UNDERSTANDING EDUCATIONAL GAME APPS THROUGH OPTIMIZATION

15.095 FINAL PROJECT REPORT

Juan José Garau Luis
PhD candidate - AeroAstro
garau@mit.edu

Iñigo de la Maza
Research Affiliate - IMES
delamaza@mit.edu

ABSTRACT

Online apps are a key element in the democratization of education all over the world, specially when it comes to early childhood education. Developing software focused to improve how kids learn is essential to make an impact on the users' abilities and skills. In this work we present an optimization-based approach to understand the relationship between the educational content of the *PBS KIDS Measure UP!* app and how effectively its users learn. This is part of the 2019 edition of the Data Science Bowl®. We tackle the problem using a feature extraction process focused on exploiting the time series nature of the data and the application of different Optimal Classification Tree models in order to both achieve a high predictive power and gain interpretable insights. Our models allow us, at the moment this report is being written, to achieve a predictive power better than 95% of the best models submitted in the competition and to understand which factors drive the children's performance. Additionally, we provide two ideas of how this work could continue leveraging the potential of optimization-based approaches.

1 Introduction

This project comprises the study of the application of predictive, optimization-based classification algorithms to a real-world task. Looking towards that aim, an online data science competition is the perfect medium due to both the availability of high-quality data, and the possibility of comparing different models and outcomes with the ones achieved by other participants using conventional heuristic methods. The online platform Kaggle is one of the most popular online competition organizer and this year's edition of the Data Science Bowl® is a perfect fit for the objectives of this project.

1.1 Data Science Bowl®

The Data Science Bowl® (DSB) is the world's most important online data science competition for social good. It was created in 2014 and is sponsored by Booz Allen Hamilton and Kaggle. Every year, the competition brings together data scientists, technologists, and domain experts across several different industries to take on the world's challenges with data and technology. In the past, the competition has addressed important issues such as improving lung cancer screening or assessing the "oceans' healt". In this year's edition, participants must focus on uncovering new insights in early childhood education and seeking solutions on how data can support learning outcomes.

PBS KIDS, a trusted name in early childhood education (3-5 years old) for decades, aims to gain insights into how media can help children learn important skills for success in school and life. One of the main products of the company is the *PBS KIDS Measure UP!* app, a game-based learning tool developed as a part of the *CPB-PBS Ready To Learn* Initiative with funding from the U.S. Department of Education.

1.2 Motivation

The increase in the number of available educational resources online has been remarkable in the last decade and the access to this huge amount of content has democratized education all over the world. People who live in remote areas

or don't have access to in-person education can now use a broad variety of resources like online masterclasses, books, videos and webinars. Additionally, the *gamification* of learning processes has been a key part of education for decades, and specially when it comes to children. This educational approach aims to motivate students to learn and practise by using video game design and game elements in learning environments. Its goal is to maximize enjoyment and engagement by capturing the interest of learners and inspiring them to continue learning. Today's smartphones and the large app variety they offer make it very easy to bring these educational apps to users all over the world. Understanding how to improve these apps is key to make sure high-quality educational content is available for children all over the world regardless of where they live.

1.3 Problem statement

The app has a diverse range of educational material including videos, assessments, activities, and games. These are designed to help kids learn different measurement concepts such as weight, capacity, or length. The DSB challenge focuses on predicting players' assessment performance based on the information the app gathers on each game session. Every detailed interaction a player has with the app is recorded in the form of an event, including things like media content display, touchscreen coordinates, and assessment answers. This data is highly fine-grained, as more than one might be recorded in less than one second. With this data, the competition participants must predict the performance of certain assessments whose outcomes are unknown, given the history of a player up to that assessment. Based on the prediction, the goal is to **classify** the player into one of four performance groups, which represent how many attempts takes a kid to successfully complete the assessment. The aim of the challenge is to help PBS KIDS discover important relationships between engagement with high-quality educational media and learning processes, by understanding what influences the most on the childrens' performance and learning rate.

2 Data Overview

The dataset provided by the competition is primarily composed of all the game analytics the *PBS KIDS Measure UP!* app gathers anonymously from its users. In this app, children navigate through a map and complete various levels, which may be activities, video clips, games, or assessments (each of them considered a different `game_session`, and its nature expressed as `type`). Each assessment is designed to test a child's comprehension of a certain set of measurement-related skills. There are five assessments: *Bird Measurer*, *Cart Balancer*, *Cauldron Filler*, *Chest Sorter*, and *Mushroom Sorter*. As explained in the previous section, the intent of the competition is to use the gameplay data to forecast how many attempts a child will take to pass a given assessment (an incorrect answer is counted as an attempt).

2.1 Data volume

Each application install is represented by an `installation_id`, which can be considered to map to one single user. The training set provides the full history of gameplay data of 17,000 `installation_id`s, while the test set has information for 1,000 players. Moreover, each `installation_id` has multiple `game_session`s of different types (activities, games, assessments...). In total there are around 300,000 `game_session`s in the training set and almost 30,000 in the test set. Finally, each `game_session` is composed by several events that represent every possible interaction between the user and the app. These events are identified with a unique ID (`event_id`), and have associated data such as screen coordinates, timestamps, durations, etc, depending on the nature of the event. In total, there are around 11.3M events in the training set and 1.1M in the test set. We can conclude the dimensionality of the problem is high and the data is presented in the form of dependencies and **time series**. To help the reader understand the data, Figure 1 shows the tree-like structure of the dataset.

2.2 Classification labels

As previously introduced, the output of the model must be a prediction of the performance of the children in a particular assessment, in the form of a classification. The model should select one performance group out of four possible `accuracy_group`s. The groups are numbered from 0 to 3 and are defined as follows:

- `accuracy_group` 3 : the assessment was solved on the first attempt
- `accuracy_group` 2 : the assessment was solved on the second attempt
- `accuracy_group` 1 : the assessment was solved after three or more attempts
- `accuracy_group` 0 : the assessment was was never solved

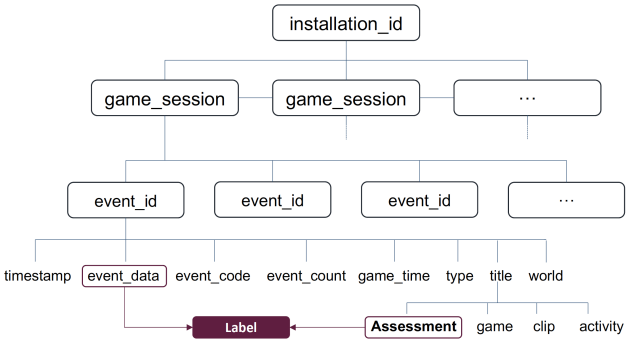

Figure 1: Structure of the dataset.

Figure 2 shows the distribution of the four possible `accuracy_groups` across the five assessments of the app. Notice that there are clearly three assessments that seem to be easier *a priori* as there is a majority of first-attempt correct answers, whereas one them has pretty much evenly distributed labels (*Bird Measurer*) and another one (*Chest Sorter*) seems to be more difficult than the others.

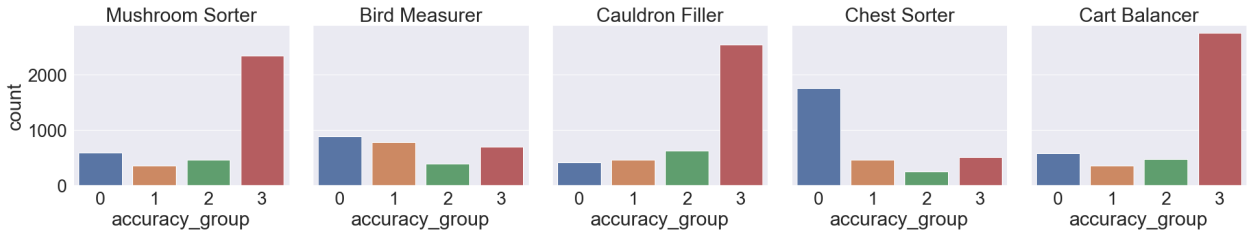

Figure 2: Label (`accuracy_group`) distribution by assessment `title`.

2.3 Training and test sets differences

Both training and test sets provided by the competition have similar structure in terms of the amount of information provided for each recorded event. The main difference is that all the information for each user (`installation_id`) in the test set stops at the beginning of an assessment, whose outcome is meant to be predicted. However, in many cases, information from previous complete assessments is given.

Every `installation_id` in the test has at least one assessment (the one to be labeled by the model), whereas many `installation_id` from the training set do not have any, and therefore have been removed from the dataset because they do not allow to generate any label for the training process. This leads to a major data volume decrease, as only 4242 `installation_id` out of the 17,000 original ones can be used.

3 Feature Engineering

The process of extracting good features train the machine learning model is time-consuming and challenging in this specific competition. Not only the provided data is in the form of a time series, but it is also has a large amount of unnecessary information that needs to be filtered. In this section the process of transitioning from the original dataset to a good dataset to make predictions upon is explained. All the code relevant to this section can be found in the first of the notebooks appended at the end of this report.

3.1 Structuring data as time series

The test set comprises all historical data from different users prior to an assessment first attempt. Therefore, the time dimension is a key feature of this dataset, as only the events that have happened before each assessment can be considered for the definition of its associated features. This leads to the first step of the featuring engineering process: redefining the data for each player in order to leverage the time series structure of the original data. Figure 3 shows how

the original data is organized for a specific player (`installation_id`) and day. Our approach is to transform all this information into multiple training data points that exploit the history of a player's interactions.

Taking this approach allows training the model with only the information it is going to have available later in the prediction process, and avoids the fatal conceptual error of using future data for any assessment. It is also true that it entails getting rid of some of the available data, as any registered event happening after an assessment completion would not be used (see events to right of the second assessment in Figure 3).

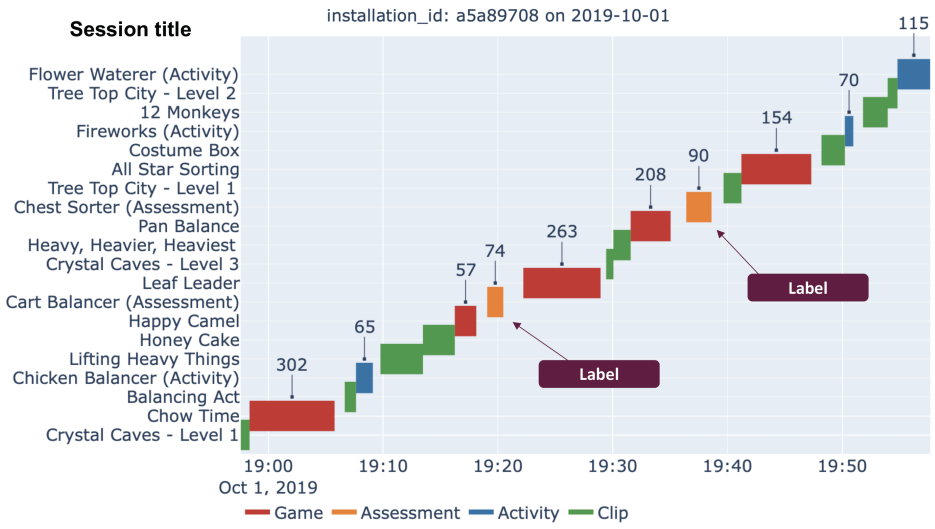

Figure 3: Time series representation of the `game_sessions` of a certain `installation_id`. The annotations above certain `game_session` show the number of events they comprise (notice that all *Clips* (green) have only one event, as there is no interaction with the player).

3.2 Training instance definition

When a player completes multiple assessments, multiple data points centered on each assessment can be considered. As observed in Figure 4, a training instance/data point is created from every assessment and its preceding information. The distribution of the number of assessments per user is very similar between the training and test sets, therefore, we can safely take all the events that happened prior to each assessment, and not only the ones between assessments. This allows to consider the whole experience of the user until the moment of facing that assessment, and therefore we can maximize the information input at every moment, without using future data.

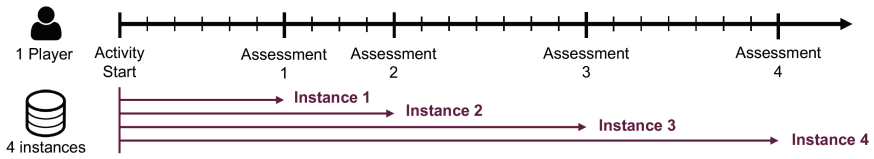

Figure 4: Approach for instance definition.

Overall, we can find approximately 21,000 assessments with this strategy, maximizing the number of training instances. In addition, this approach allows to generate extra training instances from the test set, as every completed assessment prior to the one whose outcome has to be predicted by the model can be considered as a new training instance with its associated label. This adds 2,549 more training instances to the final training dataset, summing up to a total of 23,788 rows.

3.3 Feature definition

The objectives of the feature creation approach are twofold: first, address how hard the particular assessment is by gathering assessment-related features from the whole data set. And second, determine how good a player is by looking at every recorded event before the start of an assessment and capturing skill-related features.

3.3.1 Assessment-related features

In order to provide the model with information regarding the difficulty of the assessment whose outcome has to be predicted, we compute both the mean and the median `accuracy` (continuous variable showing the percent of correct answers, different from the `accuracy_group`) as well as the mean and the median `accuracy_group` (discrete labels from 0 to 4 as it has been explain in section 2.2). Even if they might be correlated, the optimization-based models that we use are supposed not to be affected by this issue, and the metric that better describes the correlation between the difficulty of the assessment and the performance of the children will be used for the prediction process.

As shown in Figure 2, some assessments are clearly easier than others. Therefore, we include both the assessment `title` and the `world` it belongs to (related to length, capacity or weight questions) as one-hot-encoded features. The latter is meant to provide some information on the type of questions the user is going to face, allowing the model to detect certain user profiles that are better at questions related with weight than at assessments covering capacity concepts, for example.

3.3.2 Player-related features

The largest proportion of the features associated to each assessment are the ones related to the historical data of the users. The purpose of all activities, videos, and games that each user is supposed (but not obliged) to do before taking the challenge of completing an assessment is to "train" a user to increase the chances of performing better at the assessment. Therefore, using information related to how the player performs at previous games or activities might be useful to make the prediction.

Seeking the quantification of how experienced each player is at the moment of each assessment, different metrics have been developed:

- Counters (every time a particular instance has been registered) of the different:
  - `game_session` by `type` (Assessment | Clip | Game | Activity)
  - `game_session` by `title` (all the possible games, activities, etc, the app includes)
  - `event_code`
  - `game_session` and `event_code` from the same world than the assessment to be predicted (gives a sense of the experience of the player with that particular family of concepts)
  - Times the player has already tried to complete the same assessment to be predicted, and its performance
- Total number of `game_session` and `event_id`
- Time played by `game_session` `type`, and total number of minutes played
- Average time spent per `game_session` `type` (gives a sense of how engaged and focused the player is)
- Average performance of the player in previous assessments
- Day of the week and time when the assessment has been completed

All the discrete features (e.g. accuracy group of the previously completed assessments) have been encoded as one-hot. After the feature engineering process, the final training dataset has 23,788 rows, 129 features and 4 possible classification outcomes. In the following section we explain which models are used to make predictions and how we can further leverage the data to better align ourselves with the purpose of the developer.

4 Optimal Classification Trees

Since the task is to predict the accuracy group of a player before taking an assessment, in this work we focus on optimization-based classification models, namely Optimal Classification Trees (OCT). By means of Julia's package `IAI`, in this section we train several OCT models and analyze their performance. All the code that allows us to obtain the performance results from this section is presented in the second and third notebooks appended at the end of this report.

4.1 Models

Understanding the tradeoff between interpretability and predictive power they have, in this work we consider OCT models with both parallel and hyperplane splits in order to gain insights on what makes users perform well and to obtain models capable of reaching high scores in the competition, respectively.

We understand the importance of training models with an appropriate selection of hyperparameters, therefore we carry out hyperparameter tuning tasks with a validation set before evaluating the performance of each model. In the case of OCT with parallel splits, we create a grid that considers three different values for the *maximum depth* parameter (5, 10, and 15) and three different values for the *minbucket* parameter (1, 5, and 10). In the case of OCT-H, in order not to compromise runtime given the size of the dataset, we choose to fix the *minbucket* parameter to 1 and train two models with a *maximum depth* of 2 and 3, respectively. Since OCT-H considers hyperplane splits, and we decide not to enforce any sparsity constraint, choosing any other depth value beyond the selected ones does not provide impactful predictive power, degrades the interpretability of these models, and unnecessarily increases runtime.

All models are trained using the `OptimalTreeClassifier` with parallelization and using the misclassification error as criterion to determine the best combination of hyperparameters. Additionally, we decide to duplicate the number of analyses and repeat both the OCT and OCT-H cases choosing to equally prioritize the 4 classes by means of the *autobalance* attribute. The rationale behind this decision can be easily understood by looking at figures 5 and 6 in the Appendix. We cover the details of these figures in the coming section but the reader can notice that when *autobalance* is not used there is no leaf that predicts class 2. This is corrected when enforcing an equal importance among classes, helping understand the relationship between players' behaviour and this class.

4.2 Results - Predictive power

Tables 1 and 2 show the training and out-of-sample performance of the OCT and OCT-H models with and without the *autobalance* attribute. In this work we analyze the performance using two different metrics: on one hand we consider the accuracy, which takes into account the proportion of correctly classified labels; on the other hand we use the Cohen's kappa coefficient with quadratic weights, which is the official metric of the competition and is defined as

$$\kappa = 1 - \frac{\sum_{i=1}^{k} \sum_{j=1}^{k} w_{ij} x_{ij}}{\sum_{i=1}^{k} \sum_{j=1}^{k} w_{ij} m_{ij}} \tag{1}$$

where x_{ij} represents the number of datapoints from class i that have been classified as class j, m_{ij} is the expected number of datapoints from class i that have been classified as class j, and finally w_{ij} is a weight factor defined as

$$w_{ij} = 1 - \frac{i^2}{(j-i)^2} \tag{2}$$

By looking at the tables we can immediately appreciate the dominance of OCT-H models regardless of the use of the *autobalance* attribute. The *autobalance* feature works as expected, models without it perform better but that entails completely missing a class. This can be observed in figures 5 and 7 from the appendix, which do not have any leaf predicting class 2 (kids who solved the assessment in the second attempt).

	Model			
	With autobalance		**Without autobalance**	
Metric	**OCT**	**OCT-H**	**OCT**	**OCT-H**
Accuracy	0.573	0.551	0.614	**0.627**
Quadratic kappa	0.481	0.498	0.500	**0.524**

Table 1: Training performance of OCT and OCT-H with and without the *autobalance* attribute.

	Model			
	With autobalance		**Without autobalance**	
Metric	**OCT**	**OCT-H**	**OCT**	**OCT-H**
Accuracy	0.569	0.548	0.620	**0.633**
Quadratic kappa	0.473	0.490	0.514	**0.540**

Table 2: Out-of-sample performance of OCT and OCT-H with and without the *autobalance* attribute.

The model that achieves the best performance, both in terms of accuracy and the quadratic kappa coefficient, is OCT-H without balancing the outcome classes. Its kappa coefficient is 0.540, which at the moment this report is about to be submitted, is less than 0.03 points away from the first competitor in the leaderboard (with $\kappa = 0.567$). From a general perspective, only 90 competitors out of approximately 1900 (less than 5%) achieve better performance than our model. This result highlights the predictive potential of optimization-based classification trees.

4.3 Results - Interpretability

While OCT-H allows us to obtain highly competitive models, OCT (with parallel splits) offers key insights to understand why kids perform differently when using the app. The resulting OCT decision trees, shown in figures 5 and 6 from the appendix, highlight the relevance of specific groups of features:

- **Which assessments:** the presence of the variables `is_Chest_Sorter`, `is_Cauldron_Filler`, and `is_Bird_Measurer` shows us that the performance depends on which assessment is being done. The assessment *Chest Sorter* is identified as the hardest exercise and, when the player has not completed many assessments, *Bird Measurer* can also be challenging.
- **Player's experience:** both decision trees include variables such as `game_pct`, `ass_pct`, and `accumulated_accuracy_group`. When this features reflect a player with more experience in games and assessments, the chances of predicting class 3 or 2 increase.
- **Repeating an assessment:** in the decision trees it is shown (features `previous_completions` and `last_accuracy_same_title`) that kids who are repeating an assessment are more likely to perform well.
- **Specific events and titles:** features `4020` and `num_31` identify specific events and titles that, if present during an assessment, might alter the performance of the player.

In figures 7 and 8 from the appendix we also show the decision trees obtained from the OCT-H models. In this case we have omitted the split decision due to the amount of variables included as a consequence of not having enforced sparsity.

4.4 Comparison with other methods

Finally, we address the predictive performance of the OCT models compared to other popular approaches in data science competitions. Specifically, we train a XGBoost and a logistic regression model, both Python-based. Table 3 shows the accuracy and kappa coefficient performance, for both the training and test datasets, of these methods compared to the best OCT approach, namely the OCT-H model without the *autobalance* feature. We can observe that OCT-H achieves a better performance than the other approaches, which helps to emphasize the relevance of optimization-based classification tree approaches. Using the same optimization procedure we have been able to both achieve competitive performance and understand the underlying relationship between how kids interact with the app and how successfully they learn the concepts.

	Model					
	Training			**Out-of-sample**		
Metric	**OCT-H**	**XGBoost**	**Log. reg.**	**OCT-H**	**XGBoost**	**Log. reg.**
Accuracy	0.627	**0.999**	0.599	**0.633**	0.607	0.602
Quadratic kappa	0.524	**0.998**	0.466	**0.540**	0.501	0.472

Table 3: Training and out-of-sample performance of OCT-H compared to Python's XGBoost and Logistic regression.

5 Additional optimization-based approaches

Up to this point we have focused on the work that represents the goal of the DSB 2019: making predictions of players' performance before taking on assessments. In this section we take a step further and focus on the goals that align the mission of the competition organizers: understand how to improve their products to enhance the learning experience for the kids. To that end, following we broadly present two ideas based on optimization methods that could follow the work done in this project.

5.1 Optimal Splits

As presented in the previous section, in this work we have carried out hyperparameter tuning tasks by splitting the training dataset into actual train and validation data. While we have used randomization to split the data, we understand there might be better approaches considering the uniqueness of this data. As stated in Section 2, there is a wide variety of games and activities kids can do before attempting to solve an assessment. Furthermore, since there is no obligation to take these intermediate steps, and there is no order enforced, the quantity of different player profiles and feature distributions is substantially high.

Consequently, when making random splits of the dataset, there is a high chance of obtaining unbalanced splits and not having an equal distribution of data points between the train and validation datasets. In order to solve this problem, we propose formulating a mathematical program to optimally split the dataset. This formulation, which is able to provide the optimal split regardless of the amount of splits to be created, is focused on reducing the discrepancies between the different groups. Since we are dealing with a dataset with more than one feature (129 to be precise), the formulation also takes into account multicovariates.

The last of the notebooks appended contains the function `split_opt`, which contains the code of the formulation in Julia. We propose a warm start based on the rerandomization split rule. Although we can't provide results due to the dimensionality of the problem and runtime constraints, we understand this optimization-based procedure would allow us to obtain the best splits among data in order to increase the robustness of our models. In the end, making sure the models are robust is key to ensure the educational effectiveness of the app.

5.2 Optimal Prescription Trees

As part of possible extensions to this projects, it has also been considered the development of an Optimal Prescription Tree model in order to provide a solution that would directly take actions based on the way the players use the app while they are using it. The model would achieve such task by taking into account both the performance prediction and the impact that every possible performance would cause.

Different possible prescription actions have been explored. The most promising one is the design of a certain learning path on the go. Since the app allows the user to freely decide its way through the game, one possible prescription could be whether to allow or to block the possibility of taking a particular assessment. The model would predict the performance that the player would have if he/she was allowed to do the assessment. If the historical data of the player led to a bad performance prediction, the prescription would be to take a particular activity or game that may help the user to gain more experience, and therefore blocking the possibility of trying to do the assessment at that moment.

The number of possible prescriptions for each different user and assessment scenarios have made unfeasible the task of looking for suitable prescription data given the time window of this project. However, with a larger dataset (approximately 1M assessments) and additional time, it could be possible to find cases to every possible prescription.

6 Conclusions and Future Work

In this work a new application of optimization-based machine learning methods has been explored. The highly-complex dataset provided by the competition organizers has allowed the creation of a large amount of features from a time series-oriented dataset. Then, based on this dataset two OCT and two OCT-H models have been very effective in terms of selecting the most important information and achieving a high score in the competition (top 5% at the moment this report is being written). This would have been much different if classical methods had been used instead. For instance, a long and arduous process of trial and error in terms of seeking the the different features that lead to a better prediction would have been needed.

We have proved the usefulness of optimization-based methods in prediction tasks and, more important, have provided insights on which factors affected the performance the most. We believe that in this particular problem the possibility of interpreting the prediction logic was key to align ourselves with the mission of the competition organizer. Following this direction, we have opened the door to two additional optimization-based approaches that would additionally leverage the information obtained by the app to improve the learning experience of the players. Specifically, we have explored the idea of optimally splitting the dataset to ensure a higher degree of robustness and formulating a prescription problem in which the app takes into account the performance prediction to decide whether a user is ready or not to take certain assessments.

A OCT Visualizations

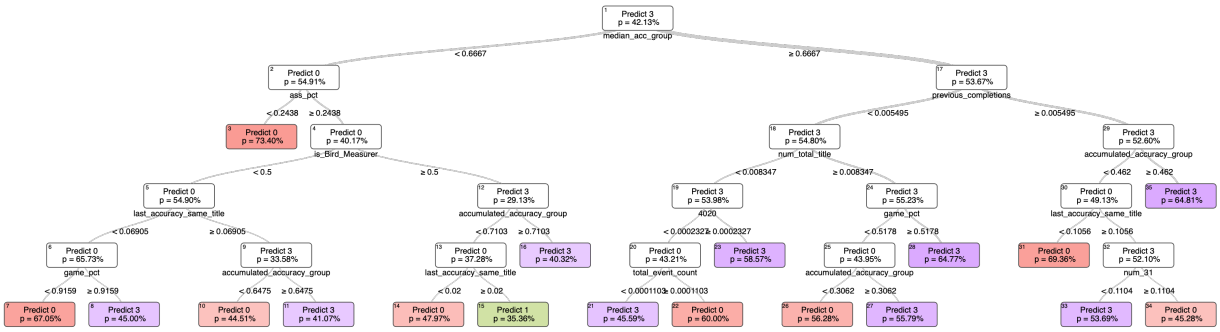

Figure 5: Decision tree result of training an OCT model without *autobalance*.

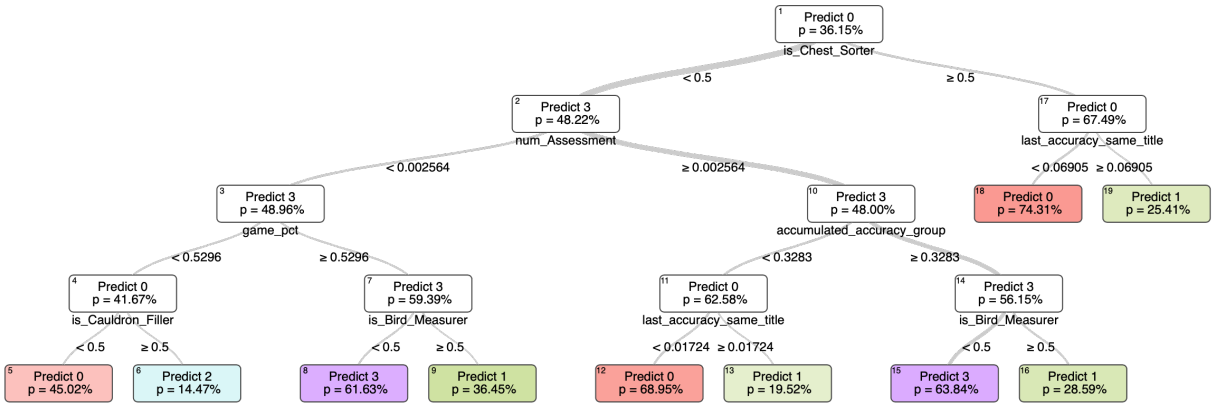

Figure 6: Decision tree result of training an OCT model with *autobalance*.

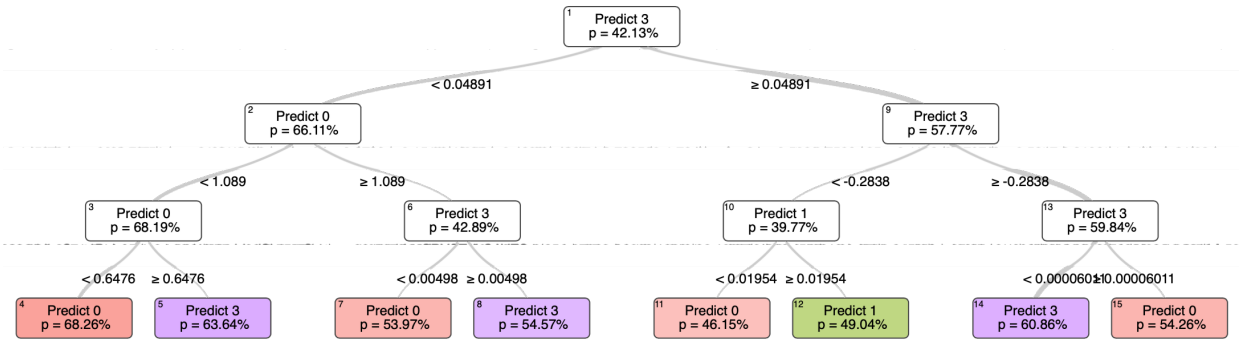

Figure 7: Decision tree result of training an OCT-H model without *autobalance*.

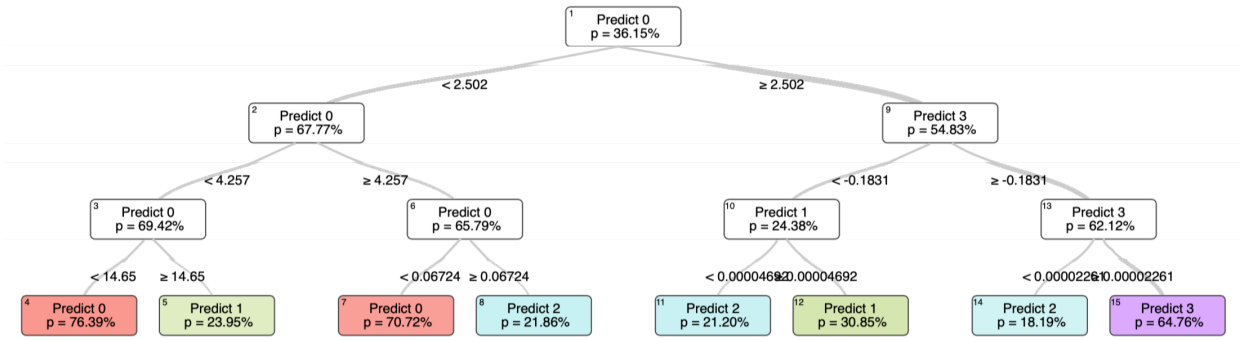

Figure 8: Decision tree result of training an OCT-H model with *autobalance*.

B Hyperparameter selection

Model class	Hyperparameter	Value
OCT with *autobalance*	Minbucket	1
	Max. depth	5
OCT without *autobalance*	Minbucket	1
	Max. depth	5
OCT-H with *autobalance*	Max. depth	3
OCT-H without *autobalance*	Max. depth	3

Table 4: Best hyperparameters obtained after the validation process for each model class.

Feature Engineering

December 8, 2019

Table of Contents

1 General information

This Notebook is meant to perform the necessary feature engineering functions in order to condition the training and test data for its input to the machine learning model.

2 Notebook setup

Library import

[ ]:
```
import pandas as pd
import numpy as np
import json
from numba import jit #high performance python compiler
from tqdm.notebook import tqdm #fast, extensible progress bar
from xgboost import XGBClassifier
from joblib import Parallel, delayed #provides lightweight pipelining in Python
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import cohen_kappa_score
#Python Standard Library
import os #miscellaneous operating system interfaces
import copy #generic shallow and deep copy operations
import gc #interface to the optional garbage collector
import time #time access and conversions
import datetime #basic date and time types
import json #JSON encoder and decoder
import re #provides regular expression matching operations
from typing import Any, List #support for type hints
import warnings #warning control
warnings.filterwarnings("ignore")
from itertools import product #cartesian product, equivalent to a nested for-loop
from collections import Counter, defaultdict #Counter: dict subclass for counting␣
 ↪hashable objects
                                              #defaultdict: dict subclass that calls a␣
 ↪factory function to supply missing values
pd.options.display.precision = 15
pd.set_option('max_rows', 500)

#Others
from IPython.display import HTML #Create a display object given raw data
import networkx as nx #creation, manipulation, and study of complex networks

#Visualization
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import altair as alt #declarative statistical visualization library based on Vega

sns.set_style('whitegrid')
my_pal = sns.color_palette(n_colors=10)
```

3 Feature engineering functions

3.1 Reading data

[ ]:
```
# Read data from CSV files
def read_data(read_train_labels_full = True):
    print('Reading train.csv file....')
    train = pd.read_csv('train.csv')
    print('Training.csv file has {} rows and {} columns'.format(train.shape[0], train.
 ↪shape[1]))

    print('Reading test.csv file....')
    test = pd.read_csv('test.csv')
    print('Test.csv file has {} rows and {} columns'.format(test.shape[0], test.
 ↪shape[1]))

    if read_train_labels_full:
```

```
        print('Reading train_labels_full.csv file....')
        train_labels = pd.read_csv('clean_data/train_labels_full.csv')
        print('Train_labels_full.csv file has {} rows and {} columns'.
↪format(train_labels.shape[0], train_labels.shape[1]))
    else:
        print('Reading train_labels.csv file....')
        train_labels = pd.read_csv('train_labels.csv')
        print('Train_labels.csv file has {} rows and {} columns'.format(train_labels.
↪shape[0], train_labels.shape[1]))

    print('Reading specs.csv file....')
    specs = pd.read_csv('specs.csv')
    print('Specs.csv file has {} rows and {} columns'.format(specs.shape[0], specs.
↪shape[1]))

    print('Reading sample_submission.csv file....')
    ss = pd.read_csv('sample_submission.csv')
    print('Sample_submission.csv file has {} rows and {} columns\n'.format(ss.
↪shape[0], ss.shape[1]))

    return train, test, train_labels, specs, ss
```

3.2 Filter `instalation_id` with no assessments

[ ]:
```
def get_installation_ids_with_assessment(train):
    """All 'installation_id' from the test set have at least one assessment,
    but in the train set some don't have any.
    """
    print('Filtering \'installation_id\' without assessments....')
    installation_id_with_assesment = train.groupby('installation_id')[['type']].
↪agg(lambda x: np.sum(x == 'Assessment') > 0).type.to_dict()
    train = train[train['installation_id'].map(installation_id_with_assesment)].
↪reset_index(drop=True)
    print('train has {} rows and {} columns\n'.format(train.shape[0], train.shape[1]))
    return train
```

3.3 Encoding text data

[ ]:
```
def encode_text_data(train, test):
    print('Encoding text data....\n')
    #Make a list with all the unique 'titles' from the train and test set
    list_of_title = sorted(list(set(train['title'].unique()).union(set(test['title'].
↪unique()))))
    #Make a list with all the unique 'event_code' from the train and test set
    list_of_event_code = sorted(list(set(train['event_code'].unique()).
↪union(set(test['event_code'].unique()))))
    #Make a list with all the unique 'event_id' from the train and test set
    list_of_event_id = sorted(list(set(train['event_id'].unique()).
↪union(set(test['event_id'].unique()))))
    #Make a list with all the unique 'world' from the train and test set
```

```
    list_of_world = sorted(list(set(train['world'].unique()).union(set(test['world'].
↪unique()))))
    short_list_of_world = list_of_world.copy()
    short_list_of_world.pop(2)
    #Make a list with all the 'title' that are assessments
    list_of_assessment_title = sorted(list(set(train[train['type'] ==␣
↪'Assessment']['title'].value_counts().index).union(set(test[test['type'] ==␣
↪'Assessment']['title'].value_counts().index))))
    #Make a list with all the 'game_session' that are assessments
    list_of_assessment_session = sorted(list(set(train[train['type'] ==␣
↪'Assessment']['game_session'].unique()).union(set(test[test['type'] ==␣
↪'Assessment']['game_session'].unique()))))

    #Create a dictionary encoding the 'title'
    title_map = dict(zip(list_of_title, np.arange(len(list_of_title))))    #(keys: str␣
↪| values: int)
    title_map_labels = dict(zip(np.arange(len(list_of_title)), list_of_title))␣
↪#reversed dictionary (keys: int | values: str)
    #Create a dictionary encoding the 'world'
    world_map = dict(zip(list_of_world, np.arange(len(list_of_world))))

    #Replace the text data with the numerical data from the dictionaries
    train['title'] = train['title'].map(title_map)
    test['title'] = test['title'].map(title_map)
    train['world'] = train['world'].map(world_map)
    test['world'] = test['world'].map(world_map)

    #Create a dictionary with the 'event_code' that has the information of the correct␣
↪answers
    correct_event_code_map = dict(zip(title_map.values(), (4100*np.
↪ones(len(title_map))).astype('int')))
    correct_event_code_map[title_map['Bird Measurer (Assessment)']] = 4110 #Change the␣
↪one for 'Bird Measurer (Assessment)' to 4110

    #Convert 'timestamp' into datetime class
    train['timestamp'] = pd.to_datetime(train['timestamp'])
    test['timestamp'] = pd.to_datetime(test['timestamp'])

    return train, test, correct_event_code_map, list_of_title, list_of_world,␣
↪short_list_of_world,list_of_event_code, title_map, title_map_labels, world_map,␣
↪list_of_assessment_title, list_of_assessment_session, list_of_event_id
```

3.4 Determining start and final events for each training instance

[ ]:
```
def identify_start_and_final_events(train,test):
    print('Determining start and final events for each training instance....\n')
    # Sort values
    train = train.sort_values(by=['installation_id', 'timestamp'])
    test = test.sort_values(by=['installation_id', 'timestamp'])

    # Identify start events
    train_min = train.groupby('installation_id')[['timestamp']].min()
```

```
    train_min['initial_time'] = 1
    train_min = train_min.reset_index()
    train = train.merge(train_min, on=['installation_id', 'timestamp'], how='left')
    train['initial_time'].fillna(0, inplace=True)

    test_min = test.groupby('installation_id')[['timestamp']].min()
    test_min['initial_time'] = 1
    test_min = test_min.reset_index()
    test = test.merge(test_min, on=['installation_id', 'timestamp'], how='left')
    test['initial_time'].fillna(0, inplace=True)

    # Identify final events (first event of each assessment)
    train_max = train[train['type'] == 'Assessment']
    train_max = train_max.groupby('game_session')[['timestamp']].min()
    train_max['final_time'] = 1
    train_max = train_max.reset_index()
    train = train.merge(train_max, on=['game_session', 'timestamp'], how='left')
    train['final_time'].fillna(0, inplace=True)

    test_max = test[test['type'] == 'Assessment']
    test_max = test_max.groupby('game_session')[['timestamp']].min()
    test_max['final_time'] = 1
    test_max = test_max.reset_index()
    test = test.merge(test_max, on=['game_session', 'timestamp'], how='left')
    test['final_time'].fillna(0, inplace=True)

    return train, test
```

3.5 Generate missing assessment labels

```
[ ]: def get_assessment_accuracy(df,assessment_session):
    assessment_df = df[df['game_session']== assessment_session]
    assessment_info = {'game_session' : assessment_session}
    assessment_info.update({'installation_id' : assessment_df.iloc[-1].
↪installation_id})
    assessment_title = assessment_df.iloc[-1].title
    assessment_info.update({'title' : title_map_labels[assessment_title]})

    #accumulated_correct_attempts & accumulated_incorrect_attempts
    all_attempts = assessment_df.query(f'event_code ==␣
↪{correct_event_code_map[assessment_title]}')
    assessment_info.update({'num_correct' : all_attempts['event_data'].str.
↪contains('true').sum()})
    assessment_info.update({'num_incorrect' : all_attempts['event_data'].str.
↪contains('false').sum()})
    assessment_info.update({'accuracy' : assessment_info['num_correct']/
↪(assessment_info['num_correct']+assessment_info['num_incorrect']) if␣
↪(assessment_info['num_correct']+assessment_info['num_incorrect']) != 0 else 0})
    #accuracy_group
    if assessment_info['accuracy'] == 0:
        assessment_info.update({'accuracy_group' : 0})
    elif assessment_info['accuracy'] == 1:
```

```
        assessment_info.update({'accuracy_group' : 3})
    elif assessment_info['accuracy'] == 0.5:
        assessment_info.update({'accuracy_group' : 2})
    else:
        assessment_info.update({'accuracy_group' : 1})

    return assessment_info

def get_train_labels_full(train_labels,train,test):
    print('Generating missing assessment labels...')
    train_labels_full = train_labels
    list_of_session_with_label = train_labels['game_session'].unique()
    print('{} sessions with labels: {}...'.
 ↪format(len(list_of_session_with_label),list_of_session_with_label[:5]))
    list_of_session_without_label = np.
 ↪setdiff1d(list_of_assessment_session,list_of_session_with_label)
    print('{} sessions without labels: {}...'.
 ↪format(len(list_of_session_without_label),list_of_session_without_label[:5]))
    list_of_sessions_to_be_predicted = []
    for assessment_session in tqdm(list_of_session_without_label, total= 6896):
        if train[train['game_session'] == assessment_session].any(axis=None):
            train_labels_full = train_labels_full.
 ↪append(get_assessment_accuracy(train,assessment_session),ignore_index=True)
        else:
            if test[test['game_session'] == assessment_session].shape[0] > 1:
                train_labels_full = train_labels_full.
 ↪append(get_assessment_accuracy(test,assessment_session),ignore_index=True)
            else:
                list_of_sessions_to_be_predicted.append(assessment_session)

    print('{} sessions to be predicted by model: {}...'.
 ↪format(len(list_of_sessions_to_be_predicted),list_of_sessions_to_be_predicted[:5]))
    print('Result: {} labels generated\n'.format(train_labels_full.
 ↪shape[0]-train_labels.shape[0]))
    train_labels_full.to_csv('train_labels_full.csv', index=False)
    return train_labels_full
```

3.6 Get general information about the performance of children in each assessment

[ ]:
```
def get_assessment_info(train_labels_full):
    """Gets the mean and median of both accuracy and accuracy_group per each assessment
    """
    print('Getting general information about the performance of children in each␣
 ↪assessment...\n')
    mean_acc_map = {title: train_labels_full[train_labels_full['title'] ==␣
 ↪title]['accuracy'].mean() for title in list_of_assessment_title}
    mean_acc_group_map = {title: train_labels_full[train_labels_full['title'] ==␣
 ↪title]['accuracy_group'].mean() for title in list_of_assessment_title}

    median_acc_map = {title: train_labels_full[train_labels_full['title'] ==␣
 ↪title]['accuracy'].median() for title in list_of_assessment_title}
```

```
    median_acc_group_map = {title: train_labels_full[train_labels_full['title'] ==␣
↪title]['accuracy_group'].median() for title in list_of_assessment_title}

    return mean_acc_map, mean_acc_group_map, median_acc_map, median_acc_group_map
```

3.7 Obtaining features for each answered assessment (from both train and test sets)

[ ]:
```
def extract_correct_pct(series, subs=False):
    tot, num_cor = 0, 0
    for s in series:
        dict_s = json.loads(s)
        if 'correct' in dict_s.keys():
            tot += 1
            if dict_s['correct']:
                num_cor += 1
    if subs and num_cor > 0:
        num_cor -= 1
        tot -= 1
    return num_cor / tot if tot > 0 else 0.0

def observation(df, k, i,test_set=False):
    # Select the time series corresponding to installation_id k and Assessment i
    df2 = df[(df['final_time_sum'] <= i) | (df['final_time_sum'] == i + 1) &␣
↪(df['final_time'] == 1)]
    df = df2.reset_index(drop=True)
    df.loc[:, 'date'] = df.loc[:, 'timestamp'].dt.date

    #Getting the features for the assessment
    features = {}
    #assesment dependent features
    assessment_row = df.iloc[-1]
    features.update({'installation_id': assessment_row['installation_id']})
    features.update({'game_session': assessment_row['game_session']})
    features.update({'is_' + assessment_title.replace(' ', '_')[:-13]: 1 if␣
↪assessment_row['title'] == title_map[assessment_title] else 0 for assessment_title␣
↪in list_of_assessment_title})
    features.update({'is_' + world: 1 if assessment_row['world'] == world_map[world]␣
↪else 0 for world in short_list_of_world})
    features.update({'hour': assessment_row['timestamp'].hour})
    features.update({'dayofweek': assessment_row['timestamp'].dayofweek})

    features.update({'num_unique_dates': df['date'].nunique()})
    features.update({'mean_acc':␣
↪mean_acc_map[title_map_labels[assessment_row['title']]]})
    features.update({'mean_acc_group':␣
↪mean_acc_group_map[title_map_labels[assessment_row['title']]]})
    features.update({'median_acc':␣
↪median_acc_map[title_map_labels[assessment_row['title']]]})
    features.update({'median_acc_group':␣
↪median_acc_group_map[title_map_labels[assessment_row['title']]]})

    #counter features
```

```
    #title_count
    title_count = df.groupby('game_session')['title'].max().value_counts().to_dict()
    for title in title_map.values():
        features.update({'num_' + str(title): title_count[title] if title in␣
↪title_count else 0})
    features['num_'+ str(assessment_row['title'])] -= 1
    features.update({'num_total_title': sum(title_count.values())})
    features.update({'previous_completions': features['num_'+␣
↪str(assessment_row['title'])]})

    #title_type_count
    title_type_count = df.groupby('game_session')['type'].max().value_counts().
↪to_dict()
    title_type_keys = ['Assessment', 'Game', 'Clip', 'Activity']
    for title_type in title_type_keys:
        features.update({'num_' + title_type: title_type_count[title_type] if␣
↪title_type in title_type_count else 0})
    features['num_Assessment'] -= 1

    #title_type_same_world_count
    title_type_same_world_count = df[df.world == assessment_row['world']].
↪groupby('game_session')['type'].max().value_counts().to_dict()
    for title_type in title_type_keys:
        features.update({'num_' + title_type + '_same_world':␣
↪title_type_same_world_count[title_type] if title_type in title_type_same_world_count␣
↪else 0})
    features['num_Assessment_same_world'] -= 1
    features.update({'num_total_title_same_world': sum(title_type_same_world_count.
↪values())})

    #event_code_count
    event_code_count = df.groupby('event_code')['event_code'].count().to_dict()
    for event_code in list_of_event_code:
        features.update({str(event_code): event_code_count[event_code] if event_code␣
↪in event_code_count else 0})
    features.update({'total_event_count': sum(event_code_count.values())})

    #min_count
    min_count_by_type = df.groupby('type')['game_time'].agg(lambda x: sum(x)/(60 *␣
↪1000)).to_dict()
    for title_type in title_type_keys:
        if title_type == 'Clip':
            pass
        else:
            features.update({'mins_' + title_type: min_count_by_type[title_type] if␣
↪title_type in min_count_by_type else 0})
    features.update({'mins_total': sum(min_count_by_type.values())})

    #avg_min_count
    avg_min_count_by_type = df.groupby('type')['game_time'].agg(lambda x: sum(x)/(60 *␣
↪1000)).to_dict()
    for title_type in title_type_keys:
```

```
        if title_type == 'Clip':
            pass
        else:
            features.update({'avg_mins_' + title_type:␣
↪avg_min_count_by_type[title_type]/features['num_' + title_type] if title_type in␣
↪min_count_by_type and features['num_' + title_type]!= 0 else 0})
    features.update({'avg_mins_total': sum(avg_min_count_by_type.values())/
↪features['num_total_title']})

    #correct_pct's
    game_pct = df[df['type'] == 'Game'].groupby('game_session')['event_data'].
↪agg(lambda x: extract_correct_pct(x)).mean()
    features.update({'game_pct': game_pct if game_pct is not np.nan else 0})
    ass_pct = df[df['type'] == 'Assessment'].groupby('game_session')['event_data'].
↪agg(lambda x: extract_correct_pct(x,True)).mean()
    features.update({'ass_pct': ass_pct if ass_pct is not np.nan else 0})

    #accumulated_correct/incorrect_attemps and accuracies
    accumulated_correct_attempts = 0
    accumulated_incorrect_attempts = 0
    accumulated_accuracy = 0
    accumulated_accuracy_group = 0
    last_accuracy_same_title = -1
    last_accuracy_group_same_title = -1

    for assessment_session in df[df['type'] == 'Assessment']['game_session'].unique():
        if assessment_session == features['game_session']: #skipping the assessment to␣
↪be labeled
            pass
        else:
            accumulated_correct_attempts +=␣
↪train_labels_full[train_labels_full['game_session'] ==␣
↪assessment_session]['num_correct'].values[0]
            accumulated_incorrect_attempts +=␣
↪train_labels_full[train_labels_full['game_session'] ==␣
↪assessment_session]['num_incorrect'].values[0]
            accuracy = train_labels_full[train_labels_full['game_session'] ==␣
↪assessment_session]['accuracy'].values[0]
            accumulated_accuracy += accuracy
            last_accuracy_same_title = accuracy if␣
↪title_map[train_labels_full[train_labels_full['game_session'] ==␣
↪assessment_session]['title'].values[0]] == assessment_row['title'] else␣
↪last_accuracy_same_title
            accuracy_group = train_labels_full[train_labels_full['game_session'] ==␣
↪assessment_session]['accuracy_group'].values[0]
            accumulated_accuracy_group +=  accuracy_group
            last_accuracy_group_same_title = accuracy_group if␣
↪title_map[train_labels_full[train_labels_full['game_session'] ==␣
↪assessment_session]['title'].values[0]] == assessment_row['title'] else␣
↪last_accuracy_group_same_title

    features.update({'accumulated_correct_attempts': accumulated_correct_attempts})
```

```
    features.update({'accumulated_incorrect_attempts': accumulated_incorrect_attempts})
    features.update({'accumulated_accuracy': accumulated_accuracy})
    features.update({'accumulated_accuracy_group': accumulated_accuracy_group/
↪features['num_Assessment'] if features['num_Assessment'] > 0 else 0})
    features.update({'last_accuracy_same_title': last_accuracy_same_title})
    features.update({'last_accuracy_group_same_title': last_accuracy_group_same_title})

    #Getting the label for the assessment if it is not an incomplete one from the test␣
↪set
    if not test_set:
        assessment_accuracy = features.update({'accuracy_group':␣
↪train_labels_full[train_labels_full['game_session'] ==␣
↪assessment_session]['accuracy_group'].values[0]})

    return features
```

3.8 Generating the final train and test sets

[ ]:
```
def get_final_data(train, test):
    train['final_time_sum'] = train['final_time'].groupby(train['installation_id']).
↪transform('cumsum')
    test['final_time_sum'] = test['final_time'].groupby(test['installation_id']).
↪transform('cumsum')
    obs_per_id_train = train.groupby('installation_id')[['final_time_sum']].max().
↪astype('int32').to_dict()['final_time_sum']
    obs_per_id_test = test.groupby('installation_id')[['final_time_sum']].max().
↪astype('int32').to_dict()['final_time_sum']
    final_train = []
    final_test = []
    print('Obtaining train features:')
    #final_train = parallelize_dataframe(obs_list,final_train,n_cores = 4)
    for k, v in tqdm(obs_per_id_train.items(),total= 4242):
        df_ = train[train['installation_id'] == k]
        for i in range(v):
            final_train.append(observation(df_, k, i))

    print('Obtaining test features:')
    for k, v in tqdm(obs_per_id_test.items(), total = 1000):
        df_ = test[test['installation_id'] == k]
        for i in range(v-1):
            final_train.append(observation(df_, k, i))
        final_test.append(observation(df_, k, v - 1,test_set=True))

    final_train = pd.DataFrame(final_train)
    final_train = final_train.reset_index(drop=True)
    X_train = final_train.iloc[:, 2:-1]
    Y_train = final_train.iloc[:, -1]

    final_test = pd.DataFrame(final_test)
    final_test = final_test.reset_index(drop=True)
    X_test = final_test.iloc[:, 2:]
```

```
    print('Final train (\'X_train\') information:\nN = {}\np= {}\nsum(y)={}\n\n'.
↪format(X_train.shape[0],X_train.shape[1],Y_train.shape[0]))
    print('Final test (\'X_test\') information:\nN = {}\np= {}'.format(X_test.
↪shape[0],X_test.shape[1]))

    return X_train, Y_train, X_test, final_train, final_test
```

3.9 Saving final train and test sets

[ ]:
```
def save_data(X_train, Y_train, X_test,final_train,final_test):
    X_train.to_csv('clean_data/X_train.csv', index=False)
    Y_train.to_csv('clean_data/Y_train.csv', index=False)
    X_test.to_csv('clean_data/X_test.csv', index=False)
    final_train.to_csv('clean_data/final_train.csv', index=False)
    final_test.to_csv('clean_data/final_test.csv', index=False)
```

4 Final dataset generation

4.1 train_labels_full generation

[ ]:
```
#If train_labels_full hasn't still been generated, run this cell

#Read data
#train, test, train_labels, specs, ss = read_data(read_train_labels_full = False)
#Filter 'installation_id' without assements from train
#train = get_installation_ids_with_assessment(train)
#Get useful dictionaries with encoded titles, worlds, types...
#train, test, correct_event_code_map, list_of_title, list_of_world,␣
↪short_list_of_world, list_of_event_code, title_map, title_map_labels, world_map,␣
↪list_of_assessment_title, list_of_assessment_session, list_of_event_id =␣
↪encode_text_data(train, test)
#Generate missing assessment labels
#train_labels_full = get_train_labels_full(train_labels,train,test)
#Storing the new data set
#train_labels_full.to_csv('clean_data/train_labels_full.csv', index=False)
```

4.2 X_train, Y_train, X_test generation

[ ]:
```
#Read data
train, test, train_labels_full, specs, ss = read_data()
#Filter 'installation_id' without assements from train
train = get_installation_ids_with_assessment(train)
#Get useful dictionaries with encoded titles, worlds, types...
train, test, correct_event_code_map, list_of_title, list_of_world,short_list_of_world,␣
↪list_of_event_code, title_map, title_map_labels, world_map,␣
↪list_of_assessment_title, list_of_assessment_session, list_of_event_id =␣
↪encode_text_data(train, test)
#Identify start and final events for each assessment in both the train and test sets
train, test= identify_start_and_final_events(train,test)
#Get general metrics about the performance of children in each assessment
```

```
mean_acc_map, mean_acc_group_map, median_acc_map, median_acc_group_map =␣
 ↪get_assessment_info(train_labels_full)
#Generate final training and test set
X_train,Y_train, X_test, final_train, final_test = get_final_data(train,test)
#Saving data
save_data(X_train, Y_train, X_test,final_train,final_test)
```

OCT model final

December 9, 2019

0.1 OCT Models

Import libraries

We also use 4 parallel processes

[ ]:
```
using JuMP, CSV, Gurobi, Random, LinearAlgebra, Statistics, DecisionTree, DataFrames,␣
 ↪PyCall, Distributed
addprocs(4)
```

Load the datasets

[ ]:
```
X_train = CSV.read("X_train.csv", header=true)
Y_train = CSV.read("Y_train.csv", header=false)
Y_train = Y_train_full[:, end]
X_test = CSV.read("X_test.csv", header=true);
```

Split into train and validation datasets

[ ]:
```
function split_iai(X_train, Y_train, train_proportion)
    (X_train, Y_train), (X_valid, Y_valid) = IAI.split_data(:classification, X_train,␣
 ↪Y_train, train_proportion=train_proportion);
    return X_train, Y_train, X_valid, Y_valid
end
```

[ ]:
```
X_train, Y_train, X_valid, Y_valid = split_iai(X_train, Y_train, 0.8)
X_train_m = Array{Float64,2}(X_train);
Y_train_m = Vector(Y_train);
X_valid_m = Array{Float64,2}(X_valid);
Y_valid_m = Vector(Y_valid);
```

0.1.1 OCT WITH autobalance

[ ]:
```
lnr_a = IAI.OptimalTreeClassifier(parallel_processes = Distributed.procs(), criterion=␣
 ↪:misclassification)
grid_a = IAI.GridSearch(lnr_a, Dict(
    :max_depth => [5, 10, 15],
    :minbucket => [1,5,10],
));
IAI.fit!(grid_a, X_train, Y_train, X_valid, Y_valid, sample_weight=:autobalance)
```

The obtained decision tree is the following

[ ]:
```
IAI.get_learner(grid_a)
```

The misclassification errors for the training and validation datasets are the following

[ ]:
```
prediction_train = IAI.predict(grid_a, X_train)
cm = confusion_matrix(Y_train_m, prediction_train)
```

[ ]:
```
prediction_valid = IAI.predict(grid_a, X_valid)
cm = confusion_matrix(Y_valid_m, prediction_valid)
```

Now we make the prediction on the test data and save it

[ ]:
```
prediction_oct_a = IAI.predict(grid_a, X_test)
submission_oct_a_df = DataFrame(accuracy_group = prediction_oct_a)
CSV.write("submission_oct_a.csv", submission_oct_a_df, writeheader=true)
```

OCT WITHOUT autobalance

[ ]:
```
lnr = IAI.OptimalTreeClassifier(parallel_processes = Distributed.procs(), criterion= :
 ↪misclassification)
grid = IAI.GridSearch(lnr, Dict(
    :max_depth => [5, 10, 15],
    :minbucket => [1,5,10],
));
IAI.fit!(grid, X_train, Y_train, X_valid, Y_valid)
```

The obtained decision tree is the following

[ ]:
```
IAI.get_learner(grid)
```

The misclassification errors for the training and the validation datasets are the following

[ ]:
```
prediction_train = IAI.predict(grid, X_train)
cm = confusion_matrix(Y_train_m, prediction_train)
```

[ ]:
```
prediction_valid = IAI.predict(grid, X_valid)
cm = confusion_matrix(Y_valid_m, prediction_valid)
```

Now we make the prediction on the test data and save it

[ ]:
```
prediction_oct = IAI.predict(grid, X_test)
submission_oct_df = DataFrame(accuracy_group = prediction_oct)
CSV.write("submission_oct.csv", submission_oct_df, writeheader=true)
```

0.1.2 OCT-H WITH autobalance

[ ]:
```
lnr2_a = IAI.OptimalTreeClassifier(hyperplane_config=(sparsity=:all,),␣
 ↪parallel_processes = Distributed.procs(), criterion= :misclassification)
grid2_a = IAI.GridSearch(lnr2_a, Dict(
    :max_depth => [2, 3],
    :minbucket => [1],
));
IAI.fit!(grid2_a, X_train, Y_train, X_valid, Y_valid, sample_weight=:autobalance)
```

The obtained decision tree is the following

[ ]:
```
IAI.get_learner(grid2_a)
```

The misclassification errors for the training and the validation datasets are the following

[ ]:
```
prediction_train = IAI.predict(grid2_a, X_train)
cm = confusion_matrix(Y_train_m, prediction_train)
```

[ ]:
```
prediction_valid = IAI.predict(grid2_a, X_valid)
cm = confusion_matrix(Y_valid_m, prediction_valid)
```

Now we make the prediction on the test data and save it

[ ]:
```
prediction_octh_a = IAI.predict(grid2_a, X_test)
submission_octh_a_df = DataFrame(accuracy_group = prediction_octh_a)
CSV.write("submission_octh_a.csv",  submission_octh_a_df, writeheader=true)
```

0.1.3 OCT-H WITHOUT autobalance

[ ]:
```
lnr2 = IAI.OptimalTreeClassifier(hyperplane_config=(sparsity=:all,),␣
 ↪parallel_processes = Distributed.procs(), criterion= :misclassification)
grid2 = IAI.GridSearch(lnr2, Dict(
    :max_depth => [2, 3],
    :minbucket => [1],
));
IAI.fit!(grid2, X_train, Y_train, X_valid, Y_valid)
```

The obtained decision tree is the following

[ ]:
```
IAI.get_learner(grid2)
```

The misclassification errors for the training and the validation datasets are the following

[ ]:
```
prediction_train = IAI.predict(grid2, X_train)
cm = confusion_matrix(Y_train_m, prediction_train)
```

[ ]:
```
prediction_valid = IAI.predict(grid2, X_valid)
cm = confusion_matrix(Y_valid_m, prediction_valid)
```

Now we make the prediction on the test data and save it

[ ]:
```
prediction_octh = IAI.predict(grid2, X_test)
submission_octh_df = DataFrame(accuracy_group = prediction_octh)
CSV.write("submission_octh.csv", submission_octh_df, writeheader=true)
```

Python models final

December 9, 2019

0.1 Python Models

Import libraries

```
[ ]: import numpy as np
     import pandas as pd
     from xgboost import XGBClassifier
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import accuracy_score
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import cohen_kappa_score
```

Read data

```
[ ]: X_train = pd.read_csv('X_train.csv')
     Y_train = pd.read_csv('Y_train.csv', header=None)
     X_test = pd.read_csv('X_test.csv')
```

Split into train and validation datasets

```
[ ]: X_train, X_valid, Y_train, Y_valid = train_test_split(X_train, Y_train, test_size=0.2)
```

0.1.1 XGBoost

Fitting an XGBoost model with the training dataset, using a maximum depth of 20 and a minbucket size of 1

```
[ ]: model = XGBClassifier(max_depth=20)
     model.fit(X_train, Y_train)
```

We make predictions on the train, test, and validation datasets

```
[ ]: Y_pred_train = model.predict(X_train)
     Y_pred_valid = model.predict(X_valid)
     Y_pred_test = model.predict(X_test)
```

We compute the accuracy and quadratic kappa scores for the training and validation datasets

```
[ ]: accuracy = accuracy_score(Y_valid, Y_pred_valid)
     kappa = cohen_kappa_score(Y_valid, Y_pred_valid, weights='quadratic')
     print(accuracy)
     print(kappa)
```

[ ]:
```
accuracy = accuracy_score(Y_train, Y_pred_train)
kappa = cohen_kappa_score(Y_train, Y_pred_train, weights='quadratic')
print(accuracy)
print(kappa)
```

We save the submission

[ ]:
```
submission_xg = pd.DataFrame(Y_pred_test, columns=['accuracy_group'])
submission_xg.to_csv("submission_xg.csv", index=False)
```

0.1.2 Logistic Regression

Fitting a Logistic Regression model with the training dataset, using the *newton-cg* solver for multiclass problems

[ ]:
```
logreg = LogisticRegression(solver='newton-cg', multi_class='multinomial')
logreg.fit(X_train, Y_train)
```

We make predictions on the train, test, and validation datasets

[ ]:
```
Y_pred_train = logreg.predict(X_train)
Y_pred_valid = logreg.predict(X_valid)
Y_pred_test = logreg.predict(X_test)
```

We compute the accuracy and quadratic kappa scores for the training and validation datasets

[ ]:
```
accuracy = accuracy_score(Y_valid, Y_pred_valid)
kappa = cohen_kappa_score(Y_valid, Y_pred_valid, weights='quadratic')
print(accuracy)
print(kappa)
```

[ ]:
```
accuracy = accuracy_score(Y_train, Y_pred_train)
kappa = cohen_kappa_score(Y_train, Y_pred_train, weights='quadratic')
print(accuracy)
print(kappa)
```

We save the submission

[ ]:
```
submission_lr = pd.DataFrame(Y_pred_test, columns=['accuracy_group'])
submission_lr.to_csv("submission_lr.csv", index=False)
```

Optimal splits

December 9, 2019

0.1 Optimal splits

Import libraries

[ ]:
```
using JuMP, CSV, Gurobi, Random, LinearAlgebra, Statistics, DecisionTree, DataFrames,␣
↪PyCall, Distributed
```

Load the datasets

[ ]:
```
X_train_full = CSV.read("X_train.csv", header=true)
Y_train_full = CSV.read("Y_train.csv", header=false)
Y_train_full = Y_train_full[:, end]
X_test = CSV.read("X_test.csv", header=true);
```

Normalize train and test dataset according to the train dataset

This action helps reducing the numerical precision errors when computing the square root of the inverse matrix

[ ]:
```
for i = 1:size(X_train_full, 2)
    X_train_full[!, i] = X_train_full[!, i] ./ maximum(X_train_full[!, i])
    X_test[!, i] = X_test[!, i] ./ maximum(X_train_full[!, i])
end
```

Auxiliar functions

`compute_avg_pairwise_discrepancy` returns the average pairwise discrepancy between `m` different groups

[ ]:
```
function compute_avg_pairwise_discrepancy(data, std_v, index_v, m, r)
    res = 0
    for i = 1:m
        for j = 1:m
            if i != j
                res = res + discrepancy(data[(index_v[i]+1):index_v[i+1], :],␣
↪data[(index_v[j]+1):index_v[j+1], :], std_v, r)
            end
        end
    end
    return res/(m*m)
end
```

`discrepancy` computes the discrepancy between two groups `g1` and `g2`

[ ]:
```
function discrepancy(g1, g2, std_v, r)
    return sum(abs(mean(g1[:, i]) - mean(g2[:, i]))/std_v[i] for i = 1:r)
```

```
end
```

`get_index_vector` specifies how many data points each group should include in case the total amount of data points `n` is not entirely divisible by the number of groups `m`

[ ]:
```
function get_index_vector(n, m)
    a = []
    k = Int(floor(n / m))
    resid = n % m
    append!(a, 0)
    for i = 2:m
        if resid > 0
            append!(a, a[end] + k + 1)
            resid = resid - 1
        else
            append!(a, a[end] + k)
        end
    end
    append!(a, n)
    return a
end
```

Initialization functions

`get_initial_x` returns matrix `x` from a split given by `best_a`

[ ]:
```
function get_initial_x(n, m, index_v, best_a)
    x = zeros(n, m)
    for i = 1:m
        for j = (index_v[i]+1):index_v[i+1]
            x[best_a[j], i] = 1
        end
    end
    return round.(Int, x)
end
```

`get_initial_M` returns a tensor as an initialization of variable `M`

[ ]:
```
function get_initial_M(n, m, r, k, x, norm_data_m)
    M = zeros(m, m, r)
    for s = 1:r
        for p = 1:m
            for q = (p + 1):m
                elem1 = 1 / k * sum(norm_data_m[i, s] * x[i, p] - x[i, q] for i = 1:n)
                elem2 = 1 / k * sum(norm_data_m[i, s] * x[i, q] - x[i, p] for i = 1:n)
                M[p, q, s] = max(elem1, elem2)
            end
        end
    end
    return M
end
```

`get_initial_V` returns a tensor as an initialization of variable `V`

[ ]:
```
function get_initial_V(n, m, r, k, x, norm_data_m)
    V = zeros(m, m, r, r)
    for s = 1:r
        for s2 = s:r
            for p = 1:m
                for q = (p + 1):m
                    elem1 = 1 / k * sum(norm_data_m[i, s] * norm_data_m[i, s2] * x[i,␣
 ↪p] - x[i, q] for i = 1:n)
                    elem2 = 1 / k * sum(norm_data_m[i, s] * norm_data_m[i, s2] * x[i,␣
 ↪q] - x[i, p] for i = 1:n)
                    V[p, q, s, s2] = max(elem1, elem2)
                end
            end
        end
    end
    return V
end
```

get_initial_d returns an initialization value for variable d

[ ]:
```
function get_initial_d(m, r, M, V, )
    d = 0
    for p = 1:m
        for q = (p + 1):m
            elem1 = sum(M[p, q, s] for s = 1:r)
            elem2 = sum(V[p, q, s, s] for s = 1:r)
            elem3 = sum(sum(V[p, q, s, s2] for s2 = (s + 1):r) for s = 1:(r - 1))
            if elem1 +  * elem2 + 2 *  * elem3 > d
                d = elem1 +  * elem2 + 2 *  * elem3
            end
        end
    end
    return d
end
```

Core function

[ ]:
```
function split_opt(X_train, Y_train, m, , reps_heuristic)

    # X_train is already normalized
    n, r = size(X_train)
    k = Int(floor(n / m))
     = [mean(X_train[:, i]) for i = 1:r]
    norm_data = deepcopy(X_train)
    norm_data_m = Array{Float64,2}(norm_data)

    # Compute matrix
     = 0
    for i = 1:n
         =  .+ ((norm_data_m[i, :] .- ) * transpose(norm_data_m[i, :] .- ))
    end
     =  ./ (n - 1)

    # Compute matrix , the square root of the inverse of
```

```
 = real(sqrt(inv()))

# Update the features using  and
for i = 1:n
    norm_data_m[i, :] =  * norm_data_m[i, :] .-
end

# Compute the standard deviation of the data
std_v = [std(norm_data_m[:, i]) for i = 1:r]

# Get an initial split based on several trial splits
index_v = get_index_vector(n, m)
best_discrepancy = Inf
best_a = [i for i = 1:n]
for i = 1:reps_heuristic
    a = shuffle(1:n)
    norm_data_aux = norm_data_m[a, :]
    disc = compute_avg_pairwise_discrepancy(norm_data_aux, std_v, index_v, m, r)
    if disc < best_discrepancy
        best_discrepancy = disc
        best_a = a
    end
end

# Get the initial x, M, V, and d variables from the initial split
init_x = get_initial_x(n, m, index_v, best_a)
init_M = get_initial_M(n, m, r, k, init_x, norm_data_m)
init_V = get_initial_V(n, m, r, k, init_x, norm_data_m)
init_d = get_initial_d(m, r, init_M, init_V, )

# Formulate the optimization problem
model = Model(solver=GurobiSolver(OutputFlag=0))
@variable(model, d, start = init_d)
@variable(model, M[i=1:m, j=1:m, k=1:r], start = init_M[i, j, k])
@variable(model, V[i=1:m, j=1:m, k=1:r, l=1:r], start = init_V[i, j, k, l])
@variable(model, x[i=1:n, j=1:m], Bin, start = init_x[i, j])

for i = 1:m
    @constraint(model, sum(x[:, i]) >= k)
    @constraint(model, sum(x[:, i]) <= k + 1)
end
for i = 1:n
    @constraint(model, sum(x[i, :]) == 1)
end
for i = 1:m
    for j = (i + 1):m
        @constraint(model, x[i, j] == 0)
    end
end
for p = 1:m
    for q = (p + 1):m
        elem1 = sum(M[p, q, s] for s = 1:r)
        elem2 = sum(V[p, q, s, s] for s = 1:r)
```

```
            elem3 = sum(sum(V[p, q, s, s2] for s2 = (s + 1):r) for s = 1:(r - 1))
            @constraint(model, d >= elem1 +  * elem2 + 2 *  * elem3)
        end
    end
    for s = 1:r
        for p = 1:m
            for q = (p + 1):m
                @constraint(model, M[p, q, s] >= 1 / k * sum(norm_data_m[i, s] * x[i,␣
↪p] - x[i, q] for i = 1:n))
                @constraint(model, M[p, q, s] >= 1 / k * sum(norm_data_m[i, s] * x[i,␣
↪q] - x[i, p] for i = 1:n))
            end
        end
    end
    for s = 1:r
        for s2 = s:r
            for p = 1:m
                for q = (p + 1):m
                    @constraint(model, V[p, q, s, s2] >= 1 / k * sum(norm_data_m[i, s]␣
↪* norm_data_m[i, s2] * x[i, p] - x[i, q] for i = 1:n))
                    @constraint(model, V[p, q, s, s2] >= 1 / k * sum(norm_data_m[i, s]␣
↪* norm_data_m[i, s2] * x[i, q] - x[i, p] for i = 1:n))
                end
            end
        end
    end
    @objective(model, Min, d)
    @show status=solve(model)
    return getvalue(x)

end
```

Apply the function to the problem

[ ]:
```
x = split_opt(X_train_full, Y_train_full, 10, 0.6, 200)
```