30 Days After Introducing Programming: Which of My Students Are Likely to Fail?

Márcio Ribeiro[1] , **Rodrigo Paes**[1] , **Baldoino Fonseca**[1] ,
Jackson Leite[2] , **Thais Castro**[2] , **Rohit Gheyi**[3]

[1] Federal University of Alagoas (UFAL) – Maceió-AL.

[2]Federal University of Amazonas (UFAM) – Manaus-AM.

[3]Federal University of Campina Grande (UFCG) – Campina Grande-PB.

`{marcio,rodrigo,baldoino}@ic.ufal.br,`

`{jlp,thais}@icomp.ufam.edu.br, rohit@dsc.ufcg.edu.br`

Abstract. *Predictors to identify whether a student will succeed or fail in introductory programming courses have been provided by previous research. However, they rely on time-consuming aptitude tests and surveys. This way, setting, executing, and replicating these studies is hard and increase the professor effort. Other predictors rely on automatic procedures, but they either do not identify the failing students early or do not provide high effectiveness. To minimize these problems, we propose a strategy to early predict the potential failing students during introductory programming courses automatically, reducing effort and allowing professors to use it in every course. By having this set of students in the first days of the course, professors and mentors would have time to act and potentially avoid such failings. The strategy consists of three steps: the use of an online judge system; the collection of metrics from this system; and the use of a clustering algorithm. To evaluate our strategy, we conduct an empirical study regarding 358 freshmen students of 12 introductory programming courses. We consider the first 30 days of the course. From the group of students our strategy points as "likely to fail," 80% of the students on average indeed fail.*

1. Introduction

Introductory programming courses are often perceived by the students as problematic [Yadin 2011]. This understanding seems to be based on the high dropout rates [Dehnadi and Bornat 2006][Porter et al. 2013], which has been observed in universities around the world, such as in Finland [Kinnunen and Malmi 2006] and ours in Brazil.

To predict the students likely to succeed or fail, previous studies provide aptitude tests regarding programming activities. These studies consider many different variables from big, complicated, and time-consuming surveys. To perform the predictions, they use diagnostic and math/logic-based tasks [Hughes and McNamara 2014], mental models [Dehnadi and Bornat 2006], games [da Silva et al. 2014], and even programming languages [Harris 2014]. Thus, although we can predict the potential failing students, these studies do not scale: setting, executing, and replicating them is hard, time consuming, and require extra effort from the professors. To avoid this problem, automatic approaches have been proposed [Watson et al. 2013, Santos et al. 2016]. However, they predict the failing students either too late or with moderate precision.

In this context, we still lack an automatic approach capable of predicting the set of students that will fail. This way, professors and mentors would be able to act and consequently help them. To achieve promising results, however, this approach must accomplish three main requirements. First, it must predict *as soon as possible*, otherwise there will be not enough time to act and the student would drop out the course anyway. In addition, due to the strong prerequisites of understanding previous classes to understand the current one in programming courses, the situation gets worse, even when acting just a bit late. Second, it must predict with *high effectiveness*, otherwise professors would spend time helping many students that actually do not need much help. Third, the approach must be *automatic*, requiring almost no effort from professors and mentors and allowing them to use it in every course they teach.

To accomplish these three requirements, we propose an automatic strategy to early predict failing students in introductory programming courses. Our strategy consists of three simple steps. The first one is to make students use an online judge system. This kind of system executes an online submitted solution to a given problem against a set of predefined test cases to check whether the solution is correct or not. Then, we collect metrics of each student from such a system. Finally, we execute a clustering algorithm to form groups so we are able to separate the potential failing students from the other ones.

To evaluate our strategy, we instantiate it with an online judge used in several Brazilian universities, with two metrics, and with the well-known *k-means* clustering algorithm. Then, we conduct an empirical study regarding 12 introductory programming courses—6 years, from 2010.02 to 2016.01, yielding 12 semesters—with 358 freshmen students (30 per course, on average). The courses have been lectured by the same professor at the Federal University of Alagoas (UFAL) in Brazil. We consider the first 30 days of the programming course (22% of each semester). The results suggest that our strategy can early predict the majority of the failing students within only 30 days. In particular, from the group of students our strategy points as "likely to fail," 80% of the students on average indeed fail. To better generalize our results, we compute the confidence interval. The population average lies between 66.4% and 92.9% with 95% confidence level. In summary, this paper provides the following contributions:

- A strategy to early predict the potential failing students in introductory programming courses automatically (Section 2);
- An empirical study assessing the potential of an instance of our strategy. We evaluate our strategy by using 358 students from 12 courses during 6 years, demonstrating significant potential (Sections 3 and 4).

2. Strategy to early predict potential failing students

In this paper, we rise three challenges we intend to address: predict the students likely to fail *as soon as possible*, with *high effectiveness*, and *automatically*. To do so, we present in this section a strategy to early predict the potential failing students in introductory programming courses automatically. Predicting the students likely to fail early is very important in the sense that professors still have enough time to act and avoid students to fail the course. Notice that in case professors act based on the results of our strategy, we can indirectly help on reducing the high rate of failing students. However, in this paper, we focus exclusively on *identifying* these students. So, interventions that could be applied by instructors to avoid students' failures are outside the scope of this paper.

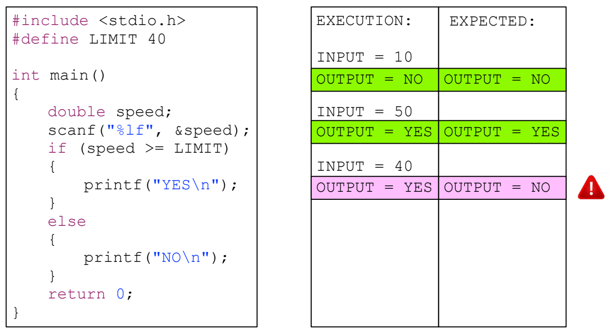
Our strategy consists of three simple steps: the use of an online judge system, metrics retrieval, and execution of a clustering algorithm. Next, we detail these steps.

2.1. Three steps

Online judge system. Monitoring the students performance is difficult and increases the professor effort, being a time-consuming task. To minimize this problem, one might rely on online learning tools, since these systems allow the information retrieval of each student automatically. A popular category of this kind of system is the online judges [uva 2014]. Online judges provide a set of programming problems so that students can submit their solutions. After submitting, the online judge executes the solution against a set of predefined test cases. In case the solution passes over all the tests, the system evaluates the solution as correct. Otherwise, the system evaluates the solution according to the error type (e.g., wrong answer, compilation error, time limit exceeded, etc) and even might give important tips so that students can solve the problem afterwards.

To better illustrate how an online judge system works, we refer to the example we present in the figure below. The programming problem is the following: given the current speed, the program should print "YES," in case the speed limit (40 mph) has been exceeded, and "NO," otherwise. The figure illustrates a solution submitted to the judge.

To check the solution, in this example the judge relies on three test cases. Two test cases pass, i.e., the outputs provided by the solution hit the expected outputs. However, the third test case does not pass: 40 mph has been reached, but not exceeded. To fix the problem, the student can submit again with `speed > LIMIT` instead of `speed >= LIMIT`.



```c
#include <stdio.h>
#define LIMIT 40

int main()
{
    double speed;
    scanf("%lf", &speed);
    if (speed >= LIMIT)
    {
        printf("YES\n");
    }
    else
    {
        printf("NO\n");
    }
    return 0;
}
```

```
EXECUTION:        EXPECTED:

INPUT = 10
OUTPUT = NO       OUTPUT = NO

INPUT = 50
OUTPUT = YES      OUTPUT = YES

INPUT = 40
OUTPUT = YES      OUTPUT = NO
```

Since students only learn how to program by programming [Jenkins 2002], the online judge consists of an important environment in the sense students can constantly practice and improve their learning process. This way, by using the online judge system, we can monitor students to automatically identify the candidates to fail the course.

Metrics. To assess the students performance during a course, it is important to closely monitor them. In this context, metrics represent an alternative to measure their performance. In our strategy the online judge represents the source for retrieving metrics regarding the students. These systems commonly contain metrics like number of logins, total amount of time using the system, number of submissions, number of correct submissions, and so forth. We consider metrics retrieval as the second step of our strategy.

Clustering algorithm. To identify potential failing students, we use clustering algorithms to define groups of students. Our idea consists of identifying different groups of students so we can clearly separate students likely to fail from the other ones. To compute the groups, the algorithm should take into account the metrics we retrieve from the online judge. The use of a clustering algorithm represents the third step.

2.2. Instantiating each step

Our strategy is general in the sense we can use any online judge system, as long as it contains problems commonly found in introductory programming courses and we can collect metrics regarding the students performance. In addition, notice that the strategy is automatic (no manual aptitude tests needed). This way, professors can apply the strategy more easily in their courses, reducing their effort.

We instantiate our strategy by using the online judge system named *Huxley*. The system is available at `http://www.thehuxley.com/`. There are 439 registered professors and 525 registered courses (all introductory in programming). More than 80 institutions around Brazil use the system. It provides a database with more than 1000 problems classified according to level of difficulty and programming topics.

From the metrics that Huxley provides, we discarded number of logins and number of days that the student logged in, since high numbers of these metrics do not necessarily mean that the student is practicing. So, to capture the students effort, we consider: **Number of submissions:** this metric represents the number of submissions a student makes. To solve a problem, a student needs to submit at least one correct solution; and **Number of correct submissions:** if a student solves one problem, we increase this metric by one. High numbers for these metrics seem to be a good indicator of the amount of practice, which is very important in programming activities [Cheang et al. 2003].

To identify potential failing students, we use an existing clustering algorithm to define groups of students at the 30th day. Our idea consists of identifying different groups of students so we can clearly separate students likely to fail from the other ones. To compute the groups, the algorithm takes into account the aforementioned metrics. To instantiate our strategy and make it parameterizable in terms of number of groups, we use the well-known k-means algorithm.

In this paper, we set the algorithm to compute two and three groups. For two groups, we have students who either fail or pass. For three groups, we have fail, pass, and students that the strategy will not conclude anything about them, which we name "inconclusive." We focus on two and three groups basically for two reasons. Using one group is useless; and more than three only brings more inconclusive groups to the table which is pretty much the same of having only one inconclusive group.

Figure 1 combines all steps of our strategy. We let the students use the online judge during the very first 30 days (Step 1), representing 22% of the semester. At the end of the 30th day, the professor invokes a software module to (i) collect metrics from the online judge system regarding the past 30 days (Step 2) and (ii) execute the clustering algorithm (Step 3). In this example, we illustrate the clustering algorithm result in a two-dimensional plot (Metric 1 = Number of Submissions and Metric 2 = Number of Correct Submissions). Also, we illustrate the groups using shapes in Figure 1. In this particular example, we have three different groups. The circles represent the students candidates to fail the course. Notice that they submitted few solutions and few of them are correct.

3. Evaluation

We now evaluate our strategy by using the instance presented in Section 2.2. To do so, we perform an empirical study. This study has the purpose of evaluating the instance of our

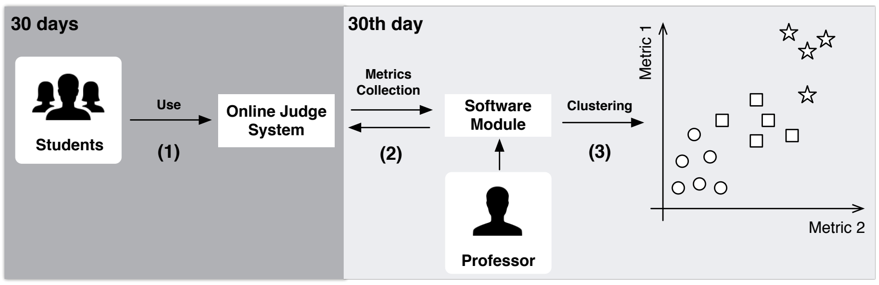


Figure 1. Summary of our strategy to identify potential failing students.

strategy to verify to what extent it is capable of identifying potential failing students in only 30 days. For brevity, we now refer to “instance of our strategy” as simply “strategy.”

3.1. Settings

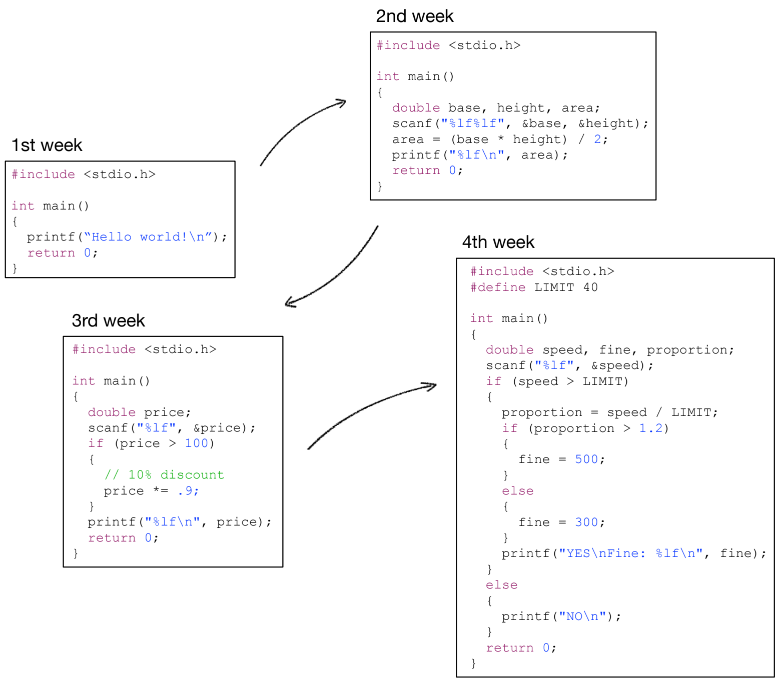
The participants of our study are freshmen students of introductory programming courses at the Federal University of Alagoas (UFAL) in Brazil. We collected the metrics we detail in Section 2.2 of each student by using Huxley.

The same professor lectured the 12 semesters. The classes happened in a laboratory and the students solved some exercises available in Huxley during the classes. The professor strongly encouraged all students to submit solutions for the problems available in Huxley as extra-class activities, once there is absolutely no penalty in case of wrong submitted solutions. So, the students were free to submit as many times as they wanted. However, during the first 30 days, the use of Huxley was mandatory for six exercises selected by the professor (used as a small percentage of the final grade) and for the first formal exam (held closely to the 30th day). The programming language was C. Notice that the focus was not to teach the language itself, but to teach logic programming.

To better illustrate the kinds of problems that the professor asks the students to solve in the Huxley during the 30 days, we refer to Figure 2. In the 1st week, the student should print messages on the screen; 2nd week: read variables and implement simple math computations; 3rd week: conditionals; 4th week: nested conditionals. Previous work suggest that the first weeks have an important influence on predicting the student outcomes [Porter and Zingaro 2014]. In particular, they find that conditionals play an important role, because the ability to trace and understand them enables the students to understand loops and functions as well [Porter et al. 2014]. These results give support to our idea of focusing on 30 days and gathering data until conditionals (4th week).

3.2. Procedure

The result of executing our strategy consists of groups of students according to the clustering algorithm. To avoid discrepancies in absolute magnitudes of our two metrics, we normalize the data. This way, the metrics are between zero and one. In Figure 1 we have three groups. We illustrate each group by using a different shape: circle, square, and star. We represent the students likely to fail by circles. These students are the ones that have a low number of submissions and correct submissions. The opposite occurs with students candidates to successfully pass (represented by stars): due to the high number of submis-



```c
#include <stdio.h>

int main()
{
  printf("Hello world!\n");
  return 0;
}
```

```c
#include <stdio.h>

int main()
{
  double base, height, area;
  scanf("%lf%lf", &base, &height);
  area = (base * height) / 2;
  printf("%lf\n", area);
  return 0;
}
```

```c
#include <stdio.h>

int main()
{
  double price;
  scanf("%lf", &price);
  if (price > 100)
  {
    // 10% discount
    price *= .9;
  }
  printf("%lf\n", price);
  return 0;
}
```

```c
#include <stdio.h>
#define LIMIT 40

int main()
{
  double speed, fine, proportion;
  scanf("%lf", &speed);
  if (speed > LIMIT)
  {
    proportion = speed / LIMIT;
    if (proportion > 1.2)
    {
      fine = 500;
    }
    else
    {
      fine = 300;
    }
    printf("YES\nFine: %lf\n", fine);
  }
  else
  {
    printf("NO\n");
  }
  return 0;
}
```

Figure 2. Examples of problems that the students solved during the first 30 days.

sions and correct submissions after 30 days, they seem to practice programming really hard. Finally, we represent the inconclusive group by squares.

To perform the evaluation, we apply the strategy and then check whether it correctly predicts the failing students after 30 days. To do so, we use the academic system of the Federal University of Alagoas (UFAL) to look for grades and check whether the students indeed failed or not. To better structure and analyze our results and, at the same time, take false positives and false negatives into account, we consider the following two well-known metrics: precision and recall [Witten and Frank 2005]. Precision is the fraction of retrieved students that are relevant. Recall, in its turn, is the fraction of relevant students that are retrieved.

After confronting the strategy results with the academic system and summarizing precision and recall, we apply a statistical test to check for significance. We rely on the proportion statistical test based on the Bernoulli distribution so that we have a binary distribution: fail or pass. We follow the convention *p-value* < 0.05.

4. Results and Discussion

In this section, we describe the results and test our hypotheses before discussing their implications. All data, materials, and R scripts we use are available at `bit.ly/whichofmystudentswillfail`. In our evaluation we use data of 12 courses totalling 358 students. Table 1 illustrates the results of precision and recall for all semesters.

Two or Three Groups? When applying our strategy considering two groups, the results indicate a higher recall average when compared to three groups. This means that the strategy with two groups can identify the majority of the failing students (raising less false negatives). However, the strategy with two groups yields many false positives. In

		Two Groups		Three Groups	
Semester	**Enrolled students**	**Precision**	**Recall**	**Precision**	**Recall**
2010.02	33	0.89	0.92	1.00	0.58
2011.01	35	0.71	1.00	1.00	0.45
2011.02	36	0.4	0.91	0.50	0.90
2012.01	32	0.94	0.77	1.00	0.27
2012.02	29	0.59	0.94	0.90	0.53
2013.01	25	0.48	0.92	0.72	0.67
2013.02	29	0.5	0.91	0.70	0.81
2014.01	33	0.63	1.00	0.71	0.63
2014.02	33	0.65	0.85	0.86	0.54
2015.01	26	0.52	1.00	0.44	0.54
2015.02	22	0.89	0.72	0.83	0.45
2016.01	25	0.76	0.87	0.9	0.60
Average	29.83	0.66	0.90	0.80	0.58
SD	4.5	0.18	0.10	0.19	0.17

Table 1. Precision and Recall per semester considering two and three groups.

fact, the precision average is lower when compared to three groups, i.e., the set we retrieve contains many students that pass. In this situation, professors and mentors might waste effort trying to recover students that actually do not need much recovering. On the other hand, with three groups we have a higher precision average, which means professors should give special attention to the students that the strategy retrieves, since 80% of them (on average) tend to fail. Nevertheless, since the recall average is lower than with two groups, we have more failing students not retrieved by the strategy for three groups.

In this context, setting the number of groups to execute our strategy seems to depend on the professors priorities and resources. In case the professor has available time and additional mentors to help her, she can probably use two groups, consisting of a more complete retrieved set (the recall is higher for two groups), even though it contains many false positives. On the other hand, if the professor wants to avoid false positives because of no available time or few mentors to help, she might prefer to use three groups, despite being aware of potential failing students not retrieved by the strategy (many false negatives, lower recall than with two groups).

To compare both groups, we also consider F-Measure. F-Measure represents the harmonic mean between both precision and recall. We achieve 74% and 65% for two and three groups, respectively. So, when considering precision and recall together, two groups perform better with respect to our data.

To better generalize our results, we apply the confidence interval for precision and recall by considering two and three groups. Our sample size is 12 (semesters) and we choose 95% as our confidence level. Because our sample size is small, we use the *t distribution*. The best result we achieve is the recall regarding two groups. In particular, the interval is 6.2%. Therefore, there is a high probability that the population average lies between 83.9% and 96.3%. When considering three groups, from the set of students our strategy points as “likely to fail,” 80% (precision) of the students on average indeed fail. The population average lies between 66.4% and 92.9% . Despite our high precision result, we acknowledge that more metrics can improve the semantics of our clusters.

Number of students. The use of our strategy only makes sense in case the number of students is high. Otherwise, if the number of students is low (e.g., 10 students), the application of our strategy is meaningless, since the professor can identify—also early—

the students that need help. When considering larger classes, the task of predicting the potential failing students becomes more difficult, time consuming, and error prone. In this scenario, applying an automatic approach like our strategy makes sense, being important to decrease the professors effort.

Applying the Strategy versus Applying One Exam. One might wonder what is the difference between applying our strategy and applying one formal exam after 30 days. In the latter, students with bad grades are candidates to fail the course. However, notice that one-day exams represent one specific chance for the student. So, it is not uncommon to confirm that this kind of exam often does not truly evaluate the students knowledge and skills. In this context, applying many exams can minimize such a problem. Nevertheless, when considering the professor perspective, grading many programming exams and reasoning at the same time about the potential failing students are time-consuming and error-prone tasks [Cheang et al. 2003], increasing effort. In contrast, by using online judges, professors might evaluate the students continuously. In addition, online judges allow students to solve many problems, instead of only a couple commonly found in one-day exams. Our strategy relies on data from 30 days, instead of only one. Thus, our precision when predicting might be higher when compared to one-day exam.

Threats to validity. Although our sample has 358 students, our study has homogeneities that might pose threats. For example, the same professor for all the 12 courses (at the same university), the same language used (C language), and the same programming curriculum threat to external validity. In this way, it is difficult to extrapolate the results to other contexts. The use of Huxley threats to internal validity. If the students somehow do not get used to the system, they might feel frustrated. We minimize this threat because the professor introduced Huxley in the very first classes. Besides, mentors helped the students on how to use Huxley. The use of k-means threats conclusion validity. This algorithm places points at random locations. So, two students might appear at different groups in two different executions. However, in our experience, we report only slight differences from one execution to another.

5. Related Work

Previous studies rely on aptitude tests, past academic achievements, and surveys to better understand the student skills. Hughes and McNamara [Hughes and McNamara 2014] introduce a survey based on math and logical questions. Dehnadi and Bornat [Dehnadi and Bornat 2006] observe the mental models that students use when reasoning about assignments and sequence of assignments. Besides previous grades, Bergin and Reilly [Bergin and Reilly 2005] also consider the number of hours per week working at a part-time job and work-style preference. Although the surveys present promising results, applying them is difficult and time consuming, specially when considering large class sizes. In addition, a systematic review showed that large classes tend to have higher failure rates [Watson and Li 2014]. Differently, our strategy automatically predicts students candidates to fail. Since it is automatic, it can reduce the effort of professors.

Estey et al. [Estey et al. 2017] introduce metrics to early identify students struggling during programming courses: baseline, a score of attempted exercises; and trajectory, to quantify changes in programming behavior over subsequent attempts. Despite the different metrics and settings (online judge *versus* programming tool, C *versus* Java, and twelve *versus* four semesters), the precision results are very similar, i.e., 80% *versus* 81%.

Bumbacher et al. [Bumbacher et al. 2013] consider code styles as a predictor of students with the help-seeking characteristic. They use metrics like number of lines of code and number of comments. Like our work, they use k-means to cluster students with similarities in their code styles. They predicted the students that needed help with a precision of 63.6% and a recall of about 71%, i.e., lower numbers than ours.

Similarly to our study, Porter et al. [Porter et al. 2014][Porter and Zingaro 2014] find that the students performance during the first weeks strongly predicts the performance on the final exam. An interesting find is that the most important exercise was a question with respect to `if` statements [Porter et al. 2014]. Also, they found that weeks 3 and 4 may be critical for success in introductory programming courses in general [Porter and Zingaro 2014]. These results are in accordance to our settings: here we cover the first 30 days and gather data until conditional structures.

Watson et. al. [Watson et al. 2013] propose an automatic predictor based upon how a student responds to different types of compilation errors compared to their peers. They apply the approach by using 45 students. The results show that, after approximately 45 days, the predictor accuracy is around 60%. Costa et. al. [Costa et al. 2017] studied the effectiveness of data mining techniques for early prediction of students likely to fail in introductory programming courses. They take into account students activity features obtained from their interactions with a web-based learning environment. The results show that the *Support Vector Machine* technique reaches the highest accuracy. Instead of using data from discussion forums and other educational tools, we rely on an online judge and evaluate problem solving skills during 30 days.

6. Concluding Remarks

This paper presented a strategy able to early predict potential failing students in introductory programming courses. The strategy is automatic, reducing effort and allowing professors to use it in their courses. It consists of the use of an online judge system, the collection of metrics from this system; and the application of a clustering algorithm. We evaluate an instance of our strategy using 12 courses (6 years) with freshmen students. From the group of students our strategy points as "likely to fail," 80% of the students on average indeed failed (precision, three groups). As future work, we intend to evaluate other instances of our strategy with different metrics and with some heterogeneity (e.g., different professors, different programming languages etc). We also intend to apply one instance during the next semesters, making professors aware of the potential failing students and checking to what extent the strategy helps in reducing the high rate of failing students. We also should compare the strategy results against the grades of the first exam.

References

[uva 2014] (2014). UVa Online Judge. `http://uva.onlinejudge.org/`.

[Bergin and Reilly 2005] Bergin, S. and Reilly, R. (2005). Programming: factors that influence success. In *Proceedings of the 36th SIGCSE*, pages 411–415.

[Bumbacher et al. 2013] Bumbacher, E., Sandes, A., Deutsch, A., and Blikstein, P. (2013). Student coding styles as predictors of help-seeking behavior. In *Artificial Intelligence in Education*, pages 856–859.

[Cheang et al. 2003] Cheang, B., Kurnia, A., Lim, A., and Oon, W.-C. (2003). On Automated Grading of Programming Assignments in an Academic Institution. *Computers & Education*, 41(2):121–131.

[Costa et al. 2017] Costa, E., Fonseca, B., Santana, M., Araújo, F., and Rego, J. (2017). Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses. *Computers in Human Behavior*, 73:247–256.

[da Silva et al. 2014] da Silva, T. R., Medeiros, T. J., and da S. Aranha, E. H. (2014). Jogos digitais para ensino e aprendizagem de programação: uma revisão sistemática da literatura. In *Proceedings of the XXV SBIE*, pages 692–702.

[Dehnadi and Bornat 2006] Dehnadi, S. and Bornat, R. (2006). The camel has two humps.

[Estey et al. 2017] Estey, A., Keuning, H., and Coady, Y. (2017). Automatically classifying students in need of support by detecting changes in programming behaviour. In *Proceedings of the 48th SIGCSE*, pages 189–194.

[Harris 2014] Harris, J. (2014). Testing Programming Aptitude in Introductory Programming Courses. *Journal of Computing Sciences in Colleges*, 30(2):149–156.

[Hughes and McNamara 2014] Hughes, J. L. and McNamara, W. J. (2014). IBM Programmer Aptitude Test.

[Jenkins 2002] Jenkins, T. (2002). On the Difficulty of Learning to Program. In *Proceedings of the 3rd annual Conference of LTSN-ICS*, pages 53–58.

[Kinnunen and Malmi 2006] Kinnunen, P. and Malmi, L. (2006). Why Students Drop out CS1 Course? In *Proceedings of the 2nd ICER*, pages 97–108.

[Porter et al. 2013] Porter, L., Guzdial, M., McDowell, C., and Simon, B. (2013). Success in Introductory Programming: What Works? *Communications of the ACM*, 56(8):34–36.

[Porter and Zingaro 2014] Porter, L. and Zingaro, D. (2014). Importance of early performance in cs1: Two conflicting assessment stories. In *Proceedings of the 45th SIGCSE*, pages 295–300.

[Porter et al. 2014] Porter, L., Zingaro, D., and Lister, R. (2014). Predicting student success using fine grain clicker data. In *Proceedings of the 10th ICER*, pages 51–58.

[Santos et al. 2016] Santos, R., Pitangui, C., Vivas, A., and Assis, L. (2016). Análise de trabalhos sobre a aplicação de técnicas de mineração de dados educacionais na previsão de desempenho acadêmico. In *Proceedings of the Workshops CBIE*, pages 960–970.

[Watson and Li 2014] Watson, C. and Li, F. W. B. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 19th ITiCSE*, pages 39–44.

[Watson et al. 2013] Watson, C., Li, F. W. B., and Godwin, J. L. (2013). Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of the 13th ICALT*, pages 319–323.

[Witten and Frank 2005] Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier.

[Yadin 2011] Yadin, A. (2011). Reducing the Dropout Rate in an Introductory Programming Course. *ACM Inroads*, 2(4):71–76.