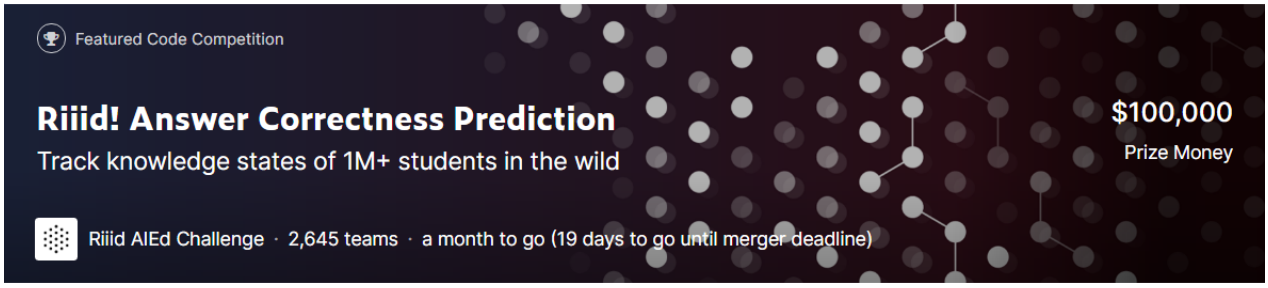

Intro to Machine Learning Applications

Final Project

Riiid! Answer Correctness Prediction

Ethan Joseph

December 13, 2020

Contents

Executive Summary

1.1 Problem

The goal of the Riiid! Answer Correctness Prediction challenge is to develop models for "knowledge tracing." Given a student's past academic performance and an overview of their educational history, these models must be able to estimate a student's understanding of the material and predict how that student will perform on future tests. More specifically, we are given millions of student interactions with an online AI-driven learning platform (Santa), and must predict how those students will answer certain questions.

Knowledge tracing models have many applications in personalized education and testing, as being able to track a student's strengths and weaknesses can help educators focus efforts and tailor their education plan.

Submissions for this challenge are scored based on the area under the ROC-curve between predictions and the target. This is a code challenge, so submissions must be Kaggle notebooks.

1.2 Data

We are given a very large scale timeseries dataset of over 100 million diverse interactions from over 700,000 students collected from *Santa*, a self-study platform equipped with an AI tutoring system.

Kaggle splits the dataset into 3 main csv files totaling 6 GB: `train.csv` (section 3.1.1) which consists of 101 million rows of student interactions, `questions.csv` (section 3.1.2) which contains metadata about the questions posed to students, and `lectures.csv` (section 3.1.3) which contains metadata about the lectures students watched. The test set is given through a python package that provides a generator for batches of test data. The test data follows chronologically from the training data. The size of the dataset and the fact that the competition requires code submissions presents the challenge of optimizing memory usage so the submitted notebook doesn't crash from running out of RAM.

1.3 Findings

Using an ensemble model of LightGBM boosted decision trees and a self-attentive transformer neural network (SAKT), I was able to achieve a submission area under ROC score of 0.772, which is only 10 spots away from scoring a bronze medal on the competition! Through my comparisons of different models, I discovered that LightGBM and SAKT perform almost equally, however the LightGBM model only achieves this score with a lot of feature engineering (almost 50 features), while the SAKT model only utilizes 3 simple features. I believe this shows that recent deep learning techniques and models might be

able to sidestep feature generation entirely and still perform comparably to traditional machine learning approaches.

Despite this, I achieved the highest scores when I ensembled the two models. I got a further increase in score after implementing state tracking during inference, where I track the state of users (number of correct answers, time taken, etc.) throughout the test set to update features. I believe this approach of ensembling decision trees and a transformer neural network and updating feature states during inference will also be very successful on other large timeseries datasets.

Data and Initial Processing

2.1 Exploratory Data Analysis

2.1.1 `train.csv`

The `train.csv` contains 101,230,332 rows and 10 variables. According to the official descriptions and some extrapolation, they are described as follows:

- `row_id` (int64) Unique ID for the row.
- `timestamp` (int32) Time between this user interaction and the first interaction from that user.
- `user_id` (int16) Unique identifier for that user.
- `content_id` (int8) ID for the interaction (also signifies if interaction was question or lecture).
- `content_type_id` (bool) 0 if content is question, 1 for lecture.
- `task_container_id` (int16) ID for the bundle of interactions. *Santa* groups questions and lectures into batches, so a user may see 3 questions in a row and a lecture before getting any explanations for them. Those interactions would share the same ID.
- `user_answer` (int8) User's answer to the question, -1 if content_id is lecture.
- `answered_correctly` (int8) If the user answered correctly (0 or 1), -1 if content_id is lecture.
- `prior_question_elapsed_time` (float32) The average time it took a user to answer each question in the bundle.
- `prior_question_had_explanation` (bool) Whether or not the user saw an explanation and the correct answer after answering the previous question bundle. Usually only false for the diagnostic test that a user sees when they initially sign up to the platform.

Checking for null values, we find that only 2 variables contain null values: `prior_question_elapsed_time` (2,351,538 null), and `prior_question_had_explanation` (392,506 null).

I computed the pairwise correlation of variables using pandas and plotted them into a heatmap (see Fig. 2.1). From the heatmap we can see two interesting correlations. The first is that `task_container_id` is correlated with `timestamp`. This is probably because each batch of questions is shown to the user in sequence, which would lead to an increasing timestamp. `answered_correctly` is also negatively correlated with `content_type_id`. This is probably because the answered_correctly is always -1 if the content type is lecture.

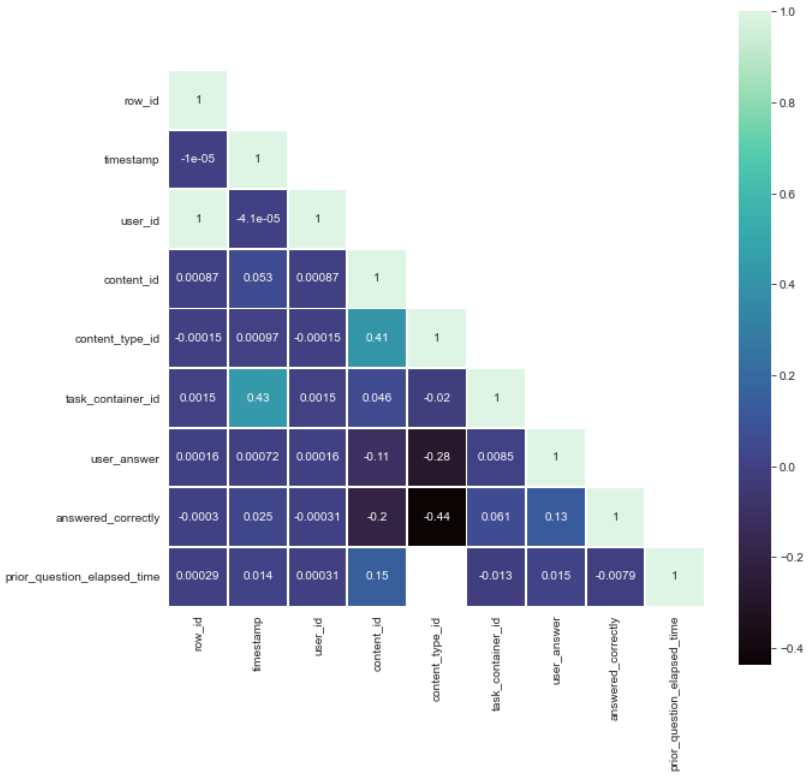

Figure 2.1: Correlation between variables in `train.csv`

I printed the total number of correct answers and incorrect answers to see if there was a 50/50 split, but it turns out that approximately 2/3rds of answers were correct ($\sim 65M$) and only 1/3rd were incorrect ($\sim 34M$).

I printed the number of unique values for each variable to see if there were any we can convert into categorical. `content_type_id`, `user_answer`, `answered_correctly`, and `prior_question_had_explanation` all have 5 or fewer unique values and should be converted to categorical.

2.1.2 `questions.csv`

- `question_id` (int32) ID that correlates with the `content_id` from `train.csv` when `content_type_id` is 0.
- `bundle_id` (int32) Code for questions served together (similar to `task_container_id`.
- `correct_answer` (int32) The correct answer for the question.
- `part` (int32) The relevant section of the TOEIC test.
- `tag` (object) Up to 6 detailed tag codes for the question in an array (meanings not provided).

I wanted to see if there was any relationship between question tags and the difficulty of the questions. After dropping null values and splitting the tag array into individual tags, I found there are 188 unique tags. I then grouped the tags by `answered_correctly` in the train.csv and plotted the 10 easiest and hardest tags (see Fig. 2.2). It does look like some question tags are much harder than others while others are easier. For example,

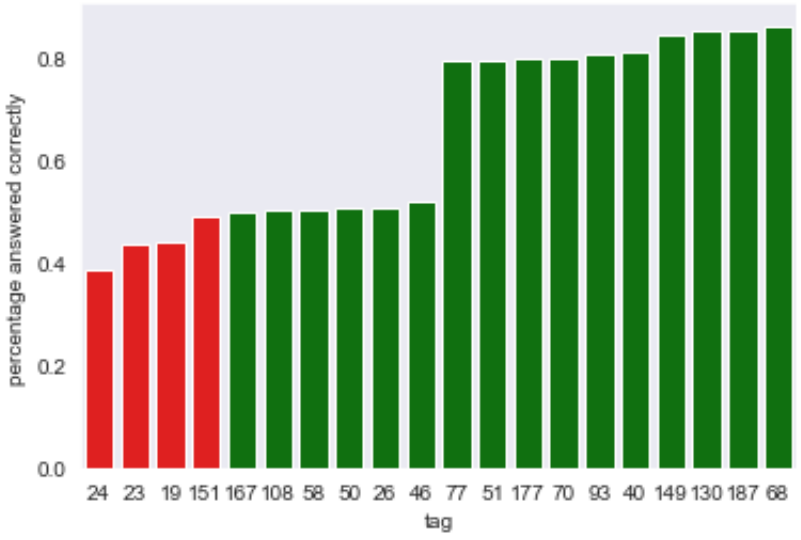

Figure 2.2: 10 hardest and easiest question tags (accuracy ≤ 0.5 is red)

questions with tag 24 have below 40% accuracy, but questions with tag 66 average almost 90%. This suggests that splitting up the question array variable into 6 individual tag variables may be worthwhile in order to preserve that relationship between individual tag and question difficulty. Since there are only 188 unique tags, I will also convert them to a categorical variable.

2.1.3 `lectures.csv`

- `lecture_id` (int32) ID that correlates with the `content_id` from `train.csv` when `content_type_id` is 1.
- `part` (int32) Category code for the lecture.
- `tag` (object) Up to 6 detailed tag codes for the question in an array (meanings not provided).
- `type_of` Brief description of the purpose for the lecture (e.g. concept, solving question).

To see if the amount of lectures in a bundle affects the percentage of questions in that bundle answered correctly, I graphed a scatterplot (see Fig. 2.3). The plot shows that there are relatively few lectures per bundle. Printing out the statistics, the data contains 9369 `task_container_ids` with lectures and 631 without, however the lectures without have 74% percent of questions correct on average while bundles with lectures average only 66% correct. I don't think this is statistically significant, and the increase in score for bundles without lectures may be due to the increased influence of outliers since there are significantly fewer bundles without lectures. From fig. 2.3 we can see that bundles with 0 lectures range from 30% correct to 80%. If anything, we can see that the bundles with lectures have a more stable percentage of questions answered correctly, ranging from only 60% to 80%.

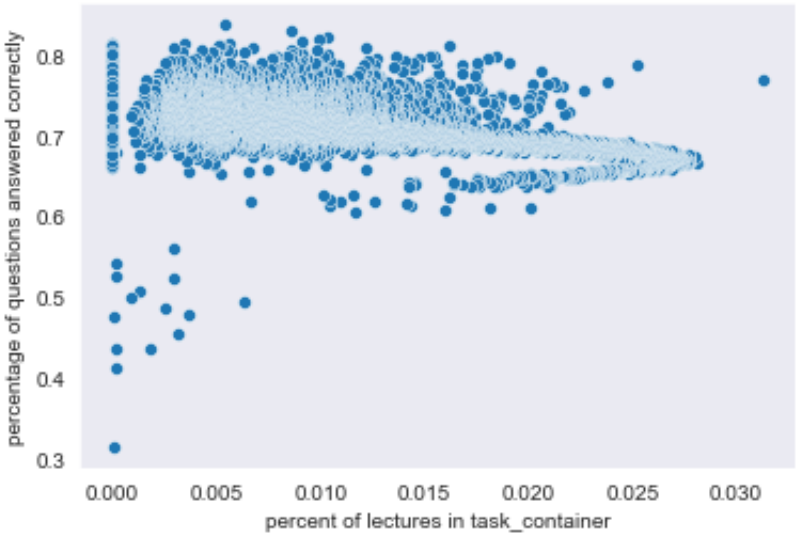

Figure 2.3: Percent of lectures in a batch vs. percent of questions in that batch answered correctly

2.2 Test data

The competition doesn't provide a test dataset file. Instead they provide a closed-source `riiideducation` package with a generator function that generates timeseries test data in batches. As a result, I have to generate features on each batch of the test data, which uses up a lot of memory.

The test data can contain new users but not new questions. It contains around 2.5 million questions and follows chronologically after the training data. This fact allows us to track user performance across test batches and use that data to update certain features (explained more in Section 4.2.5). The test data also includes a feature that wasn't included in the training data: `prior_group_answers_correct`, which tracks if a user's previous answers to questions in the test batch were correct.

2.3 Initial Processing

I did all of the feature generation and most of the model training on a google cloud AI notebook because I kept running out of memory on kaggle. I used an ec2 instance with 64GB of memory. The data was fairly clean, with only less than 2% of `prior_question_had_explanation` were NA, so I filled them with False because it was the safest assumption. An additional memory optimization step was to explicitly state the datatypes for columns in the training and question dataframes using pandas `astype`.

Since this was a code competition, the final inference step had to be done on a kaggle notebook, so after generating the features on the full training set, I exported the dataframe to a parquet format that I could load onto kaggle for submission.

I trained the LightGBM model on google cloud as well, then exported the model file once training was complete.

Benchmarks

Notebook Name	Feature Approach	Model Approach	Performance (ROC)
Riiid Classification using Naive Bayes	Question performance and student performance only	Naive Bayes	0.742
Pytorch + Entity Embedding	Tags separated, question performance features added	Deep Neural Net w/ embeddings	0.726
Riiid! Answer Correctness Prediction EDA. Modeling	Mean, median, total, standard dev., and skew of performance added	LightGBM	0.728

3.1 Riiid Classification using Naive Bayes

https://www.kaggle.com/aralai/riiid-classification-using-naive-bayes

3.1.1 Feature Selection

This notebook converted the csv documents into feather for faster loading.

There was a surprising lack of feature selection for how well the notebook performs. First, questions were grouped by `answered_correctly` into a 'total' column which contains the number of times that question was asked, and 'positive' and 'negative' columns, which contain the counts of correct and incorrect answers to the question (negative=total-positive). Students were also grouped by `answered_correctly` to get 'total', 'positive', and 'negative' columns containing the number of questions that student has answered, answered correctly, and answered incorrectly.

3.1.2 Modeling

The author used a simple Naive Bayes classifier implemented from scratch with a threshold of 10.

3.1.3 Analysis

This notebook performs better than the other two benchmarks, with an area under the ROC curve of 0.742. This is despite utilizing a less complex classifier and performing relatively little feature selection. The author selected relatively few features for modeling, but there was enough of a relationship between those two features that a classifier could perform well. This may imply that there is an increased importance on generating and selecting key features.

3.2 Pytorch + Entity Embedding (DNN)

https://www.kaggle.com/shahules/pytorch-entity-embedding

3.2.1 Feature Selection

Categorical variables were encoded into classes using scikit-learn's `LabelEncoder`, and continuous values were scaled to reduce the impact of outliers according to the scikit-learn `RobustScaler`.

The tag column was expanded into 6 separate columns for each interaction, each containing a single tag or NA if the interaction didn't have enough tags to fill the column. The notebook also adds the same features created in the Naive Bayes notebook: counts of correct answers for questions, and the counts of questions answered correctly for students.

3.2.2 Modeling

The author utilized a 4 layer neural network classifier with an embedding layer for categorical variables, and batch norm + dropout for each fully connected layer. Categorical inputs are embedded and concatenated with continuous inputs after both pass through the first fully connected layer. The model is trained with binary cross entropy loss.

3.2.3 Analysis

This notebook received the lowest score out of the benchmarks, however the neural network classifier was only trained for 1 epoch. I am curious if this was purposeful to prevent over-fitting or due to time constraints, as training on the full dataset for multiple epochs would take a significant amount of time. Since fairly good results were achieved within a single epoch of training, it might be worthwhile to try modeling with a neural network. Creating an embedding layer for categorical features also seems like a promising idea as it allows the network to better learn the relationships between those categorical classes.

3.3 Riiid! Answer Correctness Prediction EDA. Modeling (LightGBM)

https://www.kaggle.com/shahules/pytorch-entity-embedding

3.3.1 Feature Selection

The mean, median, total, standard deviation, and skew for the accuracy of each user was calculated and added as a feature. The same was done for the questions. In addition, the timestamp variable was dropped.

3.3.2 Modeling

This notebook utilized a LightGBM classifier model. They also used Optuna to search for optimal hyperparameters, and evaluated results using the hyperparameters from 70 optuna trials.

3.3.3 Analysis

This model performed reasonably well as a benchmark, but there were also many other notebooks that utilized a LightGBM and achieved higher scores. Perhaps some of the features selected in this notebook were not that important, or perhaps there weren't enough features to accurately reflect relationships in the data. Again, I believe this shows the importance of feature selection on this problem.

Modeling

4.1 Feature Selection

After looking at some of the high scoring notebooks that plotted feature importance, the most important features consistently were user's cumulative mean accuracy and the cumulative mean performance for each question. This made a lot of sense, since you can generally compare a student's performance on a test to the class average to get an idea of how that student will perform on future tests. Cumulative metrics also made sense because we want to judge a student's performance over time, so we can't simply average all of their results or we would get target leakage from the future.

Since cumulative metrics seemed to be important features I decided to add cumulative mean, std deviation, and counts for not only user correctness, but also questions, task_containers, bundles, and tags. In addition, I computed cumulative stats for the number of lectures each user watched since it seemed important from EDA. I also split the tags into 6 separate columns since each question can have up to 6 tags. This lead to a very large dataframe with almost 50 features.

4.1.1 Shifting targets

One important modification that increased accuracy was to shift the target (`answered_correctly`) down a row. This better models the test set because `answered_correctly` tells us whether or not the question in that row was answered correctly, but the test set provides `prior_group_answered_correctly` which tells whether the previous questions were answered correctly.

4.1.2 Realistic Validation Set

Since the test set only contains data from future student interactions, it was important to select a validation set that accurately reflected this. Randomly sampling the validation set from all student interactions didn't seem like a good approach because of the possibility that the model already learned the student's future performance, leading to target leakage. To generate a more realistic validation set, I simply took the last 8 interactions from each student, since this was essentially what we are trying to predict.

4.2 Model Comparison

4.2.1 Linear Model

I started by training a simple linear regressor model to serve as a baseline. It achieved an area under ROC score of 0.728 locally on the validation set, which is on par with the benchmarks, but still fairly low compared to the best performing models in the competition (around 0.8 ROC). I don't think a linear model was able to capture all the

relationships in the data, and from the EDA and Fig. 2.3 we saw that some relationships weren't linear. This was a pretty good starting baseline however, and it performed about as well as the LightGBM model and neural models I benchmarked.

4.2.2 Neural Models

I tried testing two neural models: the FastAI tabular learner, and the Self-Attentive model for Knowledge Tracing (SAKT). The FastAI tabular learner trained really fast and achieved an area under ROC of 0.748 locally after 3 epochs, which was already a large improvement over the linear model. I think a portion of the improvement can be attributed to the fact that the tabular model utilizes an embedding layer for categorical features, which was especially helpful since there were around 10 categorical features.

The SAKT model's structure is somewhat similar to the tabular learner, however it utilizes self-attention to better learn the connections between various features and questions. Self-attention is typically used in transformer neural networks to understand relationships between words in a sentence, but it makes a lot of sense to apply it to knowledge tracing, since it allows the model to learn which features are the most relevant for predicting whether or not a user will answer correctly (require the most "attention"). I downloaded a pretrained model as training it wasn't feasible on the kaggle notebook. The SAKT model implementation only allows for 3 features: `user_id`, `content_id`, and `answered_correctly`. Despite this, it performs very well with an area under the ROC of 0.760 locally.

4.2.3 LightGBM

Since gradient boosted decision trees tend to do really well on tabular data, I also trained a LightGBM model. I trained it with early stopping of 50 rounds to reduce overfitting. It achieved an area under ROC of around 0.759 locally, which was very close to the SAKT model. However, the LightGBM model had the added benefit of being able to plot the feature importance (see Fig. 4.1). Since the LightGBM model's performance was very close to the SAKT model, I decided to ensemble the two during inference on the test set.

4.2.4 Ensemble

I did a simple ensemble using the weighted average of predictions from the SAKT model and LightGBM. The notebook submission time was a few hours so it wasn't feasible to play around with the weights too much. I ended up weighting the SAKT model slightly higher than LightGBM because it performed slightly better on the validation set. I hoped to increase submission scores further by employing state tracking during inference (Section 4.2.5).

4.2.5 Tracking user's state during inference

A key improvement to the model's accuracy on the test set is tracking users during inference. The idea is to continuously update features such as cumulative mean accuracy for each user after each batch of test data since we receive the results of the previous batch through the `prior_group_answered_correctly` column. This allows the model to

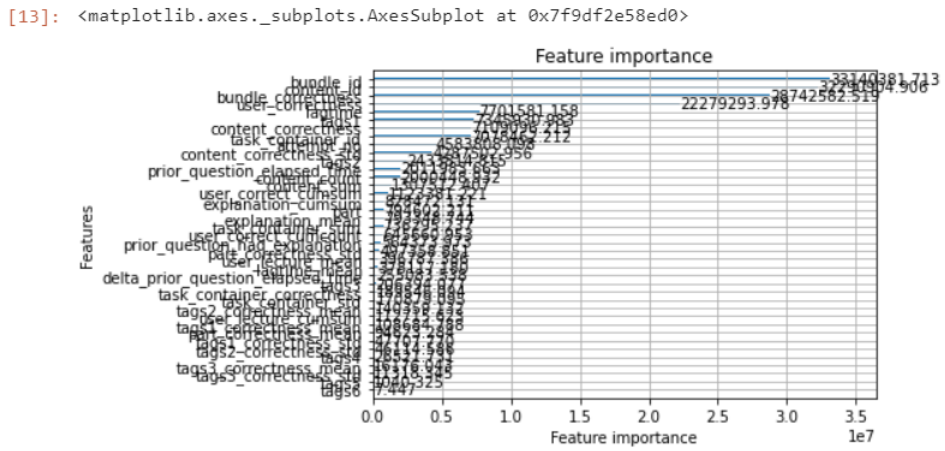

Figure 4.1: LightGBM feature importance graph

have a better understanding of new users' performance through time as we iterate through the test data, which is especially important since the test set contains users that aren't present in the training data.

4.3 Feature Importance

Figure 4.1 shows the gain of each feature for the LightGBM model. As mentioned in previous sections, `user_correctness` and `content_correctness` seem to be very important, however the bundle and content ids contributed the most to the model. This was interesting since all the notebooks I researched never had the ids as an important feature, but this makes sense since the content_id and bundle_id indicate which question is being asked, which can indicate how difficult the question is if the model learned to map question ids to difficulty. Question tags also seemed to be fairly important, which makes sense since the EDA showed how different question tags can indicate different difficulties.

4.4 Results and Conclusion

The final submission score for the ensemble model with state tracking was an area under ROC of 0.772, which is only about 10 places away from scoring a bronze medal! The competition was very challenging and required a lot of different techniques for manipulating data and modeling, but in the end I learned a lot and might try to improve my scores further while the competition is still running.

While the LightGBM model did very well, the SAKT model performed slightly better. I believe this shows that recent deep learning techniques and models might be able to sidestep feature generation entirely, since the SAKT model only uses 3 simple features and performs as well as the LightGBM model that utilized dozens of added features.

In the end, I get the best result when I ensemble the LightGBM and SAKT models and utilize state tracking. I believe this approach of ensembling tree based models and self-attentive neural networks then tracking states during inference will also be very successful when applied to other large chronological datasets, not just for knowledge tracing.

Code Repository

All notebooks are in this repository:
https://github.com/sirmammingtonham/riiid_intro_ml_final

All generated features and models are available at this link: www.kaggle.com/dataset/cd38fcd5a206d486ff7a7f97ac9c884ca1c2e34e5697de284ce52e6301d3db42