ESRB Classifier

Architecture Design Documentation

## Major Design Considerations:

### Challenges

Our software must read a dataset from disk and package it reliably so that decision trees can be generated, even if there exist missing fields. The problem with missing data in our context is that the absence of one content descriptor may influence the rating on its own. For example, the presence of gore or extreme violence in a title may necessitate a higher rating in a title that otherwise might be suitable for a younger audience. Our approach thus far is to assume the presence rather than an absence of these descriptors because receiving a higher classification, although not ideal from the perspective of the publisher, provides the greatest safety for the consumer at large. A more robust approach might involve testing both cases to see if they diverge, or aggregating data about which features correspond to the ‘escalation’ of a classification.

The implementation of a feedback loop system where a user can increase the accuracy of predictions over time is a goal that requires tracking of information generated in the user interface, and so this information must be mutated at runtime as well as being available between terminations of the software, like the model itself. This introduces the conundrum of which entity ‘owns’ the information. If the information is owned by the model layer, it may live in the <RandomForest> class itself, as outlined in the Layered Architecture Approach section of this document and be written to the disk as the model is being saved. It may also exist in an object which has the capability to increment itself, which could be passed to the user interface within the <ViewData> object.

### Layered Architecture Approach

The development of our machine learning software will be best facilitated by an architectural schema that provides encapsulation of our components with a narrow pipeline of communication between modules. This ensures the security of our internal data from being mutated by processes ‘downstream’ (as in, further along in the data’s journey from disk to screen). A layered architecture allows this modularity, making it easier to add, remove, or update individual components without affecting the entire system. Our system will feature a user interface component that allows the end user to safely manipulate the software, so this graphical implementation must be sufficiently isolated from the machine learning process as to make failure due to user error unlikely. In addition, having a narrow interface between our modules (or layers) allows the development of our components in parallel. Our communication between layers ideally should be no more complex than passing a single object, which means that the testing of each layer only requires small sample data from the layer upstream. Errors in the downstream layers are easier to track when their connections to their upstream neighbors are easily verified. Our approach will consist of three layers:

#### Data Ingestion Layer (DIL)

This layer is responsible for reading and sanitizing the dataset for use in the classifier’s training from the disk, converting the input into a HashMap structure and partitioning the data into training, validation and testing subsets, at which point the data is ready to be used in the classifier. When results are obtained or the configuration of the algorithm’s settings changes, this is where the disk operations that store this information take place.

Upstream : Read <DataSet> from disk, store <RandomForest> to disk.

Downstream : Pass <DataSet> to MVE, receive <RandomForest> from MVE.

#### Modeling & Validation / Evaluation Layer (MVE)

This layer is where the classification into an ESRB category is made. The system first takes in training data and determines the termination criteria for generating decision trees. Termination criteria (minimum tree depth, min number of samples) are chosen arbitrarily. A bootstrapped subset of equal size from each dataset is created (with replacement). It will generate decision trees in a random forest, represented by class <DecisionTree> and <RandomForest> respectively. The trees are traversed to measure the Gini index, determining the best splits. At each step, information gain is maximized by minimizing the Gini impurity. The number of times a feature was used for splitting (bagged features) is tracked throughout, as well as the majority classification in each tree leaf node. Then the validation set is compared against the majority votes to assess the accuracy of the forest’s decisions, allowing tuning of the hyperparameters. The resulting <RandomForest> is used to create a <ViewData> object that supplies the UXL with the information it needs to display to the user. The third subset of the original dataset, the testing set, can be run through the prediction model to see the outcome of the final model.

Upstream: Receive <DataSet> from DIL, Pass <RandomForest> to DIL.

Downstream: Pass <ViewData> to UXL, Receive <LikeCounter> from UXL.

#### Presentation Layer (UXL)

The Presentation Layer’s job is to display the information passed downstream from the MVE layer so that the visual elements report the result of the classification, any relevant content descriptors that would be displayed on the packaging of the game, and a similar and contrasting title (maximum and minimum shared features with the same classification) to give a potential reviewer more insight into the prediction. The user interface will allow the submission of a new hypothetical game to test or entry of a new dataset for training, likely through a Java swing form, for easy and safe manipulation of the model.

Upstream: Receive <ViewData> from MVE, Pass <LikeCounter> to MVE

Downstream: Display data to user, receive input from user.

## Diagrams

A blueprint of a computer program

Description automatically generated