Grading and Clustering Student Programs That Produce Probabilistic Output

Yunsung Kim*
Stanford University
yunsung@stanford.edu

Jadon Geathers*
Stanford University
geathers@stanford.edu

Chris Piech
Stanford University
cpiech@stanford.edu

ABSTRACT

Stochastic programs, which are programs that produce probabilistic output, are a pivotal paradigm in various areas of CS education from introductory programming to machine learning and data science. Despite their importance, the problem of automatically grading such programs remains surprisingly unexplored. In this paper, we formalize the problem of assessing stochastic programs and develop an open-source assessment framework called StochasticGrade. Based on hypothesis testing, StochasticGrade offers an exponential speedup over standard two-sample hypothesis tests in identifying incorrect programs, enables control over the rate of grading errors, and allows users to select the measure of proximity to the solution that is most appropriate for the assignment. Moreover, the features calculated by StochasticGrade can be used for fast and accurate clustering of student programs by error type. We demonstrate the accuracy and efficiency of StochasticGrade using student data collected from 4 assignments in an introductory programming course and conclude with practical guidelines for practitioners who hope to use our framework.

Keywords

Auto-Grading, Automated Assessment, Hypothesis Testing, Computer Science Education, Programming Education, Probability and Computing Education, Stochastic Programs

1. INTRODUCTION

Stochastic Programs, which are programs that generate probabilistic output, have been a major driving force behind modern computational advancements. They are fundamental to several areas within computer science, such as machine learning, randomized algorithms, cryptography, and the study of large-scale systems. As a result, fluency at the intersection of probability and computing has become a core skill for computer science students [4], leading educators to actively incorporate probability theory into early programming curricula [31]. The CS education research community has also recently begun to acknowledge "probabilistic and statistical methods for computing" as an important paradigm of programming that "should be embraced by future computing education" [10, 33].

In an area of such widespread educational demand, *automated assessment* presents a critical step towards massively expanding the opportunities of learning and practice by allowing efficient and objective evaluation of student work [2, 25]. However, the inherent non-determinism of stochastic programs poses several important challenges for grading. While the EDM research community has developed many unit testers and automated assessment tools for deterministic programs, there are no published studies on grading *stochastic* student programs to the best of our knowledge. Although fixing the pseudo-random number generator seed can make stochastic programs behave deterministically, even slight differences in implementation can cause semantically identical programs to behave completely differently under the same seed. This makes existing auto-grading systems inapplicable to grading stochastic programs, requiring a framework that can systematically account for randomness in the grading process.

In this paper, we formally pose the problem of grading stochastic programs and present an open-source[1], sampling-based assessment framework for grading and clustering. Our proposed method, StochasticGrade, is based on the principles of hypothesis testing and has four key features: (1) it achieves exponential speedup over standard hypothesis testing in identifying error programs, (2) provides a controllable rate of misgrades, (3) allows users to choose the measure of proximity to the solution that best fits their assignment, and (4) efficiently handles programs with multidimensional output. Moreover, the features calculated by StochasticGrade allow simple and effective clustering of student programs by error type. **We make the following contributions**:

- **Problem of Grading Stochastic Programs.** We formally pose the problem of grading stochastic programs and discuss its challenges.
- **Efficient and Accurate Grading with an Adjustable Error Rate.** StochasticGrade can adaptively adjust the number of samples obtained from student programs based on the complexity of the error. It also

*Equal Contribution.

Y. Kim, J. Geathers, and C. Piech. Grading and clustering student programs that produce probabilistic output. In B. Paaßen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 40–52, Atlanta, Georgia, USA, July 2024. International Educational Data Mining Society.

https://doi.org/10.5281/zenodo.12729772

[1] https://github.com/yunsungkim0908/stochasticgrade

allows users to set the desired rate of misgrades and choose a suitable "disparity function," which measures the dissimilarity between two samples throughout the grading process. We showcase several disparity functions and analyze their relative utilities.

- **Clustering Programs by Error Type.** We demonstrate how StochasticGrade's disparity measurements can be used to identify clusters of student programs with identical errors and provide 3 concrete ways to improve the quality of clustering.
- **Evaluation on Real Classroom Assignments.** Based on student-submitted responses to 4 undergraduate-level programming assignments, we empirically validate the accuracy and efficiency of our framework and conduct a comparative analysis of the disparity functions.
- **Practical Guidelines for Practitioners.** We provide a high-level summary and practical guidelines for practitioners hoping to use our open-source framework.

1.1 Related Works

Automated Assessment of Student Programs. Tools for automatically assessing programs have been in use for "as long as educators have asked students to build their own software" [2]. Research on automated assessment methods has recognized various paradigms of programming assignments such as visual [7, 23, 40], web-based [28, 35], distributed [21], and mobile [20, 36] programming, and has also developed tools for specialized programming domains such as formal language and logic [34, 3], assembly language [18], and database queries [17, 26]. Our work expands this discourse by discussing the paradigm of stochastic programs, which serves a vital role in various domains of computer science.

In much of existing automated assessment research, the notion of the "correctness" of student programs has mainly been understood to imply "providing an exactly right answer whenever it is run with valid input" [10, Chapter 20]. Several studies have also developed ways to avoid evaluating the outputs directly by assessing alternative forms of program representations such as grading based on visual output [40] or abstract vector representations of code [22, 39]. Recent progress in the zero/few-shot learning capabilities of large language models has also been leveraged for grading based on source code [15, 24], but studies [15, 30] have demonstrated that this approach may suffer from hallucination, certifiability, reproducibility, and high cost. While none of these methods allows control over the rate of grading errors, we provide a method for grading stochastic programs based on samples from the student programs with an adjustable grading error rate.

Stochastic Programs in Computing Education. The growing need for teaching probability theory in the specific context of computer science has increasingly been recognized [33]. For instance, [4] notes probability and statistics as "the most connected mathematical topic[s]" to computer science. Several early-level courses have also been designed to amalgamate probabilistic thinking with the ideas of computation and programming [31, 11, 29].

```python
def program_1():
  if (random.random() < 0.5):
    return random.gauss(0,1)
  else:
    return random.gauss(1,2)
```

```python
def program_2():
  if (np.random.uniform() < 0.5):
    return np.random.normal(0,1)
  else:
    return np.random.normal(1,2)
```

```python
def program_3():
  s0 = np.random.normal(0,1)
  s1 = np.random.normal(1,2)
  if (np.random.uniform() < 0.5):
    return s0
  else:
    return s1
```

```python
def program_4():
  s1 = np.random.normal(1,2)
  s0 = np.random.normal(0,1)
  if (np.random.uniform() < 0.5):
    return s0
  else:
    return s1
```

```python
def program_5():
  sample = np.random.randn()
  if (np.random.uniform() < 0.5):
    sample = sample*2 + 1
  return sample
```

```python
def program_6():
  s0 = np.random.normal(0,1)
  if (np.random.uniform() < 0.5):
    return s0
  return np.random.normal(1,2)
```

Figure 1: Fixing the pseudorandom number generator (PRNG) seed is insufficient for grading programs that use randomness. The programs shown above implement the same distribution but behave differently even with the same PRNG seed. (See Section 1.2)

Studies also suggest stochastic programs as an effective tool for developing probabilistic thinking. Seymour Papert in [27], for instance, noted the affordances of programming in LOGO language in helping young students understand the concept of probability. More recently, [1, 8] have used stochastic programs to develop tools to support children learn probability. Similarly, [6, 37] recognize the value of probabilistic programming for understanding the mechanics of probability and probabilistic modeling, and several courses on probabilistic modeling (in MIT and Stanford [12, 37], for instance) adopt this programming paradigm.

1.2 Grading Stochastic Programs

In this section, we formally define the problem of grading stochastic programs and explain its challenges. Suppose that a student is assigned the task of implementing a program that generates a randomized output, and this output can be translated into a number or an array of numbers. Given access to the correct solution program $\mathcal{C}$, the task of automated grading is to determine whether the student-submitted program $\mathcal{P}$ implements a probability distribution that is identical to $\mathcal{C}$ for all possible inputs to the program. The grader is allowed to make calls to $\mathcal{P}$ and observe its outputs.

As noted earlier, the use of randomness poses a unique challenge for this task that cannot be addressed using methods for grading deterministic programs. In particular, one may expect to be able to grade by fixing the seed that initializes the pseudorandom number generator, which can make student programs behave deterministically. However, this introduces systematic errors in grading that become apparent even for simple assignment tasks.

For instance, consider the simple task of implementing a program that draws samples from one of two distributions — Gaussian $\mathcal{N}(0,1)$ and $\mathcal{N}(1,2)$ — with equal probability. Figure 1 lists several correct implementations in Python, each with slight variations in the packages used, the order in which random functions are called, and the procedural logic. These variations cause each program to behave differently even when the PRNG seed is fixed to the same value. Considering that students often provide a surprisingly diverse

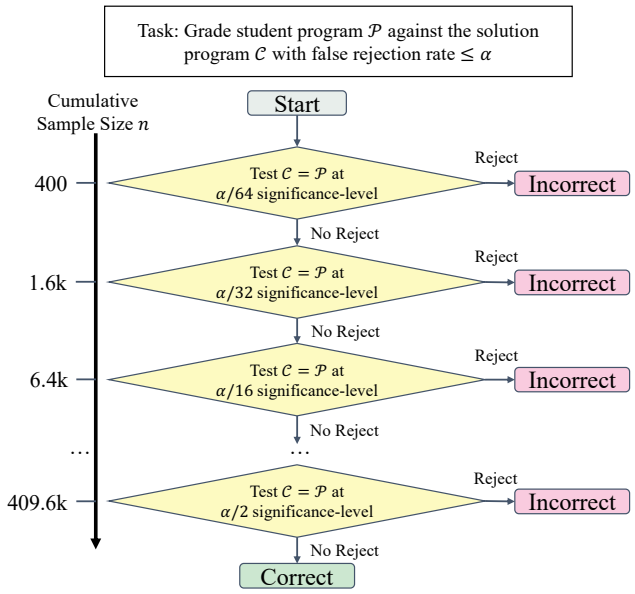


Figure 2: StochasticGrade applies a series of hypothesis tests on a cumulative set of samples obtained from student programs. The significance level at each step is adjusted to allow less elusive errors to be caught early while controlling the overall false rejection rate. In Section 5, we also show that the statistics calculated by StochasticGrade enable fast and accurate clustering of programs by error type.

set of responses even to relatively simple open-ended assignments [16], this illustrates the need for a grading framework that inherently embraces randomness in the grading process.

2. STOCHASTICGRADE FRAMEWORK

We now present StochasticGrade (Figure 2, Algorithm 1), our novel sampling-based assessment framework. Assessing programs with probabilistic output is an application of hypothesis testing at its core, as it examines the identity of the student program with the solution program by examining their samples. Unlike standard two-sample hypothesis tests that use a fixed sample size, however, StochasticGrade allows the required number of samples to vary adaptively with the student programs being graded.

This adaptive sampling is achieved by applying a *series* of two-sample hypothesis tests on an incremental, cumulative set of samples, with controlled significance levels. This multi-step process allows student programs with large discrepancies from the solution to be detected early while maintaining the overall false rejection rate (FRR) below the desired threshold.

Formally, StochasticGrade (Algorithm 1) assumes access to samples $X_{\mathcal{C}}$ of size $N_{\mathcal{C}}$ (set to a large value) obtained a priori from the solution program $\mathcal{C}$. At the time of grading, it takes the following inputs from the user: the student program $\mathcal{P}$ to be evaluated, the disparity function f, the minimum and maximum sample sizes $N_{\min}$ and $N_{\max}$, the sample growth factor R, and the desired false rejection rate (FRR) α, which is the probability of misgrading a correct program. Upon completion, StochasticGrade predicts the correctness of $\mathcal{P}$ with a desired FRR of at most α.

Algorithm 1: StochasticGrade($\mathcal{P}, f, N_{\min}, N_{\max}, R, \alpha$)

Data: Samples $X_{\mathcal{C}}$ drawn a priori from solution program $\mathcal{C}$
Input: Student program $\mathcal{P}$, Disparity function f, Minimum/Maximum sample sizes ($N_{\min}, N_{\max}$), Sample growth factor R , False rejection rate (FRR) α
Result: Grading $\mathcal{P}$ with false acceptance rate $\leq \alpha$

1 $X_{\mathcal{P}} \leftarrow []$ ▷ `For storing student program samples`
2 $L \leftarrow \log_R(N_{\max}/N_{\min}) + 1$ ▷ `Total number of steps`
3 **for** $i = 0, ..., L-1$ **do**
4 **while** $X_{\mathcal{P}}$ has fewer than $N_i = R^i \cdot N_{\min}$ samples **do**
5 Call $\mathcal{P}$ and append output to $X_{\mathcal{P}}$
6 Set the false rejection rate at step i to $\alpha_i \leftarrow \alpha/2^{L-i}$
7 Calculate discrepancy measure $D \leftarrow f(X_{\mathcal{C}}, X_{\mathcal{P}})$
8 $\varepsilon \leftarrow$ CriticalValue(f, α_i, Size($X_{\mathcal{C}}$), Size($X_{\mathcal{P}}$))
9 **if** $D \geq \varepsilon$ **then** ▷ `Reject if too discrepant`
10 **return Incorrect**
11 **return Correct**

Grading proceeds in multiple iterations, where at each step, the sample size grows by a factor of R (which we fix to 4 in our experiments) until the maximum number of samples $N_{\max}$ is reached. This results in a total of L iterations, where

$$L = \log_R\left(\frac{N_{\max}}{N_{\min}}\right) + 1. \quad (1)$$

At each step, the algorithm tests whether the samples $X_{\mathcal{C}}$ from the solution program and $X_{\mathcal{P}}$ from the student program could have originated from the same distribution. This is done by calculating a disparity measurement D between $X_{\mathcal{C}}$ and $X_{\mathcal{P}}$ using the disparity function f (Line 7), which provides a measure of dissimilarity between a pair of samples. If at any step the measurement D exceeds the critical value ε, indicating that the discrepancy between $X_{\mathcal{P}}$ and $X_{\mathcal{C}}$ is unlikely to arise from random sampling, the algorithm marks $\mathcal{P}$ as incorrect. This allows discrepant student programs to be rejected early before examining many samples.

Selecting the Critical Value ε. At step i, the threshold for the disparity measurement at which $\mathcal{P}$ is marked incorrect should be chosen so that the probability of misgrading the solution program $\mathcal{C}$ is less than $\alpha_i = \alpha/2^{L-i}$:

$$P_{Y\sim\mathcal{C}}(f(X_{\mathcal{C}}, Y) \geq \varepsilon) < \alpha_i. \quad (2)$$

For most f, finding the exact value of ε for an arbitrary $X_{\mathcal{C}}$ can be intractable. However, in the limit as the size of $X_{\mathcal{C}}$ (denoted $N_{\mathcal{C}}$) approaches infinity, Inequality 2 becomes identical to

$$P_{X,Y\sim\mathcal{C}}(f(X, Y) \geq \varepsilon) < \alpha_i, \quad (3)$$

where X represents $N_{\mathcal{C}}$ i.i.d. samples drawn from $\mathcal{C}$. In this case, the critical value ε is the $(1-\alpha_i)$-quantile[2] of $f(X, Y)$ for X and Y obtained from $\mathcal{C}$, which is equivalent to a two-sample hypothesis test on X and Y using $D = f(X, Y)$ as the test statistic with significance level α_i.

[2]The p-th quantile of the distribution of Z is the value z for which $P(Z \leq z) = p$.

Algorithm 2: Critical Value Selection Algorithm CriticalValue($f, \alpha, N_{\mathcal{C}}, N_{\mathcal{P}}, M = 1000$)

Data: Solution program $\mathcal{C}$, samples $X_{\mathcal{C}}$ drawn from $\mathcal{C}$
Input: Disparity function f, False rejection rate α, Solution sample size $N_{\mathcal{C}}$, Student program sample size $N_{\mathcal{P}}$, # of Monte Carlo iterations M
Output: Critical value ε for FRR α

1 **if** $f(X, Y)$ has a known distribution for $X, Y \sim \mathcal{C}$ with sizes $N_{\mathcal{C}}$ and $N_{\mathcal{P}}$ **then**
2 **return** The $(1-\alpha)$-quantile of $f(X, Y)$
3 Obtain M samples of $f(X_{\mathcal{C}}, Y)$ where Y is a sample from $\mathcal{C}$ of size $N_{\mathcal{P}}$
4 **return** The $(1-\alpha)$-quantile of the sampled $f(X_{\mathcal{C}}, Y)$'s

The CriticalValue subroutine (Algorithm 2) embodies this insight. If f is the statistic of a well-known hypothesis test (for instance, the Anderson-Darling statistic for distribution identity testing [32], or the Student's t-statistic for testing the identity of means), we set ε to be the critical value of the corresponding two-sample test. Otherwise, CriticalValue calculates a Monte Carlo estimate by simulating $Y \sim \mathcal{C}$ M times (line 3) and calculating the $(1-\alpha_i)$-quantile of $f(X_{\mathcal{C}}, Y)$ from these simulated samples (line 4). In practice, the estimated ε may be cached to save future runtime.

2.1 The Batch Hypothesis Testing Baseline

StochasticGrade is best understood in comparison with a standard two-sample hypothesis test. Each decision block in Figure 2 (corresponding to steps 6 through 9 in Algorithm 1) is comparable to a two-sample test of the null hypothesis $H_0 : \mathcal{P} = \mathcal{C}$ against the alternative hypothesis $H_a : \mathcal{P} \neq \mathcal{C}$ with significance level $\alpha_i = \alpha/2^{L-i}$, using the disparity measurement $D = f(X_{\mathcal{C}}, X_{\mathcal{P}})$ as the test statistic. Each test examines $N_{\mathcal{C}}$ and $N_i = N_{\min} \cdot R^i$ samples each from the solution and student programs, and these tests are conducted in series on a cumulative set of student program samples that grows each step by a factor of R.

Later in Section 4, we will compare this approach to the simple and reasonable baseline of conducting a single two-sample hypothesis test. This baseline approach, which we call *Batch Hypothesis Testing*, will use the same statistic as StochasticGrade on the full batch of student program samples with maximum sample size $N_{\max}$ and significance level α for a fair comparison against StochasticGrade.[3] Batch Hypothesis Testing will achieve a false rejection rate of α by design and is, to the best of our knowledge, the only comparable baseline against StochasticGrade with an adjustable FRR.

2.2 Controllable Rate of Misgrades

The step-wise FRR α_i starts with a small value $\alpha/2^L$, so with just a few samples from $\mathcal{P}$, StochasticGrade only marks largely discrepant samples $X_{\mathcal{P}}$ as incorrect. As more samples are collected from $X_{\mathcal{P}}$, α_i increases by a factor of 2 each step, and the algorithm rejects more proactively.

As the size of $X_{\mathcal{C}}$ becomes infinitely large, this choice for α_i guarantees that the probability of misgrading a correct program at step i is at most $\alpha_i = \alpha/2^{L-i}$ by the choice of the critical value. Consequently by the union bound[4], the probability that a correct program is marked incorrect at *any* step is at most the sum of the false rejection probabilities at each step, which is $\leq \alpha/2^L + \alpha/2^{L-1} + \ldots + \alpha/2 \leq \alpha$. Therefore, the overall probability of rejecting a correct program is bounded by α.

[3]This is equivalent to StochasticGrade with $L = 1$ step with FRR set to 2α.

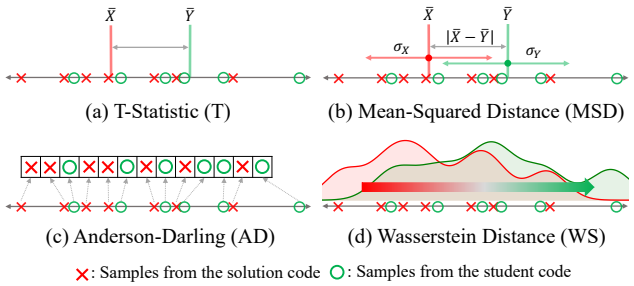


Figure 3: Intuition behind each of the disparity functions presented in Section 2.3. (a) T detects differences in means. (b) MSD measures the difference in mean and variance. (c) AD measures how well the samples "mix-in" when ordered altogether. (d) WS measures how much "work" it takes to transform one distribution to the other.

These theoretical results hold exactly in the limit as the number of samples in $X_{\mathcal{C}}$ approaches infinity. In practice, StochasticGrade samples a large enough $X_{\mathcal{C}}$ for which the empirical distribution of the samples quickly converges to the true distribution and the probability of obtaining a non-representative sample gets exponentially small with the sample size (known as the Dvoretzky–Kiefer–Wolfowitz Inequality [9]). Later in Section 4, we demonstrate empirically that this achieves the desired false rejection rate in the real-world assignments we analyzed, even with Monte Carlo estimation.

Controlling the False Acceptance Rate. To prevent student mistakes from going unnoticed, it is important to ensure that incorrect programs are not misgraded as correct. For any disparity function that converges to 0 with infinitely many samples only for a pair of identical programs,[5] any desired false acceptance rate (FAR) can be achieved for a given error program with a large enough maximum sample size $N_{\max}$ in the asymptotic limit of $N_{\mathcal{C}} \to \infty$. To choose the $N_{\max}$ that ensures a FAR of δ for all feasible errors in the limit, it suffices to consider the error $\mathcal{E}$ that is most "similar" to the solution program and choose the $N_{\max}$ for which StochasticGrade (Algorithm 1) with a given FAR parameter misgrades $\mathcal{E}$ at most δ fraction of time. Similar to estimating the critical value, the probability of error for each sample size can be estimated using Monte Carlo estimation, and the smallest possible $N_{\max}$ can be determined via exponential search.

[4]The union bound states that $P(\bigcup_{i=1}^{n} E_i) \leq \sum_{i=1}^{n} P(E_i)$.
[5]T and MSD in Section 2.3 do not, for instance, as they approach 0 for any pair of programs with identical mean and variance.

2.3 Selecting the Disparity Function

As mentioned earlier, the user can choose to use any disparity function f that is most appropriate for the assignment task. Following is the list of 4 disparity functions we studied in this work along with their descriptions and characteristics.[6] We will empirically validate these properties in Section 4. ($\overline{X}$ and σ^2 denote sample mean and variance.)

T-Statistic (T): Calculates the absolute unpaired t-statistic $f(X,Y) = \frac{|\overline{X}-\overline{Y}|}{\sqrt{\sigma_X^2+\sigma_Y^2}}$. It is useful for detecting differences in means for most errors but has low accuracy on errors with similar means.

Mean Squared Distance (MSD) Calculates $f(X,Y) = |\sigma_X^2 - \sigma_Y^2| + (\overline{X}-\overline{Y})^2$ and measures the difference in spread of the student program samples around the mean of the solution samples. It detects large differences in the overall shape of the distribution but yields low accuracy for more subtle errors.

Anderson-Darling (AD): The Anderson-Darling statistic measures how well two samples "mix-in" when ordered within the combined sample. It is sensitive to the subtle differences in probability for dense regions of the distribution, but as it depends on the rank ordering of samples, it is less sensitive to differences in sparse or extreme regions.

Wasserstein Distance (WS): The Wasserstein distance has the intuitive meaning of the minimal cost of transforming the student distribution into the solution distribution by "moving" probability masses. WS effectively detects small differences in the extreme values of the distribution, but it is less sensitive to very subtle differences in the dense regions of the distribution compared to AD.

Figure 3 graphically illustrates the intuition behind each of the disparity functions. Although T and MSD may be less accurate than AD and WS in assignments with subtle errors, one of their major appeals is their dependence only on the *summary statistics* of the samples rather than the whole sample. This can be particularly useful when no subtle errors are anticipated and one prefers not to process all solution samples.[7]

2.4 Handling Multi-Dimensional Outputs

The grading mechanism described thus far works for programs with a single output. The problem of grading programs with *multi-dimensional* output introduces new, unique technical challenges due to the complex statistical dependencies that may be present across dimensions. This makes naive adaptations of univariate grading such as calculating disparity measurements separately for each dimension ineffective, as they systematically ignore the correlation across dimensions that may be critical for grading.

[6] T and AD use a known function to calculate the critical values, whereas MSD and WS use Monte Carlo estimation with $M = 1$k.

[7] In our experiment we used 500k solution samples, which takes up only about 3.8MB of storage. This is comparable to the size of a photo taken with a smartphone.

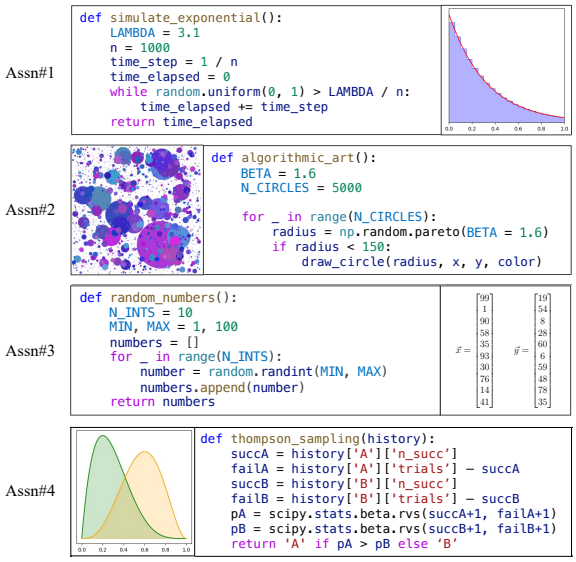


Figure 4: Illustration of the assignments in our dataset along with their solution code.

Instead we seek a simple extension of the univariate grading method through "random projections" [19], which has found many use cases in approximate multivariate hypothesis testing. The idea is to map high-dimensional samples $X_\mathcal{C}$ and $X_\mathcal{P}$ to single-dimensional samples $X_\mathcal{C}^p$ and $X_\mathcal{P}^p$ using a real-valued projection that is chosen at random. Doing so effectively casts a multivariate probability distribution into a univariate one and allows the univariate disparity functions in Section 2.3 to be used on the projected samples $X_\mathcal{C}^p$ and $X_\mathcal{P}^p$ in Algorithm 1. Moreover, the FRR α still applies after this process. We will study the following 2 types of random projections:

Orthogonal Projection [19]. A unit vector u is initially sampled uniformly at random, and each sample X_i in X is orthogonally projected onto u: $X_i^p = u^T X_i$.

Euclidean Distance. A reference point r is sampled uniformly from a bounding box around solution program samples $X_\mathcal{C}$, dilated by a factor of 2 on each side to ensure distance to points in $X_\mathcal{C}$. The Euclidean distance is calculated from r to each sample: $X_i^p = \|X_i - r\|_2$.

It is worth noting that projections are not injective, meaning that different multivariate distributions could in theory get mapped to similar univariate distributions. Yet, we expect that errors programmatically implemented by students are highly unlikely to collide with the solution under a mapping chosen at random. For Orthogonal Projection, this is supported by the fact that random linear projections preserve distance between points with high probability [38], and we also anticipate that the spherical geometry of a random Euclidean Distance projection would be unnatural to arise in student programs. While one could use multiple random projections to decrease the chances of collision, we found 1 projection to be enough in our experiments.

3. EVALUATION SETUP

3.1 Dataset

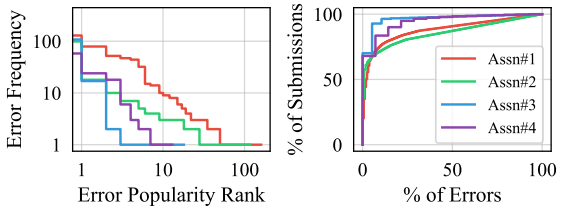
To empirically validate the performance of StochasticGrade, we curated student-written programs from 4 probabilistic programming assignment tasks (Figure 4) in an introductory computer science course at Stanford University. These programs were collected through the course's online homework system, from which we took both the final student submissions and the intermediate progress logs saved every 5 minutes. From them we randomly selected a subset of the programs that compiled and produced probabilistic output. A trained teaching assistant went through each program and marked its correctness as well as its specific error type. Figure 5 shows the statistics of the data we used for our experiments. Each of the 4 assignments exhibits unique characteristics that enable us to analyze various aspects of our grading mechanism. We now describe these assignments in greater detail.

(Assn#1) Simulate Exponential: Subtle Translations in Distribution. Students simulate a sample from the exponential distribution $\text{Exp}(\lambda = 3.1)$, using a discretized loop with incremental (1/1000) steps and making calls to a Bernoulli coin flip at each iteration. The loop structure, combined with small step sizes, lends itself to subtle errors that arise due to an incorrect starting condition, which results in distributions characterized by very small shifts from the correct distribution.

(Assn#2) Algorithmic Art: Infrequent but Large Observations. Students create an artistic collage by drawing random circles with random colors. The circles' radii R are drawn from a Pareto distribution with shape parameter $\beta = 1.6$, but values of $R > 150$ are discarded. It is worth noting that the probability of $R > 150$ is small (≈ 0.0003), but since the Pareto distribution is "heavy-tailed," programs that do not implement curtailment exactly at 150 can occasionally generate extremely large output values. Due to the exploratory nature of this problem, other subtle errors also arise from students modifying the shape parameter β.

(Assn#3) Random Numbers: Multivariate Output. Students implement a program to sample 10 random integers between 1 and 100 uniformly at random. Each run of the program consequently produces a list of 10 integers, unlike Assn#1, 2, or 4 which output a single value. In addition to subtle shifts and scaling errors, the multivariate nature of the assignment occasionally results in errors with incorrect statistical dependence across dimensions.

(Assn#4) Thompson Sampling: Binary Outputs and Functional Arguments. Students write a function that implements a single step of a binary Thompson Sampling algorithm, which chooses one option over the other based on a random estimate of the rewards for each option. The distribution of the reward estimate is determined by the "history" argument that indicates previously observed rewards. A unique feature of this assignment is that the programs require a functional argument (history), which yields a different output distribution depending on the values of the argument.



	Assn#1	Assn#2	Assn#3	Assn#4
# Labeled Programs	865	457	475	372
# Unique Errors	161	118	19	14
% Correct	5.43	21.9	70.1	15.9
500k Soln. Sampling Time (s)	41.9	7.2	3.7	33.5

Figure 5: (Top) Rank-frequency plot (Left) and CDF (Right) of student errors in each assignment dataset. (Bottom) Statistics of the assignment dataset.

3.2 Experiment Details

Grading Experiment. For each problem, 500k samples were obtained a priori from the solution program to be used as $X_{\mathcal{C}}$. For Assn#1, 2, and 3, we ran StochasticGrade with each of the 4 disparity functions in Section 2.3. The minimum and maximum sample sizes were set to 400 and 409.6k with sample growth factor $R = 4$, resulting in $L = 6$ steps. Since programs in Assn#4 additionally expect a functional argument, we first randomly chose the argument to be passed into the student program during sampling and ran StochasticGrade with the T disparity function. We set the minimum and maximum sample sizes to 50 and 3,200 each, and the sample growth factor was also set to $R = 4$.

Clustering Experiment. For the clustering experiment, we used student responses to Assn#1, 3, and 4.[8] For all problems, we used the same 500k solution samples from the grading experiment. For Assn#1 and 3, samples of varying sizes from 400 to 409.6k were obtained for each student program, and high-dimensional samples in Assn#3 were additionally processed using Euclidean Distance projection. For Assn#4 we varied the student sample sizes from 50 to 3,200 and used an input argument that was chosen at random, similar to the grading experiment. For every student program, we calculated the disparity between samples from the student program and samples from the solution program and ran hierarchical agglomerative clustering using these measurements as an input feature.[9] This process was repeated for each of the 4 disparity functions from Section 2.3 except for Assn#4 for which we only used T. The quality of the clusters was measured using Adjusted Rand Index (ARI) and Normalized Mutual Information (NMI).[10]

[8] Assn#2 was not used because an overwhelming number of student programs were unique and didn't form enough clusters to be informative.

[9] We chose the best performing number of clusters from [25, 50, 75, 100, 125, 150].

[10] ARI considers the number of pairs assigned to the same or different clusters in the true predicted clusters. NMI mea-

Table 1: Overall accuracy on grading correct programs for StochasticGrade measured over 10,000 independent trials. Disparity functions with (*) used Monte Carlo estimates to obtain the critical values.

	Assn#1				Assn#2				Assn#3				Assn#4
α	T	MSD*	AD	WS*	T	MSD*	AD	WS*	T	MSD*	AD	WS*	T
0.003	.998	.996	.998	.999	.999	.998	.999	.995	1.0	.996	.998	.997	.998
0.01	.996	.991	.996	.998	.995	.996	.998	.986	.997	.989	.996	.991	.992
0.05	.972	.966	.978	.994	.981	.974	.988	.938	.981	.938	.981	.957	.960
0.1	.939	.913	.966	.983	.951	.962	.975	.883	.956	.893	.966	.912	.902

Table 2: Overall accuracy on grading incorrect programs for StochasticGrade (SG) and Batch Hypothesis Testing (HT).

		Assn#1				Assn#2				Assn#3				Assn#4
α		T	MSD	AD	WS	T	MSD	AD	WS	T	MSD	AD	WS	T
0.003	SG	.833	.819	.999	.841	.919	.989	.510	.986	1.0	.993	1.0	1.0	.987
	HT	.833	.818	1.0	.839	.913	.983	.507	.989	1.0	.993	1.0	1.0	.978
0.01	SG	.835	.818	1.0	.840	.936	.992	.515	.986	1.0	.993	1.0	1.0	.987
	HT	.842	.818	1.0	.840	.916	.983	.510	.986	1.0	.993	1.0	1.0	.981
0.05	SG	.850	.823	1.0	.846	.947	.992	.518	.992	1.0	.993	1.0	1.0	.997
	HT	.856	.825	1.0	.841	.936	.986	.518	.992	1.0	.993	1.0	1.0	.990
0.1	SG	.858	.835	1.0	.845	.955	.992	.538	.992	1.0	.993	1.0	1.0	.997
	HT	.868	.837	1.0	.844	.961	.989	.521	.992	1.0	.993	1.0	1.0	.997

4. GRADING RESULTS

In this section, we demonstrate that StochasticGrade achieves the desired False Rejection Rate (FRR) and exhibits exponential speedup along with nearly identical grading accuracy compared to standard hypothesis testing.

4.1 Performance Analysis

Accuracy on Correct Programs. We begin by analyzing the overall False Rejection Rate (FRR) of StochasticGrade. Recall that the hyperparameter α is the desired bound on the rate of misgrading *correct* programs (Section 2.) Table 1 reports the accuracy of StochasticGrade on grading correct programs for various values of α, measured over 10,000 independent trials. For all disparity functions that do not use Monte Carlo estimates, the error rates all lie within – and are often substantially smaller than – the desired FRR α. Also, for disparity functions that rely on Monte Carlo estimates, the error rates are within a tolerable margin (at most 0.017 away) from α.

Accuracy on Incorrect Programs. Next we turn to the power of StochasticGrade in accurately recognizing an incorrect program. Table 2 shows the grading accuracy on incorrect programs for StochasticGrade and Batch Hypothesis Testing (Section 2.1) using the same test statistic and significance level α. For all problems, StochasticGrade achieves perfect or near-perfect accuracy with at least one disparity function. The performance difference compared to Batch Hypothesis Testing is negligible, and StochasticGrade even slightly outperforms Batch Hypothesis Testing by up to 2.0% on Assn#2.

Note that the high-performing disparity functions differ across problems. For Assn#1, where the popular error programs result in subtle shifts in the distribution, AD achieves perfect accuracy whereas other disparity functions perform noticeably worse. On the other hand, AD performs poorly on Assn#2 where many errors result in rare occurrences of extreme values. We will provide a more in-depth analysis of the performance of each disparity function on different error types in Section 4.2.

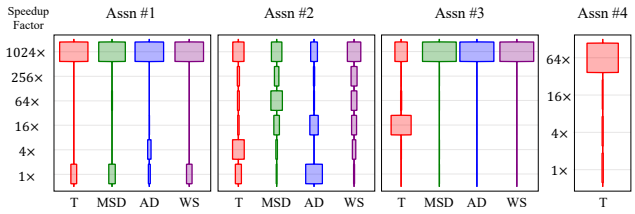


Figure 6: Grading speed of StochasticGrade relative to Batch Hypothesis Testing on all incorrect programs. Across all disparity functions and assignments, StochasticGrade is exponentially faster than HT on most programs, by up to a factor of 1,024 in many cases. (Note the log scale in the y-axis.)

Exponential Speedup. The advantage of StochasticGrade over hypothesis testing becomes most evident in runtime. Figure 6 plots the distribution of speedup achieved by StochasticGrade compared to Batch Hypothesis Testing in identifying incorrect programs, where "runtime" is measured by the number of samples required from the error program. Considering that Batch Hypothesis Testing requires the full set of samples, StochasticGrade allows *exponential* (up to $1,024$ times) speedup for many evident errors, which are often detected with only 400 samples.

sures the amount of mutual statistical dependence between the two clusterings.

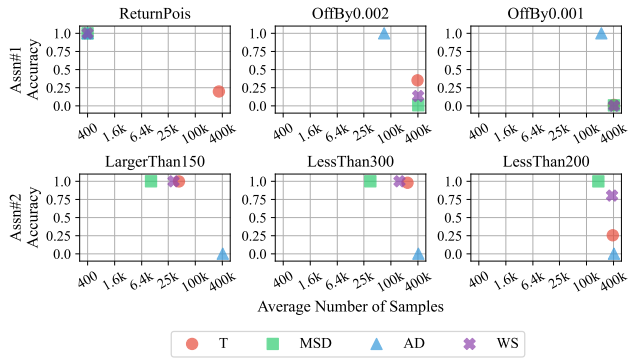


Figure 7: Accuracy and average number of samples required for different errors of Assn#1 (Top) and Assn#2 (Bottom) measured over 1,000 independent iterations with $\alpha = 0.01$.

4.2 Analysis for Different Error Types

The results from Section 4.1 suggest that the accuracy and runtime of StochasticGrade depend crucially on the choice of the disparity function. This section will study the relative strengths and limitations of each disparity function and random projection method presented in Section 2. We will choose several elusive errors from Assn#1, 2, and 3 and look at the the accuracy of StochasticGrade on identifying these errors along with the number of samples required when each disparity function and random projection method is used. For all experiments we ran StochasticGrade with $\alpha = 0.01$.

Univariate Grading. Figure 7 plots the runtime and grading accuracy for 3 elusive errors each from Assn#1 and Assn#2 with $\alpha = 0.01$, measured over 1,000 independent trials. The details of the errors we analyzed are as follows: In Assn#1, `ReturnPois` returns samples from a Poisson distribution Poi($\lambda = 1/3.1$) instead of the Exponential, which are two very different distributions that happen to have identical means. `OffBy{p}` are subtle translations of the output by `+p`. In Assn#2, `LargerThan150` omits the step of dismissing outputs larger than 150 after sampling from Pareto($\beta = 1.6$), and `LessThan{n}` rejects values smaller than `n` instead of 150. As noted earlier in Section 3.1, this changes the behavior of the program only $\approx 0.03\%$ of the time but may occasionally output extremely large values.

Most notably, all disparity functions efficiently detect the largely deviant error of `ReturnPois` with very few samples *except for* T. Although this is a particular example of an error with a mean that is identical to the solution, it illustrates an important failure point for disparity functions that only compare a summary statistic.

Next, we observe that AD is more sensitive to differences in the "typical" regions of the distributions than other disparity functions, as discussed in Section 2.3. `OffBy0.002` and `OffBy0.001` errors incur very small changes in the probabilities of the high-density regions of the solution distribution, to which AD is particularly sensitive. For instance, on detecting `OffBy0.002`, T, MSD, and WS achieve low accuracy whereas AD performs perfectly. AD also completely identifies the more subtle `OffBy0.001` errors with around 300k samples on average, whereas all other disparity functions achieve 0% accuracy.

In contrast, AD completely fails to recognize the close errors in Assn#2. Programs with these errors rarely behave differently from the solution program, but when they do, they generate values that are large or even unbounded. MSD is the most effective in detecting these errors because it is sensitive to the difference in the spread of the distribution, and it perfectly detects these errors with the fewest number of samples among all disparity functions. WS also performs well as it detects the difference in the magnitude of the output values, although it performs slightly worse than MSD on `LessThan200`, which is most subtle among the three errors.

Multivariate Grading. Figure 8 illustrates the runtime and grading accuracy for 5 errors[11] from Assn#3 for various disparity functions and two random projection methods. In `{Single/All}Coord+1`, the error programs add 1 to the last element or all elements of the output vector, which results in a small translational shift in the distribution of the output. `Repeat{All/Last/Random}Coord` errors involve duplicating values across the entire vector, the last two elements, or a randomly selected pair of elements in the output vector. While these errors do not alter the mean of the distribution, they create minor but spurious correlations among the coordinates.

First we notice that AD and WS both perform consistently well under both types of random projections, achieving perfect accuracy across nearly all error types with similar sample efficiency. This is unlike in Assn#1 and Assn#2 where the peculiarities of the error distributions (i.e., occasional extremes and very small translations) caused each disparity function to fail respectively. The errors in the figure do not exhibit such peculiarities both before and after projection, so these two disparity functions perform comparably.

On the other hand, the performances of T and MSD contrast with one another. The projections of the `{*}Coord+1` errors entail a noticeable difference in means but a relatively small difference in the spread compared to the projection of the solution distribution. As expected, T effectively picks up on the difference in means with relatively few samples, whereas the small difference in the spread misleads MSD. Conversely, the `Repeat{*}Coord` errors result in a change in covariance and no difference in means, which causes the projections to also exhibit a noticeable change in spread but no more than a small difference in means relative to the solution. MSD is effective and often more efficient than WS or AD in identifying these errors, whereas T either requires many samples or fails. These results confirm the properties of the two disparity functions as outlined in Section 2.3.

Random Projections. Figure 8 also suggests the utility of orthogonal projection (OP) and Euclidean distance projec-

[11]For analysis purposes, we added 3 artificial errors in addition to `AllCoord+1` and `RepeatAllCoord` that were observed in the dataset.

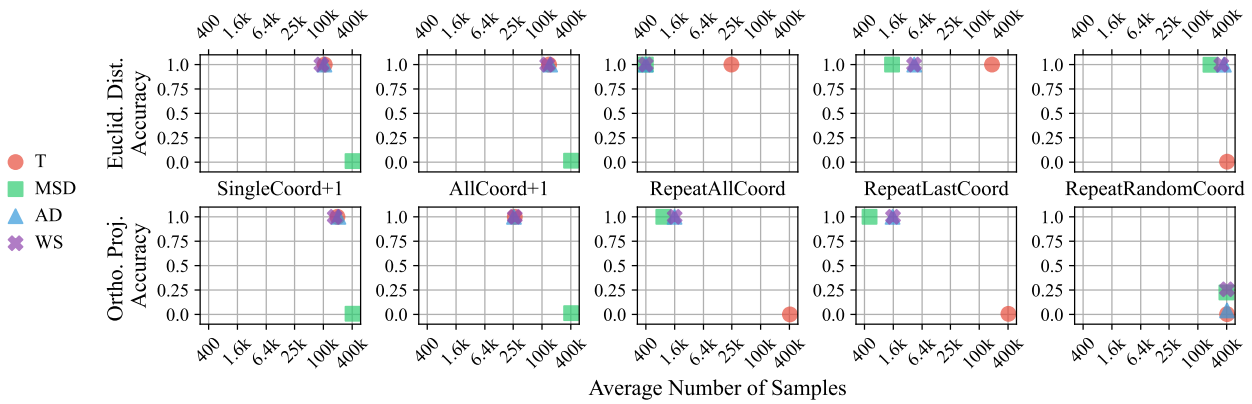


Figure 8: Accuracy and average number of samples required for 5 different errors of Assn#3 using Euclidean Distance projection (Top) and Orthogonal Projection (Bottom), measured over 1,000 iterations with $\alpha = 0.01$. See Section 4 for detail.

tion (ED) depending on the geometry of the error distribution. OP preserves noticeable linear translations or directional changes in the spread of the distribution, allowing most disparity functions to recognize `AllCoord+1` and `RepeatLastCoord` with fewer samples than with ED. However, the non-linear geometry of ED proves to be more effective at detecting a wider variety of more subtle errors, allowing `SingleCoord+1` and `RepeatAllCoord` to be noticed with fewer samples than OP. It also exposes the elusive `RepeatRandomCoord` error to most disparity functions, which OP cannot capture.

5. CLUSTERING RESULTS

In addition to evaluating programs as correct or incorrect, instructors often seek to understand the errors in the student programs more closely. *Clustering* student programs by their error type can allow instructors to quickly identify patterns of common mistakes among students, assign partial credit, and provide high-quality personalized feedback [5, 14, 13]. In this section, we demonstrate that the disparity measurements calculated by StochasticGrade convey enough identifying information about the error to be used as a simple feature for effective clustering. Moreover, we demonstrate that *concatenating*[12] the disparity measurements calculated using more than one disparity function, random projection, or random input case can profoundly enhance the quality of clustering.

The improvement effect of composing multiple disparity measurements can be understood by considering the disparity measurement as a "view" of the error distribution. Measurements from multiple disparity functions, random projections, and input cases diversify the set of perspectives on the program and yield a higher resolution representation. For each assignment, we will now present the clustering results and analyze the effect of composing different types of disparity measures.

Assn#1: Composing Measurements from Multiple Disparity Functions. Figure 9 plots the clustering performance for Assn#1 using each of the 4 disparity measurements alone (red) and an incremental concatenation of multiple disparity measurements (other colors) as input features. We observe that AD performs the best among all disparity functions while MSD and T do the worst. Notice, however, that combining measurements from AD and MSD improves clustering performance beyond any single disparity function, surpassing AD by as much as 0.13 ARI. Moreover, concatenating more disparity measurements increases the clustering performance almost monotonically, where combining all 4 measurements outperforms AD by up to 0.1 on NMI and reaches AD's NMI at 409.6k samples with 64 times fewer samples.

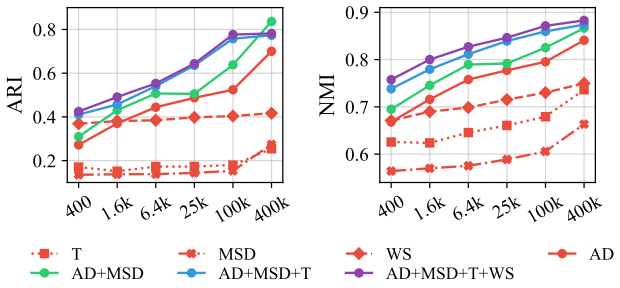


Figure 9: Clustering performance for Assn#1. Composing multiple disparity functions enhances clustering.

Assn#3: Composing Disparity Measurements Derived from Multiple Projections. Figure 10 presents the clustering results for Assn#3 using disparity measurements calculated from a varying number of random Euclidean distance projections. The figure displays the results for the best and worst performing disparity measurements, which were AD and MSD respectively. We see that concatenating disparity measurements from multiple random projections dramatically increases clustering quality for both AD and MSD even with relatively small sample sizes. Moreover, with 100k samples, composing 4 random projections for MSD results in near-perfect clustering with up to a remarkable 0.8 increase in NMI and 0.5 in ARI compared to using a single projection.

[12]More precisely, the disparity measurements were normalized to zero mean and unit variance prior to concatenation

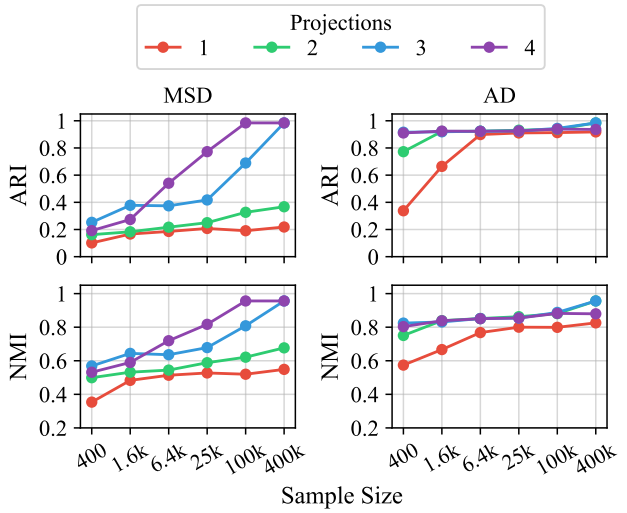


Figure 10: Clustering performance for Assn#3 using disparity measurements calculated under multiple projections. Increasing the number of projections increases performance.

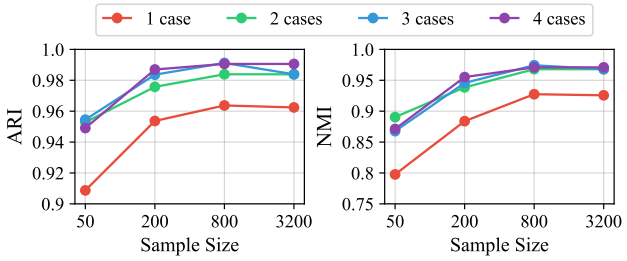


Figure 11: Clustering performance for Assn#4 using disparity measurements from multiple input cases. Using more input cases enhances clustering.

Assn#4: Composing Measurements Derived from Multiple Input Arguments. For Assn#4, Figure 11 shows the effect of combining multiple T measurements obtained by passing in several different input arguments (cases) to the student programs. Although T already achieves strong performance, using disparity measurements generated by more than one input case generally improves clustering, achieving up to 0.04 ARI and 0.09 NMI increase with only 1 additional case.

6. GUIDELINES FOR PRACTITIONERS

What problem does* StochasticGrade *address? You may want to auto-grade programs that produce probabilistic output through randomization. StochasticGrade is an open-source solution to this problem, implemented in Python.

What are the properties of an assignment for which you could apply* StochasticGrade*? You need to be able to run the student code and collect a sample, which is its output. That sample should be translatable into a number or an array of numbers. Note that you don't need StochasticGrade for programs with deterministic output, even if they use probability theory. It is designed for programs that generate probabilistic outputs.

On a high level, how does* StochasticGrade *work? The auto-grading algorithm is based on hypothesis testing theory. First, `preprocess.py` collects samples from the correct answer before grading, which could take several minutes. At the time of grading, `stochastic_grade.py` auto-grades student programs primarily based on 3 important parameters: (1) which "disparity function" to use, (2) the desired false rejection rate (FRR) which is the probability of misgrading a correct program, and (3) the maximum number of samples to collect from the student program. The grader will keep sampling from the student program until it is confident that the program is either correct or incorrect, not exceeding the maximum number of samples. In this paper, we showed that this allows up to 1,024x speedup in some cases.

Which disparity function should I choose? For detecting most errors, any function from Section 2.3 should suffice. Our default suggestion is to use AD, which is sensitive to subtle mismatches in the probability distributions. If you expect some error programs to be "heavy-tailed" or occasionally output extreme values, WS works better than AD. If you cannot afford to store or process many samples (e.g., due to tight memory constraints or you're using a web-app that grades on the client-side), you can use T or MSD which only depend on summary statistics, but these disparity functions may not detect subtle errors.

What FRR should I use? Our default value is 0.01 which will mark at least 99% of correct programs as correct, but you can also set it to a smaller value of your choice like 0.003. Note that the false rejection cases would need manual re-grading. (See the guideline below on repeated grading.)

How do I select the maximum sample size? It depends on your problem and your tolerance for misgrades. If you expect students to make very subtle errors, you will need more samples. If you have a lower tolerance for misgrades, you will also need more samples. To select `n_samples` we suggest you implement the buggy student program which is most similar to the correct answer and run our `choose_n` script. The most important inputs to this script are: `(correct_program, error_program, disparity_fn, FRR, FAR = None)`. It will either return the smallest N or an error if FAR is not applicable to the disparity function you provided (T or MSD).

What should I do with the other parameters? These parameters can be set to the default values we provide in our code, and we believe they will work well in most cases. In addition to the 3 most important parameters mentioned above, other configurable parameters include the minimum sample size $N_{\min}$, the sample growth factor R (Algorithm 1), and the number of Monte Carlo iterations M in our critical value algorithm (Algorithm 2). In our implementation, the default values are set to: $N_{\min} = 400$, $R = 4$, and $M = 1000$. However, you can adjust the behavior (e.g., runtime or precision) of our algorithm as you wish by changing these parameters. Our public repository has more detailed guidelines about how to adjust these parameters.

Can students submit multiple times? This might break the control over FAR you are trying to achieve. If you run the test on the same buggy program many times, it becomes more likely that at least one run will report that it is correct.

7. DISCUSSIONS AND LIMITATIONS

Evaluating Other Aspects of Good Programming. In this work we have shown that StochasticGrade can accurately evaluate whether student programs are functionally correct. Functional correctness is indeed critical, but it is only one of many components of writing good software. Algorithmic efficiency, program style, readability, and intuitive program decomposition are other key components of programming that StochasticGrade does not explicitly capture but are important areas for feedback to be included in grading.

Grading Efficiency. StochasticGrade is designed to terminate early for grossly discrepant error programs, and this leads to a significant speedup on identifying incorrect programs compared to standard batch hypothesis testing. For correct programs, however, the algorithm must obtain the maximum number $N_{\max}$ of samples before it determines that the program is correct.[13]

Similar issues also arise when obtaining a large solution program sample $X_{\mathcal{C}}$ during preprocessing. These runtime constraints are tolerable when student programs run fast, which is often the case for introductory programming or simple probabilistic modeling assignments. For more complex assignments that involve sophisticated algorithmic manipulation (e.g., implementing statistical inference algorithms or training routines for machine learning applications) where completing a single run of each program can be slow, the current form of StochasticGrade could be impractical.

One possible approach to reducing the required amount of samples and increasing the efficiency of grading is to incorporate insights from program traces, such as variable assignment at runtime and execution paths. While the proposed StochasticGrade algorithm relies solely on the final output of the program, additionally considering these program traces could increase the amount of diagnostic information that can be obtained with fewer samples. We leave this direction as an interesting topic for future research.

Clustering and Sampling. In Section 5 we demonstrated that StochasticGrade's disparity measurements can be used for efficient and effective clustering of student programs by their error type. One caveat of our clustering results is that they are based on disparity measures calculated using a *fixed* number of samples for all student programs (Section 3.2). StochasticGrade can adaptively decide to obtain a different number of samples from each program, and clustering should ideally be done using only those samples. Even with the same disparity function, however, different sample sizes lead to disparity measurements of disparate scales, and clustering them altogether typically leads to poorer clustering quality. Identifying clusters with identical error types based on StochasticGrade's adaptive sample sizes remains an important area for improvement.

[13]The optimal $N_{\max}$ for a target error program can be estimated based on the ideas discussed in Section 2.2 and is implemented in the `choose_n.py` script of our open-source library. This script, however, can also be slow even with parallelization due to Monte Carlo estimation.

Practical Limitations to the Theoretical Error Guarantee. As noted in Section 2.2, the theoretical error guarantees for the FRR and FAR hold in the limit where the size of the solution program sample $X_{\mathcal{C}}$ approaches infinity. To get as close to this limit as possible, StochasticGrade obtains a large $X_{\mathcal{C}}$ prior to grading (set to 500k samples by default) and uses this to grade all student programs. For the 4 real classroom assignments we studied, 500k solution samples were sufficient to achieve the guaranteed FRR, and it only took less than a minute (Figure 5) and about 3.8MB of space to obtain and store all samples for AD and WS.

In more resource-constrained settings, however, using disparity functions like AD and WS that require access to all samples can cause considerable space and transmission bottleneck for larger sample sizes. This can happen, for instance, when students in a massive online classroom run StochasticGrade locally on memory-constrained devices or browsers and $X_{\mathcal{C}}$ has to be sent over a network. Improving the sample complexity of the grading algorithm or designing new disparity functions that achieve both memory efficiency and grading accuracy[14] will greatly improve the adaptability of StochasticGrade.

8. CONCLUSION

Stochastic programs are a crucial paradigm in many areas of computer science education, but to the best of our knowledge, no prior work has addressed the problem of *auto-grading* these types of programs. In this paper, we formalized this problem and developed an open-source auto-grading framework, StochasticGrade. Our framework is able to accurately recognize errors in exponentially less time than standard two-sample hypothesis testing while allowing users to explicitly control the rate of misgrades. Moreover, it can efficiently handle programs with multidimensional output and allows a flexible choice of the metric of disparity between samples to be used throughout the grading process. We showcased 4 disparity metrics and explored their respective strengths and limitations. Measurements made using these metrics are also useful for simple and effective clustering of programs by error type. We demonstrated the grading and clustering capabilities of StochasticGrade using real programming assignment data and provided guidelines for practitioners who wish to use our open-source framework.

Several promising avenues for future research lie ahead. First, further improvements in the efficiency of the grading algorithm could be sought after. Next, noting that student errors fall across a wide spectrum, exploring the space of errors in student-written stochastic programs may help instructors anticipate the error typology when configuring our auto-grading framework and allow them to make informed decisions about the use of our algorithm. Lastly, as our present work focuses on grading and clustering, a natural next step would involve automatically building rubrics for partial grading and providing high-quality, targeted feedback to students. We hope our work illuminates exciting new directions in the study of stochastic programs in computer science education.

[14]As noted in Section 2.3, T and MSD are memory efficient as they depend only on summary statistics, but they could often be inaccurate.

9. REFERENCES

[1] Y. Akpinar and Ü. Aslan. Supporting children's learning of probability through video game programming. *Journal of Educational Computing Research*, 53(2):228–259, 2015.

[2] K. M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102, 2005.

[3] R. Alur, L. D'Antoni, S. Gulwani, and D. Kini. Automated grading of dfa constructions. In *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1976–1982, 2013.

[4] D. Baldwin, H. M. Walker, and P. B. Henderson. The roles of mathematics in computer science. *Acm Inroads*, 4(4):74–80, 2013.

[5] S. Basu, C. Jacobs, and L. Vanderwende. Powergrading: a clustering approach to amplify human effort for short answer grading. *Transactions of the Association for Computational Linguistics*, 1:391–402, 2013.

[6] A. F. Blackwell, L. Church, M. Erwig, J. Geddes, A. Gordon, I. G. Maria, A. G. Baydin, B. Gram-Hansen, T. Kohn, N. Lawrence, et al. Usability of probabilistic programming languages. In *PPIG*, pages 53–68, 2019.

[7] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin. Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 215–220, 2013.

[8] M. Dhariwal and S. Dhariwal. Let's chance: Playful probabilistic programming for children. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2020.

[9] A. Dvoretzky, J. Kiefer, and J. Wolfowitz. Asymptotic minimax character of the sample distribution function and of the classical multinomial estimator. *The Annals of Mathematical Statistics*, pages 642–669, 1956.

[10] S. A. Fincher and A. V. Robins. *The Cambridge handbook of computing education research*. Cambridge University Press, 2019.

[11] D. Ginat, R. Anderson, D. D. Garcia, and R. Rasala. Randomness and probability in the early cs courses. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 556–557, 2005.

[12] N. D. Goodman, J. B. Tenenbaum, and T. P. Contributors. Probabilistic Models of Cognition. `http://probmods.org/v2`, 2016. Accessed: 2023-8-2.

[13] S. Gulwani, I. Radiček, and F. Zuleger. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4):465–480, 2018.

[14] S. Johnson-Yu, N. Bowman, M. Sahami, and C. Piech. Simgrade: Using code similarity measures for more accurate human grading. In *EDM*, 2021.

[15] M. Jukiewicz. The future of grading programming assignments in education: The role of chatgpt in automating the assessment and feedback process. *Thinking Skills and Creativity*, page 101522, 2024.

[16] Y. Kim and C. Piech. The student zipf theory: Inferring latent structures in open-ended student work to help educators. In *LAK23: 13th International Learning Analytics and Knowledge Conference*, pages 464–475, 2023.

[17] C. Kleiner, C. Tebbe, and F. Heine. Automated grading and tutoring of sql statements to improve student learning. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, pages 161–168, 2013.

[18] Z. Kurmas. Mipsunit: A unit testing framework for mips assembly. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 351–355, 2017.

[19] M. Lopes, L. Jacob, and M. J. Wainwright. A more powerful two-sample test in high dimensions using random projection. *Advances in Neural Information Processing Systems*, 24, 2011.

[20] M. Madeja and J. Porubän. Automated testing environment and assessment of assignments for android mooc. *Open Computer Science*, 8(1):80–92, 2018.

[21] E. Maicus, M. Peveler, S. Patterson, and B. Cutler. Autograding distributed algorithms in networked containers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 133–138, 2019.

[22] A. Malik, M. Wu, V. Vasavada, J. Song, M. Coots, J. Mitchell, N. Goodman, and C. Piech. Generative grading: Near human-level accuracy for automated feedback on richly structured problems. *arXiv preprint arXiv:1905.09916*, 2019.

[23] J. Moreno-León and G. Robles. Dr. scratch: A web tool to automatically evaluate scratch projects. In *Proceedings of the workshop in primary and secondary computing education*, pages 132–133, 2015.

[24] F. Nilsson and J. Tuvstedt. Gpt-4 as an automatic grader: The accuracy of grades set by gpt-4 on introductory programming assignments, 2023.

[25] J. C. Paiva, J. P. Leal, and Á. Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)*, 22(3):1–40, 2022.

[26] F. A. K. Panni and A. S. M. L. Hoque. A model for automatic partial evaluation of sql queries. In *2020 2nd International Conference on Advanced Information and Communication Technology (ICAICT)*, pages 240–245. IEEE, 2020.

[27] S. Papert. An exploration in the space of mathematics educations. *Int. J. Comput. Math. Learn.*, 1(1):95–123, 1996.

[28] M. Peveler, E. Maicus, and B. Cutler. Automated and manual grading of web-based assignments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1373–1373, 2020.

[29] S. M. Ross. *A course in simulation*. Prentice Hall PTR, 1990.

[30] S. Sahai, U. Z. Ahmed, and B. Leong. Improving the coverage of gpt for automated feedback on high school programming assignments. In *NeurIPS'23 Workshop Generative AI for Education (GAIED). MIT Press, New Orleans, Louisiana, USA*, volume 46, 2023.

[31] M. Sahami. A course on probability theory for computer scientists. In *Proceedings of the 42nd ACM*

technical symposium on Computer science education, pages 263–268, 2011.

[32] F. W. Scholz and M. A. Stephens. K-sample anderson–darling tests. *Journal of the American Statistical Association*, 82(399):918–924, 1987.

[33] R. B. Shapiro and M. Tissenbaum. New programming paradigms. In *The Cambridge Handbook of Computing Education Research*, pages 606–636. Cambridge University Press, 2019.

[34] V. S. Shekhar, A. Agarwalla, A. Agarwal, B. Nitish, and V. Kumar. Enhancing jflap with automata construction problems and automated feedback. In *2014 Seventh International Conference on Contemporary Computing (IC3)*, pages 19–23. IEEE, 2014.

[35] A. C. Siochi and W. R. Hardy. Webwolf: Towards a simple framework for automated assessment of webpage assignments in an introductory web programming class. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 84–89, 2015.

[36] I. Solecki, J. Porto, N. d. C. Alves, C. Gresse von Wangenheim, J. Hauck, and A. F. Borgatto. Automated assessment of the visual design of android apps developed with app inventor. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 51–57, 2020.

[37] A. Stuhlmüller and N. D. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. *arXiv preprint arXiv:1206.3555*, 2012.

[38] S. S. Vempala. *The random projection method*, volume 65. American Mathematical Soc., 2005.

[39] M. Wu, M. Mosse, N. Goodman, and C. Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 782–790, 2019.

[40] L. Yan, N. McKeown, and C. Piech. The pyramidsnapshot challenge: Understanding student process from visual output of programs. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 119–125, 2019.