.2pv.1_op.LT_con.ForIf_bIf_c$_@_$EC$lv.1_op.POSTINC_con.For_upd$"    "-0.1462411177"
10 "X$E$lv.1_op.mc.push$"                                                         " 0.0409701618"
11 "X$E$lv.2_op.PLUSmc.size$"                                                     " 0.0001574543"
12 "X$EC$lv.2pv.1_op.LT_con.ForIf_bIf_c$"                                         "-0.0007513164"
```

FIGURE 7.5: LASSO non-zero coefficient values for all features without test case scores.

The most influential predictor is a Dependency feature for 2 local variables and a parameter variable with a less than operator, that is dependent on a local variable post increment. This feature occurred in four solutions in the dataset, and all have been assigned grade 1. The feature can be interpreted as follows. A for loop traverses all fishes in the river. The if statement

handles a fish going up or down, and the nest if statement says that if the size of some local variable indicates that a fish is smaller than the current fish, a fight is triggered. This only handles the first fight, and is not accustomed to the fact that either of the fishes may win and has to fight again. Combined with other errors, the solutions that contained this approach did not produce an algorithm able to handle all situations, and hence they were assigned grade 1.

The same feature is also selected including context. Positive predictors are related to data structures: Stack and push procedures. A while loop is also recognized to be a positive contributor, which makes sense because while a fish is alive it might encounter different enemies. The other positive coefficient describes a dependency of the addition of a method call to size() on a local variable assignment. It occurs 17 times in the dataset. Almost all occurrences have grade 4 or 5, one solution has grade 2 and one has grade 3. It turns out that all of the solutions with these feature use a stack based approach, adding the number of alive fishes on stack (.size() call) to a counter of alive fishes. However, the two solutions that scored below 4 have some very specific problems with the usage of the stack. The other predictors have a very small influence on the predicted score.

The model based on the basic features has an intercept of 3.20 and one non-zero coefficient of -0.08 for the basic feature that describes a "not equal to" operator. This basic features occurs in 69 solutions of which the majority has a grade below 4. However, plenty of the solutions with this feature have received grade 4 or 5. Which is not surprising as this basic feature can be applied in many contexts, and on its own is not expressive enough to capture the context properties.

The test case only model has an intercept of 2.41 and 10 non-zero coefficients. Two of these are very close to zero. Out of the other eight, the test case features "Medium Random Wrong Answer" with -0.43 and "Large Random Wrong Answer" with -0.53 are the most influential predictors. Wrong answers and timeout errors for five other test cases are also negative contributors. "Small Random Wrong Answer" is a positive contributor with coefficient value 0.19 which is odd. This may have been the best fit to the training set to balance the other negative predictors.

When all features are added to the test cases this results in 61 predictors and an intercept of 2.31. Only four test cases predictors are left, which are all "Wrong Answer" results for test cases with a negative coefficient value. The other non-zero coefficients belong to features are from all categories. While and stack related features are amongst the most positive predictors and the feature of line 7 from Fig. 7.5 is once again amongst the more influential negative contributors.

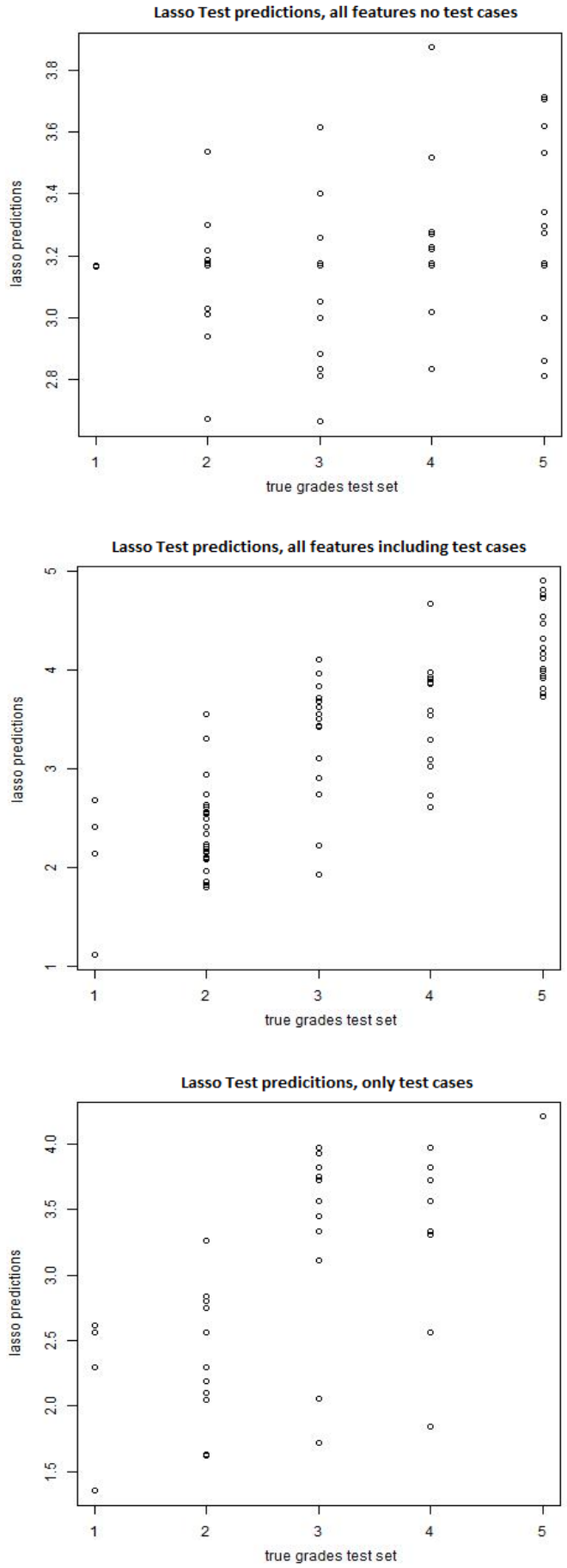

FIGURE 7.6: Predicted vs. True grades.

7.2.3 Multinomial Classification

Fig. 7.7 reports the multinomial classification results. Errors of type 2 should be avoided at all costs, as they indicate a misclassification from grade 4 or 5 to 1,2, or 3 (and vice versa). The highest accuracy and lowest type 2 error is achieved by test cases only. The various levels of feature selection based on between-predictor correlation and occurrences in the dataset did not yield any improvements over the feature-ratio filtered feature set. The bottom table shows the results of the feature sets that are filtered by removing all solutions that occurred in less than 76 solutions and removing all predictors that have a correlation above 0.90 with other predictors.

It seems as if the feature selection has a positive impact on the prediction accuracy of all features combined with the test cases. However, the features are reduced from 2.493 to 53 predictors. So the increase in performance is more likely to be attributed to the accuracy of the test cases only, which is now no longer interfered by the features. Inspection of the model predictors supports this assumption as more test cases were included and they have bigger absolute coefficient values.

Variable selection: Feature Ratio	Correct	Error 1	Error 2
Fish, all features no TC	41%	25%	34%
Fish, all features + TC	55%	17%	28%
Fish, basic features no TC	37%	25%	38%
Fish, only TC	67%	15%	18%
Fish, advanced features + TC	59%	16%	25%

Variable selection: n > 76, c <90	Correct	Error 1	Error 2
Fish, all features no TC	27%	24%	49%
Fish, all features + TC	63%	18,5%	18,5%
Fish, basic features no TC	37%	25%	38%
Fish, only TC	67%	15%	18%

FIGURE 7.7: Prediction accuracy for multinomial models. Error type 1 indicates a misclassification between 1,2 and 3 or between 4 and 5. Error type two indicates a misclassification between these two grade sets.

An interesting observation for the multinomial model predictors is that for each grade category, the features only set provides less non-zero predictors than the feature set that includes all features as well as test cases. When test cases are removed the basic features become the most dominant features. This can be explained by basic features being far less sparse than advanced features. However, basic features on their own are unreliable quality predictors. All basic features seem to have occurrences both in high and low graded solutions. I tried removing all basic features to leave only the advanced features, as the basic features might be preventing the sparser advanced features from being selected. After all, every advanced features is "shielded" by one or more basic features that refer to the same code expression. Removing the basic features yielded a slight performance increase, but is still outperformed by predictions based on test-cases only. Advanced features on their own did not provide enough information for a multinomial model to converge.

7.2.4 Binomial Classification

Since type 2 errors outweigh type 1 error in our application, we consider using a binomial model. Important information about specific grade levels gets lost, but when this yields an increase in accuracy and a significant drop in type 2 errors it may be worthwhile. All grades 4 and 5 are transformed to have value 1, and grades 1,2, and 3 to value 0. As a consequence, a prediction is either correct or a type 2 error. Fig. 7.8 reports the differences in type 2 errors obtained by binomial and multinomial models.

Variable selection: Feature Ratio	Binomial Error 2	Multinomial Error 2
Fish, all features no TC	36%	34%
Fish, all features + TC	16%	28%
Fish, TC only	16%	18%
Fish, advanced features + TC	17%	25%

FIGURE 7.8: Type 2 errors obtained by binomial and multinomial models.

The binomial model for all features including test case results contains 55 non-zero predictors: 5 test case results and 50 features. The features are from all categories but the majority belongs to the basic features. Unfortunately, removing the basic features does not yield an increase in performance. The same test case predictors are still the most important non-zero coefficients. At the moment, models based solely on test case features achieve the highest accuracy. Personally, I feel that the small drop of 2

7.2.5 Ordinal Classification

Until now the reviewed modeling approaches did not show any increase in accuracy over test case predictions by adding the semantic features. In a final attempt to discover their predictive power we resort to ordinal models. The ordinal modeling approach achieved the following accuracy on the test set:

- features only: correct: 78% , error 1: 12%, error 2: 10%.
- all features + test case: correct: 79%, error 1: 14%, error 2: 7%.

Accuracy on the training set was 100% for all features including test case results, and there was 1 misclassification for the features without test case results. The solution was predicted to have grade 5 while it should have been grade 4. It was a correct solution but it used lists to function as stacks, removing and adding from the front of the list. This is not computationally efficient and causes a time-out error in the larger test-cases. However, the model based on features only was not able to learn the difference between two correct algorithms where one uses a stack and the other a list, as there were very few occurrences of this event in our dataset.

There was no convergence for test cases only and advanced or basic features only. The accuracy of 100% on the training set is comparable to the reports resulted by the authors of the algorithm[6].The ordinal model tries

to exploit features that are monotonically associated with the ordinal response. Fig. 7.9 depicts the grade correlations for all features in the feature-ratio filtered set. We see that the correlations are centered around 0, and have a small range from -0.27 to 0.30. As we may expect from these low correlation values, inspection of the individual features shows that there is no feature that has a noteworthy monotone relationship to the grade all by itself. If there were, this would have been very remarkable due to the nature of the features.

It is very unlikely that here is a code construct that doesn't decrease code quality at some point when the code construct occurrence starts increasing. I suspect the features to have a monotone relationship with the grades in the following way: for a certain group of features, the lower the value of the combined features, the lower the grade.

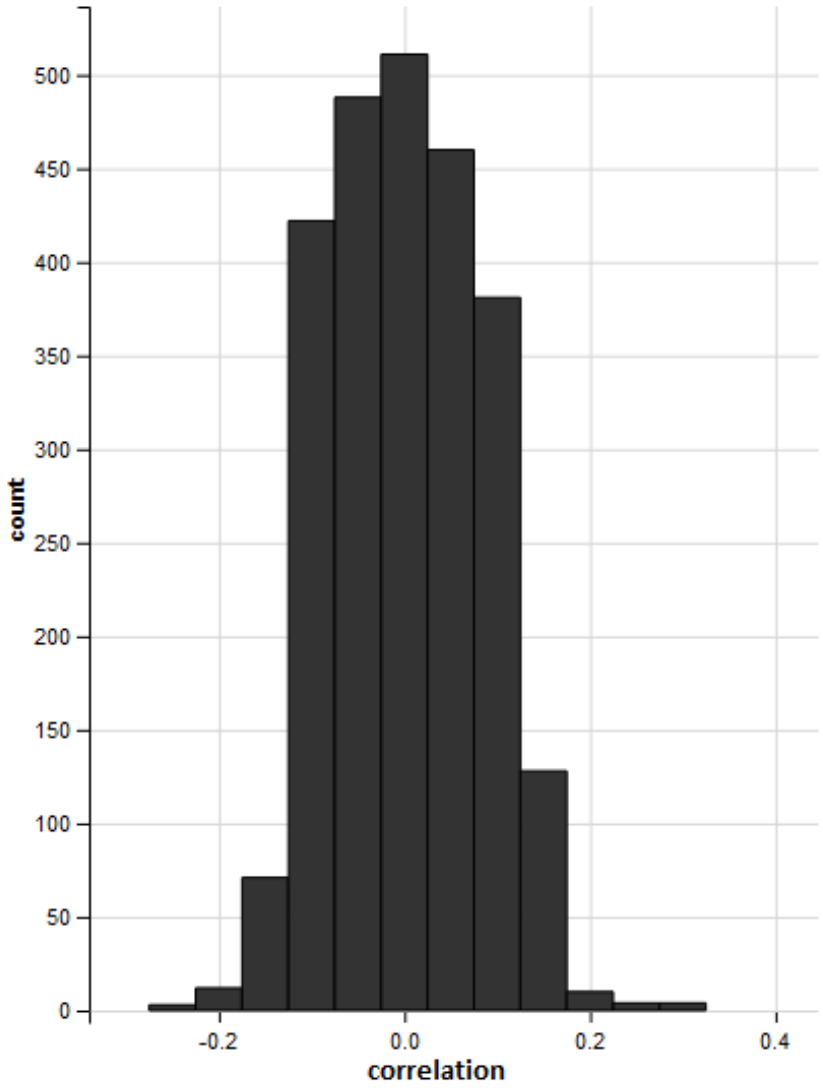

FIGURE 7.9: Grade correlation for all features in the Feature-Ratio filtered set.

Chapter 8

Conclusion

In this study I have explored the influence of Java code features on the prediction accuracy of manual grades. A feature derivation program has been developed which has been used to tag 230 solutions to a Java programming task. These solutions have been manually graded following a tailored grading approach. The grading rubric has been designed to mimic real world applications and does not take the feature properties into account. I was interested to see if the features would be able to capture specific grading information that seems to be impossible to capture with test cases alone.

8.1 Answering the Research Questions

The key aspects of this research have been the definition of the features, the exploration of applicability to a more solution oriented grading style and finally the analysis of a classification approach versus regression analysis. The main research question I aspired to answer is: can we improve automated test case based grading of programming assignments with semantic source code features? To provide a sensible answer to this question several subquestions have been defined.

The first question asked if dependency and control flow features have more predictive power than basic keyword counts. The results of section 7.1 indicate that the performance of basic features is definitely improved by the more advanced features The advanced features have shown promising results in the multinomial model. However, the other modeling approaches required the addition of the basic features in order to reach convergence. My hypothesis is that this has to do with the sparsity of the advanced features.

The second question that was answered is if the granularity of the feature definitions affects the performance with respect to human grading. Section 4.2.6 of this thesis discussed the main differences of our features compared to the proposed features by Aggarwal and Srikant[45]. Noteworthy to mention is that JFEX defines more detailed expression features. This causes our advanced features to become even sparser, and may have a negative influence on feature based predictions for our small dataset. I do not believe that the solution to this issue is to dumb down the features as this would decrease their ability to capture minor deviations from a correct solution.

The last question considers if the problem benefits from classification modeling instead of a regression approach. Binomial, multinomial and ordinal models for high-dimensional data have been considered and compared with Ridge and LASSO regression models. Figure 4.3 (page 30) indicated that simple test case predictions have a grade accuracy of 49%. Multinomial modeling increased the test case accuracy to 67%, this couldn't be improved with the addition of feature predictors. But for an ordinal model setting the features outperformed this score and achieved a prediction accuracy of 78%. This was further improved by adding test case information, which dropped the type 2 error from 10% to 7%.

To conclude, this work has not provided significant proof that automated test case based grading can be improved with semantic source code features. However, it did demonstrate some encouraging evidence that shows the features have the *capacity* to improve test case accuracy. A subset of the selected features seemed highly relevant to the problem at hand. Classification modeling with attention for the ordinal ordering between the grade levels presented itself as the best candidate to realize the potential of the features.

8.2 Future Research

The definition of the features is one of the aspects that could benefit from further research. Performing several steps of code normalization before generating the features may increase the expressiveness of the features. Clustering the features also seems a fertile future direction. Especially as the number of features grows with the number of analyzed solutions. When 3800 Fish tasks were given as input to JFEX, it found over 22.000 unique code features. This is hardly scalable.

Aggarwal and Srikant have chosen to tackle this problem with feature transformation in their newest work[44]. A distance function is used to calculate the distance in the feature space from a solution to a set of good solutions. The downside of this approach is that the expressiveness of the features gets completely lost in at most one number per feature category. A promising direction for new research is to research a scalable distance metric between feature representations of solutions, that is still able to capture very small deviations. Perhaps some work in the area of feature clustering can provide a good start.

Another option would be to redefine the features in such a way that less features are needed to capture the same amount of information. If a non-linear modeling approach is used features can be made more interactive. So instead of "2 variables and an assignment operator" and "3 variables and an assignment operator" being two different features, they could both be instances of the same feature but with different values plugged in. Obviously this is not as straight forward when multiple constructs start to play a role in the feature. Defining the interaction between the feature components would have to be one of the first things to be considered.

Building upon this research it may also be exciting to utilize the program transformation functionality that is offered by Spoon. I expect a program with a minor deviation from a good solution to perform well on the

test cases if this minor deviation is transformed to correct constructs. Applying code transformation based on the features can also provide a way to distinguish less important features from the relevant ones, by measuring the effect of the transformation on the test case performance.

Appendix A

Appendix: Grade Indicators

TABLE A.1: Fish Grade Indicators

Grade	Code	Description
1	CWL	Completely Wrong Logic: A combination of many CAT2 mistakes, that are combined in completely wrong logic. As a consequence, these mistakes cannot be categorized in isolation as a CAT2 and the code results in CAT 1.
1	ICL	Incomplete logic: the right set-up is there but it is not complete enough to compete with CAT2 efforts.
1	MAS	Missing Algorithm Structure: even though the right datastructures might be present there is not a decent structure in the algorithm to solve the problem at least partially or with some good first steps.
1		Too many CAT2 or other violations of which the combination cannot compete with the regular type2 errors.
2	CD	Casual death: a fish following a fish in the same direction is not registerd on the live stack.
2	DW	Dead Winner.
2	FO	Fish Order, for example when an upstream fish is compared against the highest downstream fish instead of the lowest (the one right above), or when a fish fights enemies even when there are predecessors that should fight first.
2	FU	Fishes Upstream: example: for a downstream fish it considers upstream swimming fishes both below and above the fish. While those above are never encountered.

TABLE A.2: Fish Grade Indicators

Grade	Code	Description
2	LE	Logical Error.
2	OFE	Only First Enemy: checks to see the first encountered fish but when we eat it we forget to check for the next encountered fish.
2	OFF	Only First Fish: checks for a contrarian fish when found it stops never checking for others.
2	SDE	Same Direction Enemy: treat fishes going in the same direction as you as enemies .
2	WD	Walking Dead: fishes stay alive that should be killed (for example when only checken on stack enemies if they are smaller and not if they are bigger and kill us).
2	WP	Wrong Push: Pushes the wrong fishes on stack based on invalid criteria.
3	1WC	One wrong condidition wrecking the algorithm, if it would have been the correct condition the algorithm was good.
3	BSF	Big sized fish (size above 9) cause problems.
3	CT	Cubic time.
3	DC, WLTW	Dead code and Wrong logic that works: when entire logic algorithm steps are wrong, but the algortihm executes correctly because the wrong algorithmic parts are dead code this is a severe mistake resulting in level 2. It shows a misunderstanding of the assignment as well as little insight in ones own code.
3	ER	Early Return: correct algorithm but it doesn't completely execute because of an early return that's not supposed to be there.
3	ESERRORCM	Empty stack error may occur Covering Mistake: there is no case for when a fish encounters an empty stack. If there was the algoritm would be perfect.
3	MB	Missing Break: a certain clause misses a break statement which would have solved the issue.
3	ORP	Object Reference Problem: forgetting to return an object from a helper method.
3	PP	Pop - Peek problem (treating pop action as if it was a Peek).
3	PPE	Pop - Peek problem (treating pop action as if it was a Peek): Edge case. Manually fixes the PP by pushing back fishes if necessary but doesn't handle the edge cases properly (the last fish, that leaves behind an empty stack when popped).
3	QT	Quadratic Time: If algorithm correct no higher than CAT 3 can be achieved (also goes for complexities between O(n) and quadratic.
3	ZSF	Zero sized fishes cause problems.
3	1SF	One sized fishes cause problems (uses size 1 to indicate dead fishes).

TABLE A.3: Fish Grade Indicators

Grade	Code	Description
4	BR	Break Reach: assuming a break will break the outer loop without a label , otherwise algorithm would be correct.
4	DSTO	Data structure time out.
4	DC	Dead code: silly mistake resulting in a level 4. But when hiding bad logic is may result in CAT 2 (see dc - wltw).
4	ESE	Empty stack error may occur because a defined nullcheck doesn't cover every stackcall.
4	II	Index Issue: usually when people simulate a stack with lists, and update the pointer by hand there might occur an index out of bounds issue. Which if it would be caught leaves a perfect algorithm. Also defined an index issue for an algorithm that timed out on the front of list and when updated to use back of list as stack top person forgot to change one index definition from 0 to list.size-1.
4	L1	Lists of size one are caught and returned with 0 survivors. Actual algorithm that is not reached without this catch would have treated it right.
4	LI1	loop instantiation 1: starts from second fish without possibly pushing first fish.
4	OM	Operator Mistake: for example greater than instead of greater or equal than.
4	OR	Operator Reach: for example a NOT operator that reaches the whole expression while it was supposed to enclose the first argument.
4	SOE	Stack overflow error for special cases due to bad recursion-call.
4	UC	Unnecessary Complicated: i do not think it's right to award full credits for these solutions as they are just bad solutions that, although they deliver the right behavior, need to be distinguished from truely good ones.
4	UPDOWN	Up and down are switched, otherwise algorithm would be ok.
4	WA	Wrong array, mistakenly switch an A-B array read or index-value read.
4	WLO	Wrong List Operation: push /add mix-up.
4	WSO	Stack.firstElement retrieves the bottom of the stack should be stack.pop() then algorithm would be perfect.
5	Sysout	Perfect algorithm but Sysout causes time-outs: it is still worth a 5.

References

[1] Beth Adelson and Elliot Soloway. "The role of domain expenence in software design". In: *Software Engineering, IEEE Transactions on* 11 (1985), pp. 1351–1360.

[2] Kirsti M. Ala-Mutka. "A survey of automated assessment approaches for programming assignments". In: *Computer science education* 15.2 (2005), pp. 83–102.

[3] Arif Altun and Sacide Guzin Mazman. "Identifying latent patterns in undergraduate Students' programming profiles". In: *Smart Learning Environments* 2.1 (2015), pp. 1–16.

[4] John R. Anderson. "Acquisition of cognitive skill." In: *Psychological review* 89.4 (1982), p. 369.

[5] John R. Anderson, Frederick G. Conrad, and Albert T. Corbett. "Skill acquisition and the LISP tutor". In: *Cognitive Science* 13.4 (1989), pp. 467–505.

[6] Kellie J. Archer et al. "ordinalgmifs: An R package for ordinal regression in high-dimensional data settings". In: *Cancer informatics* 13 (2014), p. 187.

[7] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. "Testful: an evolutionary test approach for Java". In: *Software testing, verification and validation (ICST), 2010 third international conference on*. IEEE, 2010, pp. 185–194.

[8] Karl Bell. *Automated Student Code Assessment with Symbolic Execution and Java PathFinder*. 2012.

[9] Gunnar R. Bergersen, Dag Sjoberg, and Tore Dyba. "Construction and validation of an instrument for measuring programming skill". In: *Software Engineering, IEEE Transactions on* 40.12 (2014), pp. 1163–1184.

[10] Richard Bornat and Saeed Dehnadi. "Mental models, consistency and programming aptitude". In: *Proceedings of the tenth conference on Australasian computing education-Volume 78*. Australian Computer Society, Inc., 2008, pp. 53–61.

[11] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. "Korat: Automated testing based on Java predicates". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 27. ACM, 2002. Chap. 4, pp. 123–133.

[12] John P. Campbell et al. "A theory of performance". In: *Personnel selection in organizations* 3570 (1993), pp. 35–70.

[13] Yoonsik Cheon and Carmen Avila. "Automating Java program testing using OCL and AspectJ". In: *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*. IEEE, 2010, pp. 1020–1025.

[14] Yoonsik Cheon and Carmen Avila. "Automating Java program testing using OCL and AspectJ". In: *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*. IEEE, 2010, pp. 1020–1025.

[15] D. Crookes. *Educators call for reform in how programming is taught in schools*. 2013.

[16] Lauren Csorny. "Careers in the growing field of information technology services". In: *Beyond the numbers* 2.9 (2013).

[17] Saeed Dehnadi and Richard Bornat. "The camel has two humps (working title)". In: (2006).

[18] Christopher Douce, David Livingstone, and James Orwell. "Automatic test-based assessment of programming: A review". In: *Journal on Educational Resources in Computing (JERIC)* 5.3 (2005), p. 4.

[19] Thomas Dvornik et al. "Supporting introductory test-driven labs with WebIDE". In: *Software Engineering Education and Training (CSEEandT), 2011 24th IEEE-CS Conference on*. IEEE, 2011, pp. 51–60.

[20] Stephen H. Edwards and Manuel A. Perez-Quinones. "Web-CAT: automatically grading programming assignments". In: *ACM SIGCSE Bulletin*. Vol. 40. ACM, 2008. Chap. 3, pp. 328–328.

[21] Michal Forišek. "On the suitability of programming tasks for automated evaluation". In: *Informatics in Education-An International Journal* Vol 5_1 (2006), pp. 63–76.

[22] Logo Foundation. 2015.

[23] Gordon Fraser and Andrea Arcuri. "Evosuite: automatic test suite generation for object-oriented software". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[24] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. "glmnet: Lasso and elastic-net regularized generalized linear models". In: *R package version* 1 (2009).

[25] Pedro Antonio Gutiérrez et al. "Ordinal regression methods: survey and experimental study". In: *IEEE Transactions on Knowledge and Data Engineering* 28.1 (2016), pp. 127–146.

[26] Arthur E. Hoerl and Robert W. Kennard. "Ridge regression: Biased estimation for nonorthogonal problems". In: *Technometrics* 12.1 (1970), pp. 55–67.

[27] Mahfuzul Huda, Yagya Dutt Sharma Arya, and Mahmoodul Hasan Khan. "Metric Based Testability Estimation Model for Object Oriented Design: Quality Perspective". In: *Journal of Software Engineering and Applications* 8.04 (2015), p. 234.

[28] Jens C. Huhn and Eyke Hullermeier. "Is an ordinal class structure useful in classifier learning?" In: *International Journal of Data Mining, Modelling and Management* 1.1 (2008), pp. 45–67.

[29] Petri Ihantola et al. "Review of recent systems for automatic assessment of programming assignments". In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 2010, pp. 86–93.

[30] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. "Towards a Systematic Review of Automated Feedback Generation for Programming Exercises–Extended Version". In: (2016).

[31] Max Kuhn and Kjell Johnson. *Applied predictive modeling*. Springer, 2013.

[32] Andrew S. Lan et al. "Mathematical Language Processing: Automatic Grading and Feedback for Open Response Mathematical Questions". In: *arXiv preprint arXiv:1501.04346* (2015).

[33] Jinrong Li et al. "Design and implementation of semantic matching based automatic scoring system for C programming language". In: Entertainment for Education. Digital Techniques and Systems. Springer, 2010, pp. 247–257.

[34] Hsuan-Tien Lin. In: *From ordinal ranking to binary classification* (2008).

[35] Thomas J. McCabe. "A complexity measure". In: *Software Engineering, IEEE Transactions on* 4 (1976), pp. 308–320.

[36] Kevin Alexander Naudé. In: *Assessing program code through static structural similarity* (2007).

[37] Carlos Pacheco and Michael D. Ernst. "Randoop: feedback-directed random testing for Java". In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 815–816.

[38] Renaud Pawlak. "Spoon: Compile-time annotation processing for middleware". In: *IEEE Distributed Systems Online* 7.11 (2006), p. 1.

[39] Renaud Pawlak et al. "SPOON: A library for implementing analyses and transformations of Java source code". In: *Software: Practice and Experience* (2015).

[40] Nancy Pennington and Beatrice Grabowski. "The tasks of programming". In: *Hoc et al* 307 (1990), pp. 45–62.

[41] Michael J. Rees. "Automatic assessment aids for Pascal programs". In: *ACM Sigplan Notices* 17.10 (1982), pp. 33–42.

[42] Forrest J. Shull et al. "The role of replications in empirical software engineering". In: *Empirical Software Engineering* 13.2 (2008), pp. 211–218.

[43] Fabio QB Da Silva et al. "Replication of empirical studies in software engineering research: a systematic mapping study". In: *Empirical Software Engineering* 19.3 (2014), pp. 501–557.

[44] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. "Question Independent Grading using Machine Learning: The Case of Computer Program Grading". In: ().

[45] Shashank Srikant and Varun Aggarwal. "A system to grade computer programming skills using machine learning". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 1887–1896.

[46] Michael Striewe and Michael Goedicke. "A Review of Static Analysis Approaches for Programming Exercises". In: Computer Assisted Assessment. Research into E-Assessment. Springer, 2014, pp. 100–113.

[47] Robert Tibshirani. "Regression shrinkage and selection via the lasso". In: *Journal of the Royal Statistical Society.Series B (Methodological)* (1996), pp. 267–288.

[48] Hannes Tribus, Irene Morrigl, and Stefan Axelsson. "Using Data Mining for Static Code Analysis of C". In: Advanced Data Mining and Applications. Springer, 2012, pp. 603–614.

[49] Tiantian Wang et al. "Semantic similarity-based grading of student programs". In: *Information and Software Technology* 49.2 (2007), pp. 99–107.

[50] Christopher Watson and Frederick WB Li. "Failure rates in introductory programming revisited". In: *Proceedings of the 2014 conference on Innovation and technology in computer science education*. ACM, 2014, pp. 39–44.

[51] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[52] Songwen Xu and Yam San Chee. "Transformation-based diagnosis of student programs for programming tutoring systems". In: *Software Engineering, IEEE Transactions on* 29.4 (2003), pp. 360–384.

[53] Brad Vander Zanden and Michael W. Berry. "Improving automatic code assessment". In: *Journal of Computing Sciences in Colleges* 29.2 (2013), pp. 162–168.