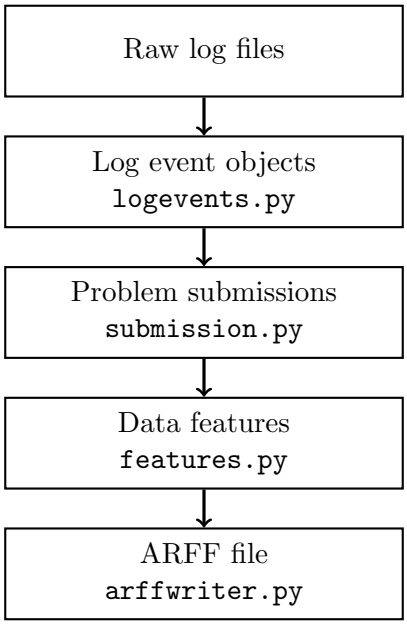
3. Data Processing

3.1. Overview

In section 2.1 the log files SQL-Tutor maintains for each student were briefly discussed. These log files contain very useful information to enable student behaviour to be analysed, and a predictor developed. However, common data mining tools require [15] the input data to be arranged into independent instances which have feature values.

The format of the log files has also changed subtly as SQL-Tutor has been developed; new events have been added, delimiters have changed and spelling errors have been corrected. After reading and understanding the structure of the log files, a Python script was developed to abstract away the differences between the log files and extract features into the ARFF (Attribute Relation File Format) file format used [1] by the Weka data mining tool. Since this is a common task for data mining from SQL-Tutor, this script was made open source[1] so that others can build on this work.

Figure 3.1.: Data flow and key Python modules comprising data extraction script.



An overview of the key abstractions and stages in the parser can be seen in figure 3.1.

[1] See `https://github.com/mattgordon/cosc366-sqltutor`.

First, the log files are read, and parsed into a series of log event objects. These log events are then combined into problem submissions, which contain data about other relevant events that occurred between submissions. Data is then extracted from these submissions into data features, which are then written out into an ARFF file for processing in Weka.

3.2. Log Parsing

The design of the log parser is object-oriented. Each type of event is represented by a subclass of an abstract base class `LogEvent`, which only handles storing the event timestamp. Each subclass of `LogEvent` must implement a single static method `is_event` which determines whether a log line could be parsed into that type of event. Some events are simple, and their meaning is fully represented by their class; other events have more data and have attributes or properties which provide further information about the event. An example of a very simple event is shown in listing 1. A more complex event, involving regular expression parsing and more logic influenced by the differences between log file formats is shown in listing 2.

```
class LoggedInEvent(LogEvent):
    """Event representing user login."""
    @staticmethod
    def is_event(log_line):
        return 'Logged in' in log_line
```

Listing 1: A simple event class representing a user login.

The event objects are then built up into submission objects that hold and 'flatten' all data that arrives in events between problem submissions. These submission objects contain all the data necessary to calculate feature values for each submission.

3.3. Feature Extraction

From the submission objects, features are then extracted. There are three types of features:

1. features representing the current problem, or current state of other aspects of the ITS (e.g. the student model) at the time of the submission;
2. features representing the previous problem, and;
3. cumulative features representing the user's progress in their current session.

Cumulative features output the arithmetic mean, standard deviation, minimum and maximum of their underlying values. The previous problem features and cumulative features are necessary for machine learning algorithms to make use of previous student

behaviour in their predictions; machine learning algorithms assume that each data point is independent from the rest, so to enable predictions to be made on past behaviour features must be included that incorporate a window of previous behaviour.

Current features are detailed in table 3.1, previous problem features in table 3.2, and cumulative features in table 3.3. Including each statistic output for cumulative features, there are 57 features overall. This set of features was chosen based on previous work [9] and our belief that additional features, particularly the SQL-Tutor student level, may be useful in predicting problem abandonment.

The logic for computing feature data resides in subclasses of `FeatureBase`. Subclasses must declare their name, feature value type, and implement a method for extracting their data value from a submission. Other base classes[2] were also implemented to provide support for two common patterns in the feature set; features representing the previous problem, and features representing the user's current session.

3.4. Final Steps

The submissions then need to be assigned as either abandoned, i.e. the student abandoned the problem after this submission, or not abandoned. The logic used here is quite simple; if a student changes to another problem before solving the current problem, or their session ends (logged out) before the problem is solved, the problem is determined as abandoned. In all other cases the problem is determined as not abandoned.

At this point the data instances representing the first two problems in each session are deleted from the data set, as by that point there is not yet enough data to add meaning to the previous problem and cumulative features with enough data. The instances are then written to an ARFF file for processing with Weka. ARFF files contain more metadata [15] than comparable formats such as CSV, which enables better validity checking; for example, nominal attributes must have all possible values declared, eliminating the possibility for errors or unknown values.

3.5. Key Dataset Information

Summary information of the processed datasets, including number of instances and proportion abandoned, is available in table 3.4. A key property of the DatabasePlace dataset is its size; the ARFF file containing its data is over 50MB. This makes this dataset very difficult to process; a J48 decision tree trained and tested on this data using 10-fold cross-validation takes over four minutes on a standard laptop; more sophisticated algorithms cause Weka [7] to crash due to insufficient memory.

It is worth noting that previous work [8] using the same datasets and a similar extraction methodology to that outlined in this section obtained different results to those in table 3.4; in particular, this work reported only 16,541 instances in the Database-

[2] An open area for improvement here would be to implement a framework for dependent features; i.e. features which combine values of other features to produce a new feature.

Table 3.1.: Features representing state as of problem submission.

Feature Name	Feature Description
Violated constraints	The number of constraints violated by the current submission.
Satisfied constraints	The number of constraints satisfied by the current submission.[1]
Help level	The current help (feedback) level requested by the student for the current submission.
Violated constraints decreased	A boolean value; true if the current submission has strictly fewer violated constraints than the previous submission.
Time since previous submission	The number of seconds elapsed since the previous submission.
Problem time from start	The number of seconds elapsed since the student began working on the problem.
Submission number	The number of the current submission to this problem.
Session time from start	The number of seconds elapsed since the user began their current session (logged in).
Problem complexity	The complexity of the current problem.
Student level	The heuristic SQL-Tutor uses to approximate the student's skill level. This corresponds with the problem complexity.
Student level complexity difference	The difference between the student level and the current problem complexity.[2]
Submission same as previous	A boolean value; true if the student has submitted a solution that is identical to the previous submission.
Completed problems	The number of problems solved in the student's current session.
Attempted problems	The number of problems attempted in the student's current session.
Attempted completed difference	The difference between the number of problems completed and the number of problems solved in the student's current session.

[1] Some constraint IDs are included more than once in the log file output; these duplicates are not removed as they are specific to the submitted solution.

[2] Defined as `current problem complexity - student level`.

Table 3.2.: Features representing the previously worked on problem.

Feature Name	Feature Description
First submit time	The number of seconds it took for the student to make their first submission to the problem.
Time taken	The number of seconds taken to solve the problem.
Completed	A boolean value; true if the student solved the problem, otherwise false.
Submission count	The count of submissions the student made to the problem.
Maximum violated constraints	The maximum number of constraints that were violated by a submission to the previous problem.
Submission time statistics	The mean, standard deviation, maximum and minimum of the number of seconds elapsed while making a submission to the problem.
Same database	A boolean value; true if the student is currently working on a problem from the same database as the previous problem.
Distinct feedback options	The number of distinct feedback options the user selected (or, in some cases, that were automatically chosen by the ITS).
Time since session start	The seconds elapsed between the beginning of the session the previous problem was started in and the time the user began working on the problem.
Problem complexity	The complexity of the problem.

Table 3.3.: Features representing cumulative data[1] from the user's current session.

Feature Name	Feature Description
Violated constraints	The number of constraints violated during the session.
Problem completion time	The number of seconds required to complete a problem.
Time between submissions	The number of seconds between submissions.
Number of submissions per problem	The number of submissions made to each problem.
Problem complexity	The complexity of the problems encountered during the session.
Time until first submission	The number of seconds elapsed until the first submission to a problem.

[1] Cumulative features include average (mean), standard deviation, maximum and minimum of values collected during the current session.

Table 3.4.: Summary of processed datasets.

Dataset	# instances	# abandoned	% abandoned
DatabasePlace	136,414	15,422	11.31%
Canterbury 2010	2409	130	5.49%
Canterbury 2014	2555	63	2.47%

Place dataset, an order of magnitude less than what was extracted here, yet a far higher proportion of these instances were abandoned (47.48%).

Without access to the previous work's source code, it is difficult to say exactly how this discrepancy arose. A basic sanity check[3] confirmed that we were extracting the correct number of instances from the log file sets.

[3]This sanity check was simply searching for `Post-process:` in all log files and counting the occurrences, then checking that this number was equal to the number of instances before deletion of the first two problems. `Post-process:` is a key phrase that is written by SQL-Tutor to the log file whenever a solution is submitted for evaluation.

```
class PostProcessEvent(MultilineLogEvent):
    """Event representing solution evaluation (post-processing)."""
    RE = re.compile(
        'Post-process:\s*' +
        'Satisfied constraints: (?:\(([0-9\s]+)\)|NIL);?\s*' +
        'Violated constraints: (?:\(([0-9\s]+)\)|NIL);?\s*' +
        'Feedback level: ([0-9])')

    def __init__(self, timestamp, line, file):
        super().__init__(timestamp, line, file)
        if 'Satisfied' in line and 'Violated' in line:
            # we have everything on one line
            self._parse_one_line(line)
        else:
            # need to iterate over lines until blank line
            self._parse_multiline(line)

    def _parse_one_line(self, line):
        match_groups = re.match(self.RE, line)
        if not match_groups.group(1):
            self._satisfied_constraints = []
        else:
            string_constraints = match_groups.group(1).split()
            self._satisfied_constraints = \
                    [int(x) for x in string_constraints]
        if not match_groups.group(2):
            self._violated_constraints = []
        else:
            string_constraints = match_groups.group(2).split()
            self._violated_constraints = \
                    [int(x) for x in string_constraints]
        self._feedback_level = int(match_groups.group(3))

    def _parse_multiline(self, init_line):
        <snip>
```

Listing 2: A more complex log event representing solution post-processing.