Distribution Agreement

Will Theodore April 10, 2024

Creativity in Programming

by

Will Theodore

Davide Fossati
Adviser

Computer Science

Davide Fossati

Adviser

Nosayba El-Sayed

Committee Member

Robert Roth Jr.

Committee Member

2024

Creativity in Programming

By

Will Theodore

Davide Fossati

Adviser

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2024

Abstract

Creativity in Programming
By Will Theodore

Creative thinking is a valuable skill in professional and academic settings. Being able to quantitatively define and measure creativity is a fundamental step toward helping students improve it. However, in the context of computer programming, effectively measuring creativity is still an open problem.

In this paper, we present a framework based on clustering to assess the creativity of computer programmers from the code they wrote. In particular, we focus on measuring three dimensions of creativity: (1) originality, i.e., how much an individual programmer's solution differs from other solutions to the same problem written by other programmers; (2) fluency, i.e., how many solutions can one programmer produce; and (3) flexibility, i.e., how many substantially different solutions to the same problem an individual programmer is able to write.

We evaluate these dimensions of creativity using a machine-learning model that transforms computer programs into code embeddings, which are real-valued vectors summarizing the semantics of a program in an abstract set of features. We use these embeddings to cluster programs into semantically similar solution types. The distance between a solution and the cluster centers can provide a measure of originality. When we have access to multiple solutions by the same programmer, we can evaluate flexibility by determining the number of clusters the solutions belong to.

We evaluate this approach using a preexisting dataset and new experimental data. The distribution of results and resulting originality scores are generally consistent with theoretical predictions. We also compare student-generated code with AI-generated code from OpenAI's ChatGPT, one of the most popular large language models. The AI-generated programs tended to have higher originality scores than students in our dataset.

Finally, we conducted an experiment with students in the secondary Computer Science course at Emory. These students solve a single problem repeatedly, allowing for measures of flexibility to be assessed. We find a lot of variation within the students' results that generally follow expectations. When evaluating the system against human graders, we found moderate agreement, demonstrating the viability of the system.

Creativity in Programming

By

Will Theodore

Davide Fossati

Adviser

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2024

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Creative thinking is a valuable skill in professional and academic settings, and it is essential to a productive society. The goal of this research is to develop a system that promotes creative thinking by creating a computational model for evaluating creativity in programming and providing direct feedback. We also explore the creativity of programs generated by large language models, since AI tools are commonly used by programmers in industry. [1] After establishing a theoretical framework for measuring creativity, we train and evaluate one dimension (originality) on an existing dataset of human-generated code. We then compare originality scores for students' code with AI-generated code. We conduct a second experiment, designed around our framework, in which we ask students to solve a single programming question as many ways as they can. The second experiment allows us to capture more dimensions of creativity beyond originality: fluency and flexibility. Finally, we compare the performance of our originality scores with consensus among human graders and previous research.

[1] Github claims that its tool "Copilot" is writing 46% of code [8].

Chapter 2

Background

2.1 Studying Creativity

Measuring creativity is not a novel challenge; researchers have been attempting to devise tests that quantify creativity since as early as 1964. One popular suite of tests called the Minnesota Tests for Creative Thinking and Writing was widely used. These tests were first created in 1959 [20], but were continuously revised until a scoring manual [21] was published in 1964. It introduces a theoretical framework that breaks creativity into four dimensions: **fluency, flexibility, originality, and elaboration**.

Fluency ”is represented by the mere number of distinct, non-repetitious ideas given by a subject in his response to the respective tasks.” Flexibility ”is obtained by paying attention to the number of *runs* of ideas belonging to a few but inclusive categories, operations, or principles.” Originality in the original design ”[takes] the basic principles or operations into consideration and gives differential weights to them according to the frequency of their occurrence in the general population.” Finally, elaboration ”is evaluated under the assumption that one must develop and work through his ideas to arrive at better responses and to communicate the results successfully.” [21].

The *circles test* from Yamamato’s revised 1964 manual illustrates these principles.

In the circles test, subjects were given ten minutes to draw in and around as many circles as they can while using the circle as a core component of the drawing.

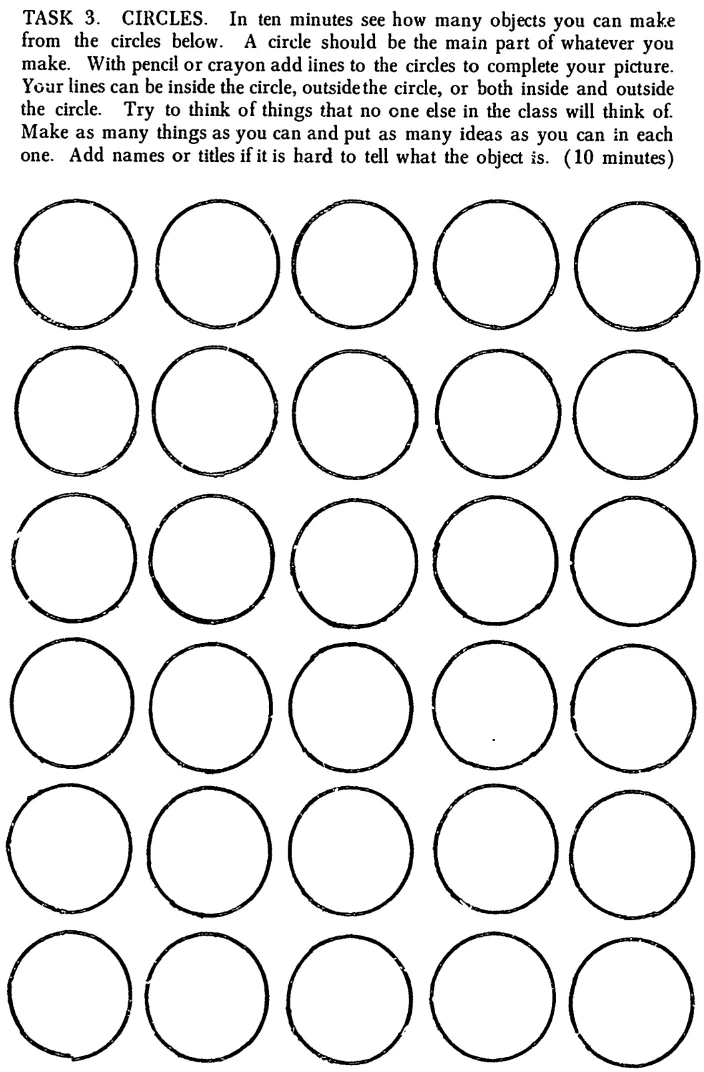


Figure 2.1: The original circles test [21]

Researchers derived fluency from the number of drawings entirely or partially completed. They determined flexibility by sorting drawings into three categories and counting the number of *runs*. They defined a run as a sequence of consecutive responses that fall into the same category. Originality came from a distribution of how often they saw certain elements in real life. The less common a certain principle was in practice, the more originality points it was worth. Elaboration was based on the detail of the drawings and how well they were labeled.

These tests received criticism for being correlated with intelligence [7], and particularly for the design of the tests with regards to fluency. Because the other dimensions require many unique responses, a more fluent subject will typically have high scores in other dimensions. Subsequent studies found that participants' scores were correlated with their intelligence and each other [15]. Correcting for the bias in fluency seemed to alleviate some of these concerns [7]. Although it has flaws, the Minnesota Tests model serves well as a starting point for evaluating creativity in the context of programming.

2.2 Measuring Creativity in Programming

Other studies have attempted to use machine learning techniques to assess creativity in computer programming. Manske and Hoppe [16] use a predictive model trained on expert assessments to directly emulate human evaluations of creativity in code. They found the model was able to correctly predict human assessments of creativity. The Hoppe model was later used to establish a correlation between creative thinking and computational thinking [9]. Kolvakov, et al. [12, 13] also use a predictive model to generate creativity scores from Scratch programs. They find their model agrees with human consensus more often than the experts agreed with each other.

Hershkovitz, Israel-Fishelson, et al. have demonstrated associations between computational thinking (CT) and creativity. After finding preliminary evidence of some association [4], they conducted a randomized experiment with Spanish middle school students. The experiment used an online learning platform, Kodetu, to assess CT and a human-scored assessment, similar to the Minnesota Tests, to assess creativity. Researchers found that creativity contributes to CT and computational creativity can be improved over time [10]. The second finding reinforces the motivations of our research, since a system for providing feedback on creativity may facilitate improvement in computational creativity.

The critical precursor to this study, written by Elijah Chou and Davide Fossati, similarly uses the dimensions presented in the Minnesota Tests [6]. They start by converting programs to an abstract syntax tree, or AST, that represents the structure of a program's operation. ASTs are commonly generated by code compilers as an intermediary step for generating machine code [11]. They then used tree edit distance to generate scores for originality, defined as "the minimum-cost sequence of node edit operations that transform one tree into another." To compute the distance they used a recursive algorithm proposed by Zhang and Shasha [22].

2.3 Word embeddings and code embeddings

To evaluate programs along these dimensions, it is necessary to encapsulate the meaning of the code in some way. Simply evaluating the raw text will not provide an accurate representation of what the program does. To achieve this, we utilize the same principle as **word embeddings** from natural language processing.

A word embedding is a distributed, one-dimensional, vector representation of the meaning of a word. One classic example of this principle shows that $vec("king") - vec("man") + vec("woman") \approx vec("queen")$. Tools such as "word2vec" [17] first applied this to a single word, and later "doc2vec" [14] was able to generate embeddings for sentences and paragraphs.

Our research relies on Code2vec [3], which applies this principle to computer programs. Simply using Doc2vec on a program wouldn't be effective, since those models are trained based on language patterns and not coding syntax. Instead, Code2vec uses an abstract syntax tree, or AST, that represents the structure of a program's operation. During the training of the model, the paths are given weights called attention scores. Attention is the model's way of determining the key aspects of a program, and it is calculated during preprocessing into ASTs. These weights are then run through the deep learning network, converting them to a vector that represents the program.

Chapter 3

Approach

3.1 Method Overview and Definitions

Creativity is an inherently subjective trait, and research on creativity has found that humans often do not agree on what is more creative. For this paper, we use the definition as outlined by the Minnesota Tests [20] consisting of the four dimensions we previously outlined.

By using code embeddings and experimental design, we can get a range of values for three of the four dimensions. The Minnesota Tests quantified **originality** using the frequencies of similar responses in the population. Our metric works similarly: we define originality as the Euclidean distance of a code embedding from the center of its membership cluster, adjusted for cluster size. In essence, this asks the question: How different is this solution from other solutions that take a similar approach?

Due to a limitation we noticed after our first experiment, we adjusted the repeated solutions experiment to the way we calculate originality scores: we also consider the number of other solutions in a program's cluster. For instance, if a program is in its cluster it is substantially different from other programs, but will only have a distance of 0 from the center. To adjust, we use the following formula to compute a cluster size adjustment that we add to the distance-based score. d_{max} is the maximum distance score of any solution without any adjustments. N is the number of solutions for a problem and n_c is the number of solutions in the cluster being evaluated.

$$adj = d_{max} * \frac{N - n_c}{n_c}$$

This adjustment is structured such that "singleton" solutions that make up their cluster are always the most original, as one would expect. It also provides a bonus for membership in a rare cluster, similar to the frequency adjustment in the Minnesota Tests. Overall, this metric provides a comprehensive picture of originality that considers the frequency of similar responses and the distance from the consensus solutions.

Our measure of **flexibility** also relies on the clustering of solutions. Each cluster represents a different way of approaching the problem. In the context of multiple solutions, we define flexibility as the number of unique clusters an individual can produce solutions for. For instance, if a subject generates solutions in 3 different clusters for a single problem, that student has a flexibility score of 3 for that problem. Producing many solutions in a single cluster is representative of less creative thinking than producing many solutions across several clusters, and our flexibility score can capture this.

Finally, we can define **fluency** simply as the total number of solutions a subject generates for a given problem. This closely follows the definition used in the Minnesota Tests, where it is defined as the "number of distinct, non-repetitious ideas given by a subject." [20]

Our research differs substantially from these tests in that they do not involve human grading to come up with the creativity scores. This is intentional since our goal is to explore methods for automated feedback, but the text analysis and program classification required to generate a meaningful score for **elaboration** cannot be done with a machine at this point. Therefore, we chose to omit elaboration scores from this study.

Dimension	**Minnesota Tests Definition**	**Our Definition**
Originality	sum of weighted scores given to each drawing based on frequency in the natural world	distance of solutions from cluster centers and weighted for cluster size
Fluency	number of unique drawings generated by a single subject	number of solutions generated by a single subject
Flexibility	number of runs of drawings belonging to a few distinct categories	number of clusters a subject's solutions have membership in
Elaboration	ability to accurately label and describe drawings	N/A

Table 3.1: Comparison of Definitions with Minnesota Tests (Circles Test) [21]

3.2 Data

3.2.1 CS170 Dataset

To evaluate the system, we used a dataset from Emory's introductory computer science course, CS170. The dataset includes questions and responses from 12 quizzes throughout the course. The quizzes were administered in a controlled setting with robust anti-plagiarism protections. It includes five semesters of data, starting in the fall of 2016 and ending in the fall of 2018. In total, there are 33,437 responses. Each response was given a score by human graders, and we exclude any responses that did not receive full credit in our analysis of the 170 data.

After determining the best clustering method on the entire dataset we test our system on a single question. We considered the capacity for flexible solutions, the period in the course when the question was asked, and the number of functions in the answers when selecting a question. Code2vec is designed to work with a single function, so although multiple functions do not cause an error, questions that require a single-function answer are best. To ensure students have the capacity for flexible solutions, we use questions from later in the course when students have learned more techniques.

For this single-question analysis, we chose the following question from week 7: "Write a method named `mergeRepeat(String[] s, int k)` that takes an array of strings `s` and an integer `k`. The method returns a string which is the concatenation of all the elements in `s` repeated `k` times." This question allows for both recursive and iterative solutions and asks students to write a single function with a consistent signature. Any variation within the code sample should, therefore, be indicative of original thought. We analyze 206 unique responses to `mergeRepeat`.

3.2.2 AI-generated Programs

To generate synthesized code to evaluate the framework against AI solutions, we used the widely available and popular GPT-3.5 model through the Chat-GPT interface [1]. We only tested a few solutions, so we generated them by interacting with the chatbot directly [1].

We generated four code samples from two conversations for clustering with our single-question analysis. In the first, we asked the LLM the exact text of the above question. In the second, I again used the exact question language, but I asked it to come up with a creative solution to the problem. I then asked the model to make the solution more creative, and finally, I asked it to come up with the most creative solution it could. The resulting programs were then cleaned of any helped functions and copied to `.java` files for preprocessing. We use these four programs to provide qualitative analysis of the differences between originality in AI-generated and human-generated code.

[1] Transcripts of these conversations are included in Appendix A.

3.2.3 Experimental Dataset

After evaluating the system to find the optimal clustering technique, we conducted a similar experiment to the circles test. In this test, each student solves a single problem as many times as they can within a two-hour time limit. The dataset includes a short survey in addition to the program submissions. The survey primarily contains questions about the students' computer programming background. To ensure no external assistance was used on the assessment, it was proctored in the Emory Computer Lab using the Qtest program.

We sourced the question for the experiment from Leetcode, a popular programming practice site [2].[2] On Leetcode, the problem had an acceptance rate of 81.7%. We chose a problem with a relatively high acceptance rate because we did not want students to get stuck on their first solution to the problem. Each submission was graded individually using the review system of the Qtest program as a pass or a fail. Any program with compilation errors or that did not generate the correct output was classified as incorrect. Of the five students who took the assessment, only one was unable to submit any acceptable response during the time limit. Overall, there were 26 programs considered. Due to the limited sample size, we restrict our analysis to a qualitative focus in this experiment.

3.2.4 Human-assessment Dataset

We compare our system's originality scores to the evaluation of human graders. Elijah Chou's previous work on creativity in programming provides the dataset for this experiment. Three human graders were asked to pick the more creative option out of two program submissions sourced from the CS170 dataset. The majority vote of these human graders can then be compared to the system to evaluate its effectiveness. There were 30 questions used in the original assessment, but we eliminated two of

[2] The full question is available in Appendix B.

the questions because the helper functions interfered with preprocessing. The specific submissions for each question were randomly selected from a pool of programs that all scored full points. To help eliminate bias over correctness, human graders were informed that all programs received full credit. Since the original survey involved problems where each solution corresponded with an individual student, there was no way to check fluency or flexibility scores against human graders using this dataset.

3.3 Preprocessing

Preprocessing Java code submissions for our analysis involves three main phases. First, each submission must be cleaned individually. We used a script to change the `.txt` files to `.java` files and split the dataset into testing, training, and validation sets. For any data that required vector conversion, we had to remove the main method and any helper methods from the code. Stripping extra methods prevents Code2vec from converting them to vectors.

Second, we run the cleaned programs through the Code2vec preprocessing script, converting each folder of submissions into a single file where each line contains a label and the abstract syntax tree for the program. Since the original Code2vec script did not retain any information from the file during preprocessing, we changed parts of the Code2vec program to retain the program metadata in the vector representation.

After training the Code2vec model on the 170 submissions, we perform the final step of preprocessing. In this final step, we run the Code2vec model on the testing set with the export vectors flag enabled. This generates a `.c2v.vectors` file where each line contains a program. The first value contains the program metadata, and the rest of the line contains the embedding of that program. Each code embedding has 384 unnamed features that describe that program.

3.4 Evaluation of Clustering Methods

After converting this dataset to vector format using code2vec, we need to determine the most effective clustering algorithm for our research. We do this by evaluating each of the clustering algorithms based on the number of points it can classify in a cluster and the number of clusters it generates. We conducted the initial analysis of clustering methods on all of the questions, and we expected to see 50-100 clusters. We try to optimize for a minimal percentage of noise points in the clustering algorithms. Nose points are defined as points without cluster membership. We try to minimize the number of noise points when comparing clustering algorithms since we cannot compute originality from our definition of a noise point. We then confirm the effectiveness of the clustering on a smaller dataset comprised of a single question. This allows us to begin calculating originality scores for the students' code.

3.5 Validation of Originality Scores on 170 Dataset

After identifying the algorithm, we tune the hyperparameters for a single-question scenario, simulating the circles test [21]. We separate all the solutions from the 170 dataset for `mergeRepeat` that received full points and fine-tune the Mean Shift clustering for the reduction of noise points. We then evaluate the framework on the originality dimension by generating distances for each sample from their respective clusters and organizing the results into a histogram. We then repeat the process for the AI-generated solutions to `mergeRepeat` and compare the originality scores between the AI and human-generated code. The system will show promise if it can correctly identify differences in originality between the two. We choose not to look at flexibility during this stage since each student is submitting a single program.

3.6 Repeated Solutions Experiment

We conducted the repeated solutions experiment after evaluating the different clustering methods. After collecting the submissions and conducting the preprocessing for the data, we clustered all the solutions using the optimal method. We changed the criteria for this experiment due to the smaller sample size. While we only used correct solutions to evaluate clustering methods, we included both correct and incorrect responses in this experiment so we could evaluate any correlation between correctness and cluster membership. We show 2D representations of the embeddings with their cluster membership, cluster centers, subject number, and grading status. We also evaluate originality scores for each submission and present them in a histogram. We expect the originality scores to show variation and be relatively unbiased for correctness.

The priority of this experiment is to evaluate the system at the individual unit of analysis instead of the submission. To achieve this, we calculate an average originality score across all responses for a given student, as well as a flexibility score and fluency score as previously defined. We then plot originality vs flexibility with fluency represented as marker size to give a comprehensive picture of the participants' creativity. We expected the system to capture variation in the students' responses. A successful system will also be able to differentiate between these dimensions. In other words, we should not see that flexibility and fluency provide similar results to average originality scores.

3.7 Comparison with Human Analysis

Finally, we compare code embedding-based originality scores with human graders and an approach based on Tree Edit Distance. As discussed previously, we eliminated two of the questions due to the different helper functions required. For each remaining question, we pulled all of the responses that received full credit into a new dataset.

We then fine-tuned the clustering for each question by re-running our algorithm while incrementing the bandwidth. We chose the bandwidth that resulted in the maximum clusters of fewer than 12 with no noise points.

After fine-tuning the clustering for each question, we clustered each question individually to generate originality scores for the programs that were randomly chosen in the human survey. The program with the highest originality score was considered more creative by our system. We then compare our system's choices for the more creative program with the human graders and Elijah's Tree Edit Distance scores. We expect our system to generally show a correlation with human graders, although we do not expect it to be perfect. Human graders only had access to two solutions when making their decision, whereas the Tree Edit Distance scores and our originality scores both depend on all solutions for a given problem. Therefore, the human graders are being asked a slightly different question than the computer systems, and we should see some inconsistency.

Chapter 4

Experiments

4.1 Evaluation of Clustering Methods

Training a code2vec model requires a training set, testing set, and validation set. Once the model is trained, vectors can be generated for any code. We randomly classified 80% of the 170 dataset into a training set, 10% into a validation set, and 10% into a test set. Each embedding is a 1D array with 384 features that represent the neural network's interpretation of the weighted AST. Since each feature is simply a real number, we can apply machine learning clustering algorithms and calculate the Euclidean distance between the samples.

K-means is a classic ML clustering algorithm that uses the distance between vectors to cluster them. We considered this algorithm first due to its simplicity, and we used the sci-kit learn python package to apply this and the other clustering algorithms [18]. Figure 4.1 shows the CS170 dataset as clustered by K-means.

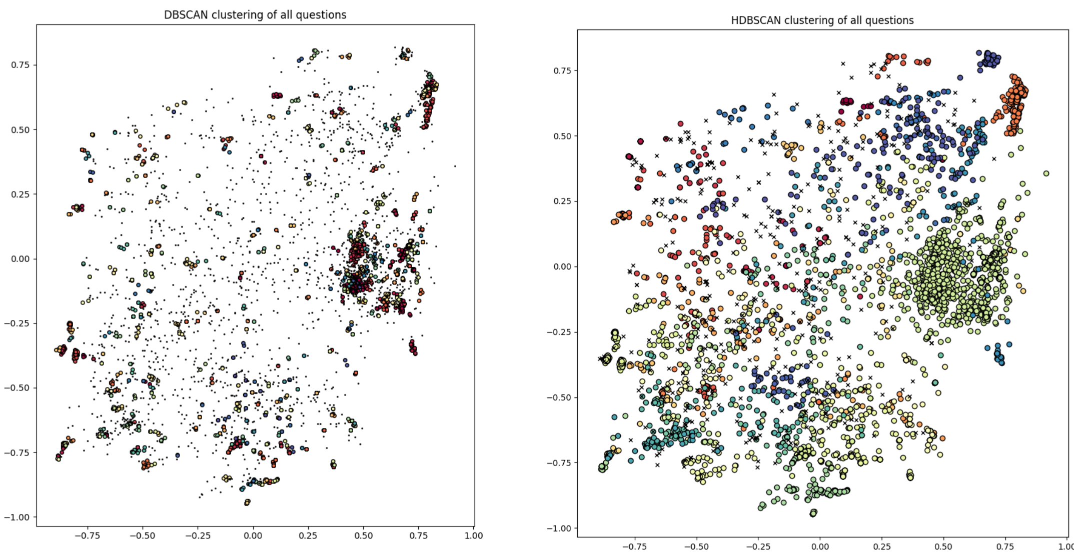
The 2D representations of vectors are only an approximation due to their high dimensionality. One significant limitation of K-Means is that it requires a set number of clusters. Since participants' results could take any number of forms, it is better for our purposes to use an approach that builds clusters dynamically. We therefore

Figure 4.1: K-means clustering of CS170 dataset

consider density-based clustering techniques: DBSCAN, HDBSCAN, and Mean Shift.

DBSCAN uses density to create clusters, allowing them to be any shape. This differs from k-means, which is distance-based and therefore requires clusters to have a convex shape [18]. DBSCAN separates high-density areas, creating *neighborhoods* that consist of *core points*. Points in low-density areas are instead classified as *noise points* and are not given a cluster.

HDBSCAN also uses density-based clustering, but it relaxes an assumption made by DBSCAN that the density requirement of neighborhoods is globally homogeneous [18]. This fits the theoretical model of creativity well since we can think of different clusters as "categories" of solutions to a specific problem. One category may be more popular than another and therefore have a higher density. It would be incorrect for our model to presume that all clusters require the same density, so we expect HDBSCAN to perform better on our data. Figure 4.2 shows the initial results of DBSCAN and HDBSCAN on the CS170 dataset.



(a) DBSCAN (b) HDBSCAN

Figure 4.2: DBSCAN vs HDBSCAN Clustering of All Questions in CS170 Dataset

DBSCAN can separate the largest clusters, but areas of lower density are not categorized. HDBSCAN does not encounter this problem; by varying the density across the solution space, fewer noise points are captured. We desire the least noise points possible; since these solutions do not belong to any cluster, we have to come up with a workaround for originality.

Finally, we consider Mean Shift. Mean Shift is a centroid-based algorithm that considers samples as candidates to be a centroid for a cluster [18]. It then iterates over each sample to find blobs of smooth density for clusters. Centroids shift as the density areas change, but they all start as part of the sample. The algorithm can be adjusted by the bandwidth parameter that sets the size of the region to search around each potential centroid. Figure 4.3 shows the initial results of Mean Shift in 2-d.

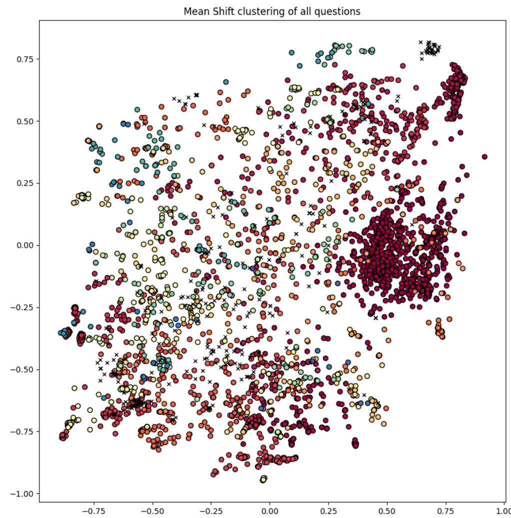


Figure 4.3: Mean Shift clustering of CS170 dataset

Table 4.1 shows the evaluation criteria for all of the density-based models.

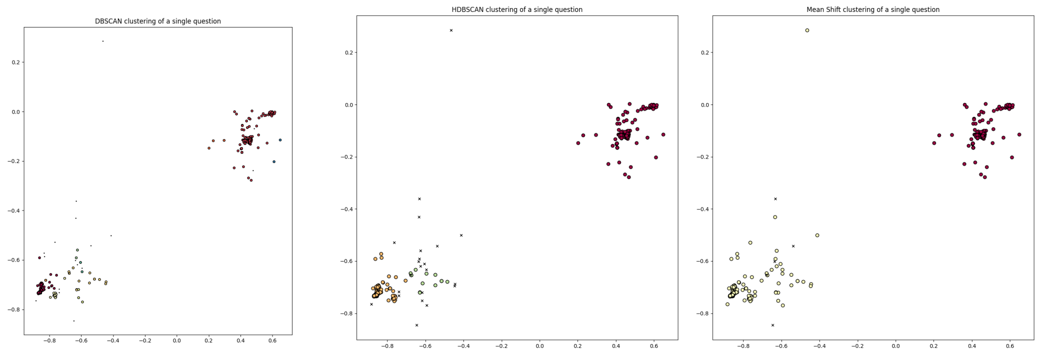
Method	# Clusters	Noise Points
DBSCAN	387	32.44%
HDBSCAN	79	7.87%
Mean Shift	72	5.97%

Table 4.1: Clusters and Noise for Initial Algorithms

DBSCAN produced far too many clusters and was unable to classify several of the points. HDBSCAN and Mean Shift performed similarly, producing around 70-80 clusters with a low percentage of noise points. Based on these criteria, Mean Shift seemed to be the best choice, due to the lower percentage of noise points when compared to HDBSCAN. However, further analysis with a single-question dataset was required to determine the optimal density-based algorithm.

4.1.1 Clustering a single question

To address these concerns, we tried each of the models again on a subset of the data comprised only of our specific question. Figure 4.4 and Table 4.2 show the results.



(a) DBSCAN (b) HDBSCAN (c) Mean Shift

Figure 4.4: Density-based algorithms applied to a single question

Method	**# Clusters**	**Noise Points**
DBSCAN	10	3.85%
HDBSCAN	3	4.71%
Mean Shift	2	1.07%

Table 4.2: Clusters and Noise for a single question

In the context of the entire dataset, we expected 50-100 clusters. Since we are now looking at one question, fewer than 5 clusters are ideal. Although the two-dimensional representation of vectors does not show the full picture, it seems that two categories are most appropriate for this question based on the scatterplots. As with the dataset comprised of all the questions, a minimal number of noise points was desired.

We selected the algorithm that shows an appropriate number of clusters with minimal noise: Mean Shift. It logically divides the sample into two clusters, which we can think of as different approaches to solving the problem. DBSCAN performs better on the reduced dataset but produces too many clusters in this situation HDBSCAN had an appropriate number of clusters but had more than four times as many noise points when compared to Mean Shift. Considering these factors, we chose Mean Shift to perform the clustering for our analysis.

4.2 Validation of Originality Scores on 170 Dataset

Once we determined that Mean Shift was optimal, we clustered the single-question responses from the 170 dataset together with the AI-generated programs. As described above, we use Euclidean distance to generate scores. At this point, we do not include any adjustments for cluster size. Figure 7 shows a histogram of the students' responses only and their distances, ranging from 0 to 10.

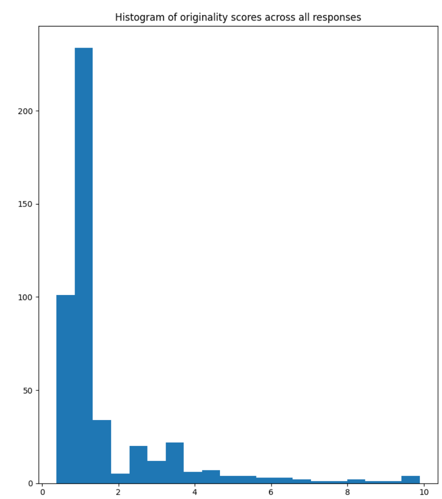


Figure 4.5: Originality scores for students

These results are encouraging, since the distribution generally follows expectations based on theory. We expect that most students will come up with similar answers, and the majority of solutions will therefore have a low originality score. There are a few students who slightly varied from these dominant solutions that make up the second "bump" in the histogram. At the tail of the distribution are the highly original solutions that varied a lot from the clusters. Then, we followed the same process to determine scores for the AI programs, shown in Table 4.3.

These results are particularly intriguing. The system finds that AI programs are generally more *original* by these criteria, earning a distance of 7.176 for the simple

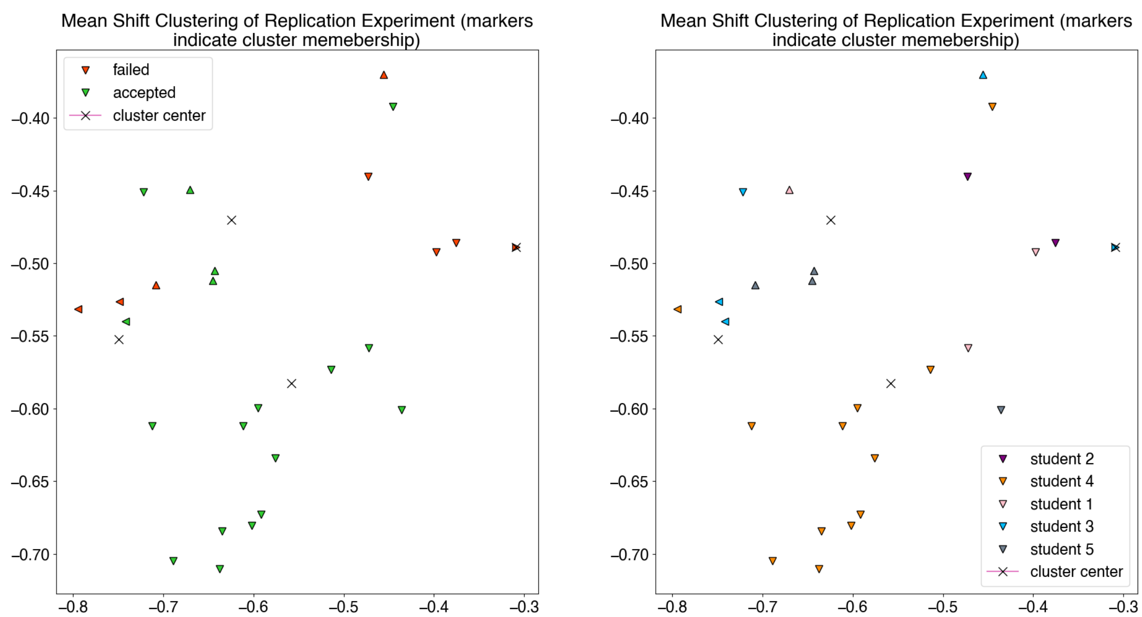
Program	Originality Distance (unadjusted)
simple	7.176
creative	9.642
more creative	8.732
most creative	8.683

Table 4.3: Originality distances for AI programs

prompt without asking for a creative solution. Another interesting finding is that asking the LLM to increase the creativity of the program actually decreased the distance to its closest center. The most *original* AI-generated solution was its first response when asked to provide a creative solution.

4.3 Repeated Solutions Experiment

4.3.1 Clustering



(a) Colored by program score (b) Colored by student number

Figure 4.6: Embeddings of Programs in the Repeated Solutions Experiment

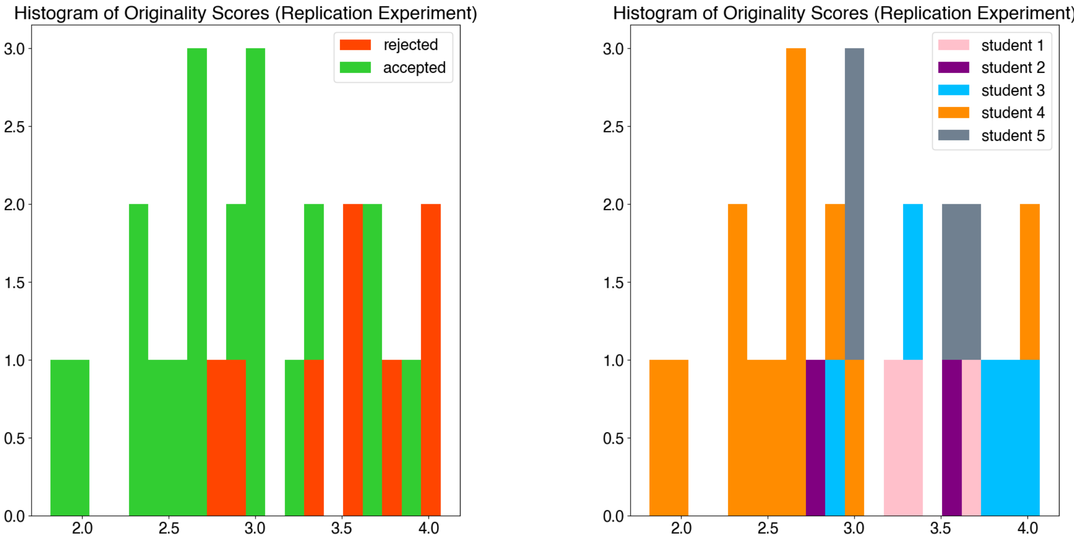
After clustering all of the solutions for the repetition experiment and fine-tuning the model for the most clusters under 10 with no noise points, we found four clusters in the solutions. One cluster contained most of the solutions, and another cluster contained only one solution. There were two other clusters that each had a few submissions. All of the clusters had at least one incorrect submission, including the singleton.

Of the 26 total solutions, the majority (17) were placed in a single cluster, denoted by the downward triangle on the charts. This cluster also contained the highest correct result percentage. 82.35% of the solutions in this cluster were accurate, and the next-highest cluster had 60%. One might interpret the large cluster as the "simple" approach to the problem and the other clusters as more "unique" approaches that weren't attempted as frequently. We see lower accuracy in the more unique approaches, as we would expect since they are less commonly practiced.

The student-by-student breakdown also provides an intriguing case study for embeddings. Due to imbalances in the fluency of students, many of the solutions in the largest cluster come from student 4. That student was an outlier in terms of their fluency, but they were not the most flexible since student 3 produced the singleton. Student 2 was the least flexible, producing two incorrect results in the largest cluster. Overall, the students showed an ability to produce a range of solutions with strengths and weaknesses in different dimensions of creativity.

4.3.2 Student Analysis

After clustering, we calculate the originality scores from the cluster distance. We also introduce the cluster size adjustment at this stage to compensate for outliers such as the singleton cluster that wasn't present in the larger dataset. We find that student 4, while being highly fluent, produced several similar solutions. The similarities limit the originality of all of them, leading to several entries at the low end of the histogram. Students 1 and 2 produced mostly original, albeit incorrect, solutions. We see from the



(a) Colored by program score

(b) Colored by student number

Figure 4.7: Histograms of Originality in the Repeated Solutions Experiment

accuracy histogram that incorrect solutions are generally more original than correct solutions in our experiment. This could suggest that the cluster center represents some amount of accuracy for a given solution, but the small sample size of this experiment prevents us from making strong claims about the correlation between accuracy and originality.

Finally, we average the originality scores for each student and plot all three dimensions together in a scatter plot. Originality is on the x-axis, flexibility is on the y-axis, and the size of each dot represents fluency. We see that student 4 is penalized in originality due to many of the solutions being similar. One might consider student 3 to have the most creative response overall; they have the strongest average originality and are considerably more flexible than the other students. Another could argue that student 4 is the most creative, since their fluency was significantly higher than the other students, and the difference in flexibility is smaller when accounting for incorrect solutions.

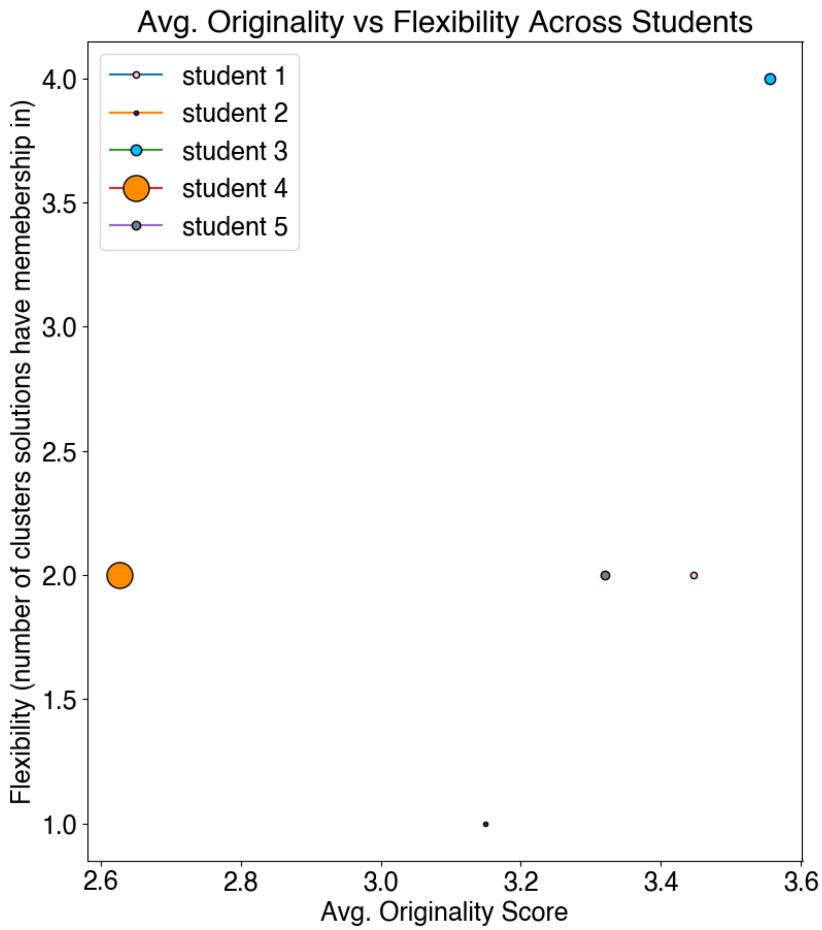


Figure 4.8: Originality vs. Flexibility

We note that many of the most original solutions are incorrect. This is true for student 3 as well, including the very original singleton cluster. We leave the balancing of the dimensions as a composite score as an open question for further research, since we cannot say whether originality, flexibility, or fluency contributes more to creativity. Answering this question involves some degree of subjectivity and is not the goal of this research.

4.4 Validation against human graders

Our results for validation against human graders largely follow expectations. Our system generates a few ties in cases where singletons run against each other. In this case, they both receive the same score and a tie occurs. We exempt these cases from the final accuracy calculations. We include a simple agreement percentage, correlation coefficient, and κ (kappa) statistics for each system and the majority opinion of the human graders. We also show agreement between the human graders themselves, who generally disagreed.

System A	System B	Agreement %	κ	Pearson Coef.
Embeddings	Humans	72	0.4337	0.447
Tree Edit	Humans	76	0.5192	0.519
Tree Edit	Embeddings	64	0.2718	0.280

Table 4.4: Comparison of Systems and Human Graders

Human A	Human B	Agreement %	κ	Pearson Coef.
Grader 1	Grader 2	40	-0.2058	-0.207
Grader 2	Grader 3	72	0.4373	0.439
Grader 1	Grader 3	36	-0.2821	-0.282

Table 4.5: Comparison Amongst Human Graders

We see that our embeddings-based system agrees with majority opinion 72% of the time. The κ and Pearson correlation coefficient show agreement outside of what we expect from random chance. These results suggest that the originality scores encapsulate some degree of creativity in the students' responses. Since humans are imperfect judges and they did not have access to the full pool of submissions, these results are inconclusive and do not suggest the system is entirely accurate.

Although Tree Edit Distance seemed to perform slightly better than code embeddings, the difference is within the range of error we expect to see from a small sample of human graders. When both systems agreed, they matched the human graders 87.5% of the time, for 14 out of 16 problems where there was consensus. Although this only happened for 64% of problems, the high accuracy of the combined systems is particularly encouraging. This suggests that both methods are worth exploring further and elucidate a degree of creativity that is clear to the majority opinion of human graders.

Chapter 5

Analysis

5.1 High AI Originality

We observe high originality distances for the AI-generated programs we analyzed. The dataset involved in training the model is predominantly written by CS170 students, who are new to programming and do not have access to every technique or library. GPT-3.5, however, is trained on a large dataset containing of millions of lines of code. Therefore, it follows that there would be a large distance between AI code and clusters that are primarily comprised of human code. For instance, it is entirely possible that the AI solutions, although original when compared to human solutions, lack flexibility and form a cluster themselves.

We observe similar cases of AI differing from humans in other areas. For instance, it has been well-documented that LLMs have a tendency to overproduce certain adjectives, such as *meticulous* or *commendable*. Liang, et al. find a sharp increase the frequency of these words in peer review since the introduction of these models [5]. Research like this would suggest that AI models, while trained on human-generated content, tend to produce content that is substantially different from humans and similar to itself.

Further research on the topic should include solutions from multiple models to compare the effect. Do solutions from the same model cluster closely together? Do all models form a cluster or does each model generate their own cluster? What about the fluency of an AI model, and how would prompt it to exhaust all solutions? These are all worthwhile questions that go beyond the scope of this paper.

5.2 Accuracy vs Originality

Our research replicates the finding from Elijah's work that performance decreases with increased originality scores. We coded program acceptance or rejection in our repetition experiment as a dummy variable, and we ran a simple linear regression with originality scores as the independent variable. We find a negative correlation between originality scores and accuracy of with a coefficient of -0.378 and a standard error of 0.137. However, the R^2 was only 0.24, indicating that performance stems from several other factors in addition to originality.

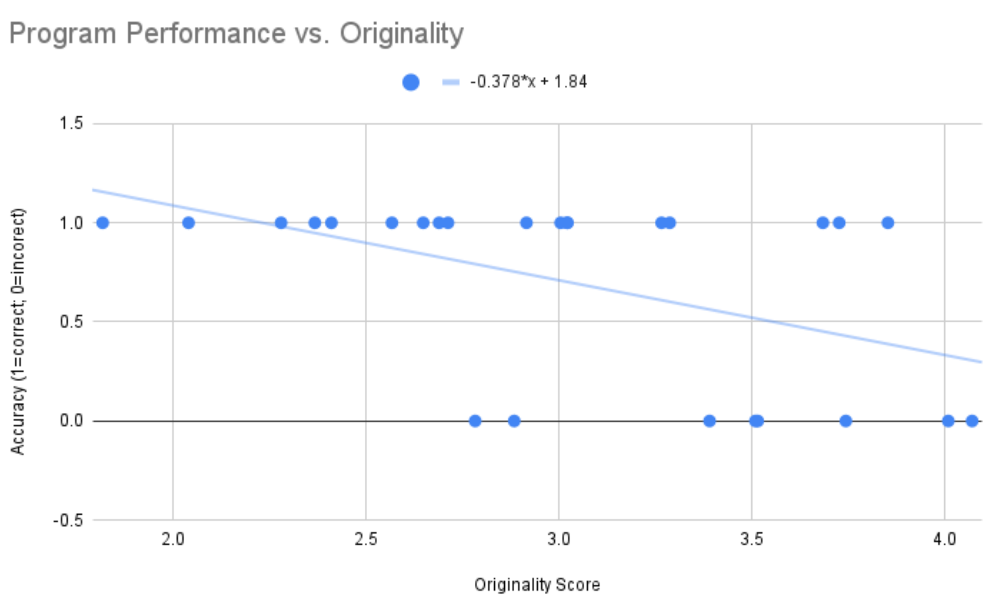


Figure 5.1: Performance vs. Originality in the Repeated Solutions Experiment

We believe this correlation occurred in our experiment because incorrect solutions were often very different from correct solutions. The correct solutions in our experiment were more similar to each other and generally closer to cluster centers. Incorrect solutions were more likely to be in smaller clusters or make up a singleton cluster. Due to this discrepancy, we expect to see a boost in originality scores for incorrect solutions.

Another theoretical framework might explain this correlation. We start by considering a theoretical clustering of all programs for a specific problem, with an extremely large number of programs. For instance, one might imagine clustering every Leetcode submission for a single problem. Due to the thousands of programs, we do not expect to see any singleton clusters. In this context, we can think of cluster centers as platonic forms [19] of solutions, where each center captures the meaning of that approach. We would expect the cluster centers that represent correct approaches to have more solutions than cluster centers that represent incorrect approaches since programmers are trying to solve the problem. In a similar theoretical clustering of every correct solution, we might think of the centers as an exhaustive set of approaches to solving a problem. We cannot confirm this framework with the small sample sizes in our research, yet it demonstrates the possibilities of clustered code embeddings and their deeper meaning.

5.3 Agreement with Human Graders

We find the tree edit distance approach has slightly better performance than the code embeddings approach when compared to the majority opinion of human graders. This could simply be due to disagreement among human graders, who seldom agreed on the more creative solution. If tree edit distance is truly closer to human consensus, it could be explained by the method behind tree edit distance when compared to code embeddings. Since the embeddings are generated through a deep learning model, the meaning of each feature is a black box. Tree edit distance, however, is computed directly from differences in the program structure. Although the embeddings also originate from abstract syntax trees, the direct distance comparison is likely closer to what humans subconsciously do when identifying an original program. For instance, one might consider other approaches to different problems they had already seen, and consider the program most different from those approaches as the most original.

We also find low levels of agreement between the two systems when compared to human graders. While each system had a moderate agreement with the majority opinion, they had a low agreement with each other. This indicates that each system has its strengths since they were able to disagree on solutions and still agree with humans most of the time. A comprehensive picture of program originality likely incorporates tree edit distances with the insights gained from clustering.

5.4 Applications

One challenge with expanding the system for automatic feedback is the model fine-tuning. Although Mean Shift was able to provide an appropriate clustering for all of the problems, each problem required individualized fine-tuning. We solved this in the human grading experiment by setting subjective parameters for the resulting clusters and retraining the model through a range of bandwidth settings to find the optimal clustering. In the context of an automated system that clusters a large number of responses, this would be computationally expensive and undesirable.

An application of clustering in an educational setting would likely rely on an existing solution pool from previous semesters. A solution pool would allow faculty to pre-train a model for each question and find an optimal bandwidth ahead of time. Existing bandwidths significantly reduce the computational load and allow for real-time feedback.

Feedback on flexibility and fluency may be impossible in existing classroom situations. While we were able to engineer an experiment that specifically assessed these dimensions of creativity, students are rarely asked to do so in the classroom. Computer Science students are seldom asked to solve the same problem again with a different approach. Providing feedback on these dimensions would likely require a specific course designed around expanding creative thinking in programming. Such a course might regularly ask students to think about problems in different ways and consider solutions that are not obvious. We believe such a course has the potential to drive students to consider new approaches to solved problems or to attempt new methods for unsolved problems in the field, leading to new discoveries.

Chapter 6

Conclusion

6.1 Limitations

There are a few limitations to the research presented here. Firstly, several aspects of the study suffered from a small sample size. We had five students in the repeated solutions experiment, and only three human graders. Additionally, we only considered responses from a single AI model, since it was not the focus of the research. Replicated findings with a larger sample size would bolster the research.

We also acknowledge limitations in the research design of the survey given to human graders. Since they were unable to compare the solutions they were grading with the rest, there was little basis for choosing the more creative program beyond their personal experiences. Another limitation is the lack of experience many programmers in the study had. Because the system is intended for an educational study, we chose to use students early in their careers. This limits the ability of some students to come up with new approaches since they haven't been exposed to every concept yet. A final limitation of our research is the applicability of fluency and flexibility. As previously discussed, these dimensions are rarely seen in the curriculum and are difficult to apply in practice.

6.2 Conclusion

Creative programmers are an asset to society, and providing direct feedback on creative thinking for students would help them acquire this critical skill. Without a standardized measurement for creativity, we explored methods for quantifying three dimensions: fluency, flexibility, and originality. Code embeddings show promise for encapsulating the meaning of a program into a set of features, and these features can be effectively clustered to show different approaches to solving a problem. By generating an originality score based on the distance to the cluster center, we can begin to quantify one aspect of creativity in programming. Our repeat solutions experiment derived from the circles test [21] demonstrates that we can apply code embeddings and experimental design to gain scores for fluency and flexibility as well. We find that more original solutions tend to be less accurate, although this is not always the case.

Our research raises questions about AI-generated code that are worthy of further consideration. With a larger sample of LLM-generated programs, what clusters form and how do solutions interact? And at what point does the model run out of new ideas and start to repeat old ones? Further research may also consider clustering with a much larger sample size for a single question. Another study might consider the repetition experiment with more experienced programmers and a more difficult question with a lower acceptance rate. A more challenging problem would more closely emulate the tasks that professional programmers and academics regularly encounter.

Improving creativity will always be a goal of education, and the potential applications of an automated feedback system for creativity could help students learn creative thinking skills and how to apply them to programming. Developing creative thinking in coordination with computational thinking is critical to developing productive citizens. By incentivizing students to be more creative, we can develop programmers that have the skills to break down barriers on unsolved problems in the field.

Appendix A

Chat-GPT Transcripts

A.1 Simple solution

USER: Write a method named mergeRepeat(String[] s, int k) that takes an array of strings s and an integer k. The method returns a String which is the concatenation of all the elements of s repeated k times.

Examples:

mergeRepeat("A", "BC", 3) returns "AAABCBCBC"

mergeRepeat("hey", "you", "hi", 2) returns "heyheyyouyouhihi"

ASSISTANT: Certainly! Here's a Java method named mergeRepeat that fulfills the specified requirements:

This Java program includes the mergeRepeat method, and in the main method, there are examples demonstrating how to use this method with different input arrays and values of k.

```java
public class StringMerger {
    public static void main(String[] args) {
        String[] example1 = {"A", "BC"};
        int k1 = 3;
        System.out.println(mergeRepeat(example1, k1));  // Output: "AAA

        String[] example2 = {"hey", "you", "hi"};
        int k2 = 2;
        System.out.println(mergeRepeat(example2, k2));  // Output: "hey
    }

    public static String mergeRepeat(String[] s, int k) {
        StringBuilder result = new StringBuilder();

        for (String str : s) {
            for (int i = 0; i < k; i++) {
                result.append(str);
            }
        }

        return result.toString();
    }
}
```

A.2 Creative solutions

USER: Write a method named mergeRepeat(String[] s, int k) that takes an array of strings s and an integer k. The method returns a String which is the concatenation of all the elements of s repeated k times.

Examples:

mergeRepeat("A", "BC", 3) returns "AAABCBCBC"

mergeRepeat("hey", "you", "hi", 2) returns "heyheyyouyouhihi"

Come up with a creative solution that others might not think of.

ASSISTANT: Certainly! Here's a creative solution for the mergeRepeat method:

This solution calculates the total length of all the strings in the array and then uses a StringBuilder to efficiently concatenate the repeated strings by iterating through each string and appending it multiple times based on the given k. The additional part result.append(str, 0, Math.min(len, i + 1)); is used to append a portion of the string in each repetition, creating a unique concatenation pattern.

USER: Please make the program more creative

ASSISTANT: Certainly! Here's a more creative solution for the mergeRepeat method using a mix of string manipulation and randomization:

```java
public class MergeRepeatExample {

    public static void main(String[] args) {
        String[] arr1 = {"A", "BC"};
        String result1 = mergeRepeat(arr1, 3);
        System.out.println(result1);  // Output: AAABCBCBC

        String[] arr2 = {"hey", "you", "hi"};
        String result2 = mergeRepeat(arr2, 2);
        System.out.println(result2);  // Output: heyheyyouyouhihi
    }

    public static String mergeRepeat(String[] s, int k) {
        int totalLength = 0;
        for (String str : s) {
            totalLength += str.length();
        }

        StringBuilder result = new StringBuilder(totalLength * k);

        for (String str : s) {
            int len = str.length();
            for (int i = 0; i < k; i++) {
                result.append(str);
                result.append(str, 0, Math.min(len, i + 1));
            }
        }

        return result.toString();
    }
}
```

In this creative solution, a generateRandomSubstring method is introduced to generate a random substring from each string in the array. This adds an element of unpredictability and uniqueness to the repeated concatenation. The random substring is generated by selecting a random starting index within the length of the original string and extracting the substring from that point onward. This results in a more varied and creative output.

USER: Please provide a more creative solution - the most creative you can think of.

ASSISTANT: Certainly! Here's a more creative solution for the mergeRepeat method, introducing a playful twist by shuffling the characters within each string before concatenating them:

In this creative solution, the shuffleString method is introduced to shuffle the characters within each string before concatenating them. This adds a playful and unpredictable element to the output, making it more creative. The shuffleString method converts the string into a list of characters, shuffles the list, and then reconstructs the string with the shuffled characters.

```java
import java.util.Random;

public class CreativeMergeRepeatExample {

    public static void main(String[] args) {
        String[] arr1 = {"A", "BC"};
        String result1 = mergeRepeat(arr1, 3);
        System.out.println(result1);  // Output: AAABCBCBC

        String[] arr2 = {"hey", "you", "hi"};
        String result2 = mergeRepeat(arr2, 2);
        System.out.println(result2);  // Output: heyheyyouyouhihi
    }

    public static String mergeRepeat(String[] s, int k) {
        StringBuilder result = new StringBuilder();

        for (String str : s) {
            for (int i = 0; i < k; i++) {
                result.append(str);
                result.append(generateRandomSubstring(str));
            }
        }

        return result.toString();
    }

    private static String generateRandomSubstring(String str) {
        Random random = new Random();
        int len = str.length();
        int startIndex = random.nextInt(len + 1);

        // Generate a random substring from the original string
        return str.substring(startIndex);
    }
}
```

```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class MostCreativeMergeRepeatExample {

    public static void main(String[] args) {
        String[] arr1 = {"A", "BC"};
        String result1 = mergeRepeat(arr1, 3);
        System.out.println(result1);  // Output: AAABCBCBC (shuffled)

        String[] arr2 = {"hey", "you", "hi"};
        String result2 = mergeRepeat(arr2, 2);
        System.out.println(result2);  // Output: heyheyyouyouhihi (shuf
    }

    public static String mergeRepeat(String[] s, int k) {
        StringBuilder result = new StringBuilder();

        for (String str : s) {
            for (int i = 0; i < k; i++) {
                result.append(shuffleString(str));
            }
        }

        return result.toString();
    }

    private static String shuffleString(String str) {
        List<String> characters = Arrays.asList(str.split(""));
        Collections.shuffle(characters);
        StringBuilder shuffledString = new StringBuilder();
        characters.forEach(shuffledString::append);
        return shuffledString.toString();
    }
}
```

Appendix B

Repeated Solutions Experiment Problem

Intro Statement

Thank you for your participation in the study. You will be asked to complete a programming question in Java. Your goal is to come up with as many distinct solutions to the question as possible. All the solutions will be tested for accuracy, and you can test your code as you go. If you're spending too much time on one approach and it still doesn't work, please submit it anyway and try another. We have included a template for your code with some test cases. Please do not add "helper methods". Write your entire solution inside the provided method. There is a time limit of 2 hours, but you may finish early if you cannot think of any other solutions.

Problem

Given an n x n binary matrix image, flip the image horizontally, then invert it, and return the resulting image.

To flip an image horizontally means that each row of the image is reversed.

For example, flipping [1,1,0] horizontally results in [0,1,1]. To invert an image means that each 0 is replaced by 1, and each 1 is replaced by 0.

For example, inverting [0,1,1] results in [1,0,0].

Example 1:

Input: image = [[1,1,0],[1,0,1],[0,0,0]] Output: [[1,0,0],[0,1,0],[1,1,1]] Explanation: First reverse each row: [[0,1,1],[1,0,1],[0,0,0]]. Then, invert the image: [[1,0,0],[0,1,0],[1,1,1]]

Example 2:

Input: image = [[1,1,0,0],[1,0,0,1],[0,1,1,1],[1,0,1,0]] Output: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]] Explanation: First reverse each row: [[0,0,1,1],[1,0,0,1],[1,1,1,0],[0,1,0,1]]. Then invert the image: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]

Constraints:

n == image.length n == image[i].length 1 ¡= n ¡= 20 images[i][j] is either 0 or 1

Starter Code

```
public class FlipInvert {
public static int[][] flipAndInvertImage(int[][] image) {
}
}
```

Exit Statement

Thank you for participating in the study. If you asked to receive results in the survey, you will receive aggregated results before the end of the semester.

Bibliography

[1] OpenAI. (2024) ChatGPT (GPT 3.5 version) [Large Language Model] http://chat.openai.com.

[2] Leetcode. (2024) Flipping an Image. https://leetcode.com/problems/flipping-an-image/description/.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL): 40:1–40:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290353. URL `http://doi.acm.org/10.1145/3290353`.

[4] Rotem Israel-Fishelson Andoni Eguíluz Pablo Garaizar Arnon Hershkovitz, Raquel Sitman and Mariluz Guenaga. Creativity in the acquisition of computational thinking. *Interactive Learning Environments*, 27(5-6):628–644, 2019. doi: 10.1080/10494820.2019.1610451. URL `https://doi.org/10.1080/10494820.2019.1610451`.

[5] Sameer Badaskar, Sachin Agarwal, and Shilpa Arora. Identifying real or fake articles: Towards better language modeling. In *International Joint Conference on Natural Language Processing*, 2008. URL `https://api.semanticscholar.org/CorpusID:4324753`.

[6] Elijah Chou and Davide Fossati. Measuring creativity in computer programming:

A code distance approach. *Emory Theses and Dissertations*, 2023. URL `https://etd.library.emory.edu/concern/etds/b8515p78f?locale=en`.

[7] Philip M. Clark and Herbert L. Mirels. Fluency as a pervasive element in the measurement of creativity. *Journal of Educational Meaurement*, 7(2):83–86, 1970.

[8] Thomas Dohmke. Github copilot x: The ai-powered developer experience. *The Github Blog*, 2023.

[9] Ekohariadi, Yeni Anistyasari, and Ricky Eka Putra. Rasch and classical test theory validation of automated assessment tool for measuring students' creativity in computer programming. In *Proceedings of the International Joint Conference on Arts and Humanities (IJCAH 2020)*, pages 1155–1160. Atlantis Press, 2020. ISBN 978-94-6239-285-4. doi: 10.2991/assehr.k.201201.193. URL `https://doi.org/10.2991/assehr.k.201201.193`.

[10] Rotem Israel-Fishelson, Arnon Hershkovitz, Andoni Eguíluz, Pablo Garaizar, and Mariluz Guenaga. A log-based analysis of the associations between creativity and computational thinking. *Journal of Educational Computing Research*, 59 (5):926–959, 2021. doi: 10.1177/0735633120973429. URL `https://doi.org/10.1177/0735633120973429`.

[11] Jaehyun Kim and Yangsun Lee. A study on abstract syntax tree for development of a javascript compiler. *International Journal of Grid and Distributed Computing*, 11(6):37–48, 2018. doi: 10.14257/ijgdc.2018.11.6.04.

[12] Anastasia Kovalkov, Benjamin Paaßen, Avi Segal, Niels Pinkwart, and Kobi Gal. Automatic creativity measurement in scratch programs across modalities. *IEEE Transactions on Learning Technologies*, 14(6):740–753, 2021. doi: 10.1109/TLT.2022.3144442.

[13] Anastasia Kovalkov, Benjamin Paaßen, Avi Segal, Niels Pinkwart, and Kobi Gal. Modeling creativity in visual programming: From theory to practice. In *Proceedings of The 14th International Conference on Educational Data Mining, EDM 2021, virtual, June 29 - July 2, 2021.* International Educational Data Mining Society, 2021.

[14] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. *Proceedings of the 31st International Conference on Machine Learning*, 2014. URL `http://jmlr.org/proceedings/papers/v32/le14.pdf`.

[15] G. F Madaus. Divergent thinking and intelligence: another look at a controversial question. *Journal of Educational Measurement*, 4:227–235, 1967.

[16] Sven Manske and H. Ulrich Hoppe. Automated indicators to assess the creativity of solutions to programming exercises. pages 497–501, 2014. doi: 10.1109/ICALT.2014.147.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR abs/1301.3781*, 2013. URL `http://arxiv.org/abs/1301.3781`.

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

[19] Plato. *Republic*. Ware, England : Wordsworth Editions, 1997.

[20] E. P. Torrance and H. W. Michie. *Explorations in creative thinking in the early school years: I. Scoring manual for "How good is your imagination?" (Form C)*. Bureau of Educational Research, University of Minnesota, 1959.

[21] Kaoru Yamamoto. *Experimental Scoring Manuals for Minnesota Tests of Creative Thinking and Writing*. Bureau of Educational Research, Kent State University, 1964.

[22] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18:1245–1262, 1989. doi: 10.1137/0218082. URL `https://doi.org/10.1137/0218082`.