Enabling Real-Time Adaptivity in MOOCs with a Personalized Next-Step Recommendation Framework

Zachary A. Pardos
UC Berkeley
Berkeley, CA
zp@berkeley.edu

Steven Tang
UC Berkeley
Berkeley, CA
steventang@berkeley.edu

Daniel Davis
TU Delft
Delft, Netherlands
d.j.davis@tudelft.nl

Christopher Vu Le
UC Berkeley
Berkeley, CA
chrisvle@berkeley.edu

ABSTRACT
In this paper, we demonstrate a first-of-its-kind adaptive intervention in a MOOC utilizing real-time clickstream data and a novel machine learned model of behavior. We detail how we augmented the edX platform with the capabilities necessary to support this type of intervention which required both tracking learners' behaviors in real-time and dynamically adapting content based on each learner's individual clickstream history. Our chosen pilot intervention was in the category of adaptive pathways and courseware and took the form of a navigational suggestion appearing at the bottom of every non-forum content page in the course. We designed our pilot intervention to help students more efficiently navigate their way through a MOOC by predicting the next page they were likely to spend significant time on and allowing them to jump directly to that page. While interventions which attempt to optimize for learner achievement are candidates for this adaptive framework, behavior prediction has the benefit of not requiring causal assumptions to be made in its suggestions. We present a novel extension of a behavioral model that takes into account students' time spent on pages and forecasts the same. Several approaches to representing time using Recurrent Neural Networks are evaluated and compared to baselines without time, including a basic n-gram model. Finally, we discuss design considerations and handling of edge cases for real-time deployment, including considerations for training a machine learned model on a previous offering of a course for use in a subsequent offering where courseware may have changed. This work opens the door to broad experimentation with adaptivity and serves as a first example of delivering a data-driven personalized learning experience in a MOOC.

Author Keywords
Adaptivity; Personalization; Real-time intervention; MOOC; RNN; Behavioral modeling; Navigational efficiency; edX

INTRODUCTION
The path towards a more democratized learner success model for MOOCs has been hampered by a lack of capabilities to provide a personalized experienced to the varied demographics MOOCs aim to serve. Primary obstacles to this end have been insufficient support of real-time learner data across platforms and a lack of maturity of recommendation models that accommodate the learning context and breadth and complexity of subject matter material in MOOCs. In this paper, we address both shortfalls with a framework for augmenting a MOOC platform with real-time logging and dynamic content presentation capabilities as well as a novel course-general recommendation model geared towards increasing learner navigational efficiency. We piloted this intervention in a portion of a live course as a proof-of-concept of the framework. The necessary augmentation of platform functionality was all made without changes to the open-edX codebase, our target platform, and instead only requires access to modify course content via an instructor role account.

The organization of the paper begins with related work, followed by technical details on augmentation of the platform's functionality, a description of the recommendation model and its back-tested prediction results, and finally an articulation of the design decisions that went into deploying the recommendation framework in a live course.

RELATED WORK
In searching for answers to the problem of dismal completion rates in MOOCs, previous research has shown that MOOC learners often feel lost or isolated in their learning experience [9]. So far, the attempts to address this problem have largely come in the form of self-regulated learning (SRL) support interventions. For example, [10] tested the effectiveness of recommending self-regulating learning strategies to MOOC learners in the pre-course survey, but did not observe any significant changes in behavior as a result. As an example of a MOOC experiment integrated in the course content, [5] ran experiments in two MOOCs evaluating the effectiveness of providing learners with retrieval cues (to facilitate the active retrieval of information from memory) and study planning support (planning and reflecting on one's learning activities each week)—both foundational techniques in self-regulation. However, in both studies the authors report null results, with no evidence that providing this support to learners was beneficial. Another approach to instructional interventions in MOOCs is found in [17] where the authors manipulated the course discussion forum. In one condition, the course instructor was active in the discussion forum and provided support to the learners in answering their questions; in the other, the instructor was absent and the learners were on their own to discuss amongst themselves. Just as in the previous two studies, this yielded no significant change in behavior between the conditions.

To address the challenge of implementing a real-time, adaptive intervention in a MOOC, we act on the need to find a way to effectively support learners in improving their navigational efficiency with the course materials. We here present a new form of support for MOOC learners in our next step recommendation system, as prior work has shown a strong relationship between the success of a MOOC learner (measured by course completion) and the characteristics of their learning path through the course [4, 6, 18]. While novel to the MOOC context specifically, such

L@S 2017, April 20-21, 2017, Cambridge, MA, USA

DOI: http://dx.doi.org/10.1145/3051457.3051471

recommender systems have been applied to educational settings in the past, namely in intelligent tutoring systems (ITS). Both [1] and [10] provide an overview of the various approaches used to recommend and adapt course content and resources to learners in the context of ITS.

To highlight some example use cases of learning path adaptivity in prior research, we begin with an early example of real-time "task-loop adaptivity" (defined in [1] as the guiding of learners from task to task) offered in [2]. The authors here present a tutoring system which models a student's learning path in terms of correct and incorrect actions, and would adaptively intervene to guide students back to the correct path of action with immediate feedback.

The authors in [13, 12] provide real-time adaptive hints to coding assignments in the context of computer programming MOOCs. Both approaches are "step-loop" [1] in that they provide adaptive hints regarding the learners' problem solving process. However, they take different approaches in doing so; [13] models the ideal process of solving the problem in a "Problem Solving Policy," as defined by an expert, and guides learners towards this behavior. [12], on the other hand, leverages the scale of MOOCs and proposes algorithms which use the surrounding context of a code snippet to identify the problem and recommend a solution to the learner. The authors in [8] present a personalized navigation support system in the context of a JavaScript programming course. By monitoring the learner's performance on previous problems, the system presented learners with a next-step suggestion to try problems of the appropriate, or "optimal," difficulty level. By addressing the issue of learners navigating themselves to tasks that are too easy or too difficult, this system increased learner achievement and engagement.

[15], [3], and [16] describe the design and deployment of an adaptive hint generator in an ITS on the topic of logic. This system uses past learner activity data as input for a Markov decision process which, when prompted by the learner requesting a hint, provides personalized support based on the current progress through the problem. This step-loop adaptivity was empirically tested in [16] where, compared to a tutor system without adaptive hints, learners receiving the adaptive hint system earned higher grades, tried more problems, and persisted deeper into the course. While the next-step recommender system we present here does not provide hints about how to solve a given quiz or assessment problem, the suggestions we provide can be thought of as hints on how to most efficiently navigate the course.

The next-step recommendation system proposed here is course content-general and concerned solely with modeling learner behavior from the navigational patterns of peers from previous offerings of the course. This is in contrast to studies described above which are based on modeling a learner's mastery of the course topic/domain or helping them through a given task. It also differs in that the system does not acknowledge any "correct" or "incorrect" learning path as described in [2]. The system could be trained to bias towards the behaviors of certificate earners but this would miss out on serving those who do not intend to complete but nevertheless wish to make use of portions of the courseware. While the objective of the recommender is not explicitly focused on improving cognitive aspects, as was attempted to be modelled in [28], it will facilitate this in so far as past behavior has been a means to these ends, for example by recommending resources for review before a quiz. These considerations are key when it comes to the eventual evaluation of recommendation quality. A review of the work in the area of recommender systems suggests that every context in which a system operates has its own special aspects against which both the system and its success metrics must be evaluated [7] Although outside of the scope of this paper, future evaluation of this intervention might include: increasing navigational efficiency (clicks per performance), affective experience (feeling supported), as well as common outcomes such as grade and completion rate.

Thinking back to the challenge of addressing MOOC learners feeling lost in the course, we propose next-step recommendations as a service that could reach learners most in need of engagement. Pointing to recent findings from HCI research, [14] found that people are stimulated and respond positively to recommendations when they are bored. The potentially-overwhelming selection of possible next steps in a MOOC compounded with the complexity of course content can, understandably, leave a learner frustrated. A friendly next-step recommendation can be the support they need to move forward and persist.

PLATFORM AUGMENTATION

Several technical hurdles had to be overcome in order to add base functionality that would enable at-scale deployment of a real-time recommendation system within the edX platform. All solutions can be achieved without modification to open-edx and only require standard instructional design team / instructor access to edit course material.

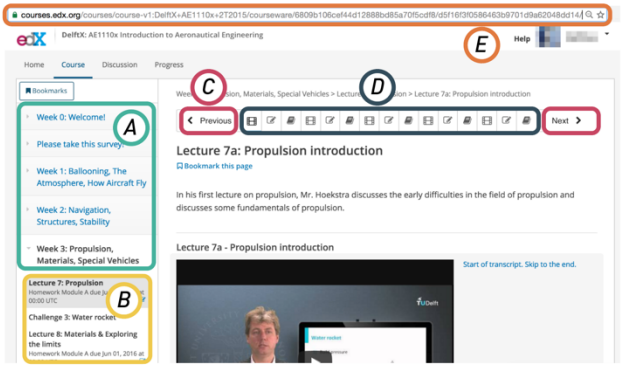

Figure 1. Annotated breakdown of edX interface components. Label (A) shows what is henceforth referred to as "Chapters," (B) refers to "Sequentials," (C) refers to navigation/goto buttons, (D) refers to "Verticals," and (E) is the page URL.

Enabling real-time logging

Our real-time recommendation requires knowledge of the student's most recent navigational events, some of which may have occurred only seconds earlier. The edX platform provides a daily event log delivery to its X consortium members but does not have a real-time data API. In order to enable access to real-time learner event logs, we set up a JavaScript logger within the xml of every page in the course which communicated to the recommendation server which events to store in the logging database. This process is illustrated in Figure 2.

The client side logging, which we describe as the sensor code, was written in JavaScript. The sensor code was responsible for gathering four items of information from the client at every page: (1) the learner's userID (2) the page's chapter (3) the page's sequential (4) the page's vertical.

The learner's anonymous ID can be queried simply enough from Segment's analytics library used by edX:

```
userid = analytics.user().anonymousId();
```

The anonymosId call has the shortcoming that it will change if the user switches devices or browsers. A non-anonymized userID call is also available, which will remain stationary throughout.

Next is the retrieval of chapter and sequential ID, both of which can be parsed from the browser URL:

```
var url = window.location.href;
var split = url.split("/");
chap = split[6];
seq = split[7];
```

The vertical ID, also known as the position ID within a sequential, is non-trivial to retrieve. While verticals can be accessed by adding the vertical number to the sequential URL, this is rarely how verticals are accessed in the course. They are most commonly accessed via the "next" and "previous" arrow buttons which are graphical navigational elements on either side of the sequential accordion view. When these "seq" events are triggered, the desired page's content dynamically replaces the current page. This dynamic loading keeps the browser URL the same (cf Figure 1) which means that the vertical position must be queried from a different source. We find this vertical position information in the edX document object model (DOM[1]).

```
var block = $('#sequence-list .nav-item.active').data('id');
vert = block.split("@").pop();
```

Arbitrarily clicking on a vertical in the accordion triggers a "seq_goto" event which is much the same as the next and previous events in how they load the page.

With all of these elements now stored, the full description of the page a learner is on can be described:

```
origin = chap+"/"+seq+"/"+vert;
```

The userID and origin are sent to a local server for logging via a cross domain aJax POST method.

Row ID	Anon Stu. ID	Origin	Rec	Followed	Previous ID	Timestamp	Time Category
100	C103	5	6	0	99	1477142712	2
101	C103	35	45	1	100	1477142732	1
102	C548	89	101	0	82	1477142736	2

Table 1. Example of entries in local mongo database

Table 1 shows the columns stored in the logging database and a few example entries. At the time of the event, only the following columns are populated: row id (transaction id), stu_id, origin, timestamp, and previous ID (the previous transaction id of the user). The remainder of the columns are populated on the subsequent event. Full client side javascript can be found here[2].

[1]All DOM related function calls used in this work are undocumented by edX and subject to change. After conducting this pilot study of the framework, we contacted edX in regards to the supportability of our approach, including providing persistent anon IDs. This support is currently under review.

[2] https://github.com/CAHLR/adaptive_mooc_LAS/

Enabling real-time recommendation

An html container is inserted at the bottom of every page which contains a template of the recommendation text. The container is marked as hidden using "display: none" until a recommendation is received successfully, upon which time the template is populated with the actual page being recommended and its title. By hiding the template until a recommendation is received, we are able to fail gracefully and shield learners from any error that may occur along the recommendation pipeline; in the case of an unsuccessful recommendation request, the page would appear to the user the same way as it would as if no intervention was added.

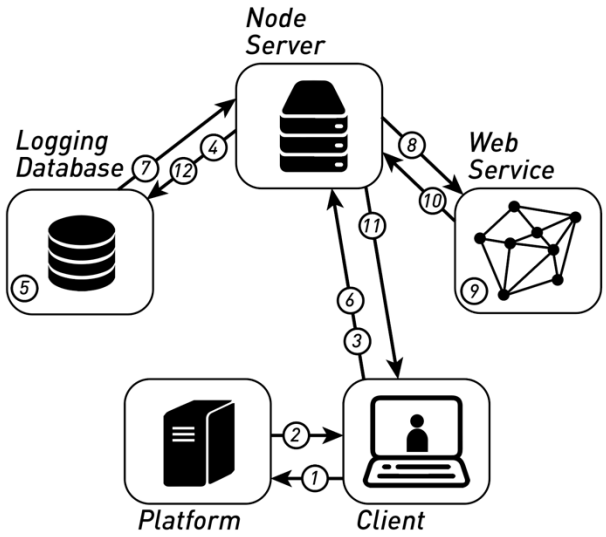

Figure 2. Diagram visualizing the entire process of delivering a recommendation to the learner. The circled numbers correspond to the numbered steps below.

The recommendation URL and title is populated by (i) sending an aJax POST to the recommendation server, which in turn (ii) looks up the learner's event history from the logging server and then (iii) passes that information to a web service which interfaces with the machine learned model. The model returns a recommendation which is passed back through to the web service. This is then sent to the recommendation server and then to the requesting client. At this point, the "Rec" column of Table 1 is filled in representing the internal index of the recommended URL, and "Followed" is set to 0. If the learner clicks on the recommended URL, a request is sent to the recommendation server, the "Followed" is set to 1, and the learner is redirected to the recommended URL. Upon loading a subsequent page, either by following the recommendation or clicking on a different navigational component, the sensor code will look up the previous event of the learner and update the time category of the past event. This is necessary since it is unknown how long the learner will spend on the page when it is first logged.

1. The learner requests a page in the course
2. The platform sends the page to the client. In the case of a "seq" event, the page is loaded in dynamically.
3. Client sensor code sends a logging event to the server
4. The server writes the event to a Mongo database
5. If a previous event exists for this student, the time category of that event is calculated and updated.
6. Client sends a request to the server for recommendation

7. The database is queried for all of the learner's past events and respective time categories.
8. The server relays this information to a Flask web service that parses the information and passes it along to the machine learned model written in python.
9. The machine learned model predicts forward until it finds a page that the user is predicted to spend more than 10 seconds on.
10. The recommended page is returned to the server.
11. The server sends this page to the client which parses a valid "200" response into a proper hyperlink and populates the to display the recommendation.
12. The server will update the logging database for this learner with the recommendation simultaneously
13. If user clicks on the recommendation, the server is contacted and the database is updated to indicate that user followed the recommendation.

The term "learner" is used when an event is triggered due to a deliberate action on the part of a human, such as clicking on a link. The term "client" is used when actions are initiated, invisible to the learner (e.g. sending a logging request), by code processed by their web browser.

Choice of Technology

In order to create this live intervention, we used a range of different technologies. NodeJS and Express were used to create the server API; Python Flask served as a light-weight web service; Python Keras was used to create the machine learned model; and Mongo was used for persistent database storage.

Server - NodeJS with Express

We decided to create our server using Node primarily because it is fast and performs well under stress. It handles operations asynchronously and facilitates a large number of simultaneous connections very well. It integrates nicely with MongoDB and can easily create routes with the Express framework.

Our API has several local lookup tables including a mapping of url to index (used for the machine learned model), index to url, url to edX path, and edX path to display name.

When the server receives a post request from the client it creates a new event with a unique user_id, origin, and timestamp. It will then check if the student has had a previous entry. If yes, the server will update the previous the timeSeq column of the previous entry and update the Recents database with this current entry. If no previous entry exists, it will skip the update in the Events database and go straight to updating the Recents database for this student. It will create an entry in the Recents database if this is the student's very first event.

After successful logging and updating, the client will ask for a recommendation for this particular student. The server will then take the student's unique user_id and query the Events database for the sequence of events and timeSeqs connected with this student. The output will then be sent to the Python web service for a recommendation.

When the web service responds, the response is checked. A lookup is then done to go from index to url as well as Edx path to name and then sent back to the client. The final JSON response will have the url, Sequential display name, and Vertical display name of the recommendation.

Web Service - Python & Flask

We decided to create web service using Python and Flask because our machine learning model was written using Python. It made it easiest to get the input into the correct format and parse the output into a simple response. Flask also allowed us to create multiple processes for parallelizability.

The web service is called after the server requests for a recommendation for a particular student. It takes in a list of the student's events and associated time categories, and then queries the machine learned model. It will receive either a -1 or an index from the machine learned model. If the response is a -1, then there is no valid recommendation (i.e., no recommendations meet the minimum time anticipated for the learner to spend on the page).

Machine Learned Model - Python Keras

Keras is a neural network machine learning framework providing functions for fast model prototyping. It has the option of utilizing Theano or tensorflow for the backend computations, both of which can utilize GPUs for accelerated training.

Database - MongoDB

We decided to use Mongo as our database of persistent storage because it is scalable and quickly handles simultaneous queries. It also has fast in-place updates and has documents stored in JSON, which makes it efficient to work with our client and server code.

Choice of Course

This framework is generally applicable to different backend recommendation algorithms with different objectives. For our purposes of navigational behavior recommendation, there were several criteria that we anticipated as important in selecting a reasonable pilot course.

Given our objective of increasing the navigational efficacy of learners, courses with more numerous pages to navigate are better candidates for demonstrating the utility of navigational recommendation. In order to learn non-trivial navigational patterns from past course events, we also wanted a course with a high amount of variation in navigational pathways exhibited by its learners. To measure this variation, we chose to treat student paths through a particular course as a Markov chain and then computed the entropy of the transition probability matrix for each course [26]. There were 13 courses evaluated offered by our deployment University partner, DelftX. Table 2 shows the entropy calculated for a variety of courses where entropy was 20 or greater. A higher amount of entropy indicates larger amounts of non-linear navigation. Since the Intro to Aeronautical Engineering course had both a high entropy and candidate assets to recommend, we selected that course for deployment.

Course	Entropy	Assets	Normalized Entropy+Assets
Intro to Aeronautical Engineering (2014)	343	1175	1.782
Intro to Water & Climate (2013)	149	1503	1.434
Intro to Drinking Water treatment (2015)	86	745	0.806
Economics of Cybersecurity (2015)	78	323	0.746

Table 2. Course suitability evaluation based on navigational entropy and asset quantity

MODELING

Modeling Navigation Behavior

The literature on cognition and learning has several theories for describing how knowledge acquisition develops over time. Far fewer theories exist for behavior, however, as it is an amalgam of many cognitive and affective factors. As such, the lack of existing theory to adequately predict navigational behavior to a high degree of accuracy means that there is also a lack of knowledge of which manually engineered features may capture student behavior. As such, we use a model that makes no assumption about behavior and instead learns these features from the raw time series data itself.

To model student navigation behavior, we chose to use the Recurrent Neural Network (RNN) architecture. RNNs are able to model time sensitive dependencies between events in arbitrarily long sequences without the need for manual feature engineering.

To provide an example, an RNN can be given a sequence of URLs a learner has already visited. The RNN maintains a hidden, continuous state that represents the past behavior exhibited by the learner. The RNN model can then output a probability distribution over the next URL the student is likely to visit. Thus, we can then take the output of the RNN as a potential recommendation to serve to the learner. The output can be augmented to also be able to predict the amount of time that a learner will spend on the resource. With this augmentation, we can then choose to only provide recommendations where there is expected to be a significant amount of time spent on the URL. This helps expedite the learner's navigation through the course by skipping less useful content.

To use an RNN model, the logs of student actions must be parsed so that each student can be represented by a single list which contains each unique course URL the student has visited. Additionally, the timestamp associated with each course URL visit is also tracked. These timestamps are used to create a proxy for the amount of time spent on a resource. We investigate whether adding time spent as an input to the RNN model improves its predictive accuracy, and investigate two model modifications to incorporate time spent as an input.

Understanding edX logging of navigational events

Parsing a data log of student actions is not trivial. In this work, the ultimate goal of parsing through the data log is to obtain the sequence of course URLs that each student has visited, as well as the timestamp associated with each visit. The data log contains other student events, such as pausing videos and answering quiz questions. For this work, such rows were dropped. Thus, only navigation events were kept, where navigation is defined as visiting a specific course URL. These navigation events were then parsed to resolve to a specific course URL. Each URL contains a chapter hash, a sequential hash (which refers to sections within a chapter), and a vertical hash (which refers to a specific course page within a section). For example, a URL represented by 'abc123/zzz444/2' would have a chapter hash of 'abc123', a sequential hash of 'zzz444', and a vertical value of '2'. Thus, each navigation event in the edX data log can be resolved to a specific URL. However, each event in the raw log unfortunately does not directly map to a URL without an extra step of processing. Navigation events can be found in rows where either:

1. The row is a ***seq event***. Seq events include *seq_next*, *seq_prev*, or *seq_goto*. Next and prev refer to moving directly forward or backwards one vertical. Goto is a jump to any vertical within a single sequential.

   OR

2. The row contains a direct course page URL. In the URL, the vertical may be given directly, or the vertical may be missing.

Both types of navigation events mentioned above have data processing quirks. *Seq_next* and *seq_prev* events contain the sequential hash and the vertical that is navigated to. Using the sequential hash, the chapter hash can be inferred, since there is only one sequential hash per section in the course, and each section only belongs to one chapter. The vertical displayed by the row, however, may need to be additionally processed when *seq_prev* is invoked on the first vertical in a section or *seq_next* is invoked on the last vertical in a section. For example, the row in the data log may contain a *seq_next* to vertical 7 in a particular section. However, that section might only contain 6 verticals. This event should actually point to vertical 1 of the next section. Thus, the processing code must be able to handle when navigating to the previous and next sections when the current vertical is at the beginning or the end of the section. Once the corresponding sequential, chapter, and vertical hashes are resolved, a URL can be constructed to represent the URL that the student is now at in this row.

For the second type of navigation event, where the row contains a direct course URL, when the vertical is included in the row, the URL can be directly taken from the row itself. When the vertical is not included in the row, which means that the row contains a chapter hash and a sequential hash, but no vertical value, then the vertical must be inferred from the student's past actions. The server stores the most recent vertical a student was at for each section in the course. Thus, the processing code must keep track of the most recent vertical accessed for each section in the course, and when a row contains a direct course URL without a vertical, the vertical must be inferred from the previously stored most recent vertical for that section.

One other important note is that the rows of the original data file may not actually be in sorted, ascending order by time. In our processing, we found that while some rows seemed to be in ascending order, some rows were actually sorted in descending order.

Thus, each student is associated with a list of URLs they visited, processed from the original data log. There are a fixed number of possible course page URLs, which can be represented by the possible combinations of chapter, sequential, and vertical hashes. If there are 200 unique URLs in a course, then the indices from 1 to 200 can each correspond to one of the URLs. Once this mapping between index and URL is established, each student's set of actions can be represented as a list of indices.

Recommendation model design

This sub-section provides context to how the RNN and LSTM architectures function. RNNs maintain an ongoing latent hidden state that persists between each input to the model. This latent state can provide a representation of what has already been seen in the input sequence. Long Short-Term Memory (LSTM) is a modification of the RNN architecture, where the hidden latent state is replaced with a more powerful memory component. We chose to use LSTMs due to their stronger performance in modeling longer range dependencies [19, 20].

RNNs maintain a latent, continuous state, represented by $\boldsymbol{h_t}$ in the equations below. This latent state persists in the model between inputs, such that the prediction at $\boldsymbol{x_{t+1}}$ is influenced by the latent state $\boldsymbol{h_t}$. The RNN model is parameterized by the input weight

matrix W_x, recurrent weight matrix W_h, initial state h_0, and output matrix W_y. b_h and b_y are biases for the latent and output units.

$$h_t = tanh(W_x x_t + W_h h_{t-1} + b_h)$$

$$y_t = \sigma(W_y h_t + b_y)$$

LSTMs, a popular variant of the RNN, augment the latent, continuous state with additional gating logic that helps the model learn longer range dependencies. The gating logic learns when to retain and when to forget information in the latent state. Each hidden state h_t is instead replaced by an LSTM cell unit with the additional gating parameters. The update equations for an LSTM are:

$$f_t = \sigma(W f_x x_t + W_{fh} h_{t-1} + b_f)$$

$$i_t = \sigma(W_{ix} x_t + W_{ih} h_{t-1} + b_i)$$

$$C'_t = tanh(W_{Cx} x_t + W_{Ch} h_{t-1} + b_C)$$

$$C_t = f_t \times C_{t-1} + i_t \times C'_t$$

$$o_t = \sigma(W_{ox} x_t + W_{oh} h_{t-1} + b_o)$$

$$h_t = o_t \times tanh(C_t)$$

f_t, i_t, and o_t represent the gating mechanisms used by the LSTM to determine when to forget, input, and output data from the cell state, C_t. C'_t represents an intermediary candidate cell state that is gated to update the next cell state.

LSTM Model Description and Training

LSTM models have several hyperparameters, which refer to values that affect how the model performs on a given set of data. Evaluating which hyperparameters work best for a given model and dataset can be done in one of several ways, and is usually resolved with some empirical experimentation. For this analysis, we varied the following hyperparameters: number of LSTM layers and number of hidden nodes per LSTM layer. Each model was trained using either 1, 2, or 3 LSTM layers, as well as 64, 128, and 256 nodes per LSTM layer. Thus, each LSTM model is trained with 9 different hyperparameter sets.

To create a behavior prediction LSTM model, the model needs to be trained to predict the next URL given a prior sequence of URLs visited. This is our baseline LSTM model, where the inputs and outputs are simply indices corresponding to unique URL accesses. The model is trained in batches of 64 student sequences at a time using back propagation through time [21]. Categorical cross entropy is used to calculate loss and RMSprop is used as the optimizer. Drop out layers were added between LSTM layers as a method to curb overfitting [22]. An embedding layer with 160 dimensions is added to convert input indices to a continuous multi dimensional space, a technique commonly used in language modeling [23]. LSTM models were created using Keras [24], a Python library built on top of Theano [25].

Figure 3 details an example pipeline where the first two timesteps of a student sequence of URL accesses is shown. The two URLs in the student's sequence are converted to an index representation of that URL, which is then fed to the LSTM model. The index is implicitly converted to a one-hot vector representation by the embedding layer used by the Keras LSTM model. The output of the model uses the softmax function to normalize the outputs to sum to 1, so that the values within the output vector could be thought of as probabilities of that index being the predicted next URL. If there are 300 unique course URLs, for example, then the output vector would be of length 300, where each value of the vector corresponds to the probability that the next URL in the sequence will be that index value. Thus, to find the most likely next URL, one needs to find the index of the vector that has the maximum probability, and then consult the one to one mapping between indices and URLs to find which URL that index corresponds to. Note that in the example figure, index 32 of the softmax output in timestep 1 has the highest probability. Thus, according to the model, the most likely next URL would be the URL corresponding to index 32. In the example, this prediction turns out to be correct, as it is shown that the actual input in the next timestep is associated with that URL.

Incorporating time into the model

The previous subsection described a baseline LSTM model, where only the sequence of URL visits was modeled. We hypothesize that prediction accuracy of the next URL can go up if the model were to incorporate the amount of time spent on each resource. Unfortunately, there is no way to know exactly how much time the student is truly paying attention to a particular URL. We can approximate time spent, however, by calculating the time difference between each URL visit. Thus, we approximate the time spent on a URL by taking the time difference before accessing the next URL.

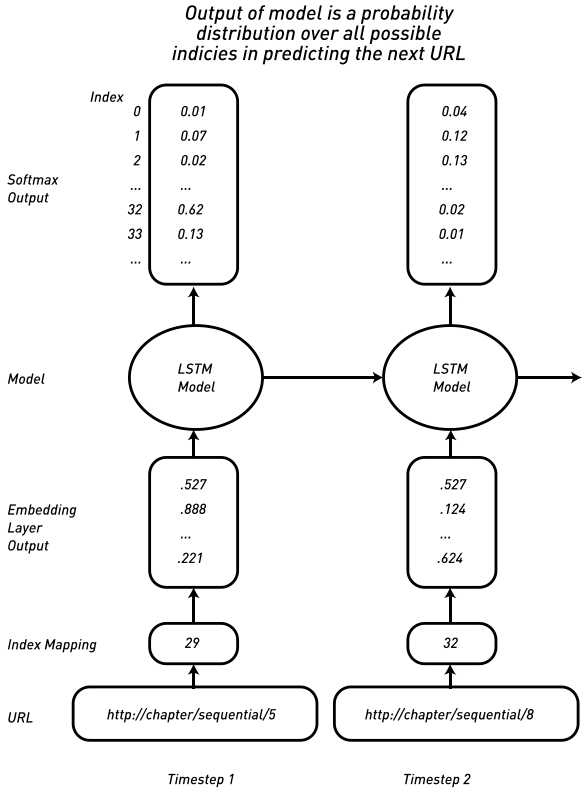

Figure 3. Depiction of baseline LSTM architecture

The baseline LSTM model [20] can be augmented to be able to incorporate time spent in addition to the standard input and output of the current and next URL index.

We propose two methods for incorporating time spent into the input of the model. These two methods are referred to as bucketed-time-input and normalized-time-input. These two methods of input are explained next.

Bucketed-time-input refers to an augmented input, where an additional one-hot vector is concatenated with the original baseline input. Figure 4 depicts this additional time input processing step in a graphical format. This additional one-hot

vector indicates the amount of time spent on the resource relative to four pre-determined buckets: between 0-10 seconds, 11-60 seconds, 61-1799 seconds, and finally 1800 and beyond seconds. These buckets were chosen qualitatively, rather than with a data driven approach, to be able to prescribe real world interpretation to the time buckets.

Normalized-time-input refers to an augmented input where an additional two-dimensional vector is concatenated with the original baseline input. Figure 4 depicts how the normalized-time-input is incorporated into the architecture. The first dimension of this vector takes a value between 0 and 1, which is calculated by dividing the time spent on the resource by 1800, or if the time spent is greater than 1800, then the value is taken to simply be 1. Thus, a time spent of 900 seconds would be converted to 0.5. This is considered normalizing the time by 1800 seconds. The second dimension of the vector is simply 1 if the time spent is over 1800 seconds, and 0 otherwise.

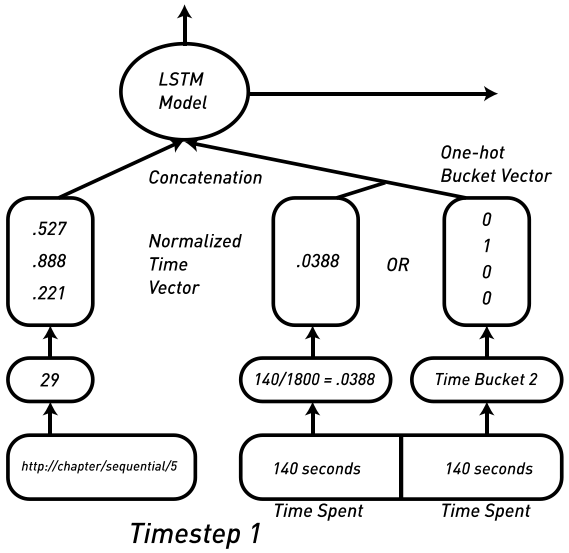

Figure 4. Depiction of two methods of adding dwell time to the model; Normalized continuous (0-1) and time bucketed (1-4)

It is also possible to incorporate time into the output of the model. The non-time version is referred to as non-concatenated-output, while the time incorporated version is referred to as concatenated-output.

Non-concatenated-output refers to the standard output, where the object of prediction is simply an index, where each index has a one-to-one mapping with a course URL. Concatenated-output refers to an output space where the number of indices possible is multiplied by four, so that one could think of a time bucket being concatenated with each index. Each possible course URL now has four associated indices with it, where each index represents a course URL and the amount of time spent on that URL, where time spent is bucketed in the same fashion as the bucketed-time-input. We can compute the overall likelihood for a particular URL by adding the probabilities among all four indices associated with a particular URL. Since each output is also associated with a time, one can look at only indices associated with a particular time bucket. Thus, the output can now be queried to find the most likely URL to visit among each possible time category.

With these methods of input and output defined, we propose the following models:

Attributes:

(a) Input time treated as continuous
(b) Input time treated as categorical
(c) Input time concatenation with vertical after embedding
(d) Time category concatenated with vertical in the output

1. Baseline LSTM model: Inputs and outputs are indices, where each index has a one to one mapping to a unique course URL.

2. Bucketed-time-input, non-concatenated-output (b,c)

3. Bucketed-time-input, concatenated-output (b,c,d)

4. Normalized-time-input, non-concatenated-output (a,c)

5. Normalized-time-input, concatenated-output (a,c,d)

Deployment Course Dataset and Prediction Results

The pilot course, DelftX Intro to Aeronautical Engineering 2015, contained log data from 27024 unique learner ids. However, for the purposes of behavior recommendation, we chose to filter the data to only include learners who attempted at least one problem check, resulting in data logs from 9,172 learners. From the data logs, we again filter the data to only include data regarding course page navigations, thus excluding events related to lecture video pausing, problem viewing, and so on. We chose to also filter out contiguous repeats of URL accesses. This means that if there are multiple visits to the same URL in a row, we removed duplicates such that there only remained one access to that URL for a student sequence representation. For the time spent associated with sole URL used in place of the duplicate contiguous URLs, we took the maximum time spent among the duplicated URL accesses. Time spent is calculated, in general, by taking the timestamp of a URL access and calculating the future difference to the timestamp of the next URL access in the sequence. There were 336127 navigation events in the 0-10 second bucket, 248918 in the 11-60 second bucket, 338144 events in the 61-1799 bucket, and 123287 events in the 1800 seconds and beyond bucket.

There was a total of 286 possible course URLs, which means there were 286 possible unique verticals to model, spread over 38 sequentials. The median number of verticals in a sequential was 6, with a maximum of 19. The course was self-paced, which means that assignment due dates were not fixed, and all of the course content was released at the beginning of the course. Log data was filtered to only include data from roughly the time period that the course officially ran, from May 31, 2015 to June 3, 2016.

Hill-climbing Validation Early Stopping

The 5 LSTM models described in the previous section were each trained under the 9 different hyperparameter settings described in section 3.1.4. The data was split into two sets, a training set and a held-out test set. The training set comprised sequences from a randomly selected 70% of the users, while the test set contained the remaining 30%. Within the training set, 10% of the sequences were held out as a hill-climbing validation set. During training of a particular model, if the loss calculated on the hill-climbing set did not obtain a best result for 3 consecutive epochs, then training was halted for that model and the best result was recorded. This was our early stopping criterion.

Baselines

An n-gram model is included as another sequential model for comparison. N-gram models capture the structure of sequences through the statistics of *n*-sized sub-sequences. The model predicts each sequence state x_i using the estimated conditional probability that x_i follows the previous *n-1* states in the training set. We trained n-grams with values of n between 2 and 10, while

also instituting a "back-off" policy when there are too few subsequences. For each n-gram, we instituted back-off policies of between 0 and 10 occurrences, so that a particular sub-sequence of size n must occur at least the number of times as the back-off policy, or else that sub-sequence is not used. The back-off policy prevents the n-gram model from using very sparse data, requiring a minimum number of occurrences for that sub-sequence to be used. If there are too few occurrences of a particular n-sized sub-sequence, the model "backs off" and uses the values for the (n-1)-gram model, and so on. The best performing n-gram model on the validation set had an n-value of 7 and a back-off value of 8. A 2-gram model is also included, representing a model predicting the most common URL following a particular URL. We call this model the "Next most common" model. The last baseline is dubbed the "Next syllabus URL" model, which predicts the next URL in the course structure; this is equivalent to the page learners are taken to when they click on the "Next" button in the native edX interface.

Back-tested Prediction Results

Validation accuracy and test set accuracy is shown in Table 2. For each model, the hyperparameter set that reached the highest validation accuracy was used. Thus, for each LSTM model listed in the table, only the highest achieving hyperparameter set results are shown, where training stopped according to the early stopping rules described previously. Accuracy refers to average accuracy per student sequence; thus a next URL prediction accuracy is established per student sequence, and then the averages from all students are averaged together. For baseline outputs, the models produce an index which has a one to one mapping with a URL. Thus, if the most likely index produced by the model matches the actual next URL in the sequence, that is counted as a correct prediction within a student sequence. For concatenated outputs, the models produce an index which has a four to one mapping with a URL, meaning there are four possible indices that all correspond to the same URL, just with a different time spent predicted. For the purpose of accuracy, as long as the URL mapping of the index is correct, then the prediction is counted as correct. Thus, accuracy for concatenated outputs drops the time component from the output in calculating correctness.

Model Input / Output	Validation Acc.	Test Set Acc.
Bucket /Non-Concat.	63.5	64.0
Norm / Concat.	62.6	63.5
Bucket / Concat.	63.0	63.3
Norm / Non-Concat.	62.9	63.3
Baseline LSTM	62.0	62.5
Best *n*-gram (7)	61.6	61.7
Next most common	55.1	55.6
Next syllabus URL	51.5	52.0

Table 2. Prediction accuracy results

REAL-TIME DEPLOYMENT

Recommendation Interface

This section describes our rationale for how to best integrate the recommendations into the learner's course experience. We primarily consider two key aspects of the interface: (i) the visual appearance of the recommendations and (ii) the linguistic framing of the accompanying text.

Visual Appearance

As the interface is housed within the edX platform and course materials, it is important that the appearance of the recommendations is seamless. This ensures both a sense of trust from the user---in that it looks like it's a natural part of the edX course---and assuages the risk that the recommendations act as distractions to the learners. Given the simplicity of the edX user interface design, this was not hard to achieve. And to make following the recommendations more intuitive, we also add a "Go" button that learners can use as an alternative to clicking on the plain text link. These appear at the bottom of every page in the course---made directly available to the learner at all times.

Linguistic Framing

Just as we did not want the visual appearance of the recommendation to be too overwhelming in the existing course interface, we likewise aimed to present the accompanying text in a way that clearly communicates the benefit of this resource while not sounding overly authoritative. While definitely an avenue for future experimentation (what is the most effective way to frame such recommendation text to learners?), we eventually decided on "Suggestion for you… Consider visiting: [Recommended next-step]." This text accomplishes the task of communicating to the learner that this recommendation is indeed personalized and unique to him or her (without explaining how) and also making it clear that following this recommendation is optional. Figure 5 shows the final design of the recommendation interface.

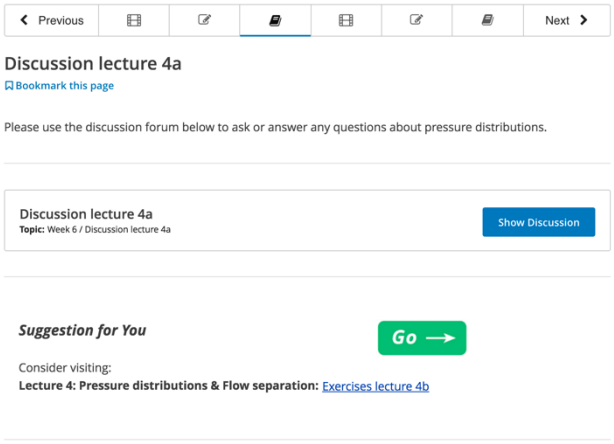

Figure 5. Final design of the recommendation interface

We are able to show the text of sequential and vertical being suggested through a lookup table we created from the course xml.

Model Usage Considerations

Training a model based on a previous offering of the course

Since the navigation behavior model proposed in this paper is behavior and data driven, a requirement to deploy such a model in a live course is that behavior from the course must already exist. To perform our live case study, we selected a MOOC that had multiple offerings over time so that we could use behavior from a completed iteration of the course to train our behavior models. Since the model is trained on a specific structure of course URLs, the current iteration of the course should not deviate too much, preferably at all, from the iteration that the model was trained on.

To deploy our behavior model in the 2016 offering of the Aeronautics Engineering course, we trained on the behavior data from the 2015 offering of the course.

Taking into Account Changes in The Courseware

In our case study, the majority of the course structure was held exactly the same. However, the first chapter to the course was re-ordered. Our behavior model implicitly incorporates the ordering of the course in its predictions, so that any re-ordering of course content would adversely affect its prediction. Therefore, we chose to drop all events related to the first chapter of the course from both the training and the live recommendation data sets. This was deemed acceptable since the first chapter for this course was an introduction to the course staff and logistics.

Additionally, one unique URL was added to the current version of the course. Thus, the trained model has no knowledge or ability to recommend that URL. However, the actual recommendation code can be altered so that when it is detected that a student is near the new URL, the recommendation code can choose to temporarily suspend usage of the behavior model and either suggest going to the new URL directly or simply not suggesting a URL temporarily.

Any deviation in course structure from the training environment needs consideration in handling. Special recommendation logic must be put in place when the live version of the course differs in ways that the original model cannot account for.

Description of the Recommendation Engine

The machine learned model is the contact point between the underlying LSTM behavioral model and the code that serves a clickable link on the learner's browser. The LSTM model has been trained to produce an output that contains a probability distribution over all possible course pages. Our time-concatenated output LSTM models additionally contain time information in each of the output indices. With this time-concatenated probability distribution, it is reasonable to simply take the most likely page and serve that as the model's recommendation. With the time-augmented output, however, the recommendation engine can instead be configured to recommend a URL that the learner is likely to spend a significant amount of time on, for example between 10 seconds and 30 minutes. The hypothesis behind this logic is that if the model only expects the learner to spend fewer than 10 seconds on a resource, then it may be the case that the learner is trying to skip over it on her way to the eventual resource of interest. The recommender gives the learner the skip directly to that eventual resource of interest. It could be reasonable to recommend pages where the learner is expected to spend more than 30 minutes on; However, we chose not to include these as part of our recommendation engine configuration, since it could be possible that such a lengthy time spent on a page could really be indicative of a time-out event, where the learner has actually just left the page, potentially after consulting an ineffective page.

Another method for producing a recommendation could instead be to repeatedly query the behavioral model until the most likely page corresponds to a desirable time bucket, where each repeated query has a "hypothetical action" appended as the most recent event. For example, if a student is currently at a quiz page (Figure 6), then the behavioral model would be queried using that student's past behavior as well as the current quiz page. The time spent on the current quiz page is not known yet since the learner has not navigated away from that page at the time of the query. Thus, time spent on the current page must be approximated in some way; we use the modal time bucket as a place holder (11-60 seconds) and the real time spent is filled in after the next navigation event. The model then produces a probability distribution over time-concatenated indices, as usual. If the most probable page is in a desirable time bucket range, then the engine recommends that URL. However, if the most probable page is not in a desirable time bucket range, then instead of recommending this page, the engine temporarily appends it to the student's "hypothetical" path until a desirable time bucket recommendation is reached. Thus, through repeated querying of the model, eventually a page in the desirable time bucket range would be reached, and the engine would use this as the recommendation. The reasoning behind such a model would be, for example, to skip through many URL accesses that are under ten seconds (undesirably short time spent) and instead recommend a URL that the student would likely have eventually dwelled on. We refer to this as a forward-stepping process, where we create hypothetical forward steps to the model. The page used in the case shown in Figure 6 is *Video 1*, a recommendation which is inserted into the page after the query completes and which the student followed in this example. After the learner in the example visits *Video* 1, she is suggested to return to the quiz but instead navigates to *Text* 3.

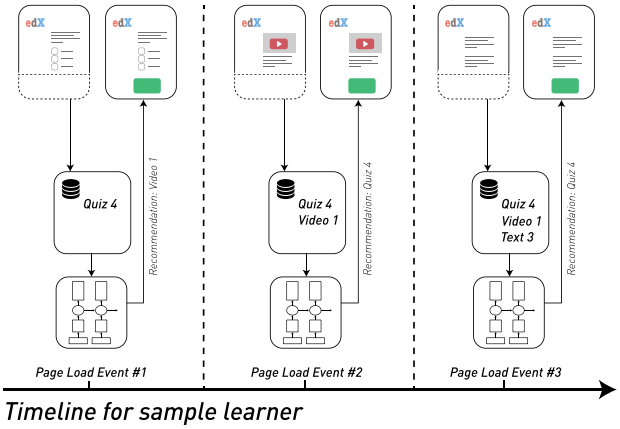

Figure 6. An example of the framework delivering three recommendations at three consecutive page visits for a learner

We chose to use the time-bucketed-input with time-concatenated output model discussed previously. For the live recommendation pilot, we chose to retrain the LSTM model on the entire set of the previous course offering's data, as opposed to the original training only using 70% of the data. We only used the validation set's best hyperparameters.

Our next step recommendation can be seen as predicting what page a learner wants or will eventually want, and directly linking them to that page in advance. When to consider if what learners want is different from what they need to achieve their goals or the goals of the course is a matter for consideration by future work as well as the appropriate role of a platform, courseware, and personalization in facilitating these goals.

CONTRIBUTIONS

In this paper, we made three contributions to adaptive personalization in a MOOC. The first was to solve the issue of real-time learner event logging required for data-driven intervention with a client side JavaScript solution that records learner navigational events. The second was to introduce a novel behavioral model which predicted the next page a learner was likely to spend significant time on which outperformed existing prediction baselines. Lastly, we combined the first two contributions to provide the first proof-of-concept realization of a real-time data-driven recommendation framework in a live MOOC along with the edge cases and design considerations that needed to be handled in order to deploy.

ACKNOLWEDGEMENTS

We would like to acknowledge our use of the edX partner's Research Data Exchange (RDX) program and the support contributed by the edX data team. We also acknowledge the support from TU Delft's Office of Online Learning, its director, and the DelftX Aeronautics Engineering course team for their cooperation.

REFERENCES

1. Aleven, V., McLaughlin, E., Glenn, R. A., and Koedinger, K. (2016) Instruction based on adaptive learning technologies. *Handbook of research on learning and instruction*. Routledge.
2. Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences* 4, 2 (1995), 167–207.
3. Barnes, T., and Stamper, J. Toward automatic hint generation for logic proof tutoring using historical student data. In *International Conference on Intelligent Tutoring Systems*, Springer (2008), 373–382.
4. Davis, D., Chen, G., Hauff, C., and Houben, G.-J. Gauging mooc learners adherence to the designed learning path (2016) In *Proceedings of the 9th International Conference on Educational Data Mining (EDM)*. 54–61.
5. Davis, D., Chen, G., van der Zee, T., Hauff, C., and Houben, G. J. (2016) Retrieval practice and study planning in moocs: Exploring classroom-based self-regulated learning strategies at scale. In *European Conference on Technology Enhanced Learning, Springer*. 57–71.
6. Guo, P. J., & Reinecke, K. (2014). Demographic differences in how students navigate through MOOCs. In *Proceedings of the first ACM conference on Learning@ scale conference* (pp. 21-30). ACM.
7. Herlocker, J. L., Konstan, J. A., Terveen, L. G., & Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1), 5-53.
8. Hsiao, I.-H., Sosnovsky, S., and Brusilovsky, P. (2010) Guiding students to the right questions: adaptive navigation support in an e-learning system for java programming. *Journal of Computer Assisted Learning* 26, 4, 270–283.
9. Khalil, H., and Ebner, M. (2014) Moocs completion rates and possible methods to improve retention-a literature review. In *World Conference on Educational Multimedia, Hypermedia and Telecommunications*, no. 1, 1305–1313.
10. Kizilcec, R. F., Pérez-Sanagustín, M., & Maldonado, J. J. (2016). Recommending self-regulated learning strategies does not improve performance in a MOOC. In *Proceedings of the Third ACM Conference on Learning@ Scale* (pp. 101-104). ACM.
11. Kopeinik, S., Kowald, D., and Lex, E. (2016) Which algorithms suit which learning environments? A comparative study of recommender systems in tel. *In European Conference on Technology Enhanced Learning*, Springer, 124–138.
12. Nguyen, A., Piech, C., Huang, J., and Guibas, L. (2014) Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web, ACM*. 491–502.
13. Piech, C., Sahami, M., Huang, J., & Guibas, L. (2015). Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second ACM Conference on Learning@ Scale* (pp. 195-204). ACM.
14. Pielot, M., Dingler, T., Pedro, J. S., and Oliver, N. When attention is not scarce-detecting boredom from mobile phone usage. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ACM (2015), 825–836.
15. Stamper, J., Barnes, T., Lehmann, L., and Croy, M. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track* (2008), 71–78.
16. Stamper, J., Eagle, M., Barnes, T., and Croy, M. Experimental evaluation of automatic hint generation for a logic tutor. *International Journal of Artificial Intelligence in Education* 22, 1-2 (2013), 3–17.
17. Tomkin, J. H., & Charlevoix, D. (2014). Do professors matter?: Using an a/b test to evaluate the impact of instructor involvement on MOOC student outcomes. In *Proceedings of the first ACM conference on Learning@ scale conference* (pp. 71-78). ACM.
18. Wen, M., and Rosé, C. P. Identifying latent study habits by mining learner behavior patterns in massive open online courses. In *CIKM '14* (2014), 1983–1986.
19. Bengio, Y., Simard, P., and Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157-166, 1994.
20. Gers, F. A., Schmidhuber, J., and Cummins, F. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451-2471, 200.
21. Werbos, P. J. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339-356, 1988.
22. Pham, V., Bluche, T., Kermorvant, C., and Louradour, J. Dropout improves recurrent neural networks for handwriting recognition. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pages 285-290. IEEE, 2014.
23. Mikolov, T., Karafiat, M., Burget, L., Cernocky, J., and Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, volume 2, page 3, 2010.
24. Chollet, F. Keras. https://github.com/fchollet/keras, 2015.
25. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.
26. Reddy, S., Labutov, I., and Joachims, T. Latent skill embedding for personalized lesson sequence recommendation. *CoRR*, abs/1602.07029, 2016.
27. Tang, S., Peterson, J. C., & Pardos, Z. A. (2016). Modeling Student Behavior using Granular Large Scale Action Data from a MOOC. *arXiv preprint arXiv:1608.04789*.
28. Pardos, Z.A., Bergner, Y., Seaton, D., Pritchard, D.E. (2013) Adapting Bayesian Knowledge Tracing to a Massive Open Online College Course in edX. D'Mello, S. K., Calvo, R. A., and Olney, A. (eds.) *Proceedings of the 6th International Conference on Educational Data Mining* (EDM). Memphis, TN. Pages 137--144.