Course Friction Explorer

Visualizing and validating indicators of student struggle

Marie Salomon
University of British Columbia
mariesal@cs.ubc.ca

Noa Heyl
University of British Columbia
falkirks@cs.ubc.ca

Shizuko Akamoto
University of British Columbia
shizuko@cs.ubc.ca

ToTo Tokaeo
University of British Columbia
tokaeo@cs.ubc.ca

Abstract

In UBC's upper-year undergraduate software engineering course CPSC 310 students work on a term-long project. CPSC 310 is a large course, with typically more than 300 registered students, which makes it extremely difficult for the course staff to determine when and why a student is struggling in the course. There could be multiple different factors to why a student is falling behind and potentially failing the course. Possible indicators for CPSC 310 staff to identify a struggling student could be a change in the programming behavior or usage of the Q and A platform. Although the staff could manually go through this data, it would require a large amount of time. We present Course Friction Explorer, an interactive visualization dashboard that leverages a suite of visualizations idioms and techniques to simplify and accelerate the exploration and understanding of CPSC 310 data collected. With our tool, the staff can determine the causes of student struggle in the course by interactively examining the data and constructing indicators for comparison.

1. Introduction

Identifying struggling students is a constant interest and challenge for course staff to ensure course quality. Struggling students, if unattended, undermine course quality as they are blocked from making effective progress in learning. Despite the criticality of this task, course staff often find it difficult to detect when a student is struggling, which delays timely intervention and brings negative repercussions to overall course quality. The issue is especially real for courses with large class sizes or those with a limited number of fine-grained assessments to gauge students' progress.

The software engineering course CPSC 310 at UBC Vancouver fits both of these properties. Each CPSC 310 offering typically has over 300 students enrolled, and its main evaluation method is a single term-long project consisting of four checkpoints, each spanning a period of 2-3 weeks. At each checkpoint, AutoTest, an automatic test runner, invokes a set of private tests against each student's solution code, giving feedback in the form of a test failure report. The AutoTest feedback, although explicitly indicating struggles, is too infrequent for achieving timely intervention. Throughout the course of the project, students have access to resources like labs, office hours, and Piazza, where they can explicitly seek help from the course staff. These resources are also important grounds for the course staff to identify and reach out to struggling students. But in reality, due to the large size of the classroom, it is impossible for the course staff to reach out to every potentially struggling student. Thus, explicit signs of struggle, although obvious, are most often not effective.

We developed Course Friction Explorer with the goal of facilitating identification of strugglers in a course like CPSC 310. It is a visualization dashboard, where a course staff can discover and compare many different indirect signs of student struggles, more useful for making early intervention than the explicit signs. We claim that Course Friction Explorer makes three major contributions:

1. We define 18 different derived student attributes from our original dataset consisting of multiple tables, thereby forming an abstraction over the raw tables. This frees our users from having to make reasoning about data across tables.
2. We design and implement Friction DSL, a domain-specific language used by our user to express an indicator using combinations of aforementioned derived attributes.
3. We provide a dashboard consisting of multiple synced visualizations, including histograms, node-link graphs, and stacked bar charts. The dashboard offers our users an overall context of their classroom in the Overview page, as well as an intuitive way of comparing different indicators of struggle against each other in the Indicators Board page. In all our visualizations, our users can change the views over time, to discover patterns over the entire progression of their course.

The rest of this paper is organized as follows. Section 2 discusses some of the existing work done, for the domain of classroom data visualization, as well as some of the concrete visualization idioms related to our solution. We then introduce our data abstraction in Section 3, followed by task abstraction in Section 4. Section 5 and 6 together present our visualization solution for the data and tasks we described, and details of our implementation. We discuss the results and findings from our solution in Section 7, and potential future works in Section 8. Finally we conclude in Section 9.

2. Related Work

2.1 Computer Science Education

There is a prevalent belief in CS education research that grades in our courses are bimodal, with some population of "strugglers" and others who do well. In past work this has been explained as a "geek gene", causing some students to be predisposed to better outcomes in Computer Science courses [1]. Robins introduced the concept of "learning edge momentum" which posits that the reason

CS1 grades appear the way they do is due to the tight linking of knowledge and how it builds on itself [11]. If a student falls behind the impact will compound itself quickly, which they suggest is unique among fields of study. Patitsas et al find the problem space is more complex [10]. They evaluated many course distributions at UBC CPSC and found that very few are actually bimodal. This indicates that there are not necessarily two separable populations of students but instead a single population that undergoes struggles in diverse ways.

Even if we can not split students into two populations based on outcomes, there is value in understanding how and why students struggle. Various authors have used different features to create models for identifying these students. In this paper we call these models “red flags”. In “Exploring the Value of Different Data Sources for Predicting Student Performance in Multiple CS Courses” the authors use grade information to predict a final course outcome [9]. They find that prerequisite grade or clicker grade strongly predicts final grade. And in “In Situ Identification of Student Self-Regulated Learning Struggles in Programming Assignments” they use measures of stagnation in grade to indicate struggle on an assignment [2]. Furthermore, Estey et al develop a model using changes in programming behaviour to identify students in need of support [4]. They find that they can identify students who require support in the first few weeks of term and target outreach to them. Neural networks have also been explored in finding students in need of assistance. One paper uses student grades and the number of submissions as features in their model [3]. Together these authors have conceptualized some feature (or “red flag”) and demonstrated its relationship to student outcomes. In this paper we contribute a tool that allows quick discovery and validation of features like these in a general-purpose way.

2.2 Classroom dashboard visualizations

Dashboards are a common tool for understanding learner behaviour. For example, Kia and their collaborators created a dashboard for visualising learner attributes in a MOOC class on edX [8]. This dashboard displays attributes such as attendance, gender, and age as bar charts. Ginda and their collaborators also investigate MOOCs, creating conceptual content hierarchies and “learner path” visualisations that show the steps a learner takes through a class [6]. These tools demonstrate the utility of visualisations for educators to understand the experience of their students throughout a course. However, these tools fall short in that they do not allow users to synthesis multiple attributes to model student struggle. This means there is not an explicit way for instructors to deduce common behaviors of struggling students and evaluate them. In this paper, instead of visualising learner attributes directly we visualise learners in terms of cohorts which are defined by models created by the visualisation user.

2.3 Visualization techniques and idioms

In “Quality based guidance for exploratory dimensionality reduction” the authors create a general tool and process for reducing a high dimensionality dataset into a single attribute one, allowing a user to pull out interesting elements for further inspection [5]. A user does this by selecting interesting variables and then inspecting their correlations. The problem of identifying struggling students also maps onto dimensionality reduction. However, in our work we are concerned about the correlation between each potential red flag and the true struggling students. This simplifies the problem considerably because we do not care about correlation between every pair of attributes. We also distinguish our work in that we identify correlations between membership in the set of struggling students instead of between quantitative attributes themselves. In ”The Attribute Explorer: information synthesis via exploration” the authors introduce techniques for synthesizing multiple attributes in the domain of home buying [12]. Their visual technique allows for filters to be brushed onto different attributes to create a synthetic derived attribute of membership in a filtered set. Although we face a similar problem to these authors, their visual technique falls short of our needs as we wanted to allow users to express a more complex set of synthetic attributes outside of standard range based filters.

LineUp introduces a method for visualising multidimensional data in a tabular format. They facilitate the task of ranking based on a user-specified model [7]. LineUp also allows for the comparison of multiple models by displaying them side by side. This idiom is essential to identifying struggling students because users of our tool need to compare their candidate models to decide which is most useful. However, LineUp stops somewhat short of the idiom we need as they do not incorporate temporal data. We need a user to be able to understand how their model for identifying struggling students varies in accuracy and sensitivity over time. LineUp does not treat attributes as time-varying.

We also use the idiom of circle packing where students are represented by smaller circles nested in larger circles representing their groups. Circle packing was previously described in ”Visualization of Large Hierarchical Data by Circle Packing” [13]. The authors of this paper use circle packing to represent tree data where multiple levels of nesting exist. In our use of the idiom, however, we only ever pack exactly one level of nesting. However, we do make use of similar techniques to layout the nested circles within the containing circle.

3. Data Abstraction

The input data abstraction consists of a series of tables representing various aspects of CPSC 310 dataset. We develop a new abstraction over this input, which we then expose to the users of Course Friction Explorer.

3.1 Input table abstraction

The input data is a series of tables with data extracted from tools that the course uses. Here, we describe each table and its source, while additional information about the exact attributes in each table is presented in Appendix A.

3.1.1 Table autotest_results

Throughout the project, students make incremental submissions of their code by committing and pushing to branches on their git repositories. Each team’s repository consists of a single master branch and a number of development branches which are oftentimes per member or per feature. Each project checkpoint has a suite of associated tests that AutoTest runs on each push to these git branches. Students can see the result of the AutoTest run on a specific commit by explicitly requesting AutoBot [1]. AutoTest result requests are rate-limited across branches, for example, one request on any branch per six hours per student.

Autotest results are identified by their feedback_id which is a unique identifier for each result entry. They also have several attributes. They have categorical attributes representing the deliverable the result is for, the branch the result is from, the user who requested the feedback, and the user who committed the code change. They have ordered attributes for the score, the current visible score, time of the request and time feedback was given.

[1] Autobot is an autograding system built at UBC, students request Autobot by commenting on a Github commit, but we run tests regardless of whether they request it.

3.1.2 Table contributions

Piazza is the most active resource where students seek and receive assistance from not only the course staff, but also other fellow CPSC 310 students. Students can create posts that can be either a note or an answer-wanted question, categorizing them using tags. A Piazza contribution includes every action from creating a post, replying to a post, creating a new followup to an existing post, etc, all of which are recorded with timestamps in this table.

Piazza contributions are identified by their cid which is a unique identifier for each contribution entry. They also have several attributes. They have categorical attributes representing whether or not the contribution was made anonymously, the kind of contribution, whether the post was tagged as "project", the user who made the contribution, and the post where the contribution was made. There is also a single ordered attribute, the time at which the contribution was made.

3.1.3 Table queue_visits

Aside from Piazza, TA-held office hours are also one resource students use for issues that benefit from more synchronous, one-to-one interaction. Access to TA assistance in office hours are regulated by Queue@UBC, an online queue service simulating "lining-up" for help. Each student seeking assistance would enqueue and wait for a notification for their turn. A TA can view all the students currently on the queue, and would pick one to "start answering" thereby dequeuing them. Upon addressing the student's question, the TA would "finish answering", recording answer_finish. Note that contrary to a conventional queue, the TA need not follow FIFO order strictly; this is to prioritize help for the students requiring more immediate assistance.

Queue visits are identified by their qid which is a unique identifier for each queue entry. They also have several attributes. These include the user who asked the question, and the TA who answered it. Ordered attributes are the time of enqueue, dequeue, and the time at which the TA started and finished answering the question.

3.1.4 Table users

A deidentified user hash, anon_id, corresponds to each student and TA. Users have a categorical attribute representing whether or not the user is a withdrawn student. They have a single ordered attribute, the time at which their first lab of the term started.

3.2 Revised data abstraction

We want our users to be able to write queries over our data to answer questions about students. We find that the complexity of joining multiple tables and filtering operations is not needed for this purpose. Therefore, we refine the existing abstraction to a single table of "students" which we generate automatically. This table starts as a filtered version of the "users" table to remove course staff and then is populated with data from the other tables.

Students are identified by their anon_id which is a deidentified hash. We generate new synthetic attributes to facilitate queries. These synthetic attributes are generated at runtime using the input dataset, so additional attributes can be added without an additional data processing step.

- num_commits - Number of commits the student has made to the course project.
- num_office_hours - Number of times the student has attended Office Hours with a teaching assistant.
- final_grade - The final project grade of the Does not vary over time.
- minutes_spent_with_ta_office_hours - Number of minutes a student has spent with a teaching assistant in Office Hours
- score_c0 - Student score on c0. The first checkpoint of the project.
- score_c1 - Student score on c1. The second checkpoint of the project.
- score_c2 - Student score on c2. The third checkpoint of the project.
- score_c3 - Student score on c3. The final checkpoint of the project.
- visible_score_c1 - Student score from the student's perspective on c1. Although we grade all commits, students only see their score when they request it.
- visible_score_c2 - Student score from the student's perspective on c2. Although we grade all commits, students only see their score when they request it.
- visible_score_c3 - Student score from the student's perspective on c1. Although we grade all commits, students only see their score when they request it.
- visible_total_delta - The total amount of project score change a student has made that is visible to them. For this metric each checkpoint is considered out of 100 and added together, so it is not bounded above by 100.
- total_delta - The total amount of project score change a student has made. For this metric each checkpoint is considered out of 100 and added together, so it is not bounded above by 100.
- visible_avg_delta - The average amount of visible project score change a student makes in any given commit.
- avg_delta - The average amount of project score change a student makes in any given commit.
- num_piazza_answers - The number of questions a student has answered on Piazza (our course discussion board). This includes both answers and edits made to answers.
- num_piazza_questions - The number of questions a student has asked on Piazza.
- num_piazza_actions - The total number of actions a student has taken on Piazza. This includes questions, answers, followups, and any edits.

4. Task Abstraction

The tool we want to build will help instructors and TAs detect struggling students early on in a course (CPSC 310). To do so, they can use certain indicators correlated with some outcome, for example low final grades, as red flags to identify the students. We assume that our intended audience, CPSC 310 staff, has knowledge in Computer Science. These indicators may be some patterns dependent on the following:

- Office hour visits
- Piazza contributions
- Auto-grading results

If a student happens to fulfill a red flag, then some intervention may be helpful in keeping the student on track. Although these red flags can be any arbitrary condition, we can also do some prior analysis using previous years' data to discover meaningful red flags. To simulate making a prediction, there should also be a way to restrict available data to a certain time-frame. We will also consider calculating some statistics that may describe how well the indicator predicts the outcome.

In order to accomplish this goal we introduce the following high-level tasks

- **T1**: **Explore** student dataset and attributes to discover indicators that identify student struggles.
- **T2**: **Validate** and evaluate indicators based on their sensitivity, accuracy, stability, and speed (at identifying struggling students).
- **T3**: **Compare** multiple different indicators based on the metrics derived from T2.
- **T4**: **Identify** the individual students whom each indicator flags as struggling.

We integrate these tasks into the scenarios presented in 8. Results.

5. Solution

Our solution allows course staff to explore their course dataset and hypothesize indicators of student struggle (**T1**). It also provides intuitive visual means to confirm patterns in student data in order to validate hypothesized indicators as red flags leading to some unfavourable user-specified outcome (**T2**). Moreover, the course staff should be able to perform comparison on multiple different indicators (**T3**), as well as directly identifying individual students flagged by the indicators (**T4**). The instructors and TAs can use established indicators of student friction from previous studies to verify against their own course, but the tool also guides them to explore their dataset and discover more novel indicators.

We propose a solution consisting of four main components split between two separate views, namely, the Overview and the Indicators Board. The Overview gives our user a summary of their student data, thereby allowing exploration (**T1**), while the Indicators Board view lets a user evaluate and compare multiple indicators simultaneously by visually conveying their effectiveness. Across all components, we share a navigation idiom of change over time via a global time slider.

5.1 Overview Histograms

When a visualization user first loads their course dataset, they are first presented with all the usable student attributes exposed for that dataset in the Overview page, as shown in Figure 1. The user can toggle any subset of the attributes to see its distribution across the entire student set. We chose a histogram to convey distribution due to its familiarity and its effectiveness at highlighting outliers. Further we use an idiom of small multiple views for the histograms to increase the information density of the Overview page. We also coordinate the multiple small histograms with shared change over time navigation, enabling attribute exploration across time as well. Initially we set the default start of global time slider to be at the start of interval being explored, but later changed to start from the end date. This is partly due to student attributes being temporal, and many attributes, such as score_c0 and score_c1, only exist after a particular time in the data. This can potentially disorient the user, thus we bring the user to the end of data by default, allowing them to start with a complete dataset. The multiple small histograms together with shared change over time navigation, let our user explore their data across multiple attributes and across time, to pinpoint the set of attributes interesting to them as part of struggle indicators (**T1**).

5.2 Indicators Board: Circular Packing

Once a user selects attributes of interest, they switch to the Indicators Board where they can then create different indicators and evaluate and compare them based on how well they predict the outcome group. The outcome group expresses the unfavourable result that the user wishes to intervene against. The user can configure outcome and create indicators in a side bar by using Friction DSL, the domain-specific language we developed for Course Friction Explorer. Figure 2 shows the elements of the Indicators Board, where the central visualization is a node-link graph. The nodes represent the indicators and outcome, where the outcome node is positioned centrally. Each node is a circular packing of unit marks, representing the individual students identified by the indicator/outcome. Circular packing technique lets us directly map the sizes of indicators and outcome sets to their node sizes. From our user's perspective, node sizes are also intuitive channel for indicator/outcome set size. In addition, every indicator node is linked to the outcome node, with link length encoding the similarity score between the two nodes. In our first iteration, we chose a similarity score based on set union and difference for its simplicity and explainability, but the score did not capture false positives (ie. those who are falsely identified by the indicator). Low false positives is a critical requirement to an effective indicator, and so we switched to using F-Score as a measure of similarity.

$$\text{F-Score} = \frac{TP}{(TP + \frac{1}{2}(FP + FN))} \tag{1}$$

$$TP = \text{True positives}$$
$$FP = \text{False postives}$$
$$FN = \text{False negatives}$$

With F-Score, we can penalize an indicator for falsely predicting strugglers and correspondingly reflect this on its link length. The Indicators Board uses a combination of visualization techniques and idioms to facilitate user's tasks, in particular, **T2** and **T3**. We now discuss each of these.

5.2.1 Attribute Synthesis

Our dataset exposes a myriad of attributes about each student which altogether may have some degree of predictive power over their final outcome. We allow the user to select some of these attributes to generate a new synthetic attribute (ie. an indicator) and then explore how well this new attribute partitions the students. Our synthetic attributes are always binary, either a student is in them or not, and they do not have a magnitude. This feature is critical for supporting **T2**-**T4** because the user must be able to combine multiple attributes derived from **T1** to create meaningful indicators. We developed Friction DSL for this purpose. The choice of DSL over other alternatives like UI-aided filtering students was due to flexibility and generalizability. Our user is free to combine many different attributes in ways supported by Friction DSL, and extending Course Friction Explorer to other attributes and datasets require minimal change in the frontend.

5.2.2 Semantic Zooming

The node-link graph of Indicators Board support the navigation idiom of panning and semantic zooming. The semantic zooming is especially helpful to display only the relevant elements of visualization based on the user's zoom level. At a high level, the user's focus is on viewing multiple indicators at once for the purpose of making comparisons (**T3**). Therefore, the details about each student contained in the indicator circles are unnecessary, thus we filter out the individual student id labels to prevent cluttering. At a lower zoom level, the user instead tries to locate the individual student captured by their indicators (**T4**). The id labels are crucial for such identify task, thus is displayed at low zoom level as can be observed in Figure 3.

5.2.3 Linked Highlighting

We apply another multiple views idiom in our node-link graph: dynamic visual layering. When the user hovers on a particular

Figure 1: The Overview page consisting of side panel, multiple small histograms, and a time slider. The user interacts with the settings in the side panel and time slider to change the histogram views.

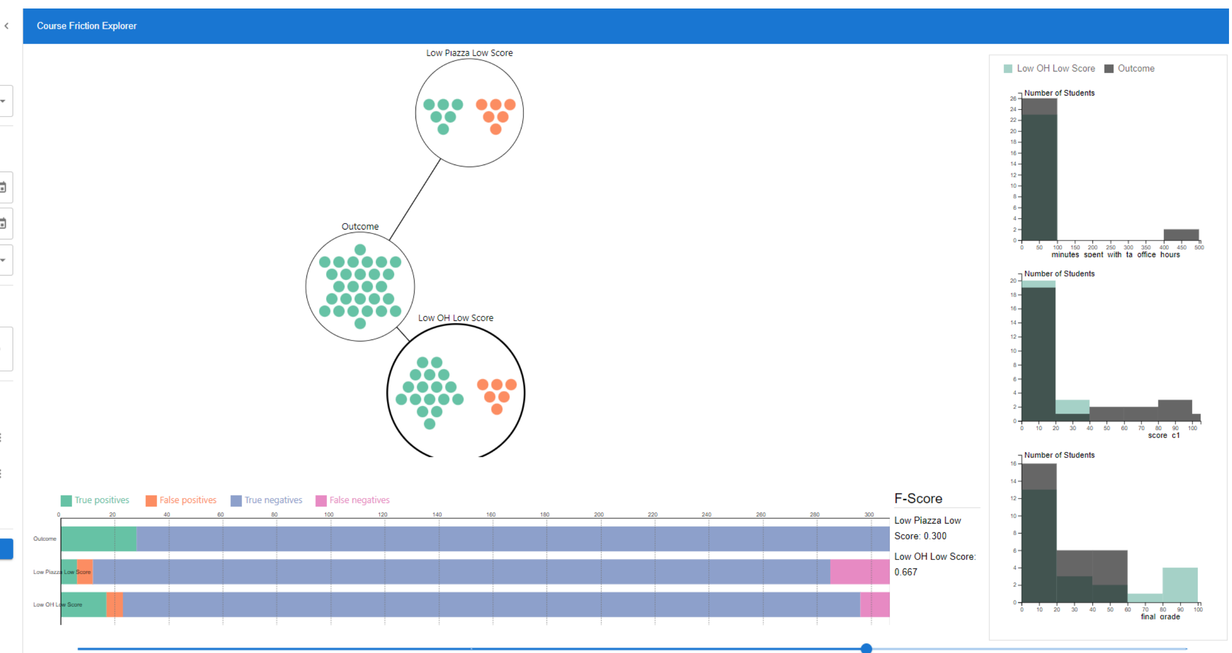


Figure 2: The Indicators Board page also has a side panel for configuring the visualization. The visualization here is linked multiform views, with a combination of node-link graph, stacked bar charts, and histograms.

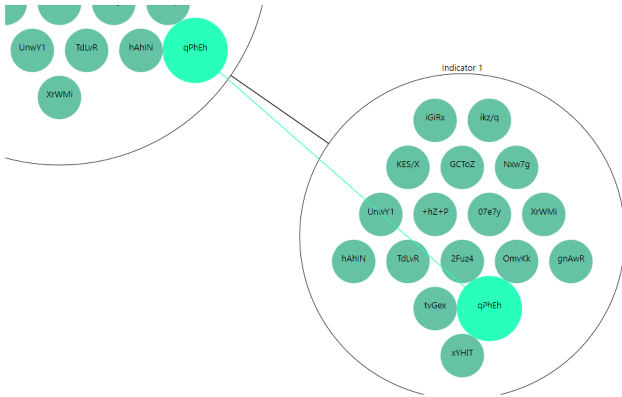


Figure 3: At low zoom level, each student id hashes are displayed for identification purpose. They are hidden once the user zooms out to certain threshold.

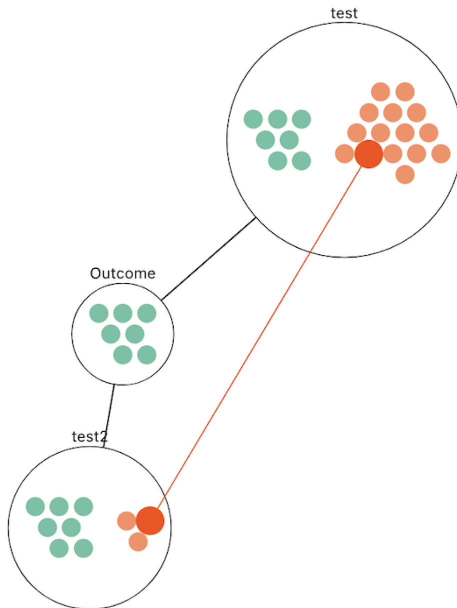


Figure 4: The hovered student is linked and highlighted across all other indicator/outcome circles, forming an overlay graph.

student mark within an indicator/outcome node, we superimpose a new graph linking the selected student to the overlapping student in other nodes (Figure 4). This is especially useful for **T3**, in finding out overlaps between the different indicators. The user can be looking at a set of highly overlapping indicators, thinking that these are distinct. We chose hovering over clicking for selection because it is more lightweight, and we expect the user to perform this selection frequently.

5.3 Indicators Board: Stacked Bar Charts

A stacked bar chart visualization supplements the node-link graph. Each outcome/indicator in the node-link graph has a corresponding bar in the stacked bar chart as can be seen in Figure 5. One pur-

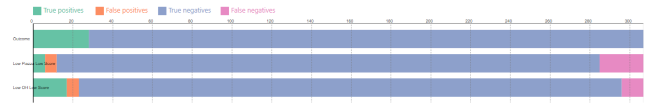


Figure 5: The stacked bar chart represents each indicator and outcome as a bar, where the stacks corresponds to their true/false positives, and true/ false negatives. It shares the same color encoding as the node-link graph.

pose of this chart is to visualize and juxtapose ratios of true/false positives and negatives across all indicators in comparison (**T3**). Another is to represent the group of students not visualized in the node-link graph: the true and false negatives. While the false negatives are indirectly encoded by link-length via F-Score, true negatives are entirely absent from the graph. Including these negatives in the stacked bar charts lets our user judge how representative their outcome and indicators are when looking at the overall. We use the color channel to encode these ratios, and make sure the same color encoding is shared with the node-link graph. We used a 4 class diverging color scale from ColorBrewer[2] to help us perform color mapping. Some alternative design choices we have considered directly embedding the ratio information into the node-link graph, by morphing each node circle into a pie chart showing the ratio. However, we chose the stacked bar charts over this to prevent the color channel from being overloaded in the graph visualization.

5.4 Indicators Board: Histogram Widgets

Click selecting an indicator node in the node-link graph brings up a series of histogram widgets as displayed in Figure 6. These widgets support tasks **T2** by visualizing indicator accuracy in terms of sensitivity. Sensitivity, applied to our visualization domain, reflects how well the indicator identifies true positives: the students belonging to the user's outcome set. Mathematically, it is the ratio of true positives to false positives. We chose to focus on sensitivity as a metric for indicator performance, as the goal of our user is identifying students that fall in outcome set. Other metrics including specificity (ie. how well can the indicator identify true negatives: the students that do not struggle) are therefore less effective in this task. In the widgets, we use the idiom of superimposing two histograms: one from the outcome student set, and the other from the selected indicator. While the layered histograms do overlap, we prevent occlusion by encoding each bar with low opacity. A highly sensitive indicator will have superimposed histograms that are highly overlapping, while a less sensitive indicator will have non-overlapping histograms. How the histogram widgets vary with sensitivity is further illustrated in Figure 7. The user is able to evaluate sensitivity of their indicators as such.

5.5 Navigation: Time slider

We implement a time slider for all views which selects some range of the data prior to the selected point. For example, in the Indicators Board, as the slider point is shifted, marks representing students selected outside of the time range are not displayed. In this way, our visualization user can evaluate the performance of an indicator across time. Some indicators may only become reliable signals close to the end of a course, while others may perform well consistently throughout. This facilitates indicator comparison based on both stability and speed (part of **T2**).

[2] https://colorbrewer2.org/

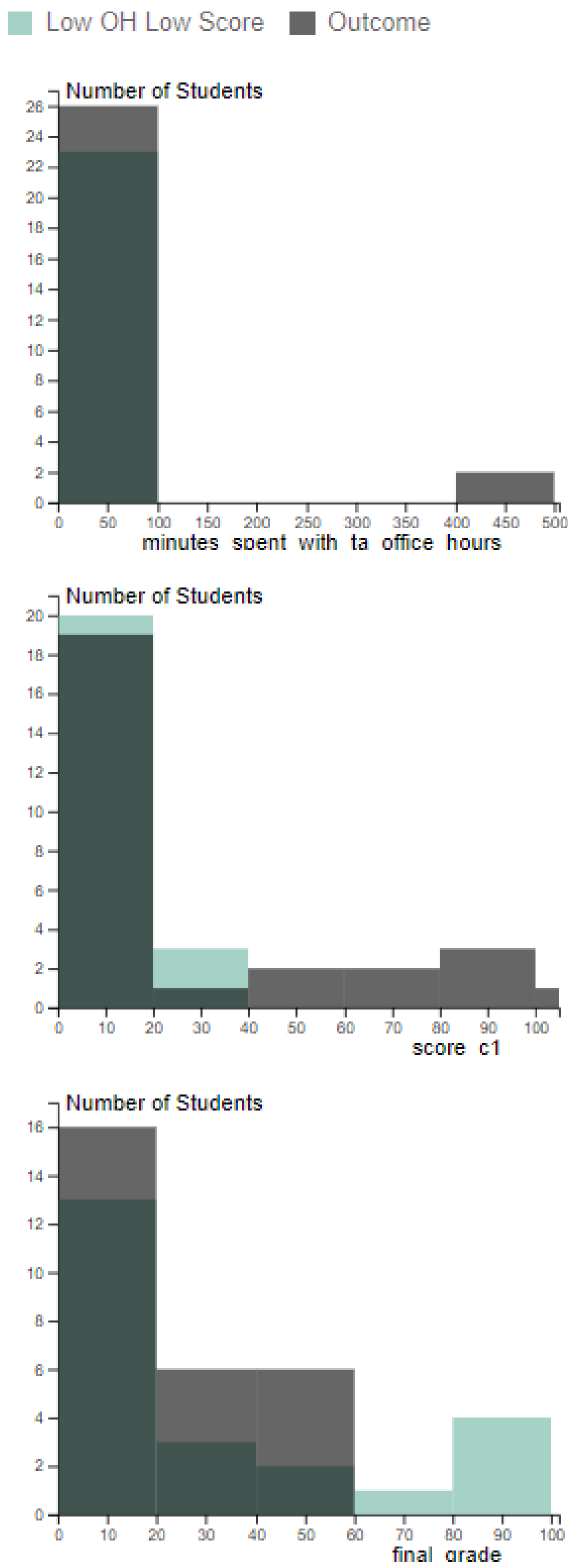


Figure 6: Each histogram widgets show the distribution of the attribute contributing to the selected indicator. The outcome distribution for the same attribute is superimposed for direct juxtaposition.

6. Implementation

We implement Course Friction Explorer as a web application, with a backend and frontend.

6.1 Backend

We use a backend written in Python [3], mainly because of its data processing capabilities. The backend assumes responsibility for loading and processing the input dataset. The dataset is then made available for querying using a DSL we designed over a REST endpoint. For serving the API, we use Uvicorn and FastAPI [4].

6.1.1 Loading dataset for querying

We load the dataset, stored in a SQLite database, into a memory representation which exposes a time varying interface. This means that the frontend can provide arbitrary timestamps and the backend can sample attributes at that time. With this, we can define our own derived attributes that the viz user can interact with through our frontend interface and DSL. As described in an earlier section, we want to abstract away complicated joins and filters that are required to extract useful information from the raw SQLite database. Combining these derived attributes are meant to be more accessible with the use of our DSL.

6.1.2 DSL: Writing queries

As part of our backend, we have designed a domain-specific language (DSL) (see Appendix B) to help process different queries in our vis tool. To create an indicator, the user would construct a query in our DSL, which gets sent to our backend for parsing and evaluation. The lexing and parsing are done with Antlr[5]. Once Antlr produces abstract syntax tree, we use a visitor to traverse the tree nodes and evaluate the query. We designed our DSL to support all the operators helpful to creating an indicator, namely logic, comparison, arithmetic, aggregation, and granularity operators. See our DSL cheatsheet[6] for more details on the DSL usage. We designed the DSL to be simple to parse and not too tightly coupled to the dataset. So, in the current state of the DSL, we use a generous number of parentheses to avoid ambiguities with order of operations, and the recurring ”student.” prefix accesses the CPSC 310 students dataset.

6.2 Frontend

Our React frontend makes use of a number of libraries to help expedite development. In particular, we use React components from Materials UI[7], applying customized CSS via styled components[8]. Redux [9] is the backbone for our frontend state management, giving all our individual components access to a consistent global state to which they can update their views on. The overall structure of the frontend is in Figure 8.

The service layer is responsible for communicating with our backend using REST API queries. For example, the `QueryService` and `StudentService` makes request to `POST /query` and `GET /students/id` endpoints surfaced by the backend server.

The state management layer lies in between our service layer and the frontend React components. We chose to offload state retrieval and update logic from the individual components into this

[3] https://www.python.org/

[4] https://fastapi.tiangolo.com/

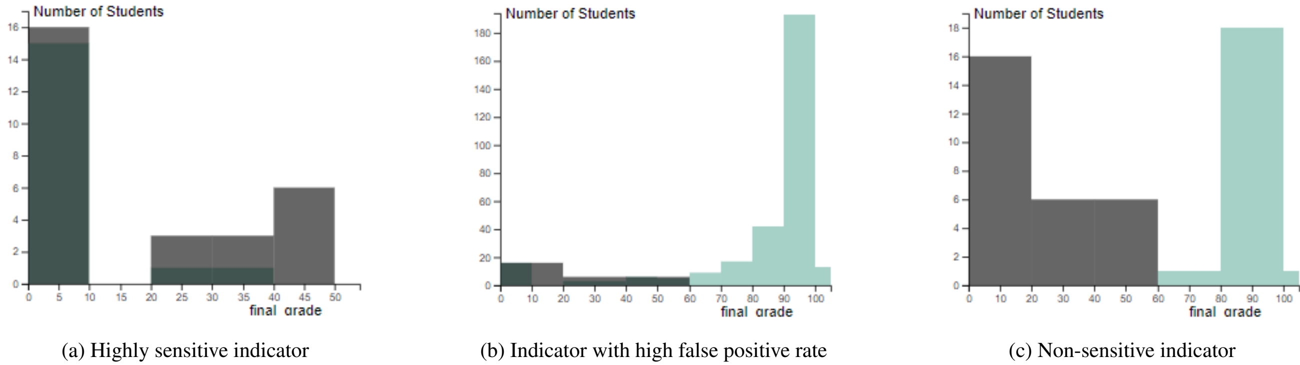
[5] https://www.antlr.org/

[6] https://gist.github.com/falkirks/74be2706cd63fc20ca1beee3e918a1ea

[7] https://mui.com/

[8] https://styled-components.com/

[9] https://redux.js.org/



(a) Highly sensitive indicator

(b) Indicator with high false positive rate

(c) Non-sensitive indicator

Figure 7: Histogram widget examples from three different indicators. The black bars corresponds to the outcome student group, and the teal bars are for the selected indicator student group. The indicator in (a) exhibits high sensitivity, with high true positive and low false positive rates. This can be seen from the almost overlapping bars of the two groups. (b) is an example of an indicator with high true positive rate but also high false positive rate. In such case, the indicator distribution extends past that of the outcome. (c) is an indicator with low true and false positive rates. The histogram visualizes this as non-overlapping bars.

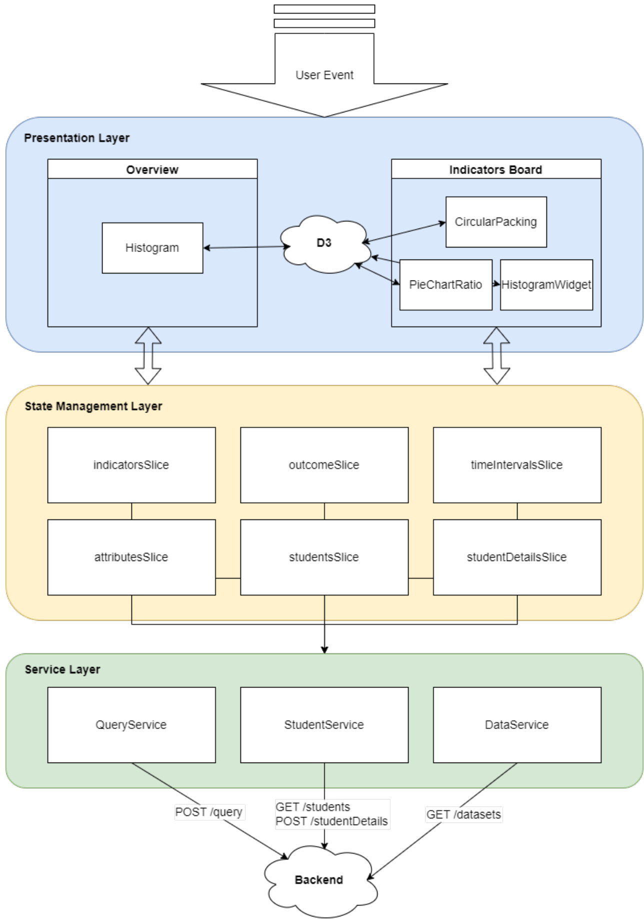


Figure 8: High level structure of Course Friction Explorer frontend. Each layer communicates with the layer below, retrieving necessary data up the layers, and propagating event updates downwards.

state layer so to minimize code duplication, and to ensure states are accessed and modified in consist manner by all components. Redux is used heavily for this purpose. We have defined a global redux store, consisting of multiple slices (subsets of the global state accessible as a single unit from frontend components). For example, the indicatorsSlice maintains the current working set of indicators configured by the user, and surfaces the addIndicator, editIndicator, removeIndicator, and queryAllIndicators as common access methods invoked from all frontend components as needed.

Finally, the React components serve as our presentation layer. Our visualization dashboard is broken down into two major page components: IndicatorsBoard and Overview pages. Each composes smaller sub-components including the Sidebar, TimeSlider, IndicatorEditorDialog. Despite the many components accessing and modifying shared state, our components remain moderately decoupled from each other, thanks to the state management layer moderating state accesses.

Integrating React with d3 came with some complexity, as both attempts to assume full control of the DOMs in view. In order to ensure they work in combination, we needed to define a clear separation in ways they can perform DOM manipulation. We wish to utilize d3's strong set of data manipulation and visualization methods, including animation and transitions. On the other hand, React's ease in receiving user input events (eg. zoom, pan, click) and compatibility with our state layer is crucial. Therefore, our approach involves using React component as a wrapper around each chart DOM manipulated entirely by d3. For example, our node-link circular packing chart in the Indicators Board has a wrapper CircularPacking component which renders a single DOM node <vis-circular-packing>. The component feeds all the data needed by d3 to manipulate <vis-circular-packing> to properly compute and visualize the chart. Upon updates from the state layer and input events from user, it is CircularPacking component's responsibility to trigger appropriate d3 methods to update the chart with new data. We have found this architecture gives us the benefits from both libraries with minimal conflicts.

7. Milestones

Table 9 displays the proposed development schedule of the Course Friction Explorer from the initial planning to writing and submitting the final paper. We estimated a total of 81 person-hours on task, with the main hours spent at implementing the Course Fric-

tion Explorer followed by writing the paper. However, estimating and planning person hours on task tends to be very difficult and error-prone. Therefore, we will be iteratively revising the allocation of hours, to confirm their accuracy or adapt where needed. After revising Table 9, the deadline for the backend of the course friction explorer got pushed back due to the fact that the development of the backend and frontend started simultaneously. Table 10 provides a detailed breakdown of the tasks including the responsible team members for each task, estimated workload and actual hours spent on each task. At the bottom of each table is the total amount of work summarized. We made the mistake of underestimating the need for additional visualizations and therefore spend 144 hours instead of the calculated 120 hours implementing the backend and frontend of the tool. The estimated workload is equally divided among the four team members.

8. Results

We contribute a visualization that allows the staff of CPSC 310 to create, evaluate, and compare indicators of student struggle (**T1-T4**). Our tool supplements infrequent course analysis tasks and allows instructors to more effectively target outreach to students. To attempt to demonstrate the efficacy of our tool, we attempt to construct and reproduce previously established metrics of friction. To evaluate the responsiveness of our tool, we perform a limited evaluation on attribute generation and indicator evaluation performance.

8.1 Assignment trajectory

In ”Automatically Classifying Students in Need of Support by Detecting Changes in Programming Behaviour”, Estey and their collaborators introduce a trajectory metric for evaluating student friction [4]. Trajectory is determined by the number of compiles required to achieve a correct solution. In our CPSC 310, we do not have instrumentation of student compilation, but we do have data for every test run. Therefore, we can define a new measure of trajectory, students who have below a specific threshold of grade change in an average week. We can express this in Friction DSL

```
(avg(weekly(student.avg_delta)) <= 1)
```

We selected 1 by using the Overview histogram for the avg_delta to determine a reasonable threshold value. Then we can enter this indicator into Course Explorer. As our outcome, we select students with a final grade less than 50%. In 9a, We see that at near the beginning of term the indicator does identify some of the outcome set, but it also generates a number of false positives. As the term goes on (shown in 9b), the indicator becomes more specific and eventually has no false positives. However, the indicator still lacks somewhat in sensitivity, as it fails to identify some of the outcome set.

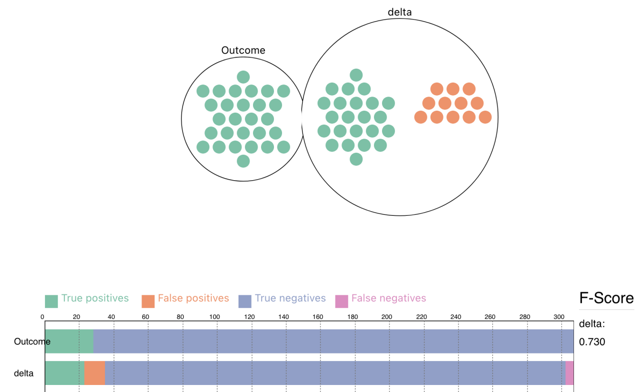
8.2 Component grades

In “Exploring the Value of Different Data Sources for Predicting Student Performance in Multiple CS Courses” the authors use grade information to predict a final course outcome [9]. We can create a similar evaluation for our dataset using Course Explorer. We start by establishing several indicators that represent failing grades on each component of the project

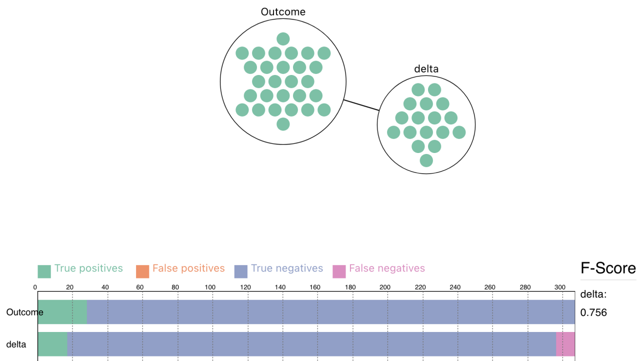
```
(student.score_c0 < 50)
(student.score_c1 < 50)
(student.score_c2 < 50)
(student.score_c3 < 50)
```

We then set our outcome set of interest to again be the students who failed the overall project. Grades are only defined after an assignment is completed so for these indicators it makes most sense to evaluate at the end of term, once all component grades are completed. In **??**, we find all of these indicators perform similarly. They are quite sensitive, very few students that fail the project are not captured by one of these indicators. However, they are somewhat inaccurate, each includes some false positives, with $c2$ having the least. $c2$ is a refactoring assignment which links heavily to $c1$ and is then needed for $c3$, so we could conjecture that a $c2$ failure represents a more major issue with the course project.

In order to more faithfully reproduce the previous work, we could integrate other sources of grade data. This would include exam data and prerequisite grade information. Our visualization system is built in such a way that these attributes could easily be added to the dataset without making any modifications to the actual visualization code.



(a) ”Assignment Trajectory” indicator shown near the beginning of term.



(b) ”Assignment Trajectory” indicator shown near the end of term.

Figure 9: Circular Packing and Stacked Bar chart views for ”Assignment Trajectory” indicator evaluation. Where outcome is the set of students with a final grade below 50%.

8.3 Attribute/indicator computation responsiveness

Course Explorer shifts complex computation from the client onto the server application. This means that vis performance is somewhat client agnostic. Once attributes are computed for the requested times, the frontend can support navigation with negligible latency.

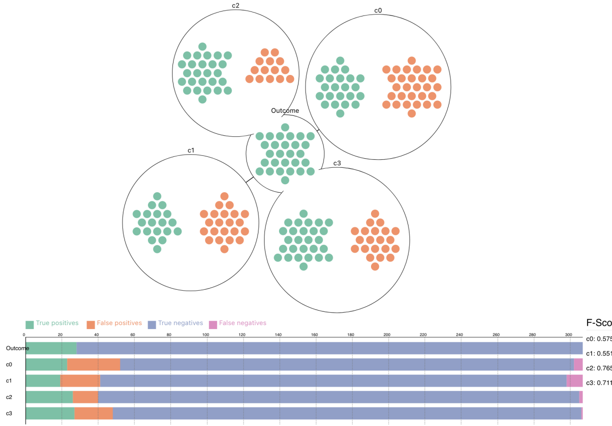


Figure 10: Course Explorer interface with four indicators created, one representing a failing grade for each checkpoint. The outcome set is defined to be an overall failing grade.

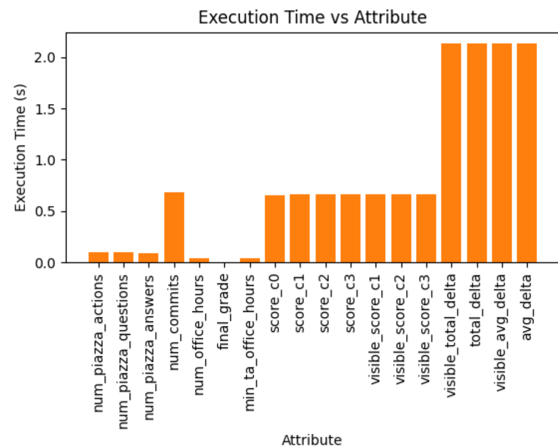


Figure 11: Execution time in seconds plotted in a bar chart again attribute type. Based on an average of 25 runs on the entire dataset.

However, computing the attributes and indicators in their current form are resource intensive operations. [10]

To evaluate our attributes we generated each synthetic attribute 25 times for every student and took the mean time to complete every student. We find that attribute complexity varies quite substantially. The simplest attribute $final_grade$ takes only 0.00002 seconds to complete on the dataset. All score related ones take under 0.7 seconds. However, computing score change consumes on average 2.14 seconds. This is due to the comparatively larger set of $autotest_results$ and the computation required to sum or average the grade change. We find that these timings are acceptable for most attributes, with further work required to bring down change related attributes. Additionally, our server currently does not cache results as we imagine our users would interact infrequently and run unrelated queries. While there is some benefit in directly caching attributes as they are queried, we think we would only see major benefit to responsiveness if we warm the cache throughout the life of the server. We could warm the cache using statically anticipated queries or dynamically based on previous queries.

We then assessed the responsiveness of our indicator evaluation system, which adds a DSL on top of the synthetic attributes. We evaluated our DSL by constructing queries of increasing complexity and plotting evaluation time as depth increases. To setup this experiment we start with a seed indicator

```
(((student.score_c0 + 10) - 10) < 70)
```

then at every iteration we nest this query in a logical AND. At the first iteration we would have

```
((((student.score_c0 + 10) - 10) < 70)
AND (((student.score_c0 + 10) - 10) < 70))
```

We choose this indicator as it uses a non-trivial attribute, and employs a comparison and two arithmetic steps. We also patched our code to disable short circuiting of AND statements, so the amount of true positives would not affect how much of the indicator was executed. We used a mean of 10 executions for this evaluation. We saw in 12a a linear trend as depth increases. Where an indicator of depth 15 takes around 10 seconds to evaluate on all students. However, drawing back on our evaluation of attribute run times we presumed this linear relationship was heavily bound to the attribute running time. To investigate we the ran experiment again, but excluded the execution time of the attribute generation code path. In 12b we find that the execution time of the DSL itself does not have a linear relationship with indicator depth once attribute generation is excluded. We infer that the improvements we suggested to caching attributes would also greatly benefit indicator execution since indicator execution is dominated by attribute generation.

[10] We performed all our responsiveness tests on a "MacBook Pro (14-inch, 2021)" using an "Apple M1 Pro" processor and with 32GB of total RAM.

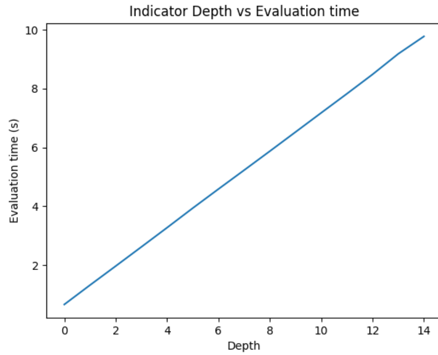
9. Discussions and Future Work

For future work, we consider implementing a more intuitive visual encoding for viewing repeated students within many indicators. Currently, to check whether a student appears in many indicators, a user must hover over individual students and check each spawned network selection. A more appropriate solution could be allowing a selection of the entire indicator, which could spawn some visuals that describe the redundancy of other indicators relative to the one selected. This could, for example, be achieved with some shading of the circle. A fully shaded indicator could represent a full overlap, and no shading means completely disjoint groups.

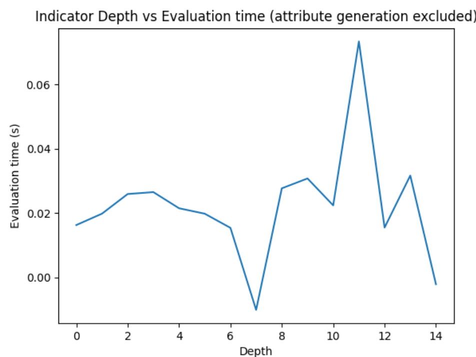
Another limitation is the information sparsity of our current visualization; indicator circles are rendered in a force-directed graph, in which their distance to the outcome circle encodes similarity. Initially, this design choice was made so our visualization would utilize position as a channel, given its effectiveness over other channels like color. However, testing showed that this resulted in indicator circles often being far apart, sometimes leaving the screen space entirely. The visualization also becomes sparse, requiring the user to pan and zoom too frequently to make comparisons or see details. With more time, we intend to remove this distance encoding, and switch to using line-width to encode similarity.

Another related issue is scalability. Once an indicator set becomes large, the visualization could contain clusters of many small student marks that each do not add much value, other than for individual identification. So instead of representing each student as its own mark, we can aggregate them in some way. One approach is to introduce additional hierarchies to the circular packing, by aggregating multiple students into a single mark. Another approach is to turn an indicator into a pie-chart like structure, representing true positives or false positives as shaded percentages.

Currently, the time slider adjusts a global time range for all indicators and attributes, so there is no option to time restrict indi-



(a) Total execution time in seconds plotted against query indicator "depth" (number of clauses nested).



(b) Total execution time with attribute generation excluded plotted against indicator "depth" (number of clauses nested).

Figure 12: Evaluation of indicator depth and its influence on execution time.

vidual indicators. This was intended to simulate being at a single time point within a term, so naturally the user would have all their attribute data up to a certain time. However, as we tested our tool, we realize that local time-ranges may be useful. For instance, the number of commits a student makes in a certain time frame, or at certain times in a week, may be an interesting indicator. Currently, the user would not be able to express this indicator without also restricting the time range of all other indicators, essentially making comparison impossible. Having a way to express more fine-grained time ranges may help address this issue.

On the Overviews page, we display histograms of individual attributes, without any option to combine them. To help users get a better overview before making an indicator, we could allow histograms that show more expressive combinations of attributes.

Lastly, we rely solely on the DSL to define our indicators. This assumes that our target users are very programming-literate instructors and staff of a course. Still, it may be unintuitive to construct an indicator without any visual cues. An interactive GUI for constructing an indicator may be helpful for our target users and could make the tool more accessible to layman users.

10. Conclusion

We introduce an interactive visualization tool for CPSC 310 staff with a number of different visualizations. This tool is intended to aid CPSC 310 staff in understanding student struggles by creating, evaluating and comparing indicators as well as developing possible interventions based on the given data. The overview allows the staff to explore changes of specific attributes over time, such as the change in office hour visits as the term evolves. The staff can decide which attributes to display and how to set the time intervals. Comparing attributes by creating indicators and predicting the outcome groups of students is also supported by our indicator board. The indicator board consisting of Stacked Bar Charts, Histogram Widgets, Time slider and Circular Packing allows CPSC staff to select specific students, identify patterns and compare different indicators by using different queries. We believe that the Course Friction Explorer is a suitable tool for the CPSC 310 staff for identifying and understanding the struggles of students, helping them better guide students through the course.

References

[1] Alireza Ahadi and Raymond Lister. 2013. Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant?. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 123–128. https://doi.org/10.1145/2493394.2493416

[2] Kai Arakawa, Qiang Hao, Tyler Greer, Lu Ding, Christopher D. Hundhausen, and Abigayle Peterson. 2021. In Situ Identification of Student Self-Regulated Learning Struggles in Programming Assignments. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 467–473. https://doi.org/10.1145/3408877.3432357

[3] Karo Castro-Wunsch, Alireza Ahadi, and Andrew Petersen. 2017. Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 111–116. https://doi.org/10.1145/3017680.3017792

[4] Anthony Estey, Hieke Keuning, and Yvonne Coady. 2017. Automatically Classifying Students in Need of Support by Detecting Changes in Programming Behaviour. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 189–194. https://doi.org/10.1145/3017680.3017790

[5] Sara Johansson Fernstad, Jane Shaw, and Jimmy Johansson. 2013. Quality-based guidance for exploratory dimensionality reduction. *Information Visualization* 12, 1 (2013), 44–64. https://doi.org/10.1177/1473871612460526 arXiv:https://doi.org/10.1177/1473871612460526

[6] Michael Ginda, Michael C. Richey, Mark Cousino, and Katy Börner. 2019. Visualizing learner engagement, performance, and trajectories to evaluate and optimize online course design. *PLOS ONE* 14, 5 (05 2019), 1–19. https://doi.org/10.1371/journal.pone.0215964

[7] Samuel Gratzl, Alexander Lex, Nils Gehlenborg, Hanspeter Pfister, and Marc Streit. 2013. LineUp: Visual Analysis of Multi-Attribute Rankings. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2277–2286. https://doi.org/10.1109/TVCG.2013.173

[8] Fatemeh Salehian Kia and Simon Fraser. 2016. Learning Dashboard: Bringing Student Background and Performance Online.

[9] Soohyun Nam Liao, Daniel Zingaro, Christine Alvarado, William G. Griswold, and Leo Porter. 2019. Exploring the Value of Different Data Sources for Predicting Student Performance in Multiple CS Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 112–118. https://doi.org/10.1145/3287324.3287407

[10] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. 2016. Evidence That Computer Science Grades Are Not Bimodal. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 113–121. https://doi.org/10.1145/2960310.2960312

[11] Anthony Robins. 2010. Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education* 20, 1 (2010), 37–71. https://doi.org/10.1080/08993401003612167 arXiv:https://doi.org/10.1080/08993401003612167

[12] Robert Spence and Lisa Tweedie. 1998. The Attribute Explorer: information synthesis via exploration. *Interacting with Computers* 11, 2 (1998), 137–146. https://doi.org/10.1016/S0953-5438(98)00022-8

[13] Weixin Wang, Hui Wang, Guozhong Dai, and Hongan Wang. 2006. Visualization of Large Hierarchical Data by Circle Packing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*. Association for Computing Machinery, New York, NY, USA, 517–520. https://doi.org/10.1145/1124772.1124851

A. Input dataset attributes

A.1 Table autotest_results

Table 1: Categorical attributes

Name	Description	# of categories	Example
deliv	Checkpoint number	4	c0, c1
ref	Git branch AutoTest was called on	1665	refs/tags/c2-rc5
is_master	True if ref is the master branch	2	1, 0
feedback_requester	Deidentified user hash of the requester	409	j+TyZUe34/c1mZOH9tppky9CF/UubYnhJmIGIdFECDY=
committer	Deidentified user hash of the committer	409	F+CFt8v9oVAaHZppCNKYTSN+KCszD8C8vHdPeIh8NiY=

Table 2: Ordered attributes

Name	Description	Min	Max	Example
score	Test score of this AutoTest run	0.00	100.00	100.00
visible_score	Test score reported back	0.00	100.00	90.00
request_time	Timestamp of when a student requests the result	1610400675825 (Mon Jan 11 2021 13:31:15)	1622674908773 (Wed Jun 02 2021 16:01:48)	
feedback_time	Timestamp of when result is returned	1610400675825	1619628202579	1610521081669

A.2 Table contributions

Table 3: Categorical attributes

Name	Description	# of categories	Example
is_anonymous	True if post is created by anonymous contributor	2	0
kind	Contribution kind	12	followup
is_project	True if contribution is tagged as project	2	1
anon_id	Deidentified id of the contributor	409	07e7yyUGH4zoF+i5UF3PH9dmjTwCIMizU+GVt2TmNnM=
post_id	Post ID	1436	Ks43Y68znhtzXws8zNnDGidHOuGi2ApOKgsHqyIOc/k=

Table 4: Ordered attributes

Name	Description	Min	Max	Example
created_at	Timestamp of contribution	1610149993000 (Fri Jan 08 2021 15:53:13)	1620093785000 (Mon May 03 2021 19:03:05)	

A.3 Table queue_visits

Table 5: Categorical attributes

Name	Description	# of categories	Example
anon_id	Deidentified user ID	409	DjN2/LSrZHkxfxAk/ka8gIigB6vY11Usc2AUFScIc3o=
answerer_id	Deidentified answerer ID	27	Nxw7gaFw+d2v0moktJ1dGKzd4Ix2fgZTgjLG0P7OFO8=

Table 6: Ordered attributes

Name	Description	Min	Max	Example
enqueue	Timestamp of enqueue	1600386235000 (Thu Sep 17 2020 16:43:55)	1618015298000 (Fri Apr 09 2021 17:41:38)	
dequeue	Timestamp of dequeue	1600386253000 (Thu Sep 17 2020 16:44:13)	1619307862000 (Sat Apr 24 2021 16:44:22)	
answer_start	Timestamp	1600386239000 (Thu Sep 17 2020 16:43:59)	1618013723000 (Fri Apr 09 2021 17:15:23)	
answer_finish	Timestamp	1600386253000 (Thu Sep 17 2020 16:44:13)	1618016558000 (Fri Apr 09 2021 18:02:38)	

A.4 Table users

Table 7: Categorical attributes

Name	Description	# of categories	Example
withdrawn	True if the user withdrew from the course	2	1

Table 8: Ordered attributes

Name	Description	Min	Max	Example
first_lab_time	Timestamp of the user's first lab, Null if user is a TA.	1610384400000 (Mon Jan 11 2021 09:00:00)	1610751600000 (Fri Jan 15 2021 15:00:00)	

B.
DSL Grammar

```
query : some_filter;
some_filter : logic | binary;
logic : '('some_filter ('AND'|'OR') some_filter')'
      | '(''NOT' some_filter')';
binary : '('comparable ('>'|'<'|'<='|'>='|'!='|'==') comparable')';
comparable : number | time | string | student_attribute;
number : Number | arithmetic | modified_attributes;
modified_attributes : student_attribute
        | granularity_operator'('modified_attributes')'
        | aggr_op'('modified_attributes')' ;
granularity_operator : 'daily'|'weekly'|'monthly'|'final'|'sofar';
aggr_op : 'avg' | 'count' | 'max' | 'min' | 'sum' | 'val' ;
arithmetic : '('number ('+'|'-'|'*'|'/') number')';
string : student_attribute | String;
time : time_lit ;
student_attribute : 'student.'attribute;
attribute : String;
time_lit : 'time' '(' String ')';

Number : DIGIT+ '.' DIGIT*
       | '.' DIGIT+
       | DIGIT+
       ;
String : [a-zA-Z0-9_\-]+;
```

C.
Milestones

	Task	Due date	(Total/Per person)	Actual hours spent	Description	Status
1	Pitch	Sep. 29	8/2	8	Create content and rehearse pitch	Complete
2	Proposal	Oct. 21	28/9	28	Discuss the project, create illustrations and write the project proposal	Complete
3	Learning and understanding the tools	Oct. 28	40/10	36	Read the documentations and examples. Learn how to use the tools and how they can interact.	Complete
	- d3.js	Oct. 24	16/4	16		Complete
	- Fast API	Oct. 26	12/3	8		Complete
	- Python	Oct. 28	12/3	12		Complete
4	Project Update I	Nov. 16	12/3	16	Prepare and provide updated paper for Peer Reviews	Complete
5	Project Update II	Nov. 24	16/4	12	Prepare for the Post-Update Meeting and demonstrate the prototype	Complete
6	Implementation	Dec. 6	160/40	180	Implement and complete the Course Friction Explorer	Complete
	- Backend: Setup, Data, Configs, Querying, DSL	Nov. 26	60/15	65	Setup the environment, clean and work with the data in the backend, do the configs and create queries.	Complete
	- Frontend: Circular packing, Table	Nov. 26	60/15	80	Implement the frontend by i.a. creating circular packing and table models.	Complete
	- Analysis	Dec, 6	40/10	35	Generating additional properties people might use	Complete
7	Draft of the final paper	Dec. 8	20/5	15	Write draft of the final paper	Complete
8	Presentation	Dec. 15	16/4	21	Prepare slides for the final presentation and talking points	Complete
9	Final paper	Dec. 17	24/6	30	Finish the paper and include final changes and conclusion	Complete
	Total hours spend		**324/81**	**346**		

Table 9: Overview of project milestones and person hours allocated to each

Task	Assignees	(Total Hr. / Per person)	Actual hours spent	Status
Backend - Environment setup	Marie, ToTo	6/3	5	Complete
Backend - Clean and load the data	Noa	6	7	Complete
Backend - Configs	Shizuko	3	3	Complete
Backend - DSL Features	Noa, ToTo	12/6	12	Complete
Backend - EBNF and parsing	ToTo	18	21	Complete
Backend - Queries	Noa	15	17	Complete
Frontend - Environment setup	Shizuko, Marie	4/2	4	Complete
Frontend - Redux integration	Shizuko	3	3	Complete
Frontend - Circular Packing	Marie	15	16	Complete
Frontend - Time slider	Shizuko	12	10	Complete
Frontend - Histograms	Shizuko, Marie	20/10	21	Complete
Frontend - Additional visualizations	All	6/3	25	Complete
Total hours spend	**All**	**120/30**	**144**	

Table 10: Breakdown of project tasks for the backend and frontend of our project