**EclipseHelper Documentation**

WEKA version 3.6

This documentation for EclipseHelper is geared towards giving a quick overview of the various features in EclipseHelper. Since EclipseHelper spanned multiple maintainers, not every aspect is covered in this documentation. As such, some assumptions about the readers are made:

1. **Is aware of the general architecture of the EclipseHelper (difficulty prediction, analyzer, and etc.)**
2. **Knows what the RatioFeatures, participant timelines, arff files, and the analyzer workflow are.**
3. **Understands how the prediction mediator and DifficultyRobot works.**

**Section 1: General Overview**

**The Analyzer Workflow**

**AnAnalyzer->DifficultyEventProcessor->DifficultyRobot(Aggregation, MAPPING)->AMediatorRegistrar->AFileReplayAnalyzerProcessor(Optional)->AnAnalyzerProcessor(Listens)->AnArffGenerator**

The above workflow details the journey of Eclipse logs to the ArffGenerator; the ArffGenerator is responsible for generating each participant’s arff files. First, AnAnalyzer reads in the Eclipse log by process ICommands one at a time. AnAnalyzer will then notify its listeners of the new ICommands. DifficultyEventProcessor will then take the ICommand and hand it to the difficulty robot. The difficultyrobot will then aggregate the ICommands and map the aggregated chunk to the appropriate ratio features with the RatioCalculator. The ratio feature will then go on an intricate path until it reaches AnAnalyzerProcessor where it is put into each participant’s participant timeline. Finally, AnArffGenerator write the participants’ ratio features to arff files.

This project is segmented into several big portions, each one to address the particular needs of the research. At first, arff file representation of our raw data was needed, so the arff file generator(section 5) was built. Then, we needed a platform to conduct the research using in code algorithms; thus, we wrote code for a custom prediction lifecycle in the package analyzer.tracker. Finally, we did explore some stuck point and interval data.

**Section 2: Table of Contents**

Table of Content

1. Custom Prediction LifeCycle (Package analyzer.tracker)
2. Attribute Mixture (Package difficultyPrediction.metrics)
3. Arff File Generation
4. Stuck Points and Intervals

**Section 3: Custom Prediction LifeCycle**

The codes for the custom prediction lifecycle are almost all uniformly under the package analyzer.tracker. The prediction life cycle is just a typical classification flow: we load the training and testing sets, do some attribute selection and dimensionality reduction, train the classifier, test the classifier accuracy, and finally output some result in a suitable format.

The prediction lifecycle is mediated by one central class **PredictionTracker.java**. It contains the methods for loading arff training and testing sets, filtering training and testing sets, training the classifier, classifying the testing instances, and outputting the testing results in a suitable format with an outputter.

The following subsections will go into greater detail on the each function.

*Section 3A*: Loading Training and Testing Sets

Loading training and testing sets only entails setting the appropriate path to the respective **arff files** and calling loadInstances(). Refer to section 5 on the peculiarities of ArffFile generation.

*Section 3B*: Filters

WEKA calls any transformation to the training and testing set “filtering.” To apply any “filtering” to the training set, set the appropriate filter and invoke the corresponding method in the PredictionTracker at any point before calling classification in your code.

Note that there is already a provided filter in the package for the SMOTE oversampling algorithm. It is a wrapper around WEKA’s actual SMOTE filter and is much easier to use. The name is SmoteFilter. Also provided is an interface called Filter for implementing other filters.

*Section 3C*: Training and Classification

The training and the classification stages only involve calling their respective methods and putting in an appropriate classification algorithm beforehand.

*Section 3D*: Outputting

Unfortunately, the prediction results are not very geared towards human readability. Normally, functional programming is fairly useful in this stage; however, Java 1.7 does not have that capability. So, the outputter classes were made to handle filtering results, printing results, and accumulating results.

There are a few outputter classes providing the aforementioned capacities, all have “Outputter” prefix to their names:

1. AccumulatorOutputter- Accumulates the prediction results into an ArrayList.
2. Wrong/RightPredictionOutputter-Outputs either the correct or incorrect prediction and writes then to a file of your choosing. The output includes the attributes and the ground truth.
3. ResamplingOutputter-Resample the output instances.
4. CombinedOutputter-Combine any number of outputters in parallel.

Interface ResultOutputter as well as AbstractOutputter are provided for easy implementations of other outputters.

Note that all outputters can be chained in serial; the abstract outputter parent class automatically provides the necessary functionality to notify the next outputter in the series (a la notifyChildren method).

**Section 4 RatioCalculator:**

Classes that implement the RatioCalculator interface can be used as the feature aggregators that convert raw eclipse user ICommands to segmented ratio features used in the research. APercentageCalculator is Jason’s old ratio calculator; the one used in the more recent research is ATestRatioCalculator. Built into the ATestRatioCalculator are the three different schemes used in the recent research: A0, A1, and A3 (See the paper for detailed descriptions of the three schemes).

Any modification to the present mixture distribution of the present features can be easily done here. Removing features can be done directly to the training set. Adding features is a little bit more involved; in such case, modifications must be made to RatioFeatures and ParticipantTimeLine.

ATestRatioCalculator and APercentageCalculator are both in the package difficulty.metrics. RatioCalculatorFactory for selecting between different RatioCalculators is also in the same package.

**Section 5:**

**ICommand Event Files -> AnAnalyzer ->DifficultyRobot(RatioFeatures)->(Notifies) AnalyzerProcesser(Put ratio feature in ParticipantTimeline)->(Notify finish Participant) ->(Timeline) ArffGenerator**

The AnArffGenerator produces the arff file, which is a data file with similar format to that of a csv. This class is integrated with the Analyzer to provide an UI for generating different arff files for each experiment participant or participants. The above workflow is activated whenever new arff files are chosen.

More specifically, AnArffGenerator uses AnAnalyzerProcessor’s participantTimeLine as data source for the arff files. The participantTimeLine is a HashMap of participant ids to their individual times lines, which are just lists of percents of each feature in each discrete time chunk. The timeline gets filled from the RatioFeatures emitted by DifficultyRobot while processing ICommands.

**Section 6: Stuck Points and Stuck Intervals**

Upon the initial call to doLoadLog in analyzer, which will run every time the analyzer loads eclipse logs, AnAnalyzer will load the stuck points and stuck intervals from their respective CSV files. The stuck points and intervals of all users will be stored into the hashmaps stuck point and stuck interval in Analyzer. The keys to hashmaps are the user ids; each user’s the stuck points and stuck intervals are in priority queues sorted by the time and date. These stuck points and intervals are used for mainly graphics currently.