Week 7 Update

**Revised proposal motivation**

Currently, we have some understanding about how students progress through different courses, as articulated by Jessica’s paper on progress networks. This progress however, is mostly measured in terms of participation in a particular problem, and likelihood to stay on/pass/continue. We do not yet have a comprehensive and detailed understanding or coherent narrative that underpins how students’ progress throughout a course, the underlying difficulties that they may face, and the effectiveness of interventions such as engaging with instructors and certain coding behaviours such as regular submission, writing more compact code and simple code.

**Enrolment data**

There are several features that can be extracted from enrolment data. These attributes can be used to build individual learner profiles. For individual students, we can find:

1. **The courses they have been assigned to and when they were assigned:** We can link this data with problems that individual students complete.
2. **Gender:** M/F
3. **Grade level:** 1 to 13

**Analysis:** How does ability vary by age? Do older students tend to perform better, potentially indicating that they have had more opportunities to learn problem solving skills?

**Question to Irena and Bryn:** Has this analysis been done before? I understand that a gender-based analysis has been done previously by Jessica.

**Comment:** This might have been done briefly to some extent. The paper that Jessica presented a few years ago had a bit of analysis based on grade and gender. One of the challenges is that there are so many other factors that are not well understood but are more dominant. There are lot of things that we don’t capture information on. We don’t have all the facts.

Causal analysis question: We find that a lot of students have a core way of attempting problems. This is because teachers have taught them in a specific way and set external goals. So students have this external goal and influence that can affect their performance. This may make students engage in surface level rather than deep level learning.

The problem with a causal model is that we may not have access to external factors/influences. This is a thing that you find with interesting methods. Often the dataset limits us. However, being able to characterise certain skills as deep and surface level approaches is interesting. If we can look at a way at activities and then look at how we might be able to differentiate between surface and deep level learning approaches. We could look at do people start off deep and then progress onto surface.

Deep vs surface learning is a growing research problem. There is a theory behind this that we can link our research to. This is unrelated to causal analysis. We can classify whether activities are surface or deep learning. Surface and deep learning can be ascertained through chat history (the way students try to get help from tutors) as well as activity data (how students approach the problem). Surface approaches might be characterised by quick saves and quickly submitting with changing a few lines.

Some observations: we do keep a record of when people look at slides in a particular problem. A module is designed such that students go through slides and attempt to problem. But a lot of students jump to the problem without having gone through the slide. The course was designed in a certain way for students to engage in the material. A common tactic is that you go to the end, try to do a problem and then copy and paste. Those kinds of behaviours probably can be captured.

We can guide our choice of features with some knowledge of what we think students would do. We can also do unsupervised learning to reveal if there are certain behaviours. However, whenever we do clustering, often we just get a spread of data rather than distinct clusters.

This would be an interesting research point. PASTA paper – we looked at improvement over several submissions. We can measure improvement in students by looking at the number of test cases passed for each module. Irena can send this paper over.

Sophia explored this to some extent – students looking through the slides and then doing some coding. Some features that Sophia extracted would be useful.

So let’s start off with some hypotheses that we expect to see:

1. We think that even though students are supposed to look at slides, we understand that some students might go straight to the problem. Let’s evaluate if that’s true and if there’s a clear distribution. This is analytical hypothesis testing where we generate the hypothesis. Then we go beyond that to see if there are other behaviours that can be tested for. The nice thing about this approach is that educators will like that because you’re trying to draw conclusions before diving straight into the data.

Next steps:

1. Starting off with Hypothesis 1 (students don’t look at slides): identify what attributes we need to get, and then show the distribution of people who do and don’t show that behaviour. Then we expand that out. This is a clear question and allows us to do a complete analysis.
2. We can do this as a Jupyter notebook. Start off as a mini report: state research question, do analysis to do the data, and then write a short report of the journey I took to get the data. Then bring it back to see if we can address the research question with an unambiguous answer.
3. Need to start working with data ASAP.

Have everything on Google Docs/Github – so we know what we discussed each week. Share the doc with Bryn and Irena. Alternatively, can use word online. Share the working doc with Bryn and Irena.

**Chat history data**

Chat history data between instructors and students can reveal several features about students and their learner profiles. Typically, students start engaging instructors by making a request for help. The instructor then provides a series of hints for the students. This interaction between instructors and students will typically feature words from a set vocabulary. Commonly appearing words have been identified and grouped into certain actions that correspond with different stages of a student’s interaction with an instructor.

**Hypothesis:** There is a vocabulary of words that can be associated with how students struggle with problems, interact with and learn from instructors, and implement feedback. There are likely associations and relationships between words e.g. ‘error’ and ‘problem’ frequently together, ‘problem’ might occur before ‘try’.

**Student raises a problem or issue with their code:**

* **Problem:** indicates that the student is stuck on a particular problem.
* **Error:** indicates that the instructor has identified the source of error.
* **Stuck:** indicates that the student is stuck on a particular problem.
* **Unsure:** indicates that the student is confused about a particular problem or aspect of the problem.

**Instructor suggests hints and improvements and provides feedback:**

* **Try:** used by the instructor to give a recommendation or suggestion to improve the student’s code.
* **Missing:** used by instructors to indicate the omission of certain elements in the student’s code.
* **Sense:** used by the instructor to check if the tips or problem are understood by the student or used by the student to signal understanding.

**Student takes (and potentially implements) the feedback:**

* **Thank you:** indicates that a problem has been resolved or advice has been given from the instructor to the student.
* **Sense:** used by the instructor to check if the tips or problem are understood by the student or used by the student to signal understanding.
* **Working:** used by the student to indicate that their solution is now working.

**Question to Irena and Bryn:** Is this a worthwhile approach? How might I be able to test this hypothesis out initially first and work out how feasible this approach is?

In addition, we can also extract the following features about individual students:

1. How frequently they interact with instructors.
2. The time between messages (potentially a proxy to indicate how they progress on a particular problem).
3. The dates and times of their interactions.

The benefit of chat history data is that it is specific to **individual problems**. For students who work on multiple problems and use the chat function for different, potentially successive problems, we may be able to extract enough features from their interactions with instructors to build a more comprehensive learner profile.

**Saves data**

There are several features related to individual code submissions that we can extract. These include:

1. **Length of code:** A start and end line are provided for each submission. This can be used to filter out submissions that are sufficiently long to analyse.
2. **Progress and development in a student’s code:** Specifically, we have access to individual saves of students, when the submissions were made, and the source code of the saves. For example in the file logo-boxed-in.csv found in challenge-beginners-2019, we can find a student who has four saves with one being made a week before the others. See Figure 1. This may help us understand how students progress for a particular problem, and how their code develops over time.



Figure 1. Various submissions for a given student in the logo-boxed-in problem in Challenge Beginners 2019.

**Submissions data**

The key contribution of the submissions data is that it allows us to clearly see which test cases were passed and which ones were not. The results column lists out the different test cases, with a status of 0 indicating that the test passed and a status of 1 indicating that the test did not pass.

Test cases are identified by their status (passed or not pass), label (name of the test case) and data (error message). Each test case is of the following form:

[{'status': 0, 'type': 0, 'label': 'bottom-line', 'data': {'message': ''}, 'msg': 'Testing the <strong>bottom</strong> of the box.'},

As such, from the submissions data, we can extract the following features:

1. Number of test cases passed and failed by the student for a specific problem.
2. The specific test cases passed and failed by the student.
3. The error messages associated with each test case.

**List of features**

Chat history

* **A vocabulary of words** that indicate different stages of the interactions between students and instructors.
* **Time between individual messages** between each student and the instructor.

Hypothesis: There are certain words that can be used to represent different stages of the interactions between students and instructors. This can be used to classify a particular message of a student. For example, messages from students that include the word ‘help’ may represent that the student is requesting help. This allows us to represent that message as a call for help. Over time, we can begin to understand **how many times a student has requested for help**, **faces difficulties and solves a problem**.

**Comments:** You can look at classification task in an unsupervised way e.g. **topic modelling** (**can weigh the different terms in the document**) to identify topics and then analyse the unsupervised and then characterise them. This is one research area that has been solved in NLP, so we can apply one of the methods. Bryn had a look briefly on topic modelling and then restrict that to what the student says or the whole conversation (tutor and student). The other thing that might be interesting is sentiment analysis – look at the tone of the conversation. This might reflect how receptive students are to help. If we could apply that technique to the existing dataset, we could potentially then expand to other courses and different dataset. Because not every uses the chat function, how would we use this information? This can show us the difficult exercises.

**To do:** Look at what sort of questions students have. We can perform global analysis on the dataset and then look specifically at problems.

**To do**: There are chat posts and chat histories. It might be useful to understand the size of the dataset. Work out how many threads are and how long the threads are. The beginners and the intermediate are the most active streams.

Code

* **Code complexity:** number of lines, Halstead measures (number of unique operators and operands, number of occurrences of unique operators and operands).
* **Cyclometric complexity:** number of linearly independent paths through the code.
* **Code repetition:** number of repeated code blocks.

**Hypothesis:** More complex code (poorly structured, large number of repeated code blocks, high number of independent paths) likely indicates that the student is struggling and not taking a methodical approach to code development. Students with more complex code are more likely to make errors, may find it harder to debug, and are less likely to do well.

**Comments:** The challenge from this dataset is that the code is quite small. A lot of these measures won’t really be useful in this case. These are measures that are more appropriate for more sophisticated exercises, potentially the advanced stream. The number of people participating in the advanced stream is only a few 100, so it’s a small dataset to work with. It is better to use the simplest versions of code complexity. A lot of the problems have some scaffold code, so it’s not necessarily all student’s individual work. Some students take scaffold code and then modify it directly. We could look at paste events and then see how common it is. If we look at code history, there are different types of saves. You can manually save code and we can also capture if someone has pasted code into the editor. However, this doesn’t lend itself to an AI task. This is more a lot of manually sifting through and understanding the data.

Saves

* **Time of individual code saves and between code saves**
* **The complexity of the code at each code save**
* **Temporal difference between code saves in terms of complexity**

**Hypothesis:** Students who regularly save their work and adjust are likely to do better in their submission and are more likely to demonstrate successful programming traits and competencies.

Submissions

* **Number of test cases passed and failed**
* **Error messages associated with each test case**
* **Concept behind the error message (manually labelled)**
* **Name of each test case passed and failed**
* **Time of submission**

**Hypothesis:** Students that consistently fail test cases may be struggling and underdeveloped in certain skill competencies. Students that do not learn from failed test cases and error messages (as well as feedback from instructors) are likely to not improve. There may also be certain competencies that cause students to pass or fail certain test cases. It would be interesting to see how the different concepts interact e.g. is one concept a prerequisite for understanding another. Students who continuously work on the problem before submitting are more likely to do well than those who cram the problem right before.

**Next steps:** We need to start focusing on one of these tasks. One of these tasks will be quite a lot of work. We need to choose one and start with that since each of these tasks will take a bit of work. We have looked at submissions previously. It would be interesting to look at overall activity. We want to find the activity of someone doing these problems. Last time, we talked about building a collection of features that characterise someone’s activity (this is the overall story of someone’s work and progression through the course). Let’s start small first (rather than analysis code and topic modelling). Let’s start with basic metrics such as time spent. We could leverage some previous work done in estimating total time spent on the task. This is an easy starting point, which makes it easier to progress to the next step. We can also characterise number of submissions, number of saves. The chat history might be another project in and of itself, though we might be able to do a bit of it using existing methods. Bryn has done some analysis before on chat history and he can dig out the notebooks that he’s used previously. We can look at where we can extend them. We can take something that looks interesting and take it further.

**Literature Review**

**Measuring code complexity**

In addition to raw metrics, such as SLOC, comment lines, blank lines, there are three main calculated measures of code complexity: Halstead metrics, cyclometric complexity and maintainability index. Each of these metrics can be estimated using [Radon](Radon:%20A%20python%20tool%20that%20computes%20various%20metrics%20from%20source%20code), a Python tool that computes various metrics from the source code.

|  |  |
| --- | --- |
| [**Halstead Metrics**](https://ieeexplore.ieee.org/abstract/document/8300883) | Halstead metrics measure the operators and operators of each line of code. The measures were first introduced in 1977, and rely on program execution and its measures, which are analysed specifically from the operators and operands from source code.  Halstead measurements allow us to assess testing time, vocabulary, estimate trouble and mistakes. The targets of Halstead measurements are to gauge certain qualities, for example, vocabulary, volume, level, trouble, programming exertion and required programming time.  The measures of Halstead are based on  n1=number of unique or distinct operators.  n2=number of unique or distinct operands.  N1=total number of occurrences of operators.  N2=total number of occurrences of operands. |
| [**Cyclometric complexity**](https://ieeexplore.ieee.org/abstract/document/7725232) **(McCabe’s complexity)** | Measures the number of linearly independent paths through a piece of code. The metric is controversial in its use. On one hand it’s regarded by academia as containing theoretical weaknesses and no better than alternatives such as ‘lines of code’.  At the same time, it is used extensively in industry and supported by most metrics collection tools.  A high cyclometric complexity (CC) indicates that the code will have a higher defect density, testing effort is higher and maintainability severely reduced. |
| [**Maintainability Index**](https://sourcery.ai/blog/maintainability-index/) | The Maintainability Index first appeared in 1992 when it was proposed at the International Conference on Software Maintenance with the goal of establishing automated software development metrics to guide “software related decision making”.  The Maintainability Index tries to give a holistic view of the relative maintenance burden for different sections of a project by blending a series of different metrics. The components are:   1. Halstead’s Volume - HV 2. Cyclomatic Complexity - CC 3. Lines of Code - LOC 4. % of Comments - perCOM   These are then blended into the original formula:  **Maintainability** = 171 - 5.2 \* ln(HV) - 0.23 \* CC - 16.2 \* ln(LOC) + 50 \* sqrt(2.46 \* perCOM) |

[**Predicting Students' Performance in an Introductory Programming Course Using Data from Students' Own Programming Process**](https://ieeexplore.ieee.org/abstract/document/6602003)

**Summary:** Investigates how students’ behaviour changes during programming process (e.g. eagerness to start working on freshly released exercises, following best programming practices) and how these changes affect course outcome.

**Processes:** Static and dynamic code analysis to create features indicating e.g. code that does not compile, has bad programming practices (e.g. style issues such as code indentation, complex or copy-paste code) or contains code that shadows existing variables, which can lead to confusion as variable names are not unambiguous.

**Features:** Submission time (minutes to deadline) were interpreted as eagerness, or as the student having good planning skills, which are both positive factors for programming performance. Students that start working early on course exercises belong more probably to the group ‘excellent’, while students that start working late are more likely to fail the course. Other indicators include: number of indentation errors in the code, that may indicate whether the student pays respect to minor details.

[**Exploring Machine Learning Methods to Automatically Identify Students in Need of**](https://dl.acm.org/doi/abs/10.1145/2787622.2787717)

[**Summary:**](https://www.mdpi.com/2227-7102/11/9/552/htm) A classification-based model to predict students’ performance is built. The chosen algorithm is an SVM algorithm that classifies students into three categories based on their performance: high, medium, and low performance levels. Student performance was predicted over 10 weeks

**Result:** Accuracy of classifier increased in the subsequent week as the data grows. Both the behavioural and learning features combined achieved high classification performance with an accuracy of 74.10% during Week 10.

[**Predicting Performance in an Introductory Programming Course by Logging and Analysing Student Programming Behaviour**](https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6601941)

**Summary:** Proposed a dynamic algorithm designed to predict student performance in a programming course. Their approach analysed directly logged, quantitative data describing aspects of a student’s programming behaviour.

Insights: The paper had a good section on identifying appropriate predictors of student programming performance. Prior research suggests that weaker students produced more compilation pairings where both events result in compilation failures, have the same generalised message, and have the same error location. The paper also hypothesised that in addition to a higher frequency of errors, weaker students would take longer to resolve errors than stronger students. Different types of errors can be more difficult for a student to resolve than others (inc. syntax, computation, identifiers, scope, exceptions, inheritance, abstraction).

[**Early Performance Prediction for CS1 Course Students using a Combination of Machine Learning and an Evolutionary Algorithm**](https://ieeexplore.ieee.org/abstract/document/8820837)

**Methodology:** Constructed a ‘programming profile’ for each student constructed using twenty code metrics which represented metrics proposed by state-of-the-art studies and self-proposed metrics. The programming profile was represented as a feature matrix. The programming profile is used to generate ML models that predict student performance. To perform optimisation in the ML pipelines, they used a combination of an automated approach with an evolutionary algorithm and hyperparameter tuning and random search.

**Results:** Achieved 75.55% accuracy using data from only the first two weeks to predict the student final grade. Shows how their pipeline outperforms state of the art work on similar scenarios**.**

**Next steps**

* Feature extraction
  + Extract the individual features listed in the confirmed list of features.
* Manually categorise the different errors that students face when presented with error messages.
* Manually annotate the problems in terms of concepts tested
  + Bryn: Would I be able to gain access to the platform?