

**Mock Data Generation and
Intelligent Hinting Integration
Final Report**

COMS W3998 sec:028
Fall 2018

Robby Costales
Alina Ying
Yicheng Shen

December 14th, 2018

1 Mock Data Generation

1.1 Introduction

In the normal data-flow diagram, sage-scratch uses a game snapshot, represented in a JSON file, and sends it to sage-node to be saved as a field to the database. The file is then converted to an SE file using ScratchExtractor. These SE files serve as a basis for statistical data analysis and machine learning so that this intelligence can be applied in the form of a hint to a student in real games when the individual encounters a problem.

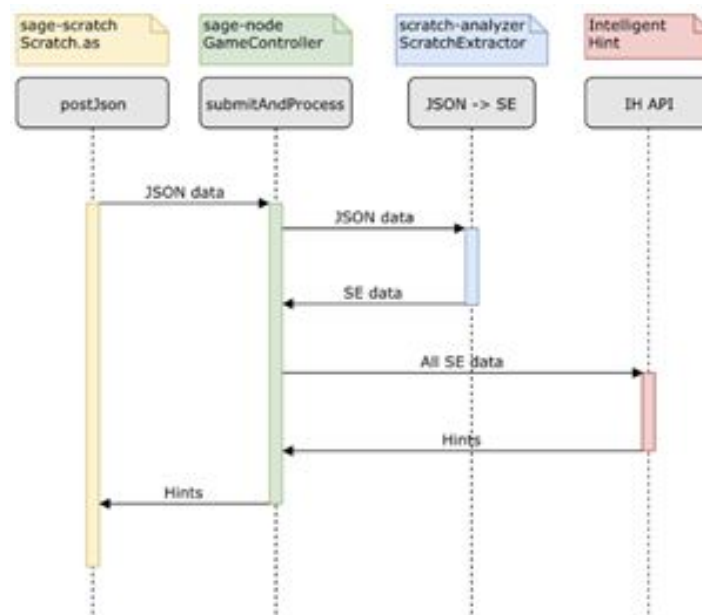


Figure 1

However, it is not easy to generate a large amount of data in short period of time for statistical analysis and generate a training data set for machine learning, as it requires many students to do the actual games, which consumes both time and resources. Instead, we propose to mock generate data from known games to simulate different “failure” cases with specific failure positions and “success” cases where students can get stuck but they eventually finish the games

successfully. These data can supplement the real data collected from the field trials for code algorithm verification and the machine learning algorithms.

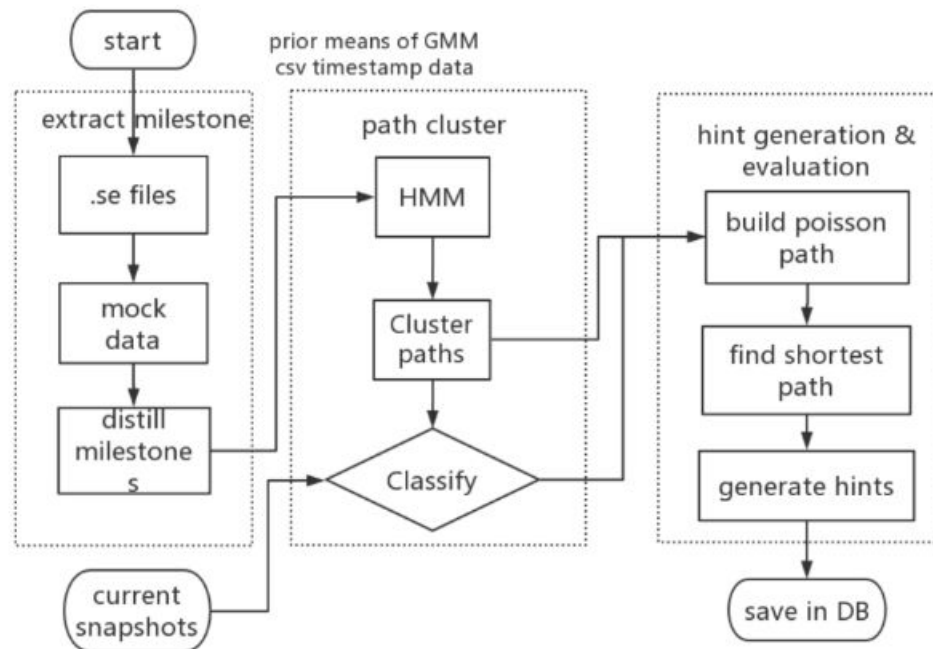


Figure 2

We have collected 21 scratch games with the number of lines of code ranging from 24 (very little) to 4543 (a lot) as a basis of reference. In the generation of the mock data for each game, the student either does not finish the game and thus stops somewhere in the code-flow, or finishes the games but gets stuck (or slows down significantly) at various points when he or she encounters problems.

Game	8 Ball	2014 Christmas Countdown	Aquarium (Piano Cover)	Cubified Logos	Face Morphing	Fizz- A Platformer	Lune of Hippocrates and Lunes of Alhazen
# of lines	56	617	218	55	681	113	282
Game	Make your own star!	Morphing from a Square to Circle	Morphing Greek alphabet Letters using Bezier curves	Morphing Numbers using Bezier curves	Mouse Maze	Music Waves	Neon Art Contest
# of lines	868	349	611	672	135	78	24
Game	Never Forget	Paint v1.0	Palette tool - Color picker!	Rotating Shapes Interactive	Santa's Sleigh	Scratch Blaster 2.21	Scratch Minigames Steve Jobs Timeline
# of lines	172	152	69	149	151	4543	563 638

Figure 3

To start with, an algorithm based on a normal distribution (user can change) determines how many times a student fails to complete the game, and where the failure point is below. This was the original failure-data-generation algorithm; the code is shown below.

```
def mover(url, l):
    if not os.path.isdir(url):
        os.makedirs(url)
    i = 1
    count = 0
    while i < len(l):
        interval = abs(int(random.normal(1,1)))
        t = 0
        while t < interval:
            ouf = open(url+str(count)+'.se', 'w')
            ouf.writelines(l[:i])
            ouf.close()
            t += 1
            count += 1
        i += 1
```

Figure 4

We can simulate different kind of students by adjusting the normal distribution. In this case, there is code for four types of students: mover, stopper, extreme mover, and tinkerer.

```
def stopper(url, l):
    if not os.path.isdir(url):
        os.makedirs(url)
    i = 1
    count = 0
    while i < len(l):
        interval = abs(int(random.normal(5,3)))
        t = 0
        while t < interval:
            outf = open(url+str(count)+'.se', 'w')
            outf.writelines(l[:i])
            outf.close()
            t += 1
            count += 1
        i += 1
```

Figure 5

These four types of students are assumed here, but more types and alternative statistical distributions can be easily added.

To demonstrate the effects and set-up, we pick two games for comparison: a very short game: Neon Art Contest (24 line of codes), and a moderately longer game: Steve Jobs Timeline (638 lines of codes).

```

1  <<Object Stage>>
2      whenGreenFlag
3      doForever
4          doPlaySoundAndWait
5          <<Object drawing1>>
6              whenKeyPressed
7              doRepeat
8                  changeGraphicEffect:by:
9                  nextCostume
10             doRepeat
11                 changeGraphicEffect:by:
12             whenGreenFlag
13             gotoX:y:
14             show
15             lookLike:
16             doForever
17                 doIfElse
18                 =
19                 costumeIndex
20                 gotoX:y:
21                 gotoX:y:
22         <<Object drawing2>>
23         whenGreenFlag
24         hide

```

Figure 6

We generate failure data, where the student “fails” at a random locations, for all the games. We also generate success data, in which the student finishes the program, but is stuck at multiple locations; this was an attempt at creating a time-series-based mock generation capability.





 extremeMover	30 incomplete SE files
 mover	18 incomplete SE files
 stopper	94 incomplete SE files
 tinkerer	116 incomplete SE files

Figure 7

For the failure data, we generate mock data from the short game (24 locs). Each generated SE file is an incomplete file where the code is cut short from the original complete SE file (ideal).





 extremeMover	693 incomplete SE files
 mover	454 incomplete SE files
 stopper	2862 incomplete SE files
 tinkerer	4463 incomplete SE files

Figure 8

We also generate mock data from the moderately-long game (638 loc); likewise, each generated SE file contains an incomplete sequence of code to mimic the student's "failure" at that point. The failure files from this game outnumber the one from the previous game, as expected, since more failures are statistically expected for longer games.

For a successful completion, in which a student does finish the whole code sequence, we do expect the student encounters difficulties from time to time. Thus, we modified the mock data generator code to create a few randomly distributed "stuck" points, with some times of students stuck more often than the other times.

```
def tinkerer (url, l):
    if not os.path.isdir(url):
        os.makedirs(url)
    w=l.copy()
    count=0
    while (count<3):
        index1 = randint(0, len(l)-1)
        # index2 = randint(0, len(temp)-1)
        if ("Object" not in w[index1]):
            w[index1]="stuck #"+str(count)+" "+w[index1]
            count+=1
    ouf = open(url+str(count)+'_new.se', 'w')
    ouf.writelines(w)
    ouf.close()
```

Figure 9

And the results:

```
1  <<Object Stage>>
2      whenGreenFlag
3  stuck #0      doForever
4      doPlaySoundAndWait
5      <<Object drawing1>>
6      whenKeyPressed
7  stuck #3      doRepeat
8      changeGraphicEffect:by:
9      nextCostume
10     doRepeat
11     changeGraphicEffect:by:
12     whenGreenFlag
13     gotoX:y:
14     show
15     lookLike:
16     doForever
17 stuck #2      doIfElse
18     =
19     costumeIndex
20     gotoX:y:
21     gotoX:y:
22 <<Object drawing2>>
23     whenGreenFlag
24     hide
```

Figure 10

These flags of stuck points can easily be picked up by subsequent analysis programs or we can incorporate them into time based events (we can add a time delay at the “stuck” points easily). We can easily generate large amount of variant data with this mock data generator utility for each game, both for failures as well for success cases. Thus, we can finish generating all 21 game mock data in short period of time.

For other future work, we can incorporate the real data and mock data together: the real data can show where students fail or get “stuck” more often. Then, the mock data generator can create statistics around these points to simulate the real cases better.

1.2 Case Study: A Failed Attempt at Mock Data Generation

In parallel with the mock data generation work done in the above section, another effort was made in attempt to incorporate the new method of mock data generation developed by our colleagues working on the Behavior Detection module. This decision was made based on the intention of making further integration more convenient between teams. The source code referred to in this attempt is [scratch-analyzer/intelligentHint/behaviordetection/mockdataBD.py](#).

This attempt did not succeed. Thus, this section is dedicated to providing an alternate source of understanding mockdataBD, an example of the types of obstacles we encountered throughout this semester's development, and potential routes for future work.

Understanding mockdataBD:

A major component and important premise to our work is understanding code. We break down mockdataBD's functionality into these functions:

- mover/stopper/extremeMover/tinkerer:

These four functions, as their names show, create snapshots that mimic what would be generated by a certain type of student. (See past semester reports for briefings on student types.) mockdataBD incorporates timestamps for actions, and records actions in a cumulative action i.e. the latest timestamp would be mapped to all the actions that have been done to that point in time.

Visually, the data looks like this:

```
{
  'Timestamp': hr/min/sec/mo/d/yr
  'Content': a, b, c
}
```

Figure 11

While this is great, we understand that under the current data flow, mock data will be passed into path clustering, which requires the data to be in differential, numeric format

(essentially representing actions using 0s -1s, and 1s; 0 for no move, -1 for undoing a move, and 1 for a move.). This is what we attempted to accomplish.

```
- mockPerProject(n, se):
```

This function creates n mocks for a given game. “se” is a list of games sourced from the completeSE directory. It looks like this:

```
[ [game 1], [game 2], ..., [game n]]  
Where game = se.readlines()
```

Figure 12

(Note that this is the argument “se”, not what the function returns)

The format of one game is a list of strings, created by calling the Python function `readlines()` on a single .se file.

```
- make_mock_data(num):
```

This function returns a list-of-lists, named `project_collections` that stores mock data generated for a single game. Each rows of games 1 to n corresponds to one type of student, and is populated by calling the `mockPerProject` function above, while each game `a1`, `a2`... is generated by the `mover/stopper/extremeMover/tinkerer` functions.

```
[[game a1], [game a2], ..., [game an]  
 [game b1], [game b2], ..., [game bn]  
 ...  
 [game m1], [game m2], ..., [game mn]]
```

Figure 13

What We Want:

Having understood what mockdataBD does, we look at how we want to transform the data. Ideally, we want the data in .csv format, looking like this:

Timestamp	Operation1	Operation2	...	Operation 138
T1	1	0	...	0
T2	0	0	...	1
...
Tn	0	1	...	0

Figure 14

Given that this data format already exists (in [scratch-analyzer/intelligentHint/csv_file](#)), we take a look at the existing method of mock data generation.

The Past Method and What We Tried to Base On:

The mock data generation method created before us is [scratch-analyzer/intelligentHint/mockdataProducer.py](#). It follows a similar structure compared to [mockdataBD.py](#), and we will not go into the greater details of this code. It calls on [scratch-analyzer/intelligentHint/server_interface/read_se.py](#), within this file is the relevant function we tried to transfer over for [mockdataBD.py](#), `convert_se_to_csv_diff`.

However, the main function, as shown in the image below, calls on `convert_se_to_csv` instead, which is a similar algorithm to `convert_se_to_csv_diff`, except that the one-hot-encoded actions are now cumulative instead of differential (i.e. instead only representing moves by 0, -1, 1, these values are

cumulatively added at each timestep.). In the main function, a file path is passed into `convert_se_to_csv`: `../mockSE/success/i/j`. Unfortunately, this file path does not exist.

```
#Reads SE file, converts it to CSV file, creates CSV file with all the operations listed
#Generates mock success and failure data
#Create 4 x 25 mock data
if __name__ == "__main__":
    # read_se("sage-frontend/machine_learning/sample_data/0/CTG-22.se")
    # convert_se_to_csv("")
    # calculate("completeSE")
    # for i in range(5):
    #     generate_mock_se_success("Face Morphing.se", i + 1)
    #     generate_mock_se_failure("Face Morphing.se", i + 1)

    operation = read_operation("operations.csv")
    # input_file = "../mockData/failure/1/1/100.se"
    # print(read_se(input_file, operation))

    for i in range(1, 5):
        for j in range(25):
            convert_se_to_csv("../mockSE/success/" + str(i) + "/" + str(j) + "/", i, j)
```

Figure 15: Screenshot of main function in `read_se.py`

We trace back to the `convert_se_to_csv` function itself. The path mentioned above fills in the `file_path` argument for the function (with `str(i)` appended), which is also called by the `read_se` function that populates the variable `list2`. `list2` is crucial because it is supposed to entail a given `.se` file, which is used for one-hot-encoding to populate the `.csv`. However, since `file_path` does not exist, we cannot know precisely what data format `list2` is in, and thus cannot confirm if this method of one-hot-encoding could be utilized.

```

#Converts the SE file to CSV file, with cumulative datapoints
def convert_se_to_csv(file_path, student_type, student_id):
    diff = []
    n = len(glob.glob(file_path + '/*.se'))
    # print(file_path)
    # print(n)
    one_hot_data = np.zeros((n, len(operation)))
    for i in range(n):
        list2 = read_se(os.path.join(file_path, str(i) + ".se"))
        for opt in list2:
            one_hot_data[i][opt] += 1

    one_hot_data = one_hot_data.astype(int)
    df = pd.DataFrame(one_hot_data)

    try:
        os.mkdir("../csv_file/original/success/" + str(student_type))
    except OSError as e:
        # print(e)
        pass

    df.to_csv("../csv_file/original/success/" + str(student_type) + "/output" + str(student_id) + ".csv")

```

Figure 16: Screenshot of convert_se_to_csv function in read_se.py

Possible Theory and Future Work:

The issue illustrated above showcases the type of obstacles we faced and our motivation to amend/comment/explain the existing code base. For the above issue in particular, our current hypothesis is the variable `n` in `convert_se_to_csv` might denote the number of lines in an `.se` file, instead of the number of `.se` files in the given path -- in this way, the `one_hot_data` variable would make sense given that it should supposedly have as many rows as a given `.se` file.

Future work in this direction could entail both implementing this theory and developing a brand-new method of one-hot encoding for the previous `project_collections` data format instead of trying to translate the current method. In rough strokes, the new method would involve separating the timestamps and actions, and then mapping the actions to the columns in `operations.csv`, which includes all possible actions in scratch, for 0/1 encoding.

2 Intelligent Hinting Polishing and Integration

2.1 Introduction

Last semester, a group of students tackled the problem of generating more accurate hints than the previous system that SAGE used. The approach utilizes a number of machine learning algorithms, and the diagram below (created last semester) highlights the different stages of the model at a high level.

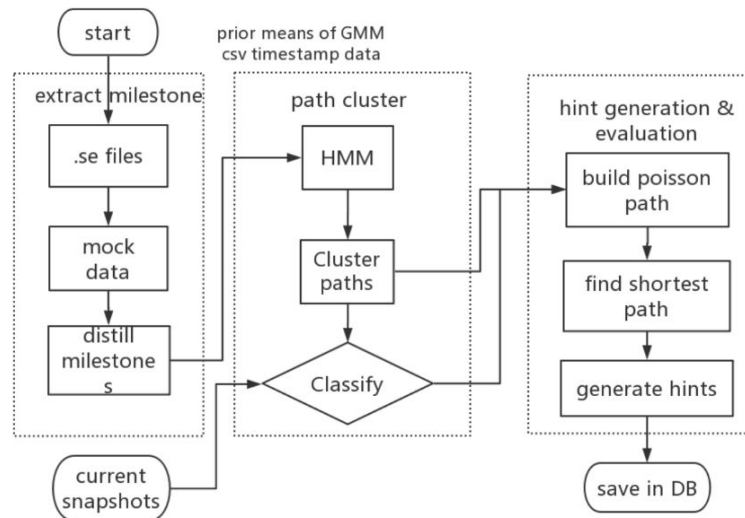


Figure 17. Previous project architecture diagram

This semester, no changes were made to the algorithms themselves. More details on the algorithms and the reasonings behind the approach can be found in the Intelligent Hinting final report for Spring 2018 [1].

At the start of the semester, some of the beginnings of integration appeared to have begun taking place for this new architecture. In the report [1], the various HTTP requests that Intelligent Hinting could receive through the new API were highlighted. However all of the requests and

underlying algorithmic functionality were still very much in a testing phase, meaning all of the algorithms were dependent on the local file structure, and only one game was supported and its information was hard-coded throughout.

The main focuses for the improvement of the intelligent hinting system this semester were to 1) run and perform surface level tests on the functionality of the inherited code, 2) separate algorithms from the flow of data and generalize functionality to any puzzle / given snapshot, 3) integrate functionality with SAGE node, and be able to accept API requests, and 4) document code and file structure so that future work can be done more readily.

2.2 Semester Progress

Testing Code

The first set of errors that arose when attempting to execute portions of the inherited code were ones presumably due to different development environments. One example of this was the simply fix of importing local modules explicitly, because previous work assumed the use of an IDE that would do this implicitly.

The second kind of errors were ones that dealt with the underlying functionality of the algorithms. Even once the algorithms appeared to run without error, there was still unexpected behavior from the `get_hints()` functionality. The most significant unexpected behavior caused a completely random hint to be generated each time, and to hunt it down required a very careful step through each line of the inherited code. In addition to the eventual fix, some light documentation and polishing of the code also arose from this point out of the need to understand each piece of code clearly.

Generalizing Functionality

Two main modifications needed to take place to the code so that it was fully generalized and ready for the requests. The first was to modify the code so that any given puzzle would be supported. Previously the code had only supported the hint generation for one particular puzzle, and this dependence was reflected all throughout the algorithmic functionality--which actually relates the second major modification. The second modification was to entirely separate the algorithmic functionality of intelligent hinting from the saving and loading of data. These two tasks very much related and were completed in parallel.

Supporting API Requests

Once the code was fully generalized, we coordinated with the SAGE Node team that sends the actual request to the Intelligent Hint API. While a handful of requests must be supported in the future, particularly ones to build intermediate files based on mock / real data, the main focus this semester was accept a get_hints POST request. The decided format of the request can be seen below:

```
{
  "seFiles": [
    {
      "content": "<<Object Stage>>\n",
      "timestamp": 1542350500031
    },
    {
      "content": "<<Object Stage>>\n                <<Object Spritel>>\n",
      "timestamp": 1542350509935
    },
    {
      "content": "<<Object Stage>>\n                <<Object Spritel>>\nwhenKeyPressed\n",
      "timestamp": 1542350515652
    }
  ],
}
```



```
"info": {  
  "studentID": "stu2",  
  "gameID": "game2",  
  "objectiveID": "obj2",  
  "puzzleID": "face_morphing",  
  "studentType": 2  
}  
}
```

The *seFiles* contains a list of se file snapshots for a certain user for a given puzzle. We are currently only using the most recent snapshot for a given user, so we only care about the last element in the array, which is why the rest are greyed out above.

info.puzzleID contains a string corresponding to a unique complete se file that represents the complete solution to a given Scratch puzzle. *info.studentType* is (already known) behavior type for the student who is sending the hint request.

The rest of the attributes in *info* are currently unused and are greyed out for now, but will be necessary for future modifications to how data is stored.

Documentation and Wrap Up

In addition to the light documentation that took place during the testing phase of the semester, some more intentional documentation efforts also were completed by the end. First, most of the main functions in the *scratch-analyzer/intelligentHint/server_interface* directory now contain docstrings that explain the parameters and purpose of each. Additionally, a *README.md* file has been created in *scratch-analyzer/intelligentHint* which contains helpful information on how to run the code, as well as the purpose of each file in the directory. This *README.md* file will also contain a todo list that is meant to be continuously updated in the future. The todo list serves a unique purpose to that of the backlog. The backlog contains high level ‘action’ tasks, but there is a layer between what needs to be done at a high level, and what can be done at a low level. The

todo list can serve as a place for all code-level issues (including bugs_ and tasks to be noted, so that current and future developers all be up to date. This allows for developers to push working code to the *development* branch on *scratch-analyzer*, but also note any assumptions, minor bugs, or future plans. Overall the *README.md* is meant to help bridge the gap between the high level documentation and reports, and the code itself.

2.3 Future work

Data Storage

While the algorithmic functionality is now separated from the data flow, and the code is generalized to accept any valid request from SAGE Node, the storage of the mock data files, and intermediate files produced by some of the algorithms are currently being stored in the local filesystem. In the future, all of the local directories that *data_flow.py* refers to should be replaced with locations in the mlab database, or some other remote filesystem. This may require a more general discussion between Intelligent Hint, Behavior Detection, and SAGE Node teams so that storage / id-ing / naming conventions are held consistent throughout.

Periodic Data File Generation

As mentioned previously (see Figure 17 for a visual reference), the hinting algorithm happens in many stages. Some stages should not, and are currently not performed each time a hint is requested, as this would be very time consuming and is simply not necessary. However, periodic updates to these files are necessary as new puzzles are created and more student data is collected (or mock data is generated / updated). The current vision we have is that a certain API request will be dedicated to running maintenance on these files, and the responsibility for calling the function will be delegated to the main SAGE functionality.

Algorithm Effectiveness

One task that was not accomplished this semester was to test the effectiveness of the algorithm. This is currently difficult because we only currently have mock data to both generate / test our models. Actual student data will be required to fully test the effectiveness of the model. However, some assumptions for ways to improve the algorithm at its current state can be made, for instance, taking multiple previous snapshots into consideration when generating a hint.

3 References

[1] Yi Ding, Weimeng Luo, Junyu Zhang. *Intelligent Hinting 1.1 Final Report*. 2018.