370 Assignment 2: System Modeling

Diwash Bhattarai

# 1. ARCHITECTURAL DESIGN DOCUMENTATION

## A. Major Design Considerations and Architectural Organization

The "A SMARTER APPROACH TO STUDENT WELL-BEING," implemented in C++ Random Forest classifier, aims to identify students at risk for depression and provide actionable recommendations via a Command Line Interface (CLI). Given its purpose and the nature of a CLI application with a self-contained ML model, the architectural design prioritizes simplicity, modularity for the core ML components, and data integrity for the single-user history.

**Modularity:** The system is designed with distinct logical components: data loading/preprocessing, the Decision Tree algorithm, the Random Forest ensemble, prediction logic, recommendation generation, and user interaction (CLI).

**Data Flow:** A clear, sequential data flow is established: User Input -> Preprocessing -> Model Prediction -> Risk Categorization -> Recommendation Generation -> Output to User. History is read before input and written after prediction.

**Self-Contained ML Engine:** The Random Forest, including its constituent Decision Trees, is implemented directly within the C++ application. This avoids external library dependencies for the core ML algorithm, aligning with the project's C++ focus, but places the onus of correctness and efficiency on the custom implementation.

**Single-User Focus (CLI)**: The current architecture is tailored for a single-user experience with local data history storage (risk\_history.txt). This simplifies data management and eliminates the need for user authentication or a database.

**Performance (CLI Context):** While not a high-throughput web server, efficient C++ implementation of the Random Forest and data handling is still a consideration to meet the non-functional requirement of quick response times in the CLI.

**Maintainability:** By encapsulating different functionalities into classes and distinct functions, the codebase aims for better readability and maintainability, crucial for this implementation.

**Accuracy of Custom Model:** A significant consideration is the accuracy and robustness of the Random Forest. Its effectiveness hinges on the correct implementation of tree-building heuristics (Gini impurity, splitting), bagging, and feature subsampling.

**Architectural Organization (Conceptual Layers for CLI Application):**

For this C++ CLI application, a strict multi-layer server architecture isn't directly applicable, but we can think of it in terms of logical components:

**User Interaction Layer (CLI):**

* Handles all input/output directly with the user via the console.
* Prompts for data, displays predictions, history, and recommendations.
* (Represented by functions within main() and get\_simplified\_user\_input\_and\_transform, display\_recommendations).

**Application Logic/Control Layer:**

* Orchestrates the flow of operations.
* Manages data transformation from simplified user input to the full feature vector.
* Invokes the prediction model.
* Calls the recommendation engine.
* Manages history file I/O.
* (Primarily within main() and the top-level logic of supporting functions).

**Machine Learning Engine Layer:**

*DecisionTree Class:* Encapsulates the logic for a single decision tree (node structure, splitting, training, prediction).

*RandomForest Class:* Manages an ensemble of DecisionTree objects, handles bootstrapping, feature subsampling (delegated to DecisionTree), training the ensemble, and aggregating predictions.

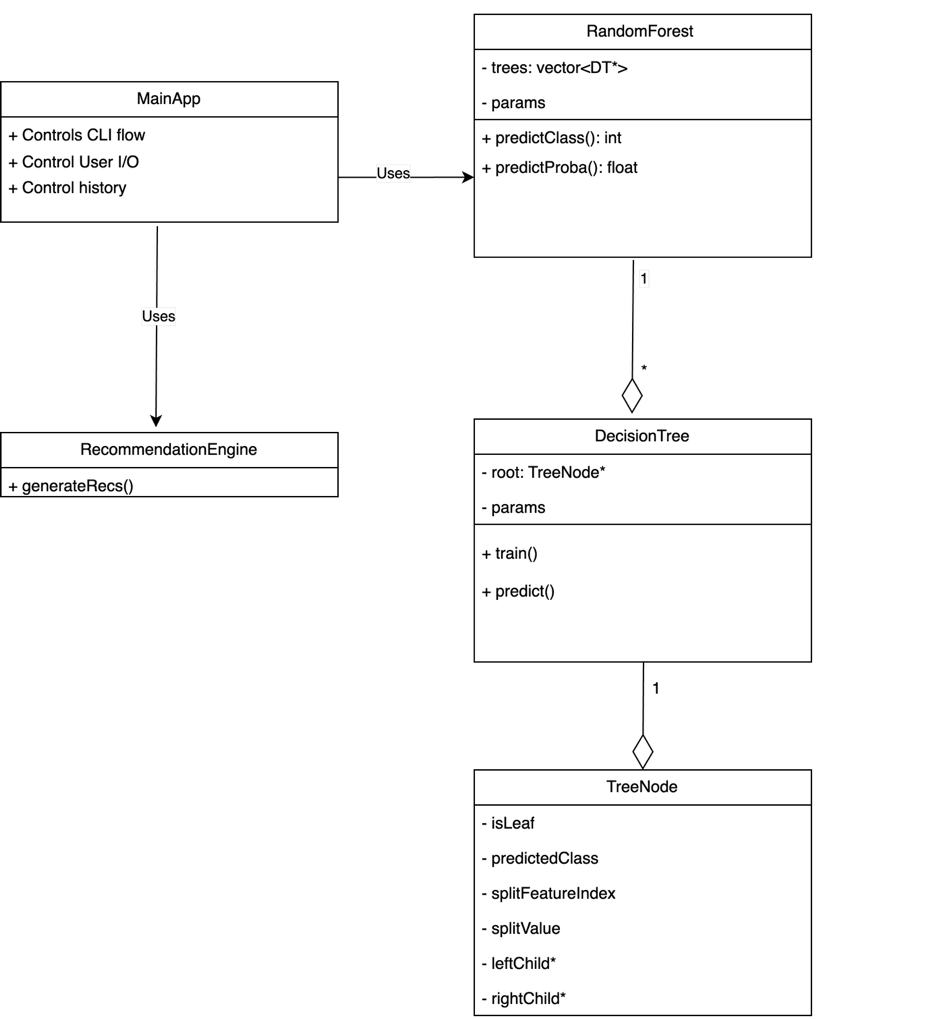
*Data Preprocessing/Loading:* Functions like load\_numeric\_csv and split\_data.

**Data Handling (Simplified for CLI):**

Primarily in-memory data structures (std::vector for datasets).

File I/O for loading the initial dataset (cleaned\_student\_data.csv) and for reading/writing the single-user risk\_history.txt. No complex database or encryption layer is implemented for this CLI version.

## B. Conceptual Class Diagram of the System



**Key Relationships:**

**MainApp (Conceptual, represented by main() and helper functions):**

* Uses RandomForest for training and prediction.
* Uses RecommendationEngine (conceptually, implemented as display\_recommendations function) to generate advice.
* Manages data loading and history file I/O.

**RandomForest:**

* Aggregates multiple DecisionTree objects. (A Random Forest "has-a" collection of Decision Trees).

**DecisionTree:**

* Aggregates TreeNode objects to form its structure (A Decision Tree "has-a" root TreeNode).

**TreeNode:**

* Can recursively point to other TreeNode objects (its children).

# 2. STRUCTURAL MODELING USING OBJECT CLASSES

## A. Class Design Definitions

**TreeNode**

*Definition*: Represents a node within a decision tree. It can be an internal node (containing a split rule) or a leaf node (containing a class prediction).

*Attributes:*

* is\_leaf: bool
* predicted\_class: int (relevant if is\_leaf is true)
* split\_feature\_index: int (relevant if is\_leaf is false)
* split\_value: double (relevant if is\_leaf is false)
* left\_child: TreeNode\* (pointer to the left child node)
* right\_child: TreeNode\* (pointer to the right child node)
* Methods (Constructors/Destructor implicitly):
* TreeNode(predicted\_class: int): Constructor for leaf node.
* TreeNode(split\_feature\_index: int, split\_value: double): Constructor for internal node.
* ~TreeNode(): Destructor to manage memory of children.

*Associations:* Aggregates child TreeNodes (a tree is composed of nodes).

**DecisionTree**

*Definition:* It handles the logic for training the tree and making predictions.

*Attributes:*

* root: TreeNode\* (pointer to the root node of the tree)
* params: DecisionTreeParams (struct holding max\_depth, min\_samples\_leaf, num\_features\_to\_consider for random feature subset at splits)
* rng\_dt: std::mt19937 (random number generator for feature subsampling during splits)

*Methods:*

* DecisionTree(params: const DecisionTreeParams&, seed: unsigned int): Constructor.
* ~DecisionTree(): Destructor (deletes the root node, which triggers recursive deletion).
* train(features: const DatasetFeatures&, labels: const DatasetLabels&): Builds the tree.
* predict(sample: const Sample&): int - Predicts the class for a single sample.
* find\_best\_split(...): SplitInfo (private helper)
* build\_tree\_recursive(...): TreeNode\* (private helper)

*Associations:* Aggregates a TreeNode (the root).

**RandomForest**

*Definition:* Encapsulates an ensemble of decision trees. Manages the creation, training, and prediction aggregation of these trees.

*Attributes:*

* trees: std::vector<DecisionTree\*> (a collection of pointers to DecisionTree objects)
* params: RandomForestParams (struct holding num\_trees, tree\_params for individual trees, bootstrap\_sample\_ratio, random\_seed)
* feature\_names\_internal: std::vector<std::string> (stores feature names used for training)

*Methods:*

* RandomForest(params: const RandomForestParams&): Constructor.
* ~RandomForest(): Destructor (iterates through trees and deletes each DecisionTree).
* train(features: const DatasetFeatures&, labels: const DatasetLabels&, feature\_names: const std::vector<std::string>&): Trains all decision trees in the ensemble using bootstrapping and feature subsampling.
* predict\_class(sample: const Sample&): int - Predicts the final class using majority vote from all trees.
* predict\_probability\_class1(sample: const Sample&): double - Predicts the probability of class 1.
* create\_bootstrap\_sample(...): std::vector<size\_t> (private helper)

*Associations:* Aggregates multiple DecisionTree objects (a "has-many" relationship).

**Helper Structs (Data Only):**

* DecisionTreeParams: max\_depth: int, min\_samples\_leaf: int, num\_features\_to\_consider: int
* RandomForestParams: num\_trees: int, tree\_params: DecisionTreeParams, bootstrap\_sample\_ratio: double, random\_seed: unsigned int
* SplitInfo: feature\_index: int, split\_value: double, gini\_gain: double, left\_indices: std::vector<size\_t>, right\_indices: std::vector<size\_t>
* HistoryEntry: timestamp: std::string, probability\_class1: double, risk\_level\_str: std::string

**Generalization Hierarchies:** In this implementation for a specific task, there are no explicit inheritance hierarchies defined for the core ML components. We are building concrete classes.

## b. Notations for the Diagrams:

**Class Boxes:** A rectangle divided into three compartments:

**Top:** Class Name (e.g., RandomForest)

**Middle:** Attributes (e.g., trees: std::vector<DecisionTree\*>, params: RandomForestParams)

**Format:** attributeName: dataType

**Bottom:** Methods/Operations (e.g., train(...), predict\_class(...))

**Format:** methodName(parameterName: parameterType, ...): returnType

**Associations:**

Aggregation (Has-A): A line with an open (hollow) diamond on the side of the "whole" class, pointing to the "part" class.

**Example:**

RandomForest ◇-----> DecisionTree (A Random Forest has Decision Trees). Multiplicity like 1..\* can be added near the DecisionTree end.

**Example:**

DecisionTree ◇-----> TreeNode (A Decision Tree has a root TreeNode).

**Usage/Dependency (Uses):** A dashed arrow -----> pointing from the using class to the used class.

**Example:** MainApp (represented by main()) -----> RandomForest.

**Pointers:** Indicated by \* (e.g., TreeNode\*).

**Visibility:**

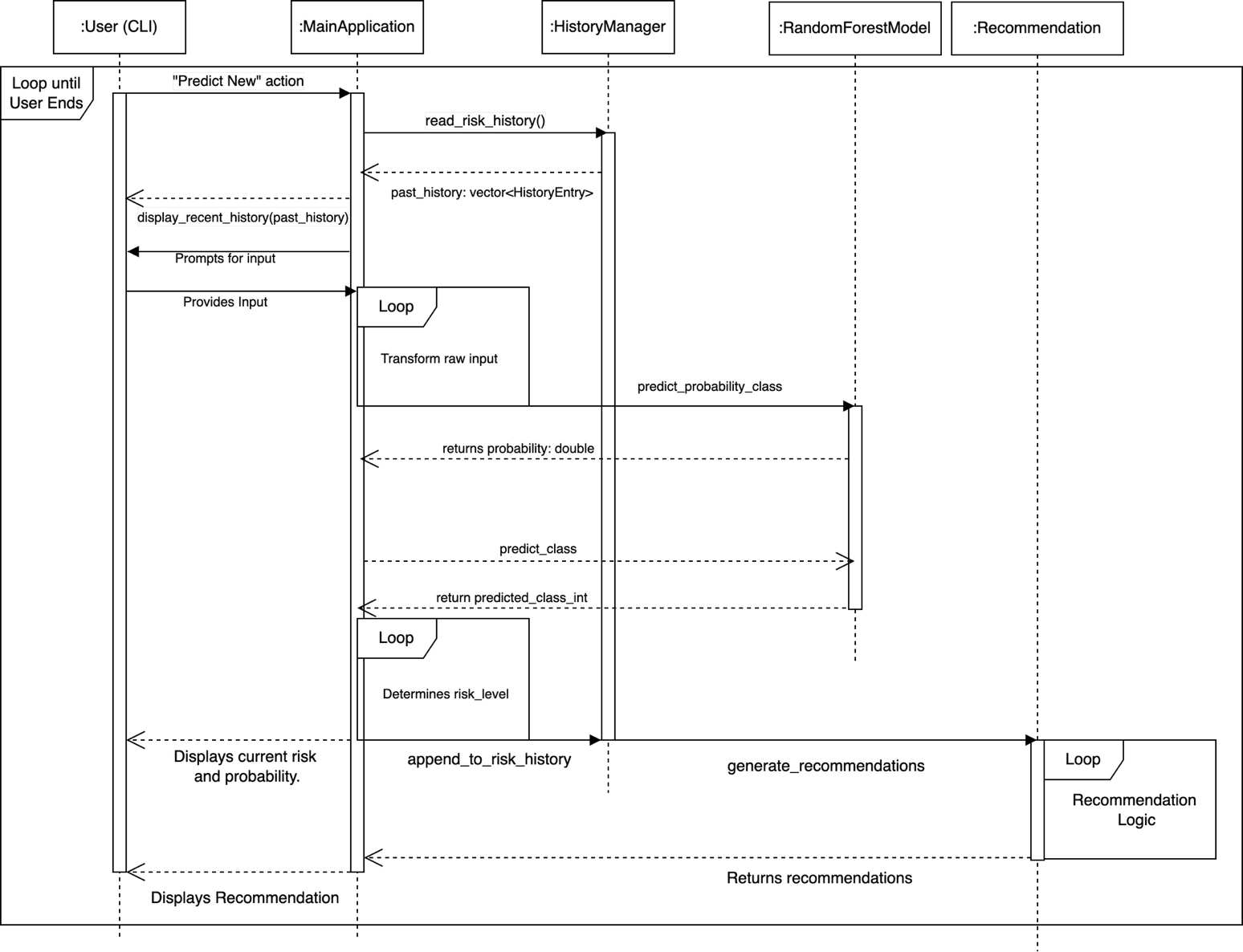
+ public

- private

# protected

# 3. Interaction Modeling using Sequence Diagrams

## a. Sequence Diagrams by Functional Requirement:



## b. Notations for the Diagrams:

**Lifelines:** A vertical dashed line for each object/participant (e.g., User (CLI), MainApplication, RandomForestModel, HistoryManager, RecommendationLogic). A box with the object name (and class) is at the top.

**Activation Bars (Execution Occurrence)**: Thin rectangles drawn on the lifelines, indicating the period during which an object is performing an operation (i.e., a method is active).

**Messages:** Horizontal arrows between lifelines.

**Synchronous Call**: Solid line with a filled arrowhead, labeled with the method name and parameters (e.g., predict\_class(sample)).

**Return Message:** Dashed line with an open arrowhead, pointing back to the caller, labeled with the return value (e.g., predictedClass).

**Loops/Conditionals:** Frames can be drawn around parts of the diagram with labels like loop or alt (for if/else).

**Execution Order:** Time progresses downwards. Messages are ordered vertically based on their sequence.