UNIVERSITY OF MASSACHUSETTS DARTMOUTH
Department of Computer & Information Science

Elegance in Code

CIS 600: Master's Project
Final Project Report

Thomas Briggs
Student ID: 01079193

Project Advisor
Dr. Adnan El-Nasan

Table of Contents

List of Tables and Figures

Abstract

Elegance is a more philosophical topic than those usually considered in the study of computer science. The algorithms, languages, applications, and systems that lie at the heart of computer science do not need to be elegant to be scalable, performant, or even correct, after all, and so our instinct is to focus on those more concrete topics. Yet every experienced software developer knows, both instinctively and empirically, that some software is clearly elegant and most is definitely not.

This project attempts to better understand the nature of elegance in software and, if possible, to build a tool to help identify it in arbitrary programs. To that end, research regarding the nature of elegance in general, and specifically elegance in software, was first performed. The results of that research made clear that while elegance is subjective, there is likely a relationship between elegance and complexity. This idea drove the development of a tool for analyzing programs to predict how elegant human evaluators would find them. Development of the tool involved designing and deploying a survey to collect information from human evaluators about the perceived elegance of a selected set of sample programs, generation of a variety of common software complexity metrics for those programs, and the creation of a predictive model that combines those two data sets to predict the elegance of other programs. Analysis of the survey data collected and the performance of the predictive model support both the theory that elegance is related to complexity and also the idea that tools to identify it can successfully be built.

1. The Concept of Elegance

1.1 Elegance in the Computer Science Literature

Less has been written about elegance in software than many of the other topics in computer science. One of the earliest treatments of the subject (and perhaps the first) was by Knuth in 1992, [1] who suggested four criteria for elegance: leanness, clarity, spare use of resources, and suitability. This model was later the basis for a discussion of the subject by Fuller. [2]

A more recent analysis of the subject by Hill [3] also proposes four properties for elegant software: minimality, accomplishment, modesty, and revelation. These overlap with Knuth's criteria in at least a few ways: minimality and leanness are related, for example, as are accomplishment and suitability. Both models make clear that simplicity is a key component of elegance, an idea discussed more directly in an often cited online article [4] and an informal discussion on Stack Overflow. [5]

Another common theme in discussions of elegance is the subjective nature of the quality. In a discussion on elegance in programming language design, MacLennan suggests that "designs that *look* good will also *be* good", [6] while Hill asserts that while programmers agree that some programs are elegant, programmers "find it difficult to articulate incisive reasons." [3] Fuller even argues compellingly that elegance cannot be objective because it is "not absolutely definable at a mathematical level." [2]

Efatmaneshnik and Ryan, however, propose a mathematical and therefore objective formula for elegance: the program with the lowest ratio of complexity relative to the problem to be solved is the most elegant.[7] This determination of elegance is relative to all possible solutions to the problem, however, and does not determine whether an individual program is (or is not) independently elegant. This approach does highlight the importance of the relationship between the complexity of a program and the problem it purports to solve, however. Interestingly, that relationship is also consistent with the models proposed by both Knuth and Hill: how lean and resource-efficient (per Knuth), or minimal (per Hill), a program can be will naturally be limited by the complexity of what it tries to accomplish.

Iandoli et al. also confirm a relationship between elegance and complexity. They asked a group of experts to evaluate a series of ATM system designs and compared the experts' evaluations to the complexity of those designs, where complexity was calculated based on the properties of the system design graph.[10] Their results show that "the level of elegance can be understood in terms of 'desirable' complexity". Here again we see parallels with Knuth's and HIll's models: all of clarity, suitability, and modesty, for example, are relative to the sophistication of the problem a program is designed to solve.

1.2 Elegance in Other Domains

1.2.1 Game Design

A thought-provoking analysis of elegance in game design is provided by Browne.[9] The author proposes a model based on three metrics: simplicity, clarity, and efficiency. Simplicity and clarity are based on a game's rule and decision complexity, respectively, while its efficiency score is based on the "degree to which all equipment contributes to play". The least of those three values then becomes the game's elegance score. These concepts clearly overlap with both Knuth's and Hill's models, and their application to software instead of game design seems fairly intuitive: must not simplicity, clarity, and efficiency be present in a program for it to be considered elegant?

Browne also evaluates games for their *shibui* - a Japanese concept of beauty. Shibui comprises seven elements: simplicity, implicitness, modesty, asymmetry, novelty, authenticity, subtlety. Here too there is obvious overlap with both Knuth's and Hill's models, though because shibui is an aesthetic quality it does not address functionality in any way.

1.2.2 Psychology

Useful insights into the nature of elegance can be found in the world of psychology as well. In one well-respected formula for identifying functional creativity, for example, elegance is one of the four criteria used to determine whether a product is considered creative.[8] It is not hard to consider software applications products; if it is possible to identify elegance in other products, then, must it not be possible to identify elegance in software?

1.3 Why Elegance Matters in Software

Since, by definition, elegance is a desirable trait, any ideas or tools that increase our ability to write elegant code are therefore desirable.

If a tool for evaluating elegance could be developed, it could be used for

- Pedagogy
    - Given an understanding of the idea of elegance in software and its underlying concepts, computer science and software engineering students could be taught how to write better code (or at least avoid writing worse code!)
    - Given tools to identify elegant (or inelegant) code, computer science and software engineering students could be taught concrete methods for improving their code
- Evaluation of existing code
    - To identify opportunities for improvement
    - To evaluate developer performance (e.g. for job interviews or performance reviews)
- Investigating the question of creativity in software
    - The Creative Solutions Diagnosis Scale[8] (CSDS) is a tool used by psychology researchers to evaluate creativity in functional products (vs. creativity in subjective endeavors, such as art). One of the four elements of creativity evaluated by the CSDS is elegance. Thus investigations into the nature of creativity in software that rely on the CSDS, such as the study recently published by Kershaw et al.[11] could benefit from an improved understanding of elegance in software.

2. Building A Tool for Evaluating Elegance in Code

2.1 Conception and Goals

The research into the nature of elegance, and particularly elegance in code, revealed two common themes: first, that elegance is subjective; and second, that there is likely a relationship between elegance and complexity. How, then, could these themes be leveraged to produce a tool for evaluating elegance?

The idea that perceptions of elegance in software are related to complexity is convenient, as complexity analysis is a popular subject in computer science. Algorithms and tools for analyzing complexity are abundant, both commercially and academically. The ready availability of these tools encouraged the idea of building a tool to evaluate elegance as a function of complexity.

The idea that perceptions of elegance in software are subjective, however, is a definite obstacle to building a software tool. First and foremost, this meant that human opinions about the elegance of programs would be required - by definition there is no other way to gather subjective information. Further, it meant that the evaluation of elegance by a tool could not be algorithmic - there is no objective way to calculate something subjective.

Conveniently, however, a statistical model that predicts how human evaluators would rate the elegance of a particular program would address the same need. A predictive model would also be a convenient way to leverage complexity metrics to evaluate elegance, as they could provide the features of the model. It was therefore decided the tool for evaluating elegance in code would be a predictive model.

With those conclusions in mind and the goal of building a predictive model established, it became clear that the project would have four main phases:

1. Identifying sample programs for analysis
2. Complexity analysis of the selected programs
3. Collecting human evaluations of the selected programs
4. Development and evaluation of a predictive model

Each phase is discussed in detail below.

2.2 Identification of Sample Programs

2.2.1 Sample Program Requirements

The first step in gathering human input on the elegance of sample programs was to identify a set of programs to be evaluated. For this project, ideal sample programs to be evaluated by humans needed a number of specific qualities. In particular, ideal sample programs would have:

- Clear problem definitions - Human evaluators need to be able to easily determine what problem the program solves in order to evaluate how elegantly it solves it.

- Verifiability - Human evaluators need to be able to easily determine that the program produces the correct result.

- Known problem complexity - Given the assumption that there is a relationship between complexity and elegance, it is necessary to know how complex the problem that each program solves is.

- Variable problem complexity - Given the assumption that there is a relationship between complexity and elegance, it is necessary to evaluate programs that solve problems with different levels of complexity.

- Variable authors - In order to avoid bias, programs to be evaluated needed to be written by multiple programmers.

- Variable programming languages - In order to avoid bias, the programs to be evaluated needed to be written in multiple languages.

2.2.2 A Source for Sample Programs

Initial suggestions for sources of sample problems included programming competitions such as Google CodeJam, as well as skills evaluation sites for prospective employers such as HackerRank (www.hackerrank.com). While these do generate different solutions to common problems written by many different authors, they have one serious limitation: the accessibility of

the resulting programs. As such they were not seriously pursued as a source of sample programs.

Further research led to the discovery of the Project Euler project (www.projecteuler.net). Project Euler is “a series of challenging mathematical/computer programming problems” created and maintained by a group of volunteers for more than twenty years. Solutions to Project Euler problems are an ideal sample data set for this project for a number of reasons; they have:

- Clear problem definitions - The Project Euler (PE) problems are clear and concise.
- Verifiability - The solutions to all PE problems are single numbers, making it easy to verify the correctness of each program.
- Known and variable complexity - The providers of the PE problems also provide a difficulty rating for each problem.
- Variable authors - Solutions to PE problems from many different authors are available.
- Variable programming languages - Solutions to PE problems are available in a variety of programming languages.

Further, Project Euler participants often make their solutions available through GitHub, addressing the accessibility issue encountered with the other sources. The sample programs analyzed for this project are therefore all publicly available solutions to Project Euler problems.

Project Euler Problem Selection

There are currently more than 800 Project Euler problems available. Four were selected for analysis in this project. Problems were selected based on the following criteria:

1. Minimal required math knowledge - Many of the PE problems are mathematical in nature. To avoid excluding human evaluators without knowledge of advanced math and/or create extra time burden by requiring evaluators to first understand the underlying math concepts, problems requiring limited mathematics knowledge are preferable.

2. Availability of solutions from multiple authors - To avoid biased results because solutions to a problem were written only by a single author, problems with solutions by multiple authors are preferable.

3. Availability of solutions in multiple programming languages - To avoid biased results because solutions to a problem were written in only a single programming language, problems with solutions available in multiple programming languages are preferable.

4. Minimal but varying difficulty - To minimize the time burden on human evaluators, two of the four selected PE problems are from the lowest difficulty level (5%), while the other two are at the second and third lowest difficulty levels (10% and 15%, respectively).

Details about the four problems selected can be found in Appendix A.

Project Euler Solution Selection

After identifying ideal Project Euler problems, five solutions to each of those four problems were selected for analysis. The result is the collection of twenty programs used through the rest of this project. The programs selected were written by a combination of eight different authors in four different programming languages (namely Python, Java, C++, and C). Specifically, nine were coded in Python, six were coded in Java, four were coded in C++, and one was coded in C. This diversity of both authorship and programming languages is intended to help ensure that the results of this project are not unintentionally (and undetectably) skewed by the characteristics of one programming language or one programmer.

Copies of the sample programs selected for and used in this project are available in the CodeSamples directory of the project's GitHub repository.

2.3 Sample Program Complexity Analysis

2.3.1 Selecting Software Complexity Metric Tools

For this project, research was performed to identify existing tools that generate complexity metrics that could be used as inputs to the predictive model. As noted earlier, there are many

algorithms and tools for generating software complexity metrics available. To be viable for use in this project, tools for generating complexity metrics needed to meet three criteria:

1. they needed to be free and open source,
2. they needed to be able to analyze code in a variety of programming languages, and
3. they needed to produce common and well-understood metrics, i.e. nothing experimental

Because of simple bias on the part of the author, preference was also given to tools written in, or at least easily usable from, Python.

2.3.2 Software Complexity Metrics Generated

After investigating more than a dozen tools and libraries for generating complexity metrics, three were selected. Complexity metrics for the selected sample programs were generated using three open source libraries. The three tools, and the complexity metrics produced by each, are:

- Pygount : https://github.com/roskakori/pygount
    - Number of lines of code
    - Number of lines of comments
    - Number of empty lines

- Lizard : http://www.lizard.ws/
    - Number of lines of code
    - Total token count
    - Minimum, maximum, and average cyclomatic complexity per function
    - Minimum, maximum, and average token count per function

- Multimetric : https://github.com/priv-kweihmann/multimetric
    - Comment to Code percentage
    - Cyclomatic complexity according to McCabe
    - Number of imports from out of tree modules
    - Number of imports from same source tree modules
    - Halstead metrics
        - Number of delivered bugs
        - Difficulty

        - Effort
        - Time required to program
        - Volume
    - Lines of code
    - Maintainability index
        - This is calculated using the “classic” method, not the SEI or Microsoft methods
    - Number of used operands
    - Number of unique used operands
    - Number of used operators
    - Number of unique used operators
    - General quality score according to pylint
    - TIOBE metrics
        - Compiler warnings score
        - Complexity
        - Coverage
        - Code duplications score
        - Fan-Out score
        - Functional defect score
        - Security score
        - Language standard score
        - General quality score

Note that these are all the metrics produced by these tools. As detailed later they were not all used in the generation of the predictive model.

2.3.3 Tool for Complexity Metrics Generation

A tool was written, in Python, that iterates over the selected sample programs, generates the complexity metrics listed above for each program using the open source libraries described, and writes them all to a single CSV file. This CSV file then serves as the primary input to another tool (described later) that builds the model for predicting the elegance score of other programs.

2.4 Human Evaluator Input Collection

2.4.1 The Need for a Survey

As previously discussed, the conclusion that perceptions of elegance in software are subjective implies a need for human input about elegance in order to perform any analysis. Ideally that input would be gathered from a variety of people from different places, with different backgrounds and varying experience levels. To that end, a web-based survey asking participants to evaluate the selected sample programs described earlier (the solutions to Project Euler problems) was created to gather information about how elegant those programs are perceived to be. The survey also asks for a small amount of demographic data.

2.4.2 Survey Design

The human evaluator survey has four main sections:

1. A brief introduction
2. Demographic questions
3. Evaluation criteria descriptions
4. Program evaluations

All demographic questions are optional. They were included in the survey in case they proved to be useful for the development of the model, and so that they would be available for future analysis. It could be interesting, for example, to determine if there are correlations between the participants' characteristics and their perceptions of elegance. The presentation of the demographic questions is detailed in a later section.

Program Evaluation Criteria

For each sample program, the survey asks about twelve evaluation criteria. Each is assessed using a five-point Likert-style scale, with the following five options:

1. Not at all
2. A little
3. Somewhat
4. Mostly
5. Very much

The program evaluation criteria on the survey are logically organized into four groups. The four questions in the first group were defined by the author, while questions in groups two and three were taken from the elegance section of the Creative Solutions Diagnosis Scale.[8] The fourth group contains only one question - the overall elegance score. Note that these groupings are for logical organization purposes only - they are not presented in groups on the actual survey.

The four groups and the criteria in each are as follows:

Group 1: Implementation	
Readability	The source code is easy to read.
Maintainability	The program would be easy to maintain over time.
Extensibility	The program would be easy to extend to meet new requirements.
Scalability	The program would scale to larger instances of the problem.
Group 2: External Elegance	
Recognition	You see at once that the solution “makes sense”.
Convincingness	You see the solution as skillfully executed, well-finished.
Pleasingness	You find the solution neat, well done.
Group 3: Internal Elegance	
Completeness	The solution is well worked out and “rounded”.
Gracefulness	The solution is well-proportioned, nicely formed.
Harmoniousness	The elements of the solution fit together in a consistent way.
Sustainability	The solution is environmentally friendly.
Group 4: Overall	
Elegance	How elegant is the program overall?

Table 1: Survey Program Evaluation Criteria

Demographic Questions

The survey begins with the five optional demographic questions. The first four questions are presented using single-select dropdown lists with predefined options. The fifth question has four parts and is thus presented as a grid of radio buttons. The demographic questions are presented this way both to make things simpler for participants and to help ensure cleaner response data.

The five demographic questions and their available response are:

Question	**Available Responses**
Age	< 18 18 - 25 26 - 34 35 - 49 50+ Prefer not to answer
Gender	Male Female Non-binary Other Prefer not to answer
Total Years of Programming Experience	None < 1 1 - 2 3 - 5 5 - 10 10+
Are you currently a student?	No Yes - college undergraduate Yes - college graduate Yes - other
How many years of experience do you have programming in… ● Java ● Python ● C++ ● C	None < 1 1 - 2 3 - 5 5 - 10 10+

Table 2: Demographic Survey Questions and Available Responses

2.4.3 Survey Delivery

The human evaluator input survey was implemented and delivered using JotForm.com. The survey is accessible at this link.

The survey form presents the evaluation criteria for each sample program on a different page. The layout and structure of the criteria are the same for all sample programs, however, and consist of the following elements:

1. The problem number and title
2. The problem description
3. The solution number and a link to the program to be evaluated
4. A grid of radio buttons for scoring the sample program on each criteria
5. A free-form text box for comments
6. A collapsible list of the descriptions of the twelve scoring criteria, for reference

Sample Survey Page

The entry form for the first solution to the first problem appears as follows.

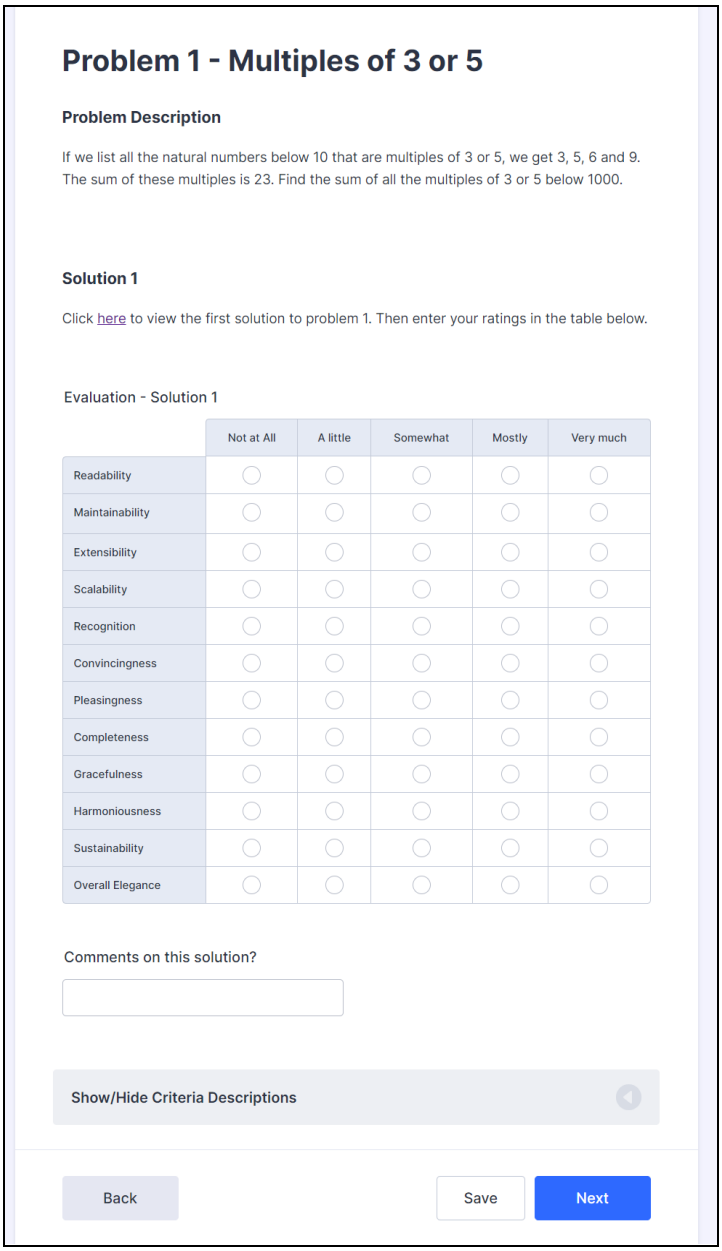

Figure 1: Sample Survey Page

2.5 Predictive Model Development

A model that predicts the elegance score that other, unknown programs would receive from human evaluators was built using Python. The model uses a random forest regressor. For all builds of the model, the training set was the data for eighty percent of the sample programs previously described, selected at random. The data for the remaining twenty percent of the sample programs were used as the test set.

2.5.1 Initial Development

Features used to train the first version of the model were

1. All the complexity metrics previously described in the Software Complexity Metrics Generated section (section number 2.3.2).
2. The complexity score for the given Project Euler problem, as defined by the Project Euler maintainers (e.g. 5, 10, 15)

The average “overall elegance” score provided by human raters was used as the prediction target.

2.5.2 Subsequent Enhancements

After a working model had been produced, a variety of enhancements were made to it.

Derived Features

Based on the premise that elegance is a function of the ratio of the complexity of the problem to the complexity of the solution to that problem, two derived values were added to the feature set:

1. Average cyclomatic complexity per function to problem complexity ratio
2. Maximum cyclomatic complexity per function to problem complexity ratio

These features attempt to incorporate the complexity of the problem into the model in a more sophisticated way than the simple “multiple of five” scores provided by the Project Euler project.

Weighted Targets

The demographic questions asked on the input survey include questions about the participant’s overall programming experience as well as their experience with individual programming languages. These values were used to generate three weighted versions of the overall elegance scores provided by the participants for the sample programs. The three weighted targets produced were:

1. Overall elegance score weighted by the participant’s overall programming experience
2. Overall elegance score weighted by the participant’s experience with the programming language in which the solution was written

3. Overall elegance score weighted by the participant's overall programming experience *and* the programming language in which the solution was written

The intent of these weighted scores is essentially to normalize them. It is reasonable to expect that more experienced programmers will have higher standards for elegance and thus provide lower scores overall, while novice programmers are expected to have more liberal standards and thus provide higher scores overall. Producing scores weighted by experience level attempts to compensate for those expected differences.

Experience Level Weights

To enable these calculations, weights were assigned to each of the available answers to the programming language experience questions on the survey, as follows:

Answer	**Weight**
None	0
< 1	0.1
1 - 2	0.2
3 - 5	0.3
5 - 10	0.4
10+	0.5

Table 3: Experience Level Weighting Factors

Weighting Formulas

Elegance scores weighted by overall or programming-language specific experience were calculated using the formula:

Weighted score = given score * (0.7 + assigned weight of answer to relevant question)

Elegance scores weighted by both overall *and* programming-language specific experience were calculated using the formula:

Weighted score = given score * (0.7 + assigned weight of answer to overall experience question) * (0.7 + assigned weight of answer to language-specific experience question)

This approach increases the elegance scores provided by participants with more experience and decreases the scores provided by participants with less experience. Responses from participants with no programming experience, for example, have their scores reduced by 30%, while participants with more than 10 years of programming experience have their scores increased by 20%.

2.5.3 Resulting Feature Vector

Combining all the complexity metrics and derived features described earlier produced a vector containing forty features. Below is an example of a feature vector provided to the model for a single sample program. While the feature names shown here are those actually used when training the model they do correspond positionally to the complexity metrics listed in section 2.3.2, with the problem complexity measure and derived features added at the end.

Feature Name	**Value**
nloc_pygount	9
comment_count	0
empty_count	5
nloc_lizard	13
token_count	87
num_functions	2
min_func_cc	1
max_func_cc	4
avg_func_cc	2.5
min_token_count	24
max_token_count	52
avg_token_count	38
mm_comment_ratio	0
mm_cyclomatic_complexity	3
mm_fanout_external	0
mm_fanout_internal	0
mm_halstead_bugprop	0.132
mm_halstead_difficulty	13.867

mm_halstead_effort	5495.855
mm_halstead_timerequired	305.325
mm_halstead_volume	396.336
mm_loc	14
mm_maintainability_index	100
mm_operands_sum	26
mm_operands_uniq	15
mm_operators_sum	54
mm_operators_uniq	16
mm_pylint	100
mm_tiobe	97
mm_tiobe_compiler	100
mm_tiobe_complexity	80
mm_tiobe_coverage	100
mm_tiobe_duplication	100
mm_tiobe_fanout	100
mm_tiobe_functional	100
mm_tiobe_security	100
mm_tiobe_standard	100
problemComplexity	5
avg_func_cc_to_complexity_ratio	0.5
max_func_cc_to_complexity_ratio	0.8

Table 4: Sample Feature Vector

The abbreviated field names used in the vector The fields in the feature vector correspond to the complexity metrics

3. Results and Discussion

3.1 Solution Evaluation Survey Responses

3.1.1 Overall

In total, 18 people participated in the Elegance in Software survey. They provided a total of 235 evaluations of the twenty sample programs. (Not all participants provided evaluations for all sample programs.)

3.1.2 Demographics

Age

The number of survey participants in each age range, as reported by the participants, was:

Age Range	Number of Participants
< 18	0
18 - 25	8
26 - 34	5
35 - 49	3
50+	1
Prefer not to answer	1

Table 5: Number of Respondents by Age Range

Gender

The number of survey participants of each gender, as reported by the participants, was:

Gender	Number of Participants
Male	12
Female	3
Non-binary	1
Other	0

Prefer not to answer	2

Table 6: Number of Respondents by Gender

3.1.3 Programming Language Experience

Total Programming Experience

The number of survey participants in each total programming experience range, as reported by the participants, was:

Experience Range	Number of Participants
None	0
< 1	1
1 - 2	0
3 - 5	4
5 - 10	5
10+	8

Table 7: Number of Respondents by Total Programming Experience Range

Per-Language Programming Experience

The number of survey participants in each programming experience range for each language, as reported by the participants, was:

Java

Experience Range	Number of Participants
None	4
< 1	3
1 - 2	5
3 - 5	1
5 - 10	3
10+	2

Table 8: Number of Respondents by Java Programming Experience Range

C

Experience Range	Number of Participants
None	3
< 1	5
1 - 2	3
3 - 5	2
5 - 10	3
10+	2

Table 9: Number of Respondents by C Programming Experience Range

C++

Experience Range	Number of Participants
None	6
< 1	4
1 - 2	1
3 - 5	1
5 - 10	5
10+	1

Table 10: Number of Respondents by C++ Programming Experience Range

Python

Experience Range	Number of Participants
None	1
< 1	5
1 - 2	2
3 - 5	4
5 - 10	5
10+	1

Table 11: Number of Respondents by Python Programming Experience Range

3.1.4 Survey Data Validation

Two types of validation were performed on the survey data: validation of the number of unique responses per solution and inspection for suspicious answer patterns.

Unique Answers Per Solution

First, the number of unique responses per sample program per respondent was calculated. This can be an indicator of invalid data at the solution level, as providing the same answer to all twelve questions for a single program could indicate that the participant did not seriously consider their answers for that solution. This per-solution count was then used to calculate the percentage of programs given all the same scores by each respondent. This provides a better indication of how seriously the evaluations were performed - giving the same score to all questions for a single solution may be a function of the evaluator's opinion of that solution, but giving the same answer to all questions for *multiple* solutions is likely an indicator that the evaluator did not take the survey seriously.

Overall, 34 out of the 235 solution evaluations, or 14%, provided the same scores for all questions. Only one evaluator gave the same score for all questions for all sample programs. The data from that participant was excluded from the data used to train and test the model, as well as all analysis of the elegance scores provided.

Inspection for Suspicious Answer Patterns

Responses were also inspected for suspicious answer patterns. Specifically, responses were checked for the pattern of answering 1 for the first question, 2 for the second question, 3 for the third, etc. The reverse pattern (5 for the first question, 4 for the second question, 3 for the third question, etc.) was also considered. Neither pattern was found in any of the survey responses.

3.1.5 Program Evaluations

Evaluations per Project Euler Problem

As previously noted, 235 program evaluations were provided by the 18 survey respondents, but not all participants provided evaluations for all sample programs. All participants did complete evaluations of either all or none of the solutions to a given Project Euler problem, however. For example, a participant might evaluate all five solutions to the first Project Euler problem and then stop, but no participants completed evaluations of all five solutions to the first problem and the first two solutions to the second problem, and then stop. The number of evaluations therefore varies per Project Euler problem but not per solution within each problem. The number of survey participants and resulting evaluations per problem is detailed below. Note that these

numbers include the one survey response deemed invalid, as discussed in the preceding section.

Project Euler Problem	Survey Participants	Evaluations Provided
1	18	90
2	12	60
3	9	45
4	8	40

Table 12: Number of Survey Participants per Project Euler Problem

Analysis of Overall Elegance Scores

A general analysis of the elegance scores provided by the survey participants for the twenty selected sample programs is provided below. All values are relative to the 1 - 5 (“Not at all” to “Very much”) scale provided by the survey. These results show that the elegance scores provided by the human evaluators do vary by program, even amongst those programs that solve the same problem. Further, average scores are usually, though not always, above the median score of three.

Note that these statistics do *not* include the one survey response deemed invalid, as discussed in section 3.1.4.

Problem Number	Solution Number	Average	Standard Deviation	Variance
1	1	3.59	1.12	1.26
	2	3.71	1.10	1.22
	3	2.94	1.25	1.56
	4	3.24	1.03	1.07
	5	3.71	0.77	0.60
Problem 1 Total		3.44	1.09	1.18
2	1	4.00	0.95	0.91
	2	4.00	0.85	0.73
	3	3.33	0.98	0.97
	4	3.58	0.67	0.45
	5	2.00	1.13	1.27

Problem 2 Total		3.38	1.17	1.36
3	1	3.22	1.30	1.69
	2	3.44	0.73	0.53
	3	3.67	0.87	0.75
	4	2.44	1.13	1.28
	5	3.33	0.87	0.75
Problem 3 Total		3.22	1.04	1.09
4	1	3.00	0.53	0.29
	2	4.13	0.64	0.41
	3	4.38	0.52	0.27
	4	4.00	0.76	0.57
	5	3.63	0.92	0.84
Problem 4 Total		3.83	0.81	0.66

Table 13: Overall Elegance Score Analysis

Distribution of Overall Elegance Scores

In order to investigate the distribution of the overall elegance scores provided by survey participants, the scores were aggregated per solution and score. The results are shown in the following graph.

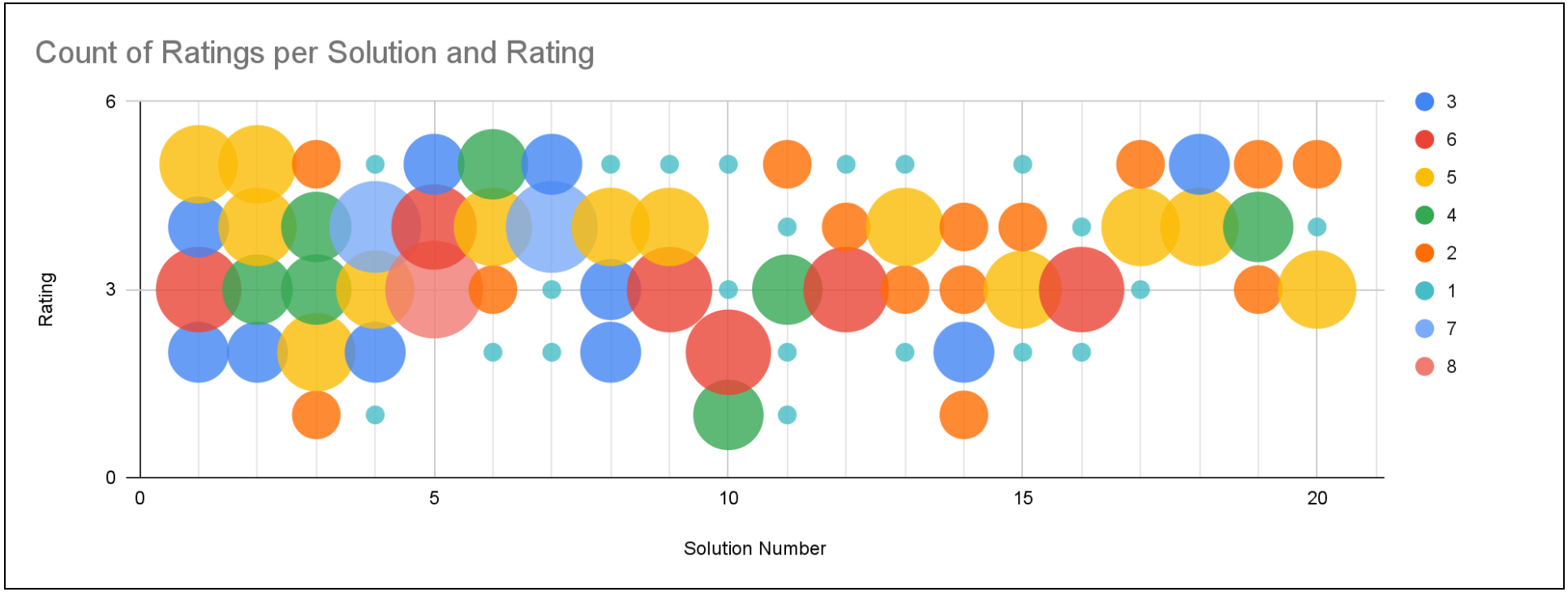

Figure 2: Distribution of Human-Provided Elegance Scores

Here, the x-axis is the sample program number (1 - 20), the y-axis is the elegance score (1 - 5, or “Not at all” to “Very much”), and the size of the circle is the number of people who gave that program the given score. The sample programs are grouped by the problem they solve, so 1 - 5 represent solutions to the first problem, 6 - 10 represent solutions to the second problem, etc. Of particular interest is how the scores for the solutions to the first Project Euler problem (positions 1 - 5 on the graph) are generally well distributed, while the scores for the solutions to the fourth problem (positions 16 - 20) are generally all positive. That may be a consequence of having more data for the first two, less complex problems and their associated ten solutions but may also suggest that more complex problems require more thoughtful, and thus more elegant, solutions.

3.1.6 Correlations Between Responses to Survey Criteria

Correlations between the answers to the twelve criteria that participants were asked to evaluate for each sample program were analyzed. Correlations were determined using a standard Pearson correlation, as implemented by the Python Pandas library. In the table below, correlation values are colored using a white-to-green gradient, with darker green reflecting stronger correlation.

	Readability	Maintainability	Extensibility	Scalability	Recognition	Convincingness	Pleasingness	Completeness	Gracefulness	Harmoniousness	Sustainability
Readability											
Maintainability	0.73										
Extensibility	0.54	0.80									
Scalability	0.28	0.54	0.65								
Recognition	0.71	0.78	0.64	0.48							
Convincingness	0.60	0.76	0.72	0.62	0.77						
Pleasingness	0.67	0.59	0.56	0.42	0.58	0.65					
Completeness	0.63	0.63	0.58	0.44	0.66	0.71	0.64				
Gracefulness	0.64	0.65	0.59	0.55	0.64	0.71	0.82	0.70			
Harmoniousness	0.56	0.68	0.58	0.53	0.68	0.75	0.67	0.65	0.76		
Sustainability	0.43	0.47	0.48	0.67	0.40	0.51	0.54	0.57	0.64	0.57	
Overall	0.69	0.71	0.65	0.57	0.70	0.73	0.80	0.68	0.82	0.71	0.66

Table 14: Number of Respondents by Java Programming Experience Range

Of particular interest for this project are the strong correlations between overall elegance and all other criteria, especially pleasingness and gracefulness.

3.2 Predictive Model Performance

3.2.1 Model Performance Evaluation Process

The performance of the model was evaluated by splitting the data available to build the model into ten randomly generated training vs. testing sets, building the model with each training set, and comparing the scores generated by each model to the human evaluator scores in each test set. All such tests used 80% of the available data for training and 20% for testing, i.e. 16 samples in the training set and four in the test set. The differences between the model-generated scores and human-provided scores were then calculated and averaged to produce an average error amount for the model. This process was repeated for all four variants of the model (the “overall average” model plus the three weighted target models; see Section 2.5.2 for details.) In total 160 total tests were run - 40 for each of the four model variants.

This approach to evaluating model performance was designed to address a number of concerns. For example, if a given training set happened to contain only data about solutions to the simplest problems, the model may not perform as well with solutions to more complex problems. Another example is skewed average elegance scores: because the number of human ratings of per sample program was variable, it is conceivable that the average elegance score for a particular program could be skewed. Generating and testing the models with multiple different combinations of the available data helps compensate for these issues and provides a better picture of how the models perform in general.

Sample Program Representation in Training and Test Splits

As previously noted, the data sets for training and testing the models were randomly generated. In theory, since there were 40 total tests (10 test runs, with four programs in every test set), each of the 20 sample programs should appear in a model test set twice on average. In practice, the distribution was not quite ideal; while six of the 20 programs did appear twice as expected, some programs appeared in test sets more frequently, including one which appeared five times and one that did not appear at all. The number of times each sample program appeared in test sets is detailed in Table 16 in the next section.

Problem Representation in Training and Test Splits

In theory, since there were 40 total tests (10 test runs, each with four programs), solutions to each of the four Project Euler problems should appear in the model test data sets ten times on average. In practice, the distribution was very close to ideal. The number of times solutions to each problem appeared in test sets is listed in the table below.

Problem Number	Appearances
1	10
2	9
3	10
4	11

Table 15: Project Euler Problem Appearance in Test Data Sets

3.2.2 Model Performance

Model Performance Evaluation Results

These results of all the tests of the four model variants are detailed below in Table 16. For space reasons, the models are listed in columns four through eight as simply “Model 1”, “Model 2”, etc.; these numbers correspond to the four model variants as follows:

Model 1: Simple Average model
Model 2: Weighted by Total Experience model
Model 3: Weighted by Language Experience model
Model 4: Weighted by Language and Total Experience model

For more information on the four model variants see section 2.5, Predictive Model Development.

Problem Number	Solution Number	Appearances in Test Sets	Average Score Human Raters	Average Score Model 1	Average Score Model 2	Average Score Model 3	Average Score Model 4
1	1	5	3.59	3.75	3.45	3.84	4.14
	2	1	3.71	3.52	3.32	3.66	3.80
	3	0	2.94	-	-	-	-
	4	1	3.24	2.82	2.54	2.77	3.05
	5	3	3.71	3.68	3.39	3.78	4.06
2	1	1	4.00	3.63	3.19	3.58	4.01
	2	1	4.00	3.52	3.20	3.40	3.82
	3	2	3.33	3.86	3.45	3.69	4.18
	4	4	3.58	3.70	3.39	3.85	4.16
	5	1	2.00	3.26	3.05	3.35	3.54
3	1	3	3.22	3.44	3.12	3.60	3.98
	2	2	3.44	3.30	3.13	3.60	3.83
	3	3	3.67	3.31	3.10	3.57	3.81
	4	1	2.44	2.99	2.74	3.05	3.35
	5	1	3.33	3.23	3.00	3.35	3.78
4	1	2	3.00	3.09	2.98	3.25	3.44
	2	2	4.13	3.99	3.66	4.23	4.51
	3	3	4.38	3.77	3.47	3.92	4.25
	4	2	4.00	3.90	3.54	3.95	4.35
	5	2	3.63	2.70	2.49	2.83	3.07

Table 16: Model Performance Evaluation Results

Overall Performance

Overall, the elegance scores predicted by all versions of the model were quite close to those provided by human evaluators. On average, all four variants of the model (the one predicting the simple average of the human scores and the three predicting the average scores weighted by experience) generated scores within a four tenths of a point (0.4), or 8%, of the human score. The standard deviation of the average error for all four models was also very small, indicating that the accuracy of the model does not vary dramatically.

Model	Average Error	Standard Deviation
Simple Average	0.30	0.07
Weighted by Total Experience	0.36	0.09
Weighted by Language Experience	0.31	0.08
Weighted by Language and Total Experience	0.35	0.11

Table 17: Average Model Error by Model

3.2.3 Feature Importances

One of the useful aspects of a random forest model is the ability to inspect which features were important for the generation of the model. This is often used during model development to remove unimportant features and thus make building later models faster and more efficient. Identifying feature importance is useful for this project independent of the model, however, as it makes it possible to determine which of the complexity metrics are best for predicting elegance.

Feature importances reported by the models varied according to the specifics of the model, e.g. whether the target was a weighted elegance score, whether derived metrics were included, etc. (See the Predictive Model Development and Testing section for details). Feature importance values also varied slightly according to how the training and test data was split. To identify the features that are truly important, the importance values reported for each feature by all versions of the model, across all test/train splits, were collected and analyzed.

This analysis was performed as follows. First, any importance values less than 0.05 were discarded as insignificant. The remaining values were then averaged and counted per feature,

and any feature with only a single occurrence of an importance value greater than 0.05 was discarded as an outlier. Interestingly, this process resulted in most of the features being deemed unimportant, and resulted in a list of only ten truly important features. These features, their average importance value reported by the model (ignoring values below 0.05), and the number of times it received an importance value greater than 0.05 are listed below.

Feature	Average Importance	Times Reported as Important
Average function cyclomatic complexity to problem complexity ratio	0.20	40
Minimum function token count	0.18	39
Average function token count	0.12	37
Multimetric Fan-out Score	0.10	26
TIOBE Fan-out Score	0.10	16
Maximum function token count	0.10	5
Multimetric comment ratio	0.07	6
Maximum function cyclomatic complexity to problem complexity ratio	0.07	5
Average function cyclomatic complexity	0.07	4
TIOBE General Quality Score	0.06	2

Table 18: Feature Importances for Important Features

3.3 Discussion

3.3.1 Survey Data Quality

While less survey data was gathered than desired, the survey data collected was generally high quality. More than half of the survey participants reported having ten or more years of programming experience, which means most of the survey data was provided by expert programmers. Conversely, only one submission was deemed invalid, as determined by the validity checks described earlier, and excluded from the data used to build and test the models.

The one potentially significant problem with the data is completeness. Unfortunately, not all participants completed the entire survey. The number of questions per solution (twelve) times the number of solutions (twenty) means that completing the survey in full requires answering 240 individual questions. For many of the participants, all of whom were volunteers, this proved to be too much. Some evaluated only the solutions to the first two problems, which are the least complex, making human feedback on the solutions to the more complex problems even more scarce.

3.3.2 Survey Response Correlations

Though the vast majority of the survey questions were not used for model development, some interesting insights can be gathered from their answers nonetheless. In general, they all correlated positively with overall elegance scores. Correlations between the seven questions taken from the Creative Solutions Diagnosis Scale[8] (CSDS) were all high, especially those for pleasingness ("finding the solution neat, well done") and gracefulness ("the solution being well-proportioned, nicely formed"). This would seem to imply that a single, overall elegance score is sufficient to gather data about the perceived elegance of a program unless the particular aspects addressed by the CSDS are of interest.

The other four questions, all defined by the author, correlate positively with overall elegance scores as well, though not as strongly as the questions taken from the CSDS. Interestingly, the weakest correlation was between the scores for elegance and scalability ("the program would scale to larger instances of the problem"). One possible explanation for this is that scalable solutions involve more complexity, and the increased complexity leads to a reduced perception of elegance.

3.3.3 Model Performance

Four models were developed and tested as part of this project, one for each of four targets. (For details see section 2.5, Predictive Model Development.) In general, all versions of the model performed fairly well, producing predictions of elegance scores within eight percent of actual scores provided by human evaluators. Interestingly, the most accurate version of the model was the one using the simple average of the human evaluator scores as the target. Whether that is a reflection on the merits of attempting to normalize the human evaluator scores according to their

self-reported experience levels, or simply on the methods used to do so here, is an open question.

3.3.4 Feature Importances

Three things are immediately obvious from a glance at the lists of important features in the “Feature Importances” section:

1. The most important feature overall was one of the derived metrics (namely, average cyclomatic complexity per function to problem complexity ratio).
2. Minimum and average function token count are strong indicators of perceived elegance.
3. Most of the complexity metrics produced for this project are not good indicators of perceived elegance.

The first - the importance of the derived metrics - provides strong evidence for the theory that perceptions of elegance reflect a ratio between the complexity of the problem and the complexity of the solution. In addition, it reflects as a valuable lesson learned about model development. While it is considered common knowledge in the data science world that feature engineering is a key part of developing good models, its significance can not be overstated. Seeing that play out firsthand was definitely enlightening.

Secondly, the fact that minimum and average token count are important features is logically consistent with the idea that simpler solutions are perceived as more elegant, in so far as simpler solutions are usually shorter. Interestingly, however, the number of lines of code was never a significant metric - only average and minimum token count. This would seem to support the idea - often lost on inexperienced programmers - that a developer’s goal should not be to “play code golf”, i.e to write the fewest lines of code possible, but instead to make the code as simple as possible and no simpler.

Finally, the fact that most of the features were deemed unimportant is a practically useful insight. Identifying a small number of complexity metrics that provide good indicators of elegance - or at least perceptions thereof - suggests that simple tools to help identify elegant or inelegant code could be built using those metrics.

4. Conclusions and Future Work

4.1. Conclusions

The goal of this project was to create a tool that evaluates a given piece of source code and predicts how elegant it is likely to be considered by human evaluators. Doing so required development and delivery of a survey to gather data from human evaluators, and development and testing of a model that uses that data to make predictions. Neither endeavor had been previously undertaken by the author. With that in mind, at least from the perspective of the author, both were a success. Designing and deploying a survey to collect input was certainly enlightening. While less data was gathered than desired, the data that was gathered was high quality, and provided a solid foundation on which to build a model. The experience of building and testing a model was also a success; while the statistical validity of the model may be debatable because of the limited data set, the experience of developing and refining a model was highly educational and very valuable.

Educational value aside, the insights gleaned from analyzing the survey data collected provide interesting perspectives on the question of elegance in software. The correlations between overall elegance scores and rating of qualities such as pleasingness and gracefulness suggest that overall elegance scores are reliable indicators of perceived elegance. Conversely, weaker correlations between overall elegance scores and ratings of qualities such as scalability may indicate that some qualities generally considered desirable in software are independent of how elegant that software is perceived to be.

The results produced by the model are equally interesting, perhaps moreso. Identifying minimum token count as the best predictor of how elegant a program will be perceived to be is a surprising result. That combined with the idea that minimum line count is *not* an important predictor of elegance may be an important lesson for novice programmers. The importance (albeit less so than minimum token count) of the cyclomatic complexity to problem complexity ratio metrics is also a key result, as it supports the theory on which this project is based - namely that the perception of elegance in software is based on a relationship between the complexity of the problem and the complexity of the solution.

4.2 Future Work

The completion of this project has generated a number of questions and ideas that may be worthy of further investigation. These include, but are not limited to, the following:

Additional Data Collection

The amount of data collected for this project was unfortunately fairly small. Further, as a function of the author's age and professional experience, the majority of participants were older and more experienced programmers. Additional data would be beneficial in general, and data from less experienced programmers in particular would enrich the data.

Investigation of Elegance at the Method Level

This project investigated elegance in complete programs. The complexity metrics generated and human evaluations gathered both concerned the program as a whole. The fact that many of the complexity metrics deemed important by the models reflect attributes of the methods, e.g. average function cyclomatic complexity to problem complexity ratio, minimum function token count, etc., may indicate that perceptions of elegance in *programs* is a reflection of the *structure of the classes and methods* in the programs more so than the content of the individual lines of code. Evaluating complexity and gathering human ratings of elegance for individual methods could instead provide insight into elegance in individual lines of code. If one assumes this type of evaluation would also require less time on the part of human evaluators it could also provide a way to address the need for additional data.

Other Approaches to Assessing Elegance

Other tools and techniques may provide better assessments of the elegance of programs. Neural networks, for example, may be able to detect patterns in code that we can not observe and/or that are not represented well by complexity metrics such as those used in this project.

Other Methods for Incorporating Demographic Data

Other methods for weighting the human evaluators' elegance scores according to their self-reported experience levels are conceivable. Building models that target scores weighted in those ways could produce better models. Simple filtering of the human evaluations according to their self-reported experience levels, e.g. including only scores provided by evaluators with a

certain level of experience overall and/or with the programming language in question, might also produce better models.

Comparison with Elegance Scores Generated by ChatGPT

It is possible to ask ChatGPT to provide an elegance score for a given program. Comparing these scores with those provided by human evaluators, and with those produced by a model such as the one built for this project, could be fascinating.

Incorporation of Campbell's Cognitive Complexity

Campbell's Cognitive Complexity is a popular software complexity metric for assessing the complexity of programs. It produces a measure of "understandability", which is similar to the idea of elegance in that it "cannot be measured by a strict[ly] mathematical approach". [12] It was not used as a model feature in this project because a library for generating the metric in all five programming languages used by the sample programs selected for evaluation could not be found. Identifying or developing such a library and incorporating that metric into the model could prove beneficial. Determining whether correlations exist between cognitive complexity scores and human ratings of elegance could prove interesting as well.

Analysis of Demographic Data and Elegance Scores

The survey data gathered as part of this project may hold some very interesting insights. For example, do more experienced programmers generally give programs lower elegance scores, possibly because they are more discerning? Do older programmers give programs higher elegance scores, possibly because they are more forgiving? Are elegance scores correlated with gender?

Acknowledgements

First and foremost I would like to thank Dr. Adnan El-Nasan for his support and guidance throughout this project. I am incredibly grateful to have been in his classes, to have learned of his work on creativity in software, and to have worked together with him on this project. He is a gifted educator whose patience and dedication to quality improved not only this project but my entire educational experience as UMass Dartmouth.

In addition, I would like to thank Project Euler for providing the rich collection of well-defined programming problems that I was able to draw from for this project. Similarly, I am grateful to the many Project Euler participants who have published their solutions to Project Euler problems, particularly Project Nayuki, for allowing me to use them for this project.

Last, but certainly not least, I owe many thanks to my wonderful wife Sherri. Nothing I do, least of all this project, would be possible without her ongoing love and support.

References

1. Knuth, Donald Ervin. Literate Programming. Stanford, CA: Center for the Study of Language and Information, 1992. Print.

2. Fuller, Matthew. Software Studies a Lexicon. Cambridge, Mass: MIT Press, 2008. Print.

3. Hill, R.K. (2018). Elegance in Software. In: De Mol, L., Primiero, G. (eds) Reflections on Programming Systems. Philosophical Studies Series, vol 133. Springer, Cham.

4. Perrin, C. (2006). ITLOG Import: Elegance. Retrieved from http://sob.apotheon.org/?p=113 Retrieved from web.archive.org

5. https://softwareengineering.stackexchange.com/a/98303/426803

6. MacLennan, Bruce. "'Who Cares About Elegance?' The Role of Aesthetics in Programming Language Design." SIGPLAN notices 32.3 (1997): 33–37. Web.

7. Efatmaneshnik, Mahmoud, and Michael J. Ryan. "On the Definitions of Sufficiency and Elegance in Systems Design." IEEE systems journal 13.3 (2019): 2077–2088. Web.

8. Cropley, David H., and James C. Kaufman. "Measuring Functional Creativity: Non-Expert Raters and the Creative Solution Diagnosis Scale." The Journal of creative behavior 46.2 (2012): 119–137. Web.

9. Browne, C. "Elegance in Game Design." IEEE transactions on computational intelligence and AI in games. 4.3 (2012): 229–240. Web.

10. Iandoli, Luca et al. "Elegance as Complexity Reduction in Systems Design." Complexity (New York, N.Y.) 2018 (2018): 1–10. Web.

11. Kershaw, Trina C. et al. "An Initial Examination of Computer Programs as Creative Works." Psychology of aesthetics, creativity, and the arts (2022): n. pag. Web.

12. Campbell, G. Ann. "Cognitive Complexity - An Overview and Evaluation." 2018 IEEE/ACM International Conference on Technical Debt (TechDebt). ACM, 2018. 57–58. Web.

Appendices

Appendix A: Project Source Code and Data

All source code written for this project, along with all the data gathered and produced for it, is publicly available in the project's GitHub repository, which can be found at https://github.com/tombriggs/elegantCode.

Appendix B: Selected Project Euler Problems

Following are the details of the four Project Euler problems selected for this use in this project. This information is provided by, and taken directly from, the Project Euler project web site.

Selected Problem 1: Multiples of 3 or 5

Difficulty: 5 (out of 100)
Description

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

Selected Problem 2: Even Fibonacci Numbers

Difficulty: 5 (out of 100)
Description

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

Selected Problem 3: Even Fibonacci Numbers

Difficulty: 10 (out of 100)

Description

In the card game poker, a hand consists of five cards and are ranked, from lowest to highest, in the following way:

- High Card: Highest value card.
- One Pair: Two cards of the same value.
- Two Pairs: Two different pairs.
- Three of a Kind: Three cards of the same value.
- Straight: All cards are consecutive values.
- Flush: All cards of the same suit.
- Full House: Three of a kind and a pair.
- Four of a Kind: Four cards of the same value.
- Straight Flush: All cards are consecutive values of same suit.
- Royal Flush: Ten, Jack, Queen, King, Ace, in same suit.

The cards are valued in the order:
2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace.

If two players have the same ranked hands then the rank made up of the highest value wins; for example, a pair of eights beats a pair of fives (see example 1 below). But if two ranks tie, for example, both players have a pair of queens, then highest cards in each hand are compared (see example 4 below); if the highest cards tie then the next highest cards are compared, and so on.

Consider the following five hands dealt to two players:

Hand	Player 1	Player 2	Winner
1	5H 5C 6S 7S KD Pair of Fives	2C 3S 8S 8D TD Pair of Eights	Player 2

2	5D 8C 9S JS AC Highest card Ace	2C 5C 7D 8S QH Highest card Queen	Player 1
3	2D 9C AS AH AC Three Aces	3D 6D 7D TD QD Flush with Diamonds	Player 2
4	4D 6S 9H QH QC Pair of Queens Highest card Nine	3D 6D 7H QD QS Pair of Queens Highest card Seven	Player 1
5	2H 2D 4C 4D 4S Full House With Three Fours	3C 3D 3S 9S 9D Full House with Three Threes	Player 1

The file, poker.txt, contains one-thousand random hands dealt to two players. Each line of the file contains ten cards (separated by a single space): the first five are Player 1's cards and the last five are Player 2's cards. You can assume that all hands are valid (no invalid characters or repeated cards), each player's hand is in no specific order, and in each hand there is a clear winner.

How many hands does Player 1 win?

Selected Problem 4: Digit Factorial Chains

Difficulty: 15 (out of 100)
Description

The number 145 is well known for the property that the sum of the factorial of its digits is equal to 145:

1! + 4! + 5! = 1 + 24 + 120 = 145

Perhaps less well known is 169, in that it produces the longest chain of numbers that link back to 169; it turns out that there are only three such loops that exist:

169 → 363601 → 1454 → 169
871 → 45361 → 871
872 → 45362 → 872

It is not difficult to prove that EVERY starting number will eventually get stuck in a loop. For example,

69 → 363600 → 1454 → 169 → 363601 (→ 1454)
78 → 45360 → 871 → 45361 (→ 871)
540 → 145 (→ 145)

Starting with 69 produces a chain of five non-repeating terms, but the longest non-repeating chain with a starting number below one million is sixty terms.

How many chains, with a starting number below one million, contain exactly sixty non-repeating terms?