Intelligent Hinting 1.1
Final Report

Yi Ding, Weimeng Luo, Junyu Zhang

May 7, 2018

Contents

1 Introduction

Intelligent hinting is an important part for the Social Addictive Gameful Engineering (SAGE) project. This project aims to improve the intelligent hints generation approach by implement different kind of machine learning algorithms, and then build an evaluation system for teachers to see the validation of the hints. The architecture of our project is shown figure 1, we have three main blocks in this chapter: 1. moke data generation. 2. Hidden-Markov Model(HMM); in order to model students' learning path; 3. hint generation.

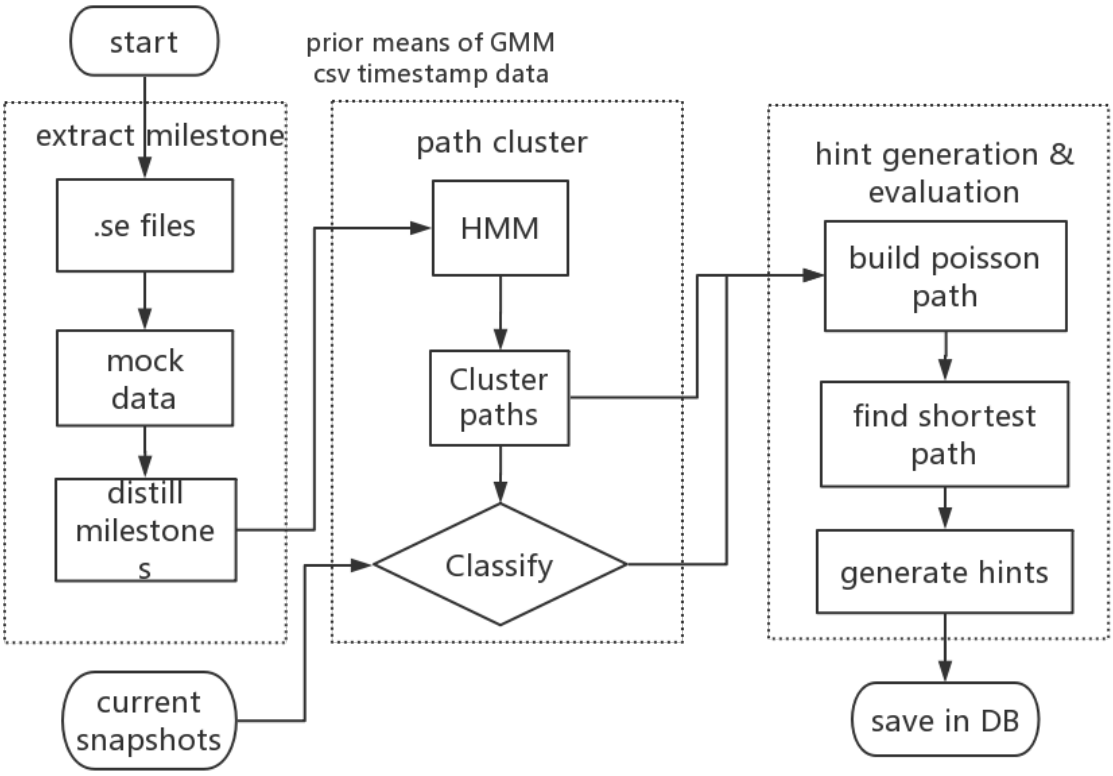

Figure 1: Project Architecture Diagram

The previous hinting generation method in SAGE gives users hints according to their learning behavior, the current state and the sequence frequency in hinting pool to generate hints.[2] The limitation is that time-series information is lost during feature extraction. Considering such limitations, the motivation for the project is first implement Hidden-Markov Model for time series data to model students' learning path rather than the learning behavior.

Another important thing is that after cluster students according to their learning path,

how to generate hints more precisely and friendly for students to choose the hints. This is useful for teacher to know whether your intelligent system works on s/he students, and which hinting method is more helpful to students. After implement HMM, we use the cluster result, student's type, and the current snapshots as the input to generate hints. This part is based on Poisson Path and Dikjstra's Algorithm to find the shortest path on the learning graph. Finally, we think about how to encapsulate these methods in RESTful API. We choose Python Flask framework to host our code, which is easy for front-end to call and for the future back-end development groups to maintain.

In the second chapter, we introduce the related work of this project, the first part is the summary of the previous hinting generation approach. This could help us compare different methods and design a reasonable hinting evaluation system; the second part focuses on the implementation; The third part is about limitations and assumptions and then we discuss what could be improved in the future in the last part.

2 Related Work

2.1 Existing Hinting System in SAGE

Current Hinting System in SAGE already embeds several machine learning methods to extract project features and also models student behaviors according to their history data. It first uses K-medoid algorithm to cluster the user snapshots for a specific project. The cluster's center is called project "milestones". Then users behavior is classified into different types according to their paths compared with the milestones. For each type user, the hinting system implements sequential pattern mining to select the next step with the highest possibility.

The processing method is shown as below:

1. Feature extraction: The system first parses the .se file to transform the snapshots to a set of trigrams (consists of 3 blocks), then extracts and clusters the features according to these trigrams, finally matches k 1 centroids to k 1 .se snapshots.

2. Behavior classification: After feature extraction, the milestones of each project is defined. For each student that participates in this project, n snapshots with equal time interval are used as the input. For each snapshot, it implements Needleman-Wunsch algorithm to calculate the similarity between it and the k 1 milestones. Several machine learning methods have been utilized in the system to classify user behaviors.

3. Hinting generation: In order to use sequential pattern mining algorithm, it first transforms the snapshots into binomial feature vectors. The training data is collected from students who successfully finish the project (do this processing for each type of students after 2.). Hints are generated according to Generalized Sequential Pattern (GSP). The hinting generation frequency is based on what kind of the user behavior is.

2.2 HMM for hints generation

This paper [1] mainly proposed that (1) how to build models of student's progress pathways using machine learning methods (2) how to autonomously extract features of a student's progress (3) how to predict student's future performance by their grades.

Student's coding progress(path) is modeled using a Hidden Markov Model(HMM). The HMM model they used that at each move assumes that a student is in the state of some high-level "milestone". We can observe the source code of each snapshot as a noisy output of a latent variable - milestone. For example, milestones could be "student has just started" or "student got frustrated". HMM is relevant to our problem because student's coding progress is generally incremental and the hidden state(milestone) is more likely to be independent to one's previous time step.

HMM makes Markov assumption that the future state is independent to the past state given the current state of a student. Markov assumption is not entirely correct, but this assumption helps to simplify student's progress, find patterns and make predictions of future state.

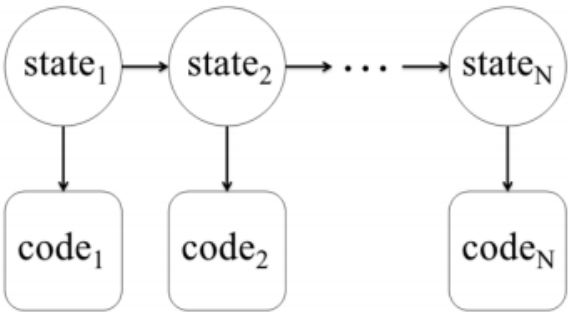

Figure 2: HMM clustering method

2.3 Alternative hinting method

This paper [2] focus on how to autonomously generating hints. This process can be decoupled into two tasks: 1. Deciding for any learner what partial solution an educational expert would

suggest they transition to next. 2. Choosing how to communicate that information to the student such that they improve on a learning objective.

This paper uses Code.org, which is the largest MOOC to date, as a case study. They define the problem as follows: At each point in time, the student's current work can be represented by a partial solution $\phi \in S$, where S is the set of all possible responses. Students will traverse through a series $T = \phi_0, \phi_1, \ldots, \phi_n$ of partial solutions from step 0. They try to solve the first problem by what they call a Problem Solving Policy (PSP), which is a decision for any partial solution as to what next partial solution a student should take (from expert view). They develop a family of algorithms to predict the way how an expert would encourage a student to make forward progress.

	P_A Accuracy	P_B Accuracy
Algorithm		
Random	8.2%	4.3%
Shortest Path	49.5%	33.6%
Min Time	67.2%	42.2%
Rivers Policy†	72.9%	78.2%
Expected Success	77.9%	56.2%
MDP†	80.5%	47.6%
Most Common Next	81.1%	49.0%
Static Analysis†	86.3%	-
Most Popular Path	88.3%	52.8%
Ability Model	88.4%	63.3%
Independent Probable Path ‡	**95.5%**	**83.3%**
Poisson Path ‡	**95.9%**	**84.6%**
Variation		
Unconditioned on Success	72.2%	68.3%
No Interpolation	86.8%	70.5%
Allow Cycles	94.3%	82.0%
Poisson Interpolation	94.6 %	83.2%

Figure 3: Comparison between different methods

They tested three classes of solutions:

Desirable Path algorithms, previously proposed algorithms and vanilla baseline algorithms. Desirable Path Algorithms: Poisson Path, Independent Probable Path Baseline Algorithms: Markov Decision Problem, Rivers Policy, Static Analysis, Most Likely Next, Most Common Path, Expected Success, Min Time, Ability Model

The results shows that Poisson Path policy had the highest accuracy for both two problems.

2.4 Conditions of learning in novice programmers

This paper [3] describes several different behaviors of novice programmers. Researchers conducted a series of clinical studies of novice programmers at Harvard Graduate School of Education, and reflected on the why some youngsters learned much better than others.

They suggested that underlying reason for their difference is their different patterns of learning. They further classified students into several different types based on their learning patterns: stoppers who will give up their learning procedure easily and just stop long enough to appear stuck, movers who consistently try different approaches and never seem to get stuck, extreme movers who move too fast without reflection on their previous ideas. And they found out that students often tend to adapt the approach of tinkering, which means they will consistently try new ideas and make small changes to their codes.

3 Implementation

3.1 Code Repository

Table 1: Code Repository

Project aspect	Repository location	Branch name
Mock data generation	http://dev.cu-sage.org:8080/tfs/SAGE/SAGE/	
HMM & Clustering	_git/scratch-analyzer?path=%2FintelligentHint&	development
Hint generation	version=GBdevelopment&_a=contents & development	

3.2 Technology used

3.2.1 Pandas

Pandas [4] is an open source, BSD-licensed library which provides some high-performance, easy-to-use data structures. It is also a powerful data analysis tool for the Python programming language.

3.2.2 Scikit-learn

Scikit-learn [5] is a free software machine learning library for the Python programming language. It provides the features including various kinds of classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to cooperate with the Python numerical and scientific

libraries NumPy and SciPy. It is a powerful and widely used tool for data mining and data analysis.

3.2.3 hmmlearn

Hmmlearn (`https://hmmlearn.readthedocs.io/en/latest/`) is a library which provides the implementation of the Hidden Markov Models (HMMs).

3.3 Mock Data

As we don't have any real data for students, we need to generate some mock data to build our hint generating system. This part of work need two phases: first, we generate some fake projects from the original complete .se file; then, we imitate the behaviors of different types of students. data.

1. We can choose any complete .se files provided by SAGE project. We choose one project from completeSE directory: completeSE/Face Morphing.se to generate the mock. We use read_se.py to generate some fake projects, these projects are classified into two types: success, failure. They are located in mockSE.

    - Success: We shuffle the order of some random blocks to generate fake successful .se data from completeSE.
    - Failure: We delete some blocks to generate fake failure se data from completeSE.

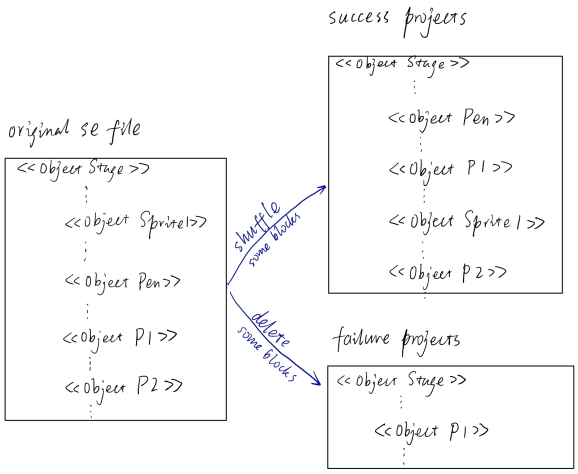

Figure 4: Mock projects

```
# shuffle some blocks in the compleseSE file
def generate_mock_se_success(input_file, n):
    with open("completeSE/" + input_file, encoding='utf-8') as f:
        content = f.readlines()

    blocks = calculate_blocks(content)
    # the beginning block
    begin = random.randint(1, len(blocks) - 5)
    old_index = list(range(begin, begin + 5))
    new_index = old_index[:]
    np.random.shuffle(new_index)
    new_content = content[:]
    # shuffle the blocks between begin, begin + 5
    for t in range(5):
        new_content[blocks[old_index[t]][0]:blocks[old_index[t]][1]] \
            = content[blocks[new_index[t]][0]:blocks[new_index[t]][1]]

    output_file = "mockSE/success/" + input_file.split('.')[0] + str(n
        ) + "." + input_file.split('.')[1]
    fo = open(output_file, "w")
    fo.writelines(new_content)
```

2. From mockSE files, we use mockdataProducer.py to imitate the operations of four kinds of students: extremeMover(1), mover(2), stopper(3), tinker(4).

   - **Extreme movers(1)**

     Extreme movers are students who move too fast. They often write some code without long consideration, clearly these codes will not work. They also don't learn much experience from their failure, and will some retry unworkable approaches. As a fact, they often go round in circles. Typically, extreme movers don't do well in programming.

   - **Movers(2)**

     Movers are students that consistently try one idea after another, writing or modifying their code and testing it, never stopping long enough to appear stuck. Generally, movers can do well in their programming task, making progress on a problem and carrying it to a successful completion of the project.

   - **Stoppers(3)**

     Stoppers are students who try the simplest expedient and then just stop. They

write some code, and stop there long to appear stuck. Clearly, this type of students also won't do well.

- **Tinkers(4)**
  Tinkerss are students who uses an approach called tinkering. They try to solve the programming problem by writing some code and making small changes to it. In a word, they often move forward and then backward consistently. Clearly, this approach also don't perform well.

Each type students have 25 mock data.

3. We then convert the .se file to .csv file according to the look up table saved in **operation.csv**. Each **.csv** file represent the timestamps of a student completing one project. We also process the raw data into two types of .csv files:

   - original: Represent the original snapshots of students.
   - differential: The differences between two neighbored snapshots. 1 represent append operation, -1 represent delete operation.

```
def convert_se_to_csv_diff(file_path, student_type, student_id):
    '''
    Convert se file to csv file. 1 represent append operation, -1
       represent delete operation
    '''
    diff = []
    n = len(glob.glob(file_path + '/*.se'))
    print(glob.glob('*.se'))
    one_hot_data = np.zeros((n - 1, len(operation)))
    for i in range(n - 1):
        list1 = read_se(os.path.join(file_path, str(i) + ".se"))
        list2 = read_se(os.path.join(file_path, str(i + 1) + ".se"))
        operations_append = list(set(list2) - set(list1))
        operations_delete = list(set(list1) - set(list2))
        for opt in operations_append:
            one_hot_data[i][opt] = 1
        for opt in operations_delete:
            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/differential/success/" + str(student_type))
    except OSError as e:
        print(e)

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

3.4 Probabilistic Student Path Clustering

In this section, we'll present how we cluster student paths (student's history snapshots indicating how students developed the program over time)[6] using a probabilistic approach (hidden Markov model). Assume that we have observed the source code of each snapshot, which can be treated as a noisy output of a latent variable (hidden state) through a random process. Students' hidden state are transferred based on first-order Markovian assumption

$$p(state_{t+1}|state_t, state_{t-1}, ..., state_1) = p(state_{t+1}|state_t)$$

and each hidden state can be interpreted as a high-level state of the student's progress. Because the number of different snapshots is massive, we can assume that we have K milestone snapshots and each observed snapshot is generated according to some milestone snapshot. For example, milestones could be "student has just started" or "student got frustrated". HMMs are parameterized by the probabilities of a student going from one state to another and the probability that a given snapshot came from a particular milestone. We may cluster the student's path according to these probabilities.

We first compute the set of high-level states using the original csv files with K-Medoids method. This method will distill K *.se* files as the center for each cluster. These centers are the milestones of the project, and we utilized the clustered result as the prior means and prior probabilities of multivariate Gaussian Model. And then we use the Expectation-Maximization(EM) algorithm to compute the transition and emission probabilities.

Now we have built HMMs for each student and calculate student's pairwise distance under HMM's log-likelihood space for clustering. For every student pair i, j, distance is calculated as

$$dist_{i,j} = score_i(j) + score_j(i)$$

and $score_i(j)$ is the log-likelihood score for model i and sequence j. Finally, we use spectral clustering with our distance matrix to cluster all student paths.

3.4.1 Distill and Data Preprocessing

To formalize our problem, let's assume that our input is preprocessed .se file describing student snapshot at some specific time step. Those .se files could be fetched through SAGE node databases using our sage node interface.

Input: .se files with student id and time stamp.

Output: A list of milestones $[m_1, m_2, ..., m_k]$ and preprocessed student paths.

Here's how an example .se file looks like:

```
<<Object Stage>>
        <<Object Sprite6>>
        <<Object Sprite2>>
                whenGreenFlag
                doForever
                        doIf
                        =
                        stringLength:
                        -
                        computeFunction:of:
                        timestamp
                setVar:to:
...
```

First we will encode each .se file into a vector. We identify each block name and replace them with corresponding index (138 different blocks in total) for each block. Then we count the occurrence of each block and encode them into a $(138, 1)$ vector storing the number of each block in a snapshot. This simplified representation of raw .se file comes from Bag-of-Words model in natural language processing.

Next we'll store each student sequenced snapshots (path) into a .csv file. Snapshot vectors are stored in each rows of .csv files according to their time sequenced order. Those .csv files could be accessed and loaded into python objects using our Snapshot class.

Our snapshot object is defined as follow:

```
class Snapshot:
    def __init__(self, group, order, success, csv_file):
        # snapshot groups, used as ground-truth for evaluating clustering
           quality
        self.group = group
        # snapshot identification (student number)
        self.order = order
        # success / failure path tag
```

```
8           self.success = success
9           # csv file path
10          self.csv_file = csv_file
11          # snapshot vector
12          self.snapshot = pd.read_csv(csv_file).values[:,1:]
```

K-Medoids Now we have encoded all snapshots in a project into vectors, the next thing we want to do is calculating the median snapshot for HMM's milestone.

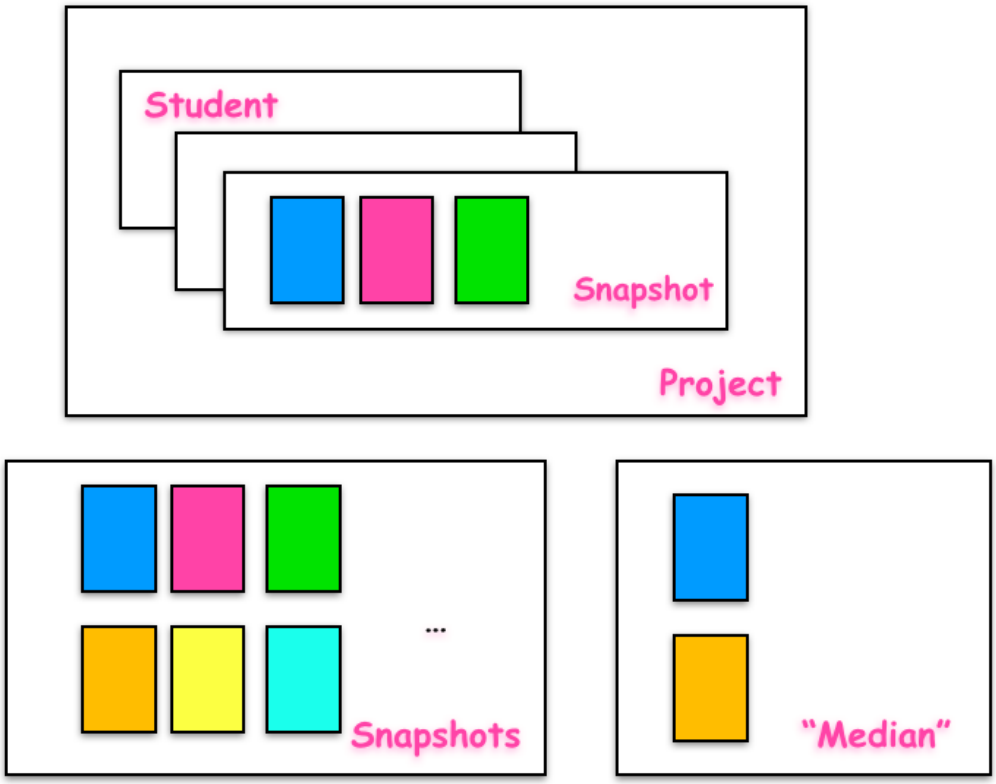

Figure 5: Example of snapshot distilling

Each project is composed of student paths, and each student path is composed of many snapshots. We first collect all snapshots in a project, then calculate some median snapshots amongst them.

Shown in the figure above, we try to use K-Medoids clustering algorithm[7] to distill the milestones from a project. Previous semester also use the algorithm, but we have different usage for the result. The milestone means that students are often stuck on this status (it takes students a long time around this cluster, and the milestone is the center of this cluster), so K-Medoid is suitable for this circumstance.

We use these milestones as the prior means in the hidden state of HMM. This process can decrease the calculate complexity during HMM. We separate the Expectation Maximization (EM) algorithm into two step: learning the hidden states and learning the transition matrix and emission matrix.

Because we only care about snapshot changes in the HMM part, differentiated snapshots is practically applied as the final stage before we finally feed our data into HMM. In this stage, we calculate the difference of two neighbouring snapshot. Because our Gaussian HMM processes continuous variables instead of discrete 0/1 vectors, we apply principal component analysis (PCA) to our processed vectors and project them into a lower dimension space as the final input to our HMM model.

```
data = np.concatenate(np.array([x.snapshot for x in snaps]))

from sklearn.decomposition import PCA
pca = PCA(n_components=20)
principalComponents = pca.fit_transform(data)

print('PCA dimension=', principalComponents.shape)

sampled = principalComponents[::20, :]
```

3.4.2 Hidden Markov Model

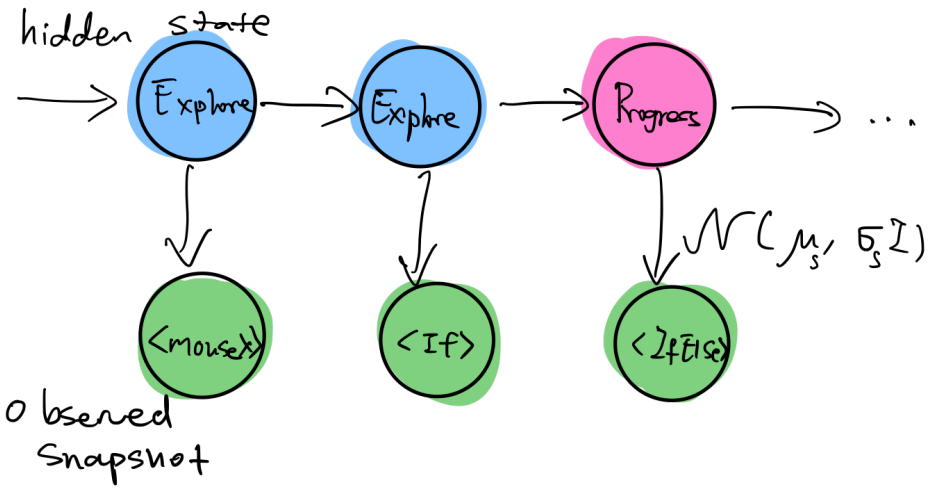

Figure 6: HMM clustering method

Assume that we have observed data and hidden states:

Observed data(snapshots): $x_1, x_2, ..., x_t$

Hidden states: $s_1, s_2, ..., s_t$

Model parameters: θ

Figure 7: Example of snapshot and hidden state sequence

Observed snapshots and its corresponding hidden states. The advantage of using a probabilistic model is that we can infer a interpretable high-level state of every observable student snapshot, then clustering them under a HMM space.

The probability of the occurrence of such sequence is defined using likelihood function

$$L(\theta) = \sum \log p(x_1, ...x_t, s_1, ..., s_t; \theta)$$

Maximum-likelihood estimates of transition probability is

$$t(s'|s) = \frac{\sum_{i=1}^{n} count(s \to s')}{\sum_{i=1}^{n} \sum_{s'} count(s \to s')}$$

Maximum-likelihood estimates of emission probability is

$$e(x|s) = \frac{\sum_{i=1}^{n} count(s \to x)}{\sum_{i=1}^{n} \sum_{s'} count(s \to x)}$$

But if we don't have the real hidden states sequences, the likelihood function could also converge to a local maximum using Expectation-Maximization(EM) algorithm.

Define $\alpha(j, s)$ to be the sum of of probabilities of all paths ending in state s at position j and $\beta(j, s)$ to be be the sum of of probabilities of all paths starting in state s at position j.

$$\alpha(1, s) = t(s)e(x_1|s)$$

$$\alpha(j, s) = \sum_{s'} \alpha(j-1, s') \times t(s|s') \times e(x_j|s)$$

$$\beta(m, s) = t(s)e(x_1|s)$$

$$\beta(j, s) = \sum_{s'} \beta(j+1, s') \times t(s|s') \times e(x_{j+1}|s)$$

Then we can represent the conditional probability of any state at any position

$$p(S_j = s|x_1, ..., x_m; \theta) = \frac{\alpha(j, s) \times \beta(j, s)}{\sum_s \alpha(m, s)}$$

This is called Forward-Backward algorithm for computing forward and backward probabilities using dynamic programming as the Expectation stage (E-step) in EM algorithm. For Maximization stage (M-step) in EM algorithm we update the transition and emission probabilities using θ_{t-1} calculated in E-step. Finally the EM algorithm will converge to a local maximum of the likelihood function.

3.4.3 Path Clustering

Now we can compute each student's HMM parameters. We use the project's milestone from our distillation as the prior mean of our Gaussian HMM and run inference on every student in the project. An straight-forward way to evaluate similarity between student i and sequence j is that using HMM likelihood function

$$score_i(j) = L(j; \theta_i)$$

And symmetric pairwise student distance can also be defined as

$$dist(i, j) = score_i(j) + score_j(i)$$

Once we have distance matrix, we can cluster student paths using spectral clustering.

```
def log_likelihood_space_metric(model, X):
    return model.score(X)

def symetric_score(i, j):
    score = log_likelihood_space_metric(hmm_models[i], principalComponents
        [snaps[j].start_idx : snaps[j].end_idx])
    score += log_likelihood_space_metric(hmm_models[j],
        principalComponents[snaps[i].start_idx : snaps[i].end_idx])
    return score

dist = np.zeros((num_routes, num_routes))
for i in range(num_routes):
    for j in range(num_routes):
        dist[i, j] = symetric_score(i, j)
sc = SpectralClustering(n_clusters=4, affinity='precomputed')
xmin, xmax = dist.min(), dist.max()
dist = (dist - xmin) / (xmax - xmin)
y_pred = sc.fit_predict(dist)
```

Here's our result of clustering using our generated mock data.

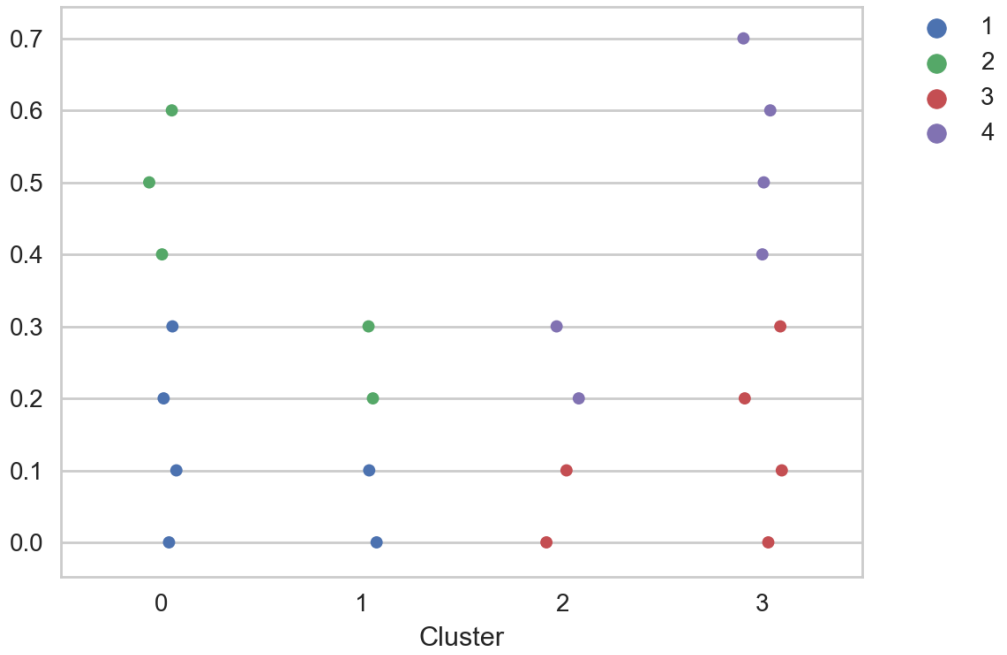

Figure 8: Result of HMM clustering

1: Mover; 2: Extreme Mover; 3: Stopper; 4: Tinker. Movers and extreme movers are identified to be more similar in our HMM clustering algorithm.

Currently we're using single snapshot to build N hidden Markov models. However, using just one sequence to build a model is prune to get noisy parameters of student model. Ideally we want to build K hidden Markov models instead of N so that we could take advantage of more training data for each model (N/K on average instead of 1). More discussions on how to build K HMMs after clustering N HMMs into K groups could be found in "Clustering Sequences with Hidden Markov Models" [6].

3.5 Hint Generation and HMM Evaluation

We change the rule for hint generation in order to quantify the performance of the cluster method. Borrowed the idea from [2], we choose Poisson Path. The Poisson process is widely used to model random points in time and space, such as the times of radioactive emissions, the arrival times of customers at a service center, and the positions of flaws in a piece of material. We regard the learning process as a Poisson Process, so this hint generation method is called Poisson Path. Specifically, the path from a node s to a terminal which has the least expected time required, to generate the hints. We model the students' data for a specific type and

a specific project as a graph. In each graph, the node represent a partial solution (all the snapshots came up by one student on the way to the final success snapshots are called partial solution). The weights for each edge in the graph depends on the number of students that comes up with this learning path. The equation to assign weights for each node is shown as below:

$$\gamma(s) = \arg\min_{p \in Z(s)} \sum \frac{1}{\lambda_x}$$

Where $Z(s)$ are all the paths-to-solution from s, λx is the number of times partial solution x in successful student pool. The code to reweight the graph is shown as below:

```
def poi_reweight(graph):
    # Merge duplicate edges
    gprime = Graph()
    gprime.nodes = graph.nodes
    for node in graph.nodes:
        se = defaultdict(lambda: 0)
        for v, w in graph.edges[node]:
            se[v] += w
        # Reweight edges
        for v in se:
            se[v] = 1.0 / se[v]
            print(se[v])
        gprime.edges[node] = list(se.items())
    return gprime
```

To find the path, we create a virtual node called 'final', and start from the current snapshot. Since the nodes saved in the graph is in a set, we need to transform the integer vector into strings to make them hashable.

The next step is run Dijkstra algorithm, it take $O(n^2)$ to find the shortest path between the current snaphsot and the correct solution. Dijkstra's algorithm initially marks the distance to every other intersection on the graph as infinity. The current intersection will be the starting point, and the distance to it will be zero during the first iteration. Then the current intersection will be a closest unvisited intersection to the starting node.

During the calculating process we save each node as the key in a new dict and the predecessor node as the value. Then search from the final node to the current snapshot to generate the path, and then the last node is our hint. We compare the difference between the hint and the current snapshot, finding the different blocks, mapping their dimension to

their original name, and then choose the top 3 block names to return:

```
    def get_hints(code, type):
    print("current_snapshot:{0}".format(code))

    # get block name
    operation = read_operation(os.path.abspath(os.path.join(os.path.
        dirname(__file__)))+"/operations.csv")
    pool = [key for key, value in operation.items()]

    #generate hints
    hint, distance = generate_hints(code, type)
    print("hint_snapshot:{0}".format(hint))
    hint1 = hint.split(',')
    code1 = code.split(',')
    result = []

# calculate the difference between current snapshot and the hints
    for idx in range(0, len(code1)):
        if hint1[idx] != code1[idx]:
            result.append((int(hint1[idx]), pool[idx]))

# return the top 3 blocks if there are more than three dimension different
    .
    result.sort()
    blocks = [r[1] for r in result[-1:]]
    print("distance to the final snapshot: %d" % distance)
    print("generated hint: {0}".format(blocks))

    return blocks
```

3.6 Interface for Front-end

We derived an easy-to-use interface for Frontend using Python Flask Framework. Since we only work on back-end, we simulate the behavior of front-end in Postman software. The screenshots are shown as below.

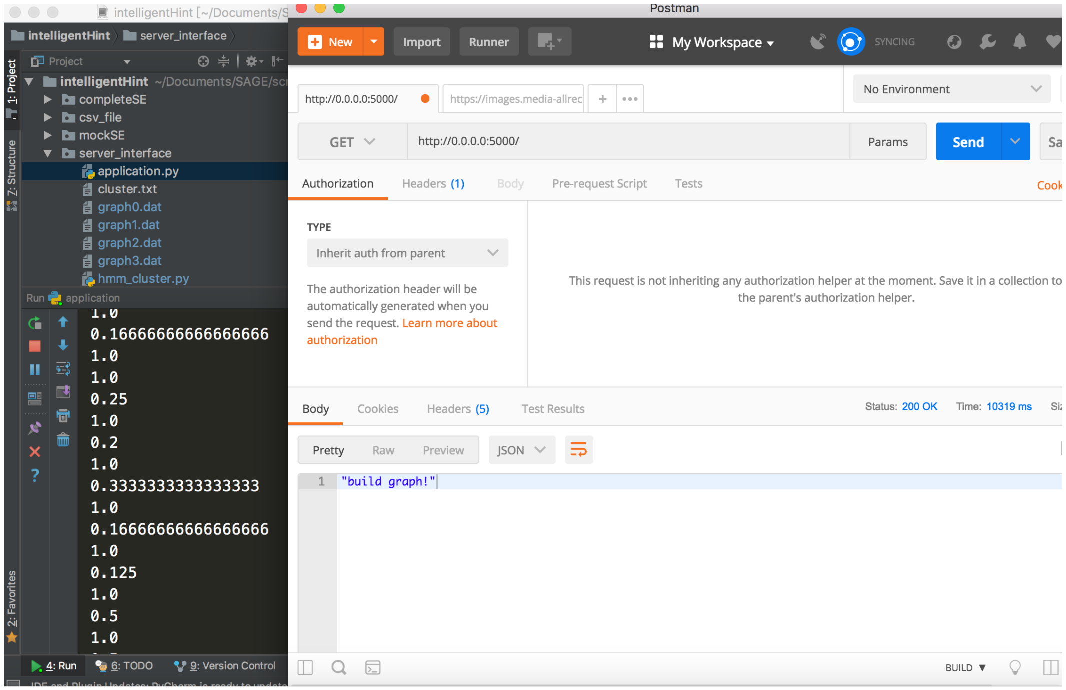

Figure 9: Graph Buiding Screenshot

The number list shown above in the left part is the reweight result.

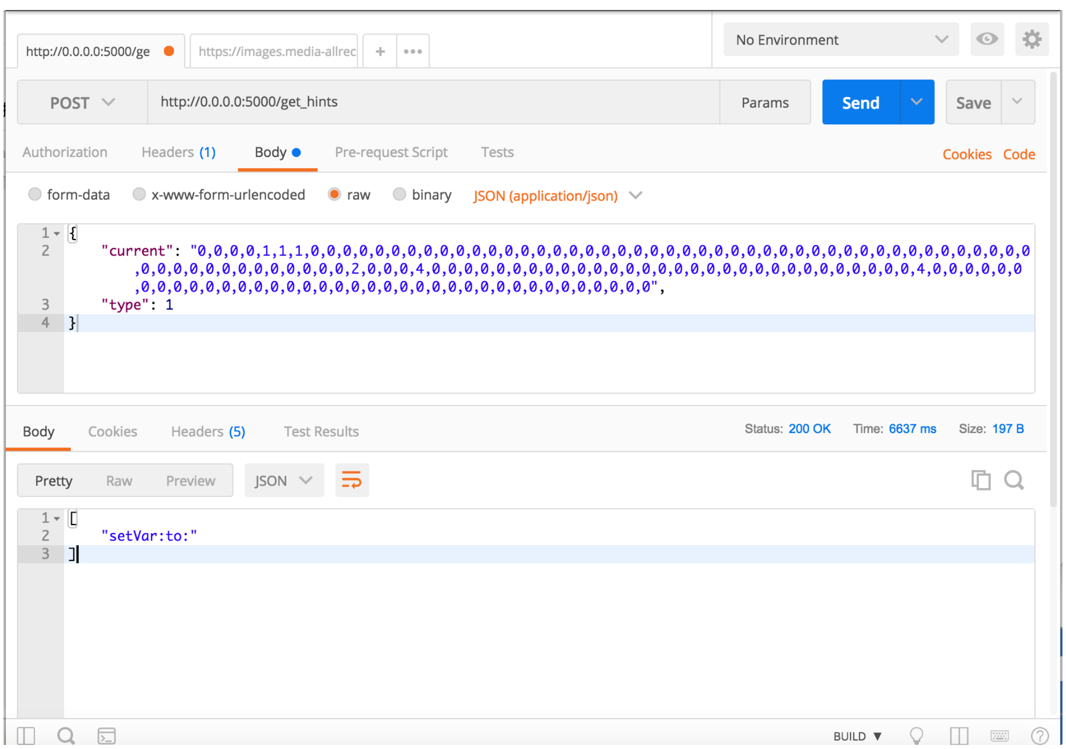

Figure 10: Hint Generation Screenshot

We generate a hint for this student including the block "setVar: to". It is contained in a string list. The maximum length of this list is 3, and the minimum length is 1.

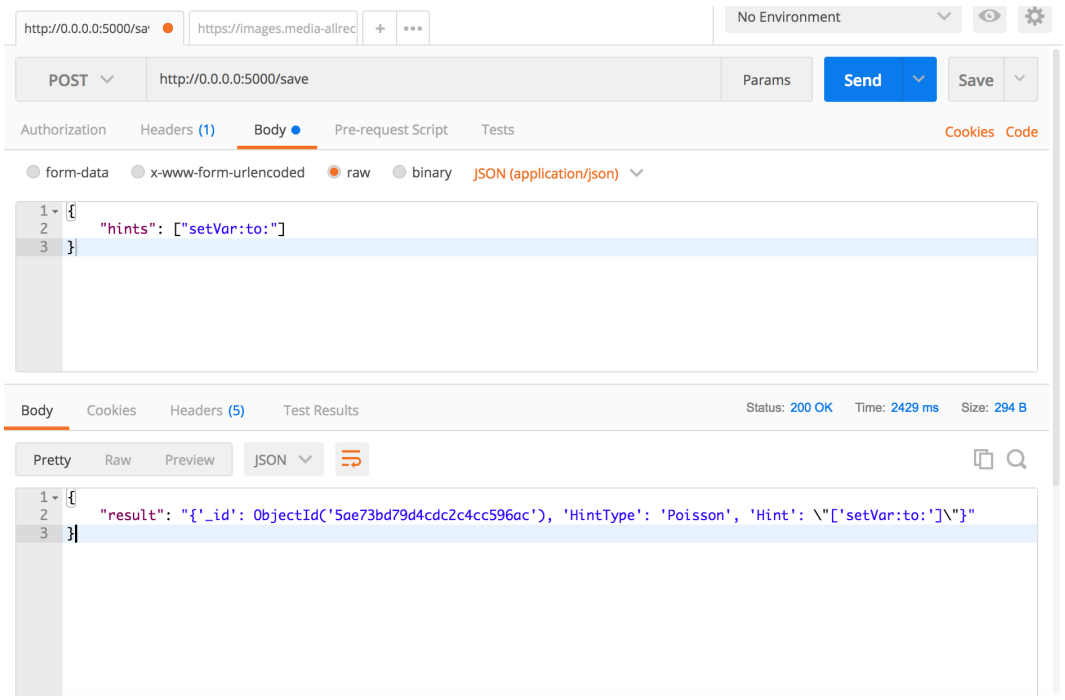

Figure 11: Save the Hints in Database Screenshot

We constrain the input and output data type as 'json'. And our API is developed according the principles of RESTful architecture over HTTP. To run the application, open 'server_interface', run 'python application.py', and open localhost:5000 to call the APIs. It is also easy to deploy on AWS or other cloud platform.
The status code definition:

- 404: Page Not Found
- 401: Unexpected Error
- 200: OK

We suppose the type of students already saved in the database, and the snapshots have been transformed into the integar vectors, using the method we described in previous chapters. After get the hints, the fronend may call 'save' to save it in the database for futher use. Following is the definition of each API and their corresponding URL, input and output data instruction, and notation.

API	Input & Output	Notes
GET /	None	build learning graph

POST /get_hints	input:
	{
	“current”: string,
	“type”: string
	}
	return:
	{“hints”: [string]
	}
		enter the student type and his or her current snapshot, return the corresponding intelligent hints.

POST /save	input:
	{
	”hints”: [string]
	},
	
	return data:
	{
	“result”: [string]
	}
		save the hints in MongoDB, return the ObjectId in the database for this hints.

GET /get_data	return:
	{
	“gameID”: string,
	“Text”: string
	}
		return the data saved in MongoDB

4 Limitations and Assumptions

4.1 Assumption during HMM Clustering

For HMM evaluation, we use the distance between the hints and the destination built by the graph as the metric. For example, after cluster the student into a specific HMM group, if the hint shows to be closer to the final state compared with some other cluster method, then HMM is helpful for the student. This method is limited since we don't have the data of student's interaction, for example, whether s/he accept this hint. We leave this problem

into the evaluation of the hinting system part.

4.2 Lack of real students data

Currently we don't have any real students data. We are only using the fake data which we generated from our scripts based on some relatively simple assumptions. For this reason, our generated HMM model may be not reliable and realistic enough for the real-world problem.

4.3 Cold start problem

For our HMM model, we need to train a new model for each new student. However, at the very beginning, we don't have enough data for building the model, thus we need to wait for the accumulation of the student's data, which could be intolerable for practical usage.

5 Future Work

5.1 More variants of mock projects

Currently, we are using a relative simple approach to generate mock data, which is, shuffle some code blocks to generate successful ones, and delete some blocks to generate failing ones. There is not enough variants for possion path algorithm.

To make the possion path algorithm run more robustly, we need to add more variants of mock projects. For successful projects, we can substitute some equivalent operations in the .se file to make a new project. We can also shuffle some codes in the line level to generate new ones. For failing projects, we can add some wrong code to generate unsuccessful projects.

5.2 Improve the imitation of students' behavior

We imitate the behavior of four types of students according to the paper [3]. However, the imitation algorithm seems to be simple. We only control the probability and time interval of moving forward and moving backward in our script.

To better capture different characristics of different types of students, we can design more complicated algorithms to imitate the behavior of different students.

5.3 Use multiple snapshots while generating hints

Currently, we are only using a single snapshot to generate the hints considering the structure of our learning graph, this might be leading to some biases. Actually, to get more accurate result, we can utilize a series of snapshots of the student to take the learning path in a short time interval into consideration (not only used in the HMM clustering).

5.4 Deal with the cold start problem

As previously mentioned, the HMM model may suffer from the problem called cold start. This problem may occur if the student's data is not enough. We currently build a general graph for student's who's type could not be classified immedietely (e.g. the student just start coding this project). But we can think about better model if there is some other project data for this student.

5.5 Deep integration to the SAGE system

Since the data processing pipeline for our hint generating system is quite complicated, currently some parts of our system are still running offline. Besides that, as most of our codes are written in Python, which may be different from Scratch Analyzer which is written in Java, the integration work would be harder. In the future, we are planning to integrate our system deep into the SAGE system, including the data input, data preprocessing, model building, hint generating, and saving to database.

References

[1] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[2] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, pages 195–204, New York, NY, USA, 2015. ACM.

[3] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.

[4] Wes McKinney. pandas: a foundational python library for data analysis and statistics.

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

[6] Padhraic Smyth. Clustering sequences with hidden markov models. In *Advances in neural information processing systems*, pages 648–654, 1997.

[7] Christian Bauckhage. Numpy / scipy recipes for data science: k-medoids clustering. 02 2015.