Intelligent Tutoring System: Outer Loop

Harsimran Bath (COMS W3995 sec 028) and Weiman Sun (COMS E6901 sec 028)

May 8, 2018

Contents

1 Abstract

In this project, we have successfully implemented the architecture for the recommendation engine and the algorithms for the recommendation engine in SAGE (Socially Addictive Gameful Engineering). The users can now very easily invoke provided API methods to get recommended games for any student to help them move forward in their progression or improve their skills. Furthermore, by calling a separate API, games will be recommended for entire classrooms to improve their curriculum and progress. The robust architecture is decoupled to support short-term changes as SAGE is being changed and long-term changes for maintenance. The work is part of the Gameful Intelligent Tutoring (GIT) Epic in TFS. The feature implemented is ITS Outer Loop.

2 Introduction

SAGE is a platform that infuses computational thinking and gamification to bring computational skills to students in K-12 grades. It is built to ensure students can learn computational concepts and instructors can teach, with the aid of a centralized platform. Jeannette Wing, in her influential article "Computation Thinking", writes "computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science."[6]

In this project, we have added intelligent system capabilities to the outer loop of SAGE. The outer loop is a process that runs when the users complete games and quests. It determines which games the student can pursue next. We have built a sophisticated recommendation engine to make recommendations for the next games the student can pursue individually. These games are recommended to help students improve skills where they are struggling, and find new challenges as they master current games. Hence, this is an item- and user- based collaborative filtering system.

Furthermore, the outer loop also offers recommendations to the instructors. The recommender recommends games the instructor can add to their class to help their students and improve the class's curriculum.

Lastly, since SAGE is still in rapid development and there are lots of things changing, we have architected the system to withstand and support changes to everything, from models to recommendation logic. It has been designed to require minimum changes, and changes that won't break other parts of the recommender system.

3 Related Work

3.1 "The Behavior of Tutoring Systems[5]"

This paper is a very helpful read in understand how the outer and inner loop work and their functions. It begins by exploring the various aspects of the outer loop:

1. Task Domain: The skills being taught by the tutor)
2. Task: An activity within the domain.
3. Step: A user interface event within the task.
4. Knowledge Component: A domain-specific concept being taught as part of the task and/or step.

The outer loop essentially runs when the user completes a task and determines what task the user should handle next. There are four main methods to selecting tasks for a student:

1. The outer loop displays all the tasks and lets the user select which the one they want to work on next. This is to some extent similar to Khan Academy, where there is a sequence of assignments, but the student can jump around.
2. The tutor assigns tasks in a predetermined sequence. In this approach, the instructor will have predefined a sequence of tasks and the tutor would automatically assign the next task in the sequence.
3. The tutor assigns tasks from a unit's pool of tasks until the student masters the unit. In this approach, there is likely a mastery level to each unit that the student needs to achieve in order to move on to the next unit. Until the student does so, the tutor will continue assigning tasks from the same unit.
4. The last approach is macro-adaptive learning. Essentially, the tutor tracks traits, such as learning styles and mastery of knowledge components, to assign tasks that would be best correlate with user's progress in mastery.

3.2 "Evaluating Collaborative Filtering Recommender Systems"[3]

Although, we were not able to implement a test harness, we considered this paper in its potential design. This paper offers different approaches in evaluating recommender systems. The paper shows key decisions to consider when building evaluation systems for recommenders. First, we have to decide between Live User Experiment and Offline Analyses. Offline analyses is quick and

economical, but the natural sparsity of data sets limit the items that can be evaluated. Live User Experiment, on the other hand, evaluate real user performance, but are expensive and prone to user bias. In addition, we have to decide between natural data sets and synthesized data sets.

After much discussion, we decided Offline analyses was the best way to move forward, given the problem of Cold Start. Hence, we need a smart mock data generation tool that generates mock data with real results and real possible recommendations to compare against the recommender's recommendations. We also decided that the "Rank Accuracy Metrics" was the best way to evaluate users in the test harness. This approach measures how well the recommender orders the recommended items similar to the user's ordering. Our user's ordering would be manually generated mock data

3.3 "A Multi-Criteria Item-based Collaborative Filtering Framework"[2]

In the traditional Item-based Collaborative Filtering method, each user reveal her admiration about an item based on a single criterion. This paper considers user can rate an item from different aspects, and proposes a multi-criteria collaborative filtering (CF) algorithm. Specifically, the multi-criteria CF computes the similarity of each criteria using the cosine-based similarity in traditional CF, then performs average among these similarities. Another proposed approach to determining the overall rating is to find an aggregation function among criteria.

3.4 "User-Based and Item-Based Collaborative Filtering Recommendation Algorithms Design"[7]

This paper builds the recommendation system based on item-based and user-based collaborative filtering. It gives the insight of the details of collaborative filtering recommendation algorithms, such as computing the similarity between users and items and making the prediction based on the similarities. It also compares the results of two models.

3.5 "Combination of machine learning algorithms for recommendation of courses in E-Learning System based on historical data"[1]

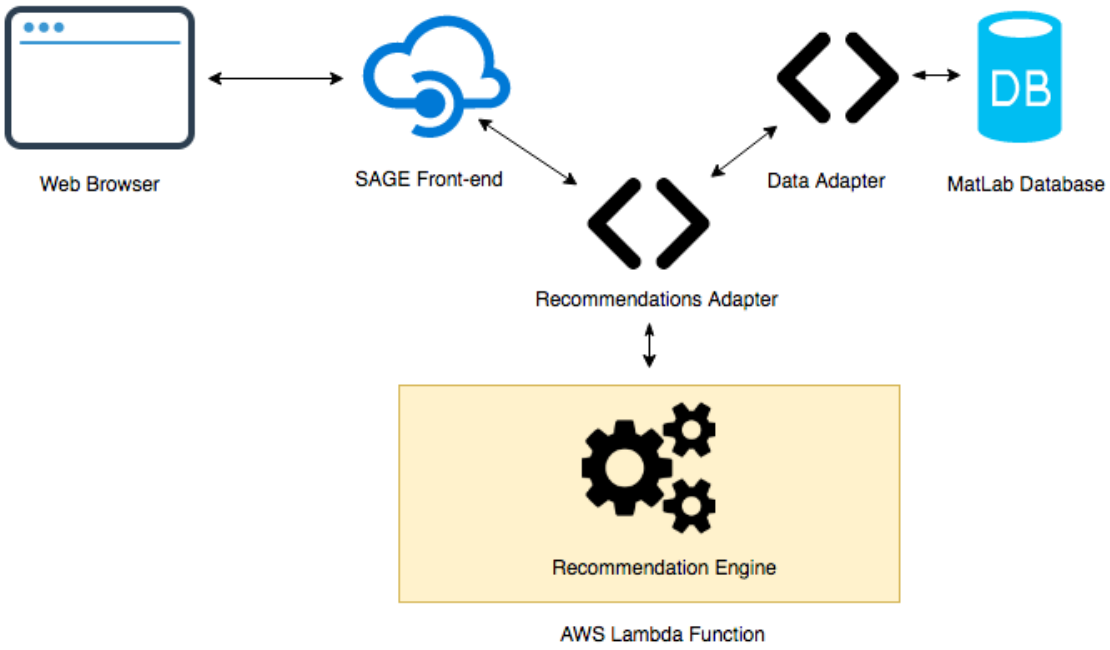
In this paper, the authors use machine learning algorithms such as clustering and association rule algorithm in a Course Recommendation System. The approach combines clustering technique – Simple K-means and association rule algorithm – Apriori and finds the result. When it applied the association rule algorithm, it only takes into account the students' interests for courses.

3.6 "Hybrid User-Item Based Collaborative Filtering"[4]

This paper proposes a hybrid user-item based collaborative filtering algorithm to address the data sparsity and scalability challenge in traditional CF. It uses Case Based Reasoning (CBR) combined with average filling to handle the sparsity of data set. It clusters users sharing common attributes using Self-Organizing Map (SOM) network with Genetic algorithms (GA) optimization. At the time of recommendation, for a target user, the closest cluster is first identified for the user, then the traditional item based CF is performed within the respective user cluster.

4 Accomplishments

4.1 Architecture



The overall architecture of the system has been implemented to allow support for long term maintenance, as well as, support for short-term dynamic changes that will result from SAGE's continued development. The Adapter Pattern and Dynamic Features described in the following sections explain how that is accomplished.

Essentially, the client running on the browser will make a simple API call to SAGE Front-end to get recommendations. The SAGE Front-end server will invoke the Recommendations Adapter to get recommendations. The Recommendations Adapter will get all relevant data it needs from the Data Adapter (which connects to MatLab) and prepares that data for the recommendation engine running in AWS. It then makes the call to AWS Lambda, invoking the engine, and getting the result. The result is then returned to the API, which returns it to the client application in JSON format.

4.1.1 API Method(s)

Two Rest API Methods have been created to get recommendations. These API methods can be invoked from the client AngularJS Application or any other client to return recommendations for a student or teachers. The primary focus of these API methods is ease of use. As long as the student Id is provided for student, or instructor Id and class Id are provided for the teacher, the APIs will automatically call the relevant adapters to return the recommendations in standard JSON format (as shown above). The complexity of recommendation is hence reduced to mere GET API requests with IDs, making it very easy for future researchers to use this.

1. Student Recommendation Route: [GET] /**recommendations/student/games/:studentId**
2. Teacher Recommendation Route: [GET] /**recommendations/instructors/:instructorId/classes/:classId/**

4.1.2 Data Adapter

```
const mockGamesDataAdapter = (function() {

  var allGames = {…
  };

  function getAllGames() {…
  }

  function getGameByIds(idsArr) {…
  }

  function getAllGamesOfAllStudents() {…
  }

  function getAllGamesInClass(classId) {…
  }

  return {
    getAllGames,
    getAllGamesOfAllStudents,
    getAllGamesInClass,
    getGameByIds,
  }
})();

module.exports = mockGamesDataAdapter;
```

The Data Adapter is the layer of interaction between the Database and the Recommendation Engine. The image above shows a skeleton contract of what is required of the Data Adapter. Essentially, this has four required methods:

1. **getAllGames()**: Returns all games and their respective features in the current system. This aides in recommending new games.

2. **getAllGamesOfAllStudents()**: Returns all games that all students have played. This is used by the recommendation engine to find trends in other students' progress with current student's progress to recommend games and build a similar path to mastery for the current student.

3. **getAllGamesInClass(classId)**: Returns all games and their respective features and scores played in current classroom. The algorithms use this to recommend new games based on mastery of current games played.

4. **getGameByIds([id1, id2, ...]**: Returns all games with Ids from the supplied array. Once the recommendation engine returns all recommended games' Ids, this method is used to fetch those actual games before retuning them to the client.

The Data Adapter is a new concept in the system. Up until now, all API routes have directly called the Mongoose Data Models to retrieve data, which couples the system too deeply with the data and prevents us from doing unit tests and maintaining the system. With this adapter, we can easily switch between Mock Data and real Data without changing any other parts of the system. In the future, it will allow us to easily build a test harness.

4.1.3 Mock Data Adapter

The current implementation is built with manually generated Mock Data. This is because we ran into the Cold Start problem; the existing system did not have useful data for us to use. Additionally, the existing Data Models in MatLab were either incomplete or outdated. To further avoid confusion, we implemented a Mock Data Adapter to not only demonstrate the feasibility of the engine, but built it in a way that, it would be very easy to swap it with Real Data Models once they are finalized. This is because Data Adapter simply needs to return the data in the format required by the engine, as long as they do that, it does not matter where and how that data is stored.

4.1.4 Recommendation Engine Adapter

```
const recommendationsAdapter = (function(dataAdapter) {
  // Validate Argument
  if (…
  ) {
    throw new Error("Invalid Argument: [dataAdapter]");
  }

  function isFunction(obj) {
    return typeof obj === "function";
  }

  function createStudentsRequest(studentId, allGames, allStudentGames) {…
  }

  function createTeachersRequest(allGames, gamesInClass) {…
  }

  function getStudentRecommendations(studentId) {…
  }

  function getTeacherRecommendations(classId) {…
  }

  return {
    getStudentRecommendations,
    getTeacherRecommendations
  };
})(mockGamesDataAdapter);

module.exports = recommendationsAdapter;
```

The Recommendation Engine Adapter is the layer of interaction between the recommendation engine and the rest of the system. It can be invoked by the system to get recommendations for

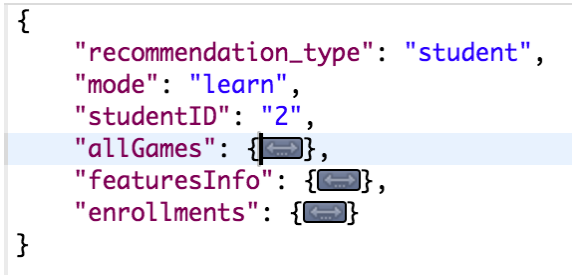
students and teachers. As evident from the interface above, it has a dependency on the Data Adapter, which it injected via the invoked function. Secondly, it exposes two simple functions to perform the recommendations:

1. **getStudentRecommendations(studentId)**: When invoked, the recommendation adapter will return game recommendations for the supplied student. It does so by first loading all the needed data through the Data Adapter, preprocessing the data, and then calling the Recommendation Engines via a request it expects. However, all these details are encapsulated from the user of the API, making this a very simple API to consume.

2. **getTeacherRecommendations(classId)**: Similarly, a user can invoke this function and provide a class Id for which they need recommendations. The function will internally perform the required steps, call the recommendations engine, and return the recommendations to the user, resulting in a very simple and user-friendly API.

A big benefit of this Adapter is that it is maintainable and it can easily "adapt" to changes in the recommendation engine and the SAGE Front-End system. Whether the data model changes or the requirements of the recommendation engine changes, only changes to the data adapter are needed. It is decoupled from the recommendation engine and the outer system, resulting in simplicity and ease of use. We were very aware of this requirement and designed the system in a way to ensure this.

4.1.5 Recommendation Engine



Finally, the recommendation engine is built to be decoupled from the entire system, requiring only an abstract input to run the algorithms on. It is written in Python (because of the availability of resources on Artificial Intelligence) and hosted on AWS Lambda Functions (more on this later).

The image above describes what the request of the Recommendation engine looks like. We have implemented this to be very easy to build, understand, and modify. These are the parameters required by the engine to work:

1. **recommendation_type**: Whether we are running *student* recommendations or *teacher* recommendations.

2. **mode**: If set to *learn*, we run a more aggressive algorithm that returns games to move forward in the curriculum. If set to *practice*, we run a relaxed algorithm to return games to improve skills.

3. **studentId**: If student recommendation, the current student's Id must be provided.

4. **allGames**: All the games that exist in the system.

5. **featuresInfo**: All the features to consider in the recommendation (more on this later).

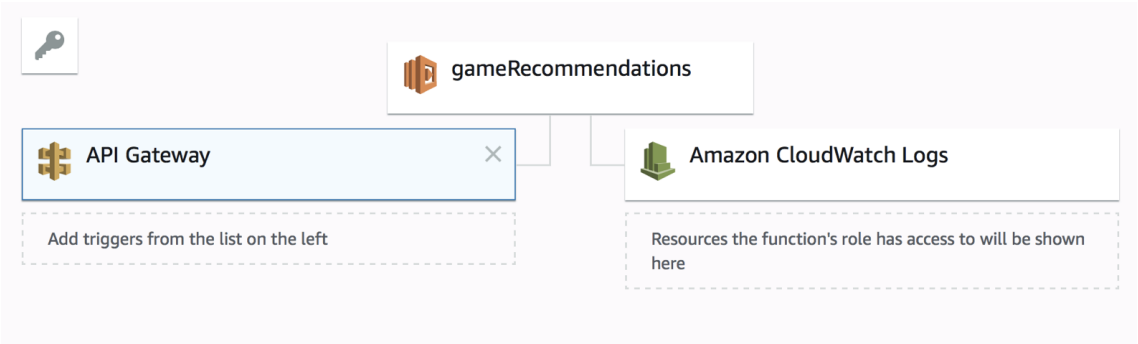
6. **enrollments**: All students to consider to base recommendations off of.

Note: This is a high level understanding of these parameters, for more information, refer to SAGE documentation.

The recommendation engine is built on the principle of single responsibility, and hence, would only perform its core duties, decoupled from the rest of the system. After processing the request, it returns an array of Ids of the games which it recommends.

Implementation details on how the recommendations work are presented in Section 6.

4.1.6 AWS Lambda Function



Since the recommendation engine is written in Python, it cannot run on SAGE Front-End, which is written in JavaScript and requires NodeJs. We found a much better alternative in AWS Lambda Functions. AWS Lambda Functions provides 1 million free requests a month, so it it ample for initial development and testing. It also scales well, for the possibility that recommendations are time consuming and require more resources.

Therefore, we built a AWS Lambda Function and exposed it through an AWS Endpoint. It is invoked and consumed by the Recommendations Adapter (described above). This is decoupled nicely from rest of the system, so in the future, we could run a Python server locally and run a test harness, giving us full coverage.

Note: More information on utilizing, deployment, and managing this service is also provided in SAGE documentation.

5 Dynamic Features

```
features_info = {
    'difficulty': {
        'weight': .3,
        'type': 'scalar',
        'maxValue': 3,
    },
    'abstractionProblemDecompositionCtScore': {
        'weight': .1,
        'type': 'scalar',
        'maxValue': 3,
    },
    'parallelismCtScore': {
        'weight': .1,
        'type': 'scalar',
        'maxValue': 3,
    },
    'logicalThinkingCtScore': {
        'weight': .1,
        'type': 'scalar',
        'maxValue': 3,
    },
```

An important part of this project is Feature Distillation; that is, the ability to dynamically add, remove features, as well as, modify their weights in recommendations. We have built such a dynamic recommendation engine, that takes into account dynamic features and weights, and applies them to the algorithms to return proper results. The image above shows the *features_Info* parameter of the Recommendation Engine request in AWS Lambda Function. Essentially, the user can provide an array of features. Features have the following:

1. **weight**: The weight of this feature. All features add up to 1.
2. **type**: Is this a scalar feature or an array.
3. **maxValue**: What is the highest value this feature can have, assuming the lowest is 0.

Note: More information on these features and how they work is provided in SAGE documentation.

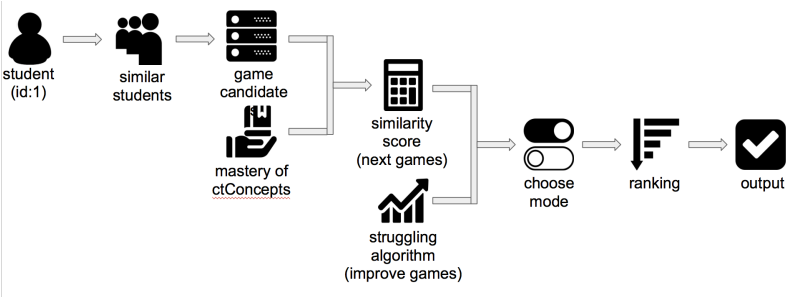
These features can dynamically changed, as the system changes. For example, if the system changes how scores are computer, or adds new metrics to games and features to students, all those can be added into the *featureInfos* and the recommendation engine would adjust.

6 Recommendation Algorithms

6.1 Student recommendation

6.1.1 Overview

The system can recommend student additional unassigned games to complete. The flow chart of recommendation engine is shown below.



Here is the overview of the recommendation algorithms: Suppose we want to recommend games for student 1, we first find the similar students of student 1. We then select the games that student 1 has not yet played from all the games of similar student to generate game candidates. At the same time, we compute the student 1's mastery of each ctConcept. We then compute the similarity score of each game in game candidates using the mastery score of each ctConcept, the games list generated here is called the next games. At this stage, we also apply the struggling algorithm to find the improve games. Then student 1 can chooses the mode to combine next games and improve games. Finally, we ranks the result and output.

In the following subsections, I will discuss more details about our algorithms.

6.1.2 Finding similar users

When finding similar users, we employ the similarity computation in the collaborative filtering method. The formula is:

$$sim(a,b) = \frac{\sum_{p\in P}(s_{a,p}-\bar{s}_a)(s_{b,p}-\bar{s}_b)}{\sqrt{\sum_{p\in P}(s_{a,p}-\bar{s}_a)^2}\sqrt{\sum_{p\in P}(s_{b,p}-\bar{s}_b)^2}} \tag{1}$$

where a, b are two users, $s_{b,p}$ is the score of user b in game p, and P is the set of games taken both by a and b. Each game has several features (which are dynamic, see Section 5). The score in our case now is the average value of every features. After we computing the similarity between two users, we will then rank the similarity and find the most k (user-defined) similar users. So the games that the similar users have taken but the given user has not taken consist of the game candidates.

ctConcepts	loop	condition	sequence	algorithms
mastery level	1	2	3	1

Table 1: Computed mastery level

game candidates	ctConcepts	similarity score
game 1	loop, condition	(0.33+0.66)/2
game 2	loop,condition,sequence	(0.33+0.66+1)/3
game 2	condition,sequence,algorithms	(0.66+1+0.33)/3

Table 2: The similarity score of each game candidate

6.1.3 Compute the mastery level of ctConcepts

We define the mastery level of a ctConcept as the highest score this ctConcept ever got in the student's history games. Below is the code snippet to compute the mastery level of each ctConcepts.

```
# compute the mastery of each ctconcepts
for game in current_games:
    score = final_score[game]
    b = games.get(game)
    c = b['ctConcepts']
    for ct in c:
        if ct not in mastery or mastery.get(ct) < score:
            mastery[ct] = score
```

6.1.4 Compute the similarity score of each game candidate

The most important part of our recommendation engine is to compute the similarity score of each game in game candidates. We employ the multi-criteria collaborative filtering method to compute the similarity. The basic idea is: for a game candidate, compute the mastery level similarity of each ctConcepts of that game, then the final similarity score is the average of all the mastery level similarity of each ctConcepts. Here is an example: from Table 1, suppose we already computed the mastery level of four ctConcpts; Table 3 tells the mapping from mastery level to similarity score; we can see from the Table 2, to compute the similarity score, we will average the score of all the ctConcepts involved in this game.

6.1.5 Struggling Students

So far, we only consider the games that student does well in, what if the student is struggling in some games? In that case, we need to provide more games that contain similar ctConcepts with

mastery level	similarity score
$\leq$1	0.33
$\leq$2	0.66
other	1

Table 3: The relationship bewteen mastery level and similarity score

the weak games to student for practicing. So here's what struggling algorithm does. First, if the mastery level of a ctConcept is less than 1, we will mark this ctConcept as week ctConcept. In the struggling algorithm, if a game contains more than 2/3 week ctConcepts among all its ctConcepts, we will add this game to the improve games list, and the proportion of the week ctConcepts in all the ctConcepts in this game is assigned as the similarity score of that game.

```
# Struglling algorithm
improve_games = {}
for game in games:
    a = games[game]
    b = a['ctConcepts']
    count = 0
    for ct in struggling:
        if ct in b:
            count += 1
    if count/len(b) >= 0.66:
        improve_games[game] = count/len(b)
```

6.1.6 choose mode

The last step before the recommendation output is to choose mode between practice mode and learn mode. In the practice mode, the weight of the improve games is 0.6, and the weight of the next games is 0.4, while learn mode is the opposite. We times the similarity score of games in both improve games and next games with the associated weight, then combine the improve games list and next games list, and finally rank the games based on their similarity score. So in summary, in the practice mode, we will recommend more improve games and less next games to student to strengthen their weakness; in the learn mode, we will recommend more next games and less improve games to student to let them learn more new things. Below is the code snippet that assigns weights to improve games list and next games list in different modes. Below is the code snippet that we use to generate improve games list.

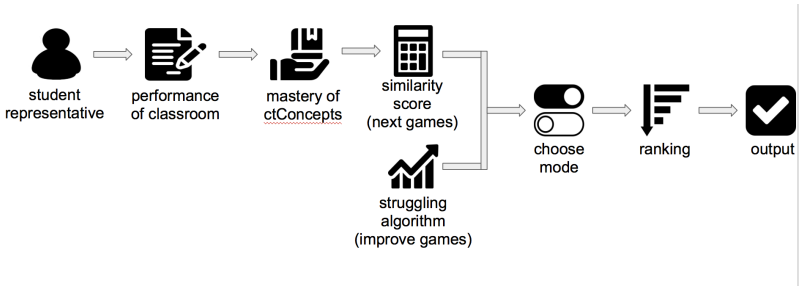
```
if(mode == 'practise'):
    # Practise Mode: improve game: 0.6, next game: 0.4
    improve_weight = 0.6
    next_weight = 0.4
elif(mode == 'learn'):
    # Learn Mode: improve game: 0.4, next game: 0.6
    improve_weight = 0.4
    next_weight = 0.6
```

6.2 Teacher Recommendation

6.2.1 Overview

The system can help the teacher to identify games to add to the quests. The basic idea of teacher recommendation engine is a bit similar with the student one, so in the implementation we reuse some code of it. The flowchat of the teacher recommendation engine is shown below. First, we need

to find a student representative. Then we compute the performance of classroom, and compute the classroom mastery of each ctconcept at the mean time. Then compute the similarity score of each game from all the other games in the system, and find the improve games using struggling algorithm. Then the teacher chooses mode between practice mode and learn mode. Finally, we rank the results and output it.



6.2.2 student representative

The student representative can be seen as a student that has average learning ability in the class. We compute the average score of each game in the classroom, these are the scores the student representative gets in each game.

6.2.3 compute the performance of classroom

Compute performance of classroom is to compute performance of each game in the classroom. There are some features to evaluate the performance of games, which are the dynamic features in Section 5. So we will weight all the features together and generate a final performance score for each game. Below is the function that we use to average all the features.

```
def process(games, features_info):
    result = {}
    average = 0
    for game in games:
        a = games[game]
        # Each ctConcept can have max score of 3
        total_score = 0
        for feature in features_info:
            f_info = features_info[feature]
            f_type = f_info['type']
            f_weight = f_info['weight']
            value = a[feature]
            if (f_type == 'array'):
                # This is an array
                max_possible_score = len(value) * f_info['maxValue']
                total_score += (sum(value)/max_possible_score)**f_weight
            elif (f_type == 'scalar'):
                # This is a scaler
                total_score += (value/f_info['maxValue'])*f_weight
            else:
                raise Exception('Unsupported Feature Type' + f_type)
        total_score = total_score / len(features_info)
        result[game] = total_score
        average += total_score
    return average/len(games), result
```

7 Challenges and Limitations

Early in the semester, we ran into lots of issues related to the Database. The databases were not documented well, and some of the obsolete tables/data still existed in the database. Some of the

important data we needed, like scores, was not persisted. Hence, this added some delays to our work. Eventually, we went the Mock Data route. Jeff and Johan were very helpful in providing guidances, cleaning the database, and starting documentation. This delay, however, pushed back our deadline and we were unable to complete User Story 130. In the future work, we have added recommendations to improving the database.

8 Documentation

1. SAGE Front-end API Endpoints: https://gudangdaya.atlassian.net/wiki/x/CoCBFw 2. Lambda Deployment and Consumption: https://gudangdaya.atlassian.net/wiki/x/HYCFFw

9 Future Work

9.1 Integration of Recommendations in the Affinity Space

Now that the APIs and the recommendation engine is built, the next step is to integrate these APIs into the front-end and show recommendations to the user, and allow them to pursue those recommendations. This should be relatively simple and easy, as everything is already in place and the API calls have been simplified to call, these are documented in SAGE Wiki (above).

9.2 Test Harness

This is a problem we were not able to solve in the semester, we simply ran out of time. We need a test harness to test and ensure the recommendation engine is working as expected. The first problem here is cold start. To address this, the test harness will require proper mock data with expected recommendations. Therefore, the mock data generator will require logic to generate mock data and the expected recommendations in order for all students. The test harness can run recommendations on those students and classrooms and compare them to the expected recommendations. A success rate can be calculated.

9.3 Real Data Models

The last but not least is to integrate real data models in the engine. Given time constraints and challenges in the current data model, we chose to create mock data. However, the design of the data adapter should make it very easy to integrate real data models into the system. This, however, will require the data models to be completed and finalized in SAGE. That is out of the scope of this task.

9.4 Database Updates

This is unrelated to our project, but caused some limitations for us. Hopefully, these suggestions will help make the schema and data models clean and simple.

1. **Combining Databases**: Right now we have Sage-Login and Sage-Node databases. The main confusion here is that we have data in two databases that is related. For example users (students and teachers) exist in Sage-Login, but Sage-Node has the actual scores for the games User played. Hence, this adds confusion and it becomes hard to protect the integrity of the system. It would be much simpler to merge these databases, so we can enforce some relationships and simplify the data models.

2. **Locking down Schema**: The data models are very fluid at the moment, and they are not always well defined. The relationships are also not simple, and can be confusing at times. I think it would help to revisit all the collections and lock down their schemas. A more radical change would be to consider switching to a relational database. It is worth considering given the benefits of clearly defined tables and relationships in schema, and SAGE is relational by design.

10 Conclusion

The intelligent outer loop now adds a significant improvement to the SAGE platform. It eases the stress on teachers to build the right curriculum by recommending games. Furthermore, it helps successful students more challenging games to continue their growth and improvement, and struggling students more games to continue improving their skills and striving for success. The recommendation engine have much better insights into how to make students successful given the data at hand, and can make more accurate and successful recommendations than the instructor.

Moreover, we have designed and built a very dynamic and robust system. It can not only adapt to new changes and features in the system, but also, its APIs make it very easy to interact with it. This project was successfully completed.

11 References

References

[1] Sunita B Aher and LMRJ Lobo. Combination of machine learning algorithms for recommendation of courses in e-learning system based on historical data. *Knowledge-Based Systems*, 51:1–14, 2013.

[2] Alper Bilge and Cihan Kaleli. A multi-criteria item-based collaborative filtering framework. In *Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on*, pages 18–22. IEEE, 2014.

[3] Jonathan L Herlocker, Joseph A Konstan, Loren G Terveen, and John T Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53, 2004.

[4] Nitin Pradeep Kumar and Zhenzhen Fan. Hybrid user-item based collaborative filtering. *Procedia Computer Science*, 60:1453–1461, 2015.

[5] Kurt Vanlehn. The behavior of tutoring systems. *International journal of artificial intelligence in education*, 16(3):227–265, 2006.

[6] Jeannette Wing. Computational thinking. *Communications of the ACM*, pages 33–36, 2006.

[7] Guanwen Yao and Lifeng Cai. User-based and item-based collaborative filtering recommendation algorithms design. *University of California, San Diego*.