MindTrack

A SMARTER APPROACH TO STUDENT WELL-BEING

Diwash Bhattarai

Faiza Khan

Durapatie (Priya) Ramdat

Table of Contents

1. INTRODUCTION

In today's demanding academic environment, students frequently encounter significant pressure from coursework, examinations, and various external responsibilities. This can lead to heightened stress, anxiety, and burnout, which detrimentally impact both academic performance and overall mental well-being. Many students grapple with effective time management, suffer from insufficient sleep, and feel overwhelmed by their workloads, often resulting in diminished mental health and reduced productivity. While traditional support systems exist, they often provide generic advice that fails to address the specific and unique challenges individual students face.

The Behavioral Optimization & Mental Wellness System aims to bridge this critical gap. By analyzing individual behavioral patterns and lifestyle data, the system is designed to provide students with personalized, actionable recommendations. These tailored insights will focus on optimizing study techniques, promoting effective wellness strategies, and offering practical time management tips, all derived from patterns of what has proven effective for others facing similar challenges. Unlike static advice, this system is envisioned to be dynamic, continuously adapting based on user feedback and evolving data to ensure ongoing relevance and efficacy. More than just a habit tracker, it empowers students to proactively manage their routines, mitigate stress, and cultivate sustainable habits essential for long-term academic success and personal well-being. The core of this system will be a predictive model capable of assessing a student's likelihood of experiencing depression, enabling timely and targeted support.

2. Glossary

Academic Pressure: The stress and anxiety experienced by students due to coursework, exams, and overall academic expectations.

Behavioral Optimization: The process of refining routines and habits to improve efficiency, reduce stress, and enhance overall performance and well-being.

Burnout: A state of emotional, physical, and mental exhaustion caused by excessive and prolonged stress.

CLI (Command Line Interface): A text-based interface used for interacting with a software program.

Depression: A mental health disorder characterized by persistent sadness, lack of motivation, and decreased interest in daily activities.

Feature Engineering: The process of creating new input variables (features) for a machine learning model from existing raw data to improve model performance or interpretability.

Functional Requirement: A statement that defines a specific function or behavior the system must perform.

Hyperparameter Tuning: The process of optimizing the parameters of a machine learning algorithm that are set prior to the learning process itself.

Lifestyle Habits: Behavioral factors such as sleep patterns, social engagement, physical activity, and diet, which may influence mental well-being.

Machine Learning (ML): A field of artificial intelligence that enables computer systems to learn from data and make predictions or decisions without being explicitly programmed for each task.

Non-Functional Requirement: A statement that describes a quality or constraint of the system, such as performance, security, or usability.

One-Hot Encoding: A process of converting categorical data variables into a numerical format suitable for machine learning algorithms, where each category becomes a new binary (0 or 1) feature.

Prediction Model: A machine learning algorithm that processes input data and forecasts an outcome, in this case, the likelihood of depression.

Random Forest Algorithm: A supervised learning model that operates by constructing multiple decision trees during training and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

Recommendation System: A feature that provides personalized suggestions to students on how to reduce their risk of depression or improve well-being, based on data analysis and comparisons.

3. User Requirements

This section outlines the key needs of the users (students and potentially counselors)

UR1: Know My Current Mental Wellness Risk Level (Functional Requirement)

Description:

The system should tell the student's current risk level for experiencing depression (e.g., high, medium, or low) based on the information provided about their academics, lifestyle, and stress.

Rationale:

This core feature provides immediate awareness. Understanding the current risk level is the first step towards seeking help or making positive changes if needed. It allows for early identification of potential issues.

UR2: Get Advice on How to Improve My Well-being (Functional Requirement)

Description:

If the system indicates that a student might be at risk, or even if the student is doing okay, students want to receive personalized, simple, and actionable suggestions on what they can do to improve or maintain my mental well-being and study habits.

Rationale:

Simply knowing the risk isn't enough; the students need guidance. Personalized advice is more likely to be effective than generic tips, helping the students make practical changes to their daily routine or seek appropriate support.

UR3: Track Changes in My Risk Over Time (Functional Requirement)

Description:

A student will want to be able to see how their risk level changes if they use the system multiple times, perhaps by seeing a simple chart or history.

Rationale:

This helps the students understand if the changes they are making is having a positive impact or if certain periods (like exam season) increase their risk, allowing for better long-term planning and self-awareness.

UR4: Trust the System's Predictions (Non-Functional Requirement)

Description:

A user needs to feel confident that the risk level the system predicts is reasonably accurate and reliable.

Rationale:

If the predictions are often wrong, the user won't trust the system or its recommendations, making it useless. High accuracy is crucial for user trust and the system's credibility.

UR5: Get Results Quickly (Non-Functional Requirement)

Description:

A student, when provides input of their information, wants to see the risk prediction and any advice without waiting for a long time.

Rationale:

A slow system is frustrating and discourages use. Quick feedback makes the interaction smooth and encourages regular check-ins.

UR6: Know My Personal Data is Safe (Non-Functional Requirement)

Description:

A student needs to be sure that their sensitive personal information they share about their mental state, habits, and academics is kept private and secure.

Rationale:

Mental health data is highly personal. The students will only use the system if they trust that their information is protected from unauthorized access and won't be misused.

4. System Requirements

This section details the system's specific functionalities and qualities, mapped to the user requirements.

4 A. Functional Requirements:

ID: SR1

User Requirement Mapping: UR1: Know My Current Mental Wellness Risk Level

System Requirement: Risk Prediction: Predict the student's depression risk level.

Description:

The system shall accept student input. Based on this input, it will utilize a trained Random Forest machine learning model to predict the student's likelihood of experiencing depression, classifying it into a risk category and providing an associated probability score.

Inputs & Source:

Inputs:

- Age (integer)
- Academic Pressure (integer, scale 1-5)
- CGPA (float, scale 0-10)
- Study Satisfaction (integer, scale 1-5)
- Suicidal Thoughts (integer, 0 for No / 1 for Yes)
- Work/Study Hours (float, hours per day)
- Financial Stress (integer, scale 1-5)
- Family History of Mental Illness (integer, 0 for No / 1 for Yes)
- Sleep Ordinal (integer, representing categories like <5h, 5-6h, 7-8h, >8h)
- Gender (integer, 0 for Female/Other, 1 for Male)
- Degree Type (integer, representing simplified categories like Science/Tech vs. Non-Science/Arts)

Source: Direct user input via the Command Line Interface (CLI)

Outputs & Destination:

Outputs:

- Predicted Risk Level (e.g., "HIGH RISK", "LOW RISK")
- Probability of High Risk (float, e.g., 0.75)

Destination: Displayed on the Command Line Interface (CLI) screen for the user.

Precondition:

- User must have entered valid data for all required input fields.
- The system must have a trained Random Forest model loaded and accessible.
- Input data values must be within their expected ranges and formats.

Algorithm/Process:

- Receive raw simplified inputs from the user via the CLI.
- Internally transform these simplified inputs into the full numerical feature vector expected by the Random Forest model. This includes:
- Calculating derived features (e.g., Total_Stress from Academic and Financial Pressure).
- Mapping categorical choices (e.g., user's simplified Degree Type, Age) to the appropriate one-hot encoded Age_Group_* and Degree_Type_* columns, ensuring other related one-hot encoded columns are set to 0.
- Ensuring all other one-hot encoded features (e.g., for specific cities, regions, detailed degrees if they were part of model training but not direct CLI input) are set to 0 as a baseline.
- Feed the complete, ordered feature vector to the loaded Random Forest model.
- The model outputs a probability for the positive class (e.g., likelihood of being 'High Risk').
- Apply a predefined threshold (e.g., 0.4 or 0.5) to the probability to determine the final risk category (e.g., "HIGH RISK" or "LOW RISK").

Postcondition:

- The user is presented with their predicted depression risk level (e.g., High/Low).
- The user is presented with the calculated probability score associated with the high-risk category.

ID: SR2

User Requirement Mapping: UR2: Get Advice on How to Improve My Well-being

System Requirement: Personalized Recommendations: Provide well-being and study habit suggestions.

Description:

Based on the predicted depression risk level and key input factors provided by the student, the system shall generate and display a set of personalized, actionable recommendations aimed at improving or maintaining mental well-being and academic habits.

Inputs & Source:

Inputs:

- Predicted Risk Level (from SR1: e.g., "HIGH RISK", "LOW RISK")
- Specific raw user input values used for conditional advice, such as:
- Academic Pressure value
- CGPA value
- Work/Study Hours value
- Financial Stress value
- Suicidal Thoughts status (Yes/No)
- Sleep Ordinal value
- Source: Output of SR1 (predicted risk) and the preprocessed feature vector containing user inputs.

Outputs & Destination:

Outputs: A list of textual recommendations.

Destination: Displayed on the Command Line Interface (CLI) screen for the user.

Precondition:

A depression risk prediction (SR1) must have been successfully generated for the user.

Algorithm/Process):

- Receive the predicted risk level and the user's (transformed) input features.

- Apply a rule-based logic engine:
- Provide general advice based on the overall predicted risk (High or Low).
- Conditional Advice (Examples):
- IF Suicidal Thoughts == YES: Prioritize and strongly advise seeking immediate professional help and provide crisis contact information.
- IF Academic Pressure > 3 (on a 1-5 scale): Suggest stress reduction techniques related to academics, time management, and seeking academic support.
- IF CGPA < 7.0: Offer tips for improving study habits, seeking academic resources, or forming study groups.
- IF Work/Study Hours > 8: Advise on the importance of work-life-study balance and strategies to manage workload.
- IF Financial Stress > 3 (on a 1-5 scale): Suggest resources for financial planning or support.
- IF Sleep_Ordinal indicates insufficient sleep: Recommend sleep hygiene practices.
- Compile the applicable recommendations into a list.

Postcondition:

The user is presented with a list of personalized and actionable recommendations.

ID: SR3

User Requirement Mapping: UR3: Track Changes in My Risk Over Time

System Requirement: Display Risk History and Current Prediction.

Description:

The system will maintain a simple history of the single user's past depression risk probabilities. When the user requests a new prediction, the system will first display their recent risk history before showing the current prediction. This allows the user to see trends in their self-assessed risk over time.

Inputs & Source:

Inputs for current prediction: Same as SR1 (Age, Academic Pressure, etc.).

Inputs for history display: Data from a local text file (risk_history.txt) storing past prediction dates and probabilities.

Source: User input via CLI for current prediction; local text file for history.

Outputs & Destination:

Outputs:

- A list of past risk probabilities with timestamps (if available).
- The current predicted risk level (e.g., "HIGH RISK", "LOW RISK").
- The current probability of high risk.

Destination: Displayed sequentially on the Command Line Interface (CLI) screen.

Precondition:

- User must enter valid data for all required input fields for a new prediction.
- The system must have a trained Random Forest model loaded and accessible.
- The risk_history.txt file may or may not exist. If it doesn't, no history will be shown for the first prediction.

Algorithm/Process:

- When the user initiates a prediction:
- Attempt to read risk_history.txt.
- If the file exists and contains data, display the last few entries (e.g., date and probability) to the user.
- Receive raw simplified inputs from the user via the CLI for the current assessment (as per SR1).
- Internally transform these simplified inputs into the full numerical feature vector (as per SR1).
- Feed the feature vector to the trained Random Forest model.
- Model outputs a probability for the positive class.

- Convert probability to a risk category.
- Display the current predicted risk level and probability.
- Append the current prediction's date/timestamp and probability to risk_history.txt.

Postcondition:

- The user is shown their recent risk history (if any).
- The user is presented with their current predicted depression risk level and probability.
- The current prediction result (date, probability) is saved to the local history file for future reference.

4 B. Non-Functional Requirements:

ID	User Requirement (Description)	System Requirement (Verifiable)	Method for Verification
SR4	System Accuracy: Trust the system's predictions. (Maps to UR4)	The machine learning model (Random Forest) must achieve at least 80% accuracy in predicting depression risk when evaluated on a held-out, unseen test dataset.	Execute the C++ program's evaluation module on the test dataset. Calculate accuracy Ensuring the value is ≥ 0.80.
SR5	Response Time: Get results quickly. (Maps to UR5)	The system, when running via the CLI, must process user inputs for a single prediction and display the risk level and initial recommendations within 5 seconds of the user submitting the final input.	Conduct 10 timed test runs using the CLI. For each run, manually time from the moment the last piece of required input is entered by the user to when the prediction and recommendations are displayed on the screen. Average the times. Ensuring the average is ≤ 5 seconds.
SR6	Data Privacy: Know my personal data is safe. (Maps to UR6)	For the current CLI single-user version, no personal identifiable information (like name) is explicitly requested or stored beyond the session, other than the anonymized risk history. The risk history file (risk_history.txt) will contain only timestamps, probabilities, and risk levels, without direct user identifiers.	Review the C++ code to confirm no personally identifiable information is written to any persistent storage, except for the risk_history.txt which contains anonymized prediction outcomes. Inspect the contents of risk_history.txt to ensure it only contains the defined anonymous data points.

5. ARCHITECTURAL DESIGN DOCUMENTATION

5 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.

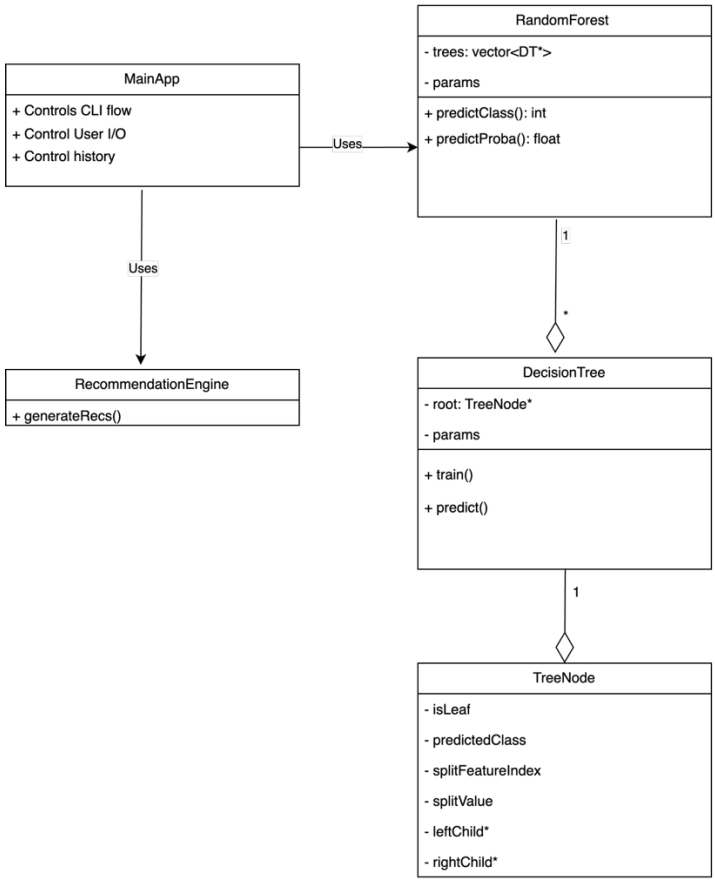
Data Preprocessing/Loading: Functions like load_numeric_csv and split_data.

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.

5 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).

6. STRUCTURAL MODELING USING OBJECT CLASSES

6 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.

6 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:

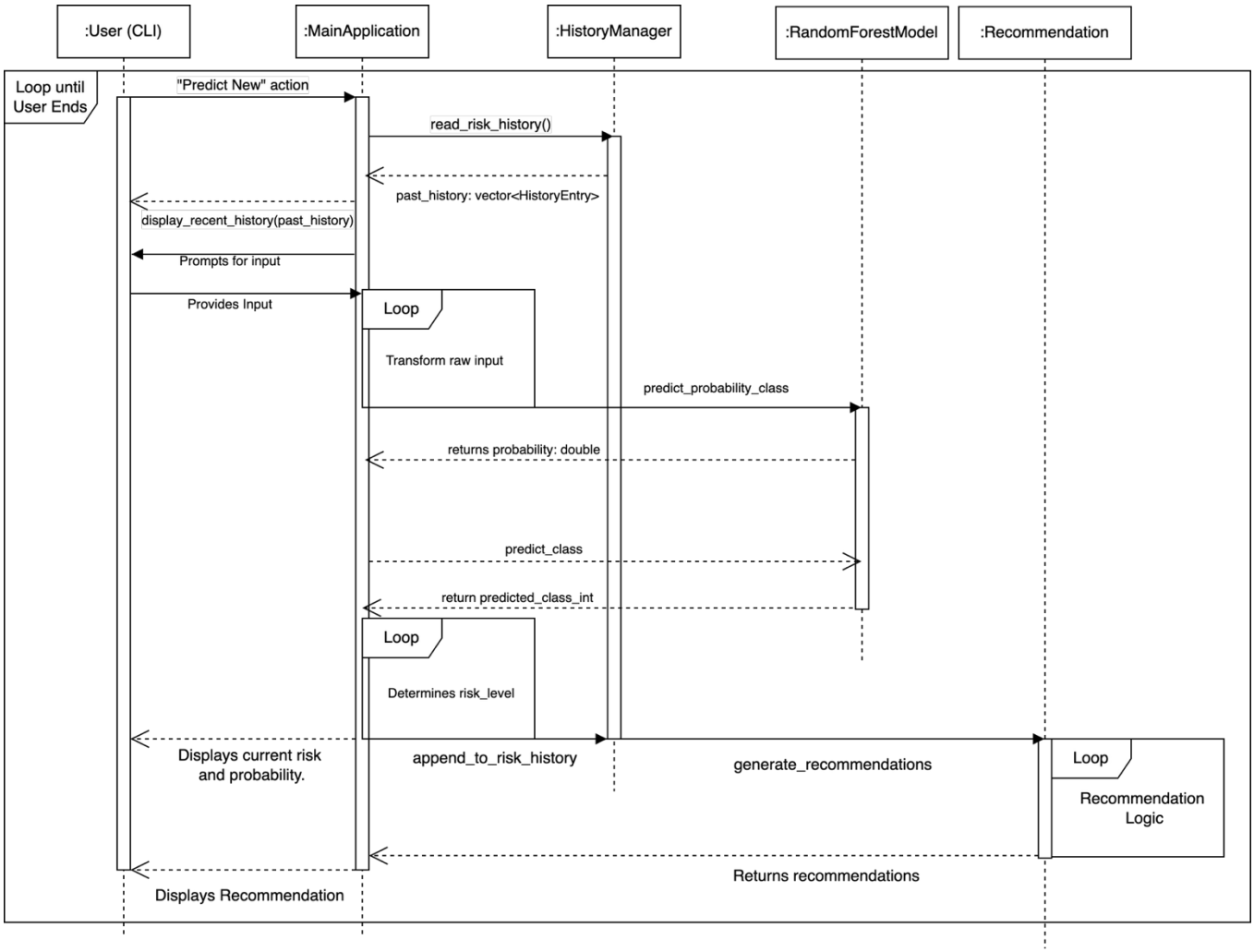
+ public

- private

protected

7. Interaction Modeling using Sequence Diagrams

7 A. Sequence Diagrams by Functional Requirement:



7 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.

8. Test Plan

8 A. DATASET DESCRIPTION

The primary dataset used for training, validating, and testing was compiled to capture various aspects of a student's life that could potentially correlate with mental well-being and depression.

Source: The dataset was a CSV file named “Student Depression Dataset”, sourced from Kaggle

url: https://www.kaggle.com/datasets/hopesb/student-depression-dataset

INITIAL SIZE: 27901 Rows with 18 attributes each

SIZE AFTER PREPROCESSING: 27851 Rows with 91 attributes each

TARGET VARIABLE: 'Depression', a binary indicator

KEY FEATURES: The dataset originally comprised features categorized as follows:

Demographics:

- id: Unique identifier (dropped during preprocessing).
- Gender: Male, Female.
- Age: Numerical age of the student.
- City: The city where the student resides/studies.

Academics:

- Profession: Student
- Degree: The student's current or highest completed academic degree.
- CGPA: Cumulative Grade Point Average.
- Academic Pressure: Self-reported level (e.g., 1-5).
- Study Satisfaction: Self-reported level (e.g., 1-5).
- Work/Study Hours: Average hours spent per day.

Well-being & Lifestyle:

- Work Pressure: (This column was found to have constant values and was dropped).
- Job Satisfaction: (This column was found to have constant values and was dropped).
- Sleep Duration: Categorical (e.g., "Less than 5 hours", "5-6 hours").
- Dietary Habits: Categorical (e.g., "Healthy", "Moderate", "Unhealthy").

Mental Health Indicators:

- Have you ever had suicidal thoughts?: Yes/No.
- Family History of Mental Illness: Yes/No.

Socioeconomic Factors:

- Financial Stress: Self-reported level (e.g., 1-5).

DATA QUALITY AND PREPROCESSING:

Initial Cleaning:

- Dropped uninformative columns
- Handled junk values identified in the city column
- Outlier Treatment

Feature Engineering:

- Suicidal_Thoughts, Family History of Mental Illness: Converted from Yes/No to binary (1/0).
- Sleep_Ordinal: Created from Sleep Duration categorical values.
- Total_Stress: Derived by summing Academic Pressure and Financial Stress.
- Region: The City column was mapped to broader geographical regions
- The Degree column was generalized into Degree_Level, Degree_Field, Degree_Type.

Encoding:

- Categorical features were one-hot encoded

8 B. Unit + Component Testing

UT = Unit Testing

CT = Component Testing

Test Case ID: UT_001

Class::Function:

DecisionTree::calculate_gini_impurity()

Input*:*

- labels = {0, 0, 1, 1}
- indices = {0, 1, 2, 3}

Scenario*:* Calculate Gini impurity for a perfectly balanced two-class node.

Expected Outcome: Gini Impurity = 1.0 - ((2/4)^2 + (2/4)^2) = 1.0 - (0.25 + 0.25) = 0.5

Assertion Method:

Check if the returned double is approximately equal to 0.5 (within a small epsilon).

Status*: PASS*

Test Case ID: UT_002

Class::Function: DecisionTree::calculate_gini_impurity()

Input:

- labels = {0, 0, 0, 0}
- indices = {0, 1, 2, 3}

Scenario: Calculate Gini impurity for a pure node (all class 0).

Expected Outcome: Gini Impurity = 1.0 - ((4/4)^2 + (0/4)^2) = 1.0 - 1.0 = 0.0

Assertion Method: Check if the returned double is approximately equal to 0.0.

Status*: PASS*

Test Case ID: UT_003

Class::Function: DecisionTree::majority_class()

Input:

- labels = {0, 1, 1, 0, 1}
- indices = {0, 1, 2, 3, 4}

Scenario: Find majority class in a mixed set.

Expected Outcome: 1

Assertion Method: Check if the returned int is 1.

Status*: PASS*

Test Case ID: UT_004

Class::Function: DecisionTree::find_best_split()

Input:

- features = {{1.0, 10.0}, {2.0, 20.0}, {3.0, 10.0}, {4.0, 20.0}}
- labels = {0, 1, 0, 1}
- current_indices = {0, 1, 2, 3}
- feature_subset_indices = {0, 1} (consider both features)
- params.min_samples_leaf = 1

Scenario: Simple 2-feature dataset where splitting on feature 1 at value 15.0 (or feature 0 at 2.5) should yield a good Gini gain.

Expected Outcome:

SplitInfo.gini_gain > 0. SplitInfo.feature_index and SplitInfo.split_value should correspond to the optimal split (e.g., feature 1, value around 15.0, or feature 0, value around 2.5). Left/Right indices should be correctly partitioned.

Assertion Method:

Verify best_split.gini_gain > 0. Manually calculate or inspect the expected optimal split and compare feature_index, split_value. Check sizes of left_indices and right_indices.

***Status**: PASS*

Test Case ID: CT_001

Class::Function: RandomForest::train() and RandomForest::predict_class()

Input:

- Use the same simple linearly separable dataset as UT_DT_006.
- rf_params.num_trees = 5
- rf_params.tree_params.max_depth = 2, rf_params.tree_params.min_samples_leaf = 1
- rf_params.bootstrap_sample_ratio = 1.0

Scenario: Train a small random forest and predict on training data.

Expected Outcome:

Predictions on training data should be highly accurate (likely 100% for this simple case, but bagging might introduce slight variations if a tree gets a very skewed bootstrap sample). Each tree should be trained.

Assertion Method:

Call train. Verify model.trees.size() is 5. For each sample in input features, call predict_class and compare to the true label. Aim for high accuracy.

***Status**: PASS*

Test Case ID: CT_002

Class::Function: RandomForest::predict_probability_class1()

Input:

- Train with UT_RF_001.
- sample = {1,1} (belongs to class 0)
- sample = {2,2} (belongs to class 1)

Scenario: Check probability output.

Expected Outcome:

For {1,1}, probability of class 1 should be low (e.g., <= 0.2, ideally 0.0 if all trees agree). For {2,2}, probability of class 1 should be high (e.g., >= 0.8, ideally 1.0).

Assertion Method:

Call predict_probability_class1 and check if the returned double is within the expected range.

Status*: PASS*

8 C. System-wise Testing (Functional Requirements)

These tests evaluate the end-to-end functionality based on the CLI interaction.

Test Case ID: SYS_001

Input:

User inputs via CLI leading to a known "High Risk" prediction (e.g., low sleep, high academic pressure, high financial stress, suicidal thoughts=YES, low CGPA, etc. - define a specific set of values).

- Age: 20
- Academic Pressure: 5
- CGPA: 5.5
- Study Satisfaction: 1
- Suicidal Thoughts: 1 (Yes)
- Work/Study Hours: 10
- Financial Stress: 5
- Family History: 1 (Yes)
- Sleep Ordinal: 0 (<5 hours)
- Gender: 1 (Male)
- Degree Type: 0 (Science/Tech)

Scenario: User enters data indicating high likelihood of depression. This is the first run (no history).

Expected Outcome:

- CLI displays "No past prediction history found."
- CLI displays "Predicted Risk: HIGH (Likely Depressed)".
- CLI displays a probability of High Risk > 0.5
- CLI displays recommendations appropriate for high risk, including urgent advice for suicidal thoughts, and tips related to high academic/financial stress, low sleep.
- risk_history.txt is created/updated with one entry (timestamp, probability, "HIGH").

Assertion Method: Observe CLI output. Inspect risk_history.txt content.

Status: PASS

Test Case ID: SYS_002

Input:

Precondition: risk_history.txt contains the entry from SYS_FN_001.

User inputs via CLI leading to a known "Low Risk" prediction (e.g., good sleep, low academic/financial stress, no suicidal thoughts, high CGPA, etc.).

- Age: 22
- Academic Pressure: 1
- CGPA: 9.0
- Study Satisfaction: 5
- Suicidal Thoughts: 0 (No)
- Work/Study Hours: 6
- Financial Stress: 1
- Family History: 0 (No)
- Sleep Ordinal: 2 (7-8 hours)
- Gender: 0 (Female)
- Degree Type: 1 (Non-Science/Arts)

Scenario: User enters data indicating low likelihood of depression. History exists.

Expected Outcome:

- CLI displays the previous "HIGH" risk entry from history.

- CLI displays "Predicted Risk: LOW (Likely Not Depressed)".
- CLI displays a probability of High Risk < 0.5 (e.g., < 0.3 based on input).
- CLI displays recommendations appropriate for low risk (e.g., maintain habits).
- risk_history.txt is updated with a new second entry (timestamp, probability, "LOW").

Assertion Method: Observe CLI output. Inspect risk_history.txt content (should now have two entries).

Status: Pass

Test Case ID: SYS_003

Input:

Test with values at the boundaries and within typical ranges for each simplified CLI input field (e.g., Age=17, Age=40; Academic Pressure=1, Academic Pressure=3, Academic Pressure=5; CGPA=0, CGPA=6.9, CGPA=7.1, CGPA=10).

Scenario: Test various combinations of inputs to see how predictions and recommendations change.

Expected Outcome:

- System handles all valid inputs within defined ranges without crashing.
- Predictions and recommendations appear logical and consistent with the input changes (e.g., increasing multiple stressors should generally increase risk probability).

Assertion Method: Observe CLI output for sensible predictions and recommendations. Ensure no crashes or exceptions for valid inputs.

Status: Pass

8 D. Acceptance Testing for Non-Functional Requirements

ID	System Requirement (Verifiable)	Verification Criteria & Method	Status
SR4	The machine learning modelmust achieve at least 80% accuracy in predicting depression risk when evaluated on a held-out, unseen test dataset.	Criteria: Test set accuracy ≥ 80.0%. Method: After training the model on the training split, run the C++ program's evaluation part on the remaining 20% test split.	PASS
SR5	The system, when running via the CLI, must process user inputs for a single prediction and display the risk level and initial recommendations within 5 seconds of the user submitting the final input.	Criteria: Average response time ≤ 5.0 seconds. Method: Perform 10 interactive prediction sessions via the CLI. For each session, use a stopwatch to measure the time from hitting "Enter" after the last required input field to the moment the prediction and first line of recommendations appear. Calculate the average time.	PASS
SR6	For the current CLI single-user version, no personal identifiable information (like name) is explicitly requested or stored beyond the session, other than the anonymized risk history. The risk history file (risk_history.txt) will contain only timestamps, probabilities, and risk levels, without direct user identifiers.	Criteria: risk_history.txt contains only timestamp, probability, and risk category string. No other user-identifying data is persistently stored by the application. Method: 1. Run several prediction sessions. 2. Manually inspect the contents of risk_history.txt. 3. Review the C++ source code to ensure no other files are written with user-specific session data (beyond the feature values passed to the model in memory).	PASS

9. Programming Environment

9 A. Programming Language:

C++ (Standard: C++17):

The core application, including data loading, preprocessing, the Random Forest machine learning model (decision trees, ensemble logic), prediction, recommendation engine, and the command-line interface (CLI), is implemented in C++. C++17 was chosen for its modern features, performance capabilities, and control over memory management, suitable for building a computational application from fundamental components.

9 B. External Software Packages/Libraries and Their Roles:

Standard C++ Library: Extensively used for core functionalities:

iostream: For console input and output (CLI interactions).

vector: For dynamic arrays to store datasets, features, labels, and collections of objects string,

sstream: For string manipulation and parsing CSV data.

fstream: For file input/output (loading the dataset, reading/writing history).

algorithm: For functions like std::sort, std::shuffle, std::min, std::max, std::iota, std::transform.

random: For random number generation

map: For counting class occurrences

cmath: For mathematical functions like std::sqrt.

limits: For std::numeric_limits

iomanip: For output formatting

chrono, ctime: For generating timestamps for the prediction history.

(No External Machine Learning Libraries for Core RF): Notably, for the Random Forest algorithm itself (decision tree construction, splitting logic, ensemble management), no external pre-built machine learning libraries were used in this C++ phase. The decision tree and random forest logic were implemented from scratch as per the project's C++ focus.

10. User's Guide

This guide demonstrates how a user interacts with the Command Line Interface (CLI) of the Behavioral Optimization & Mental Wellness System to achieve their requirements.

Pre-requisite:

The C++ application (depression_classification) has been compiled and is ready to run in a terminal/console environment. **The cleaned_student_data.csv file must be in the same directory as the executable, or its path correctly specified in the C++ code.**

10 A. Execution Steps:

Step 1: Compile the main.cpp file to get the C++ application

Navigate to you project dir (can be done by running the following command)

```
git clone https://github.com/DiwBhat/Depression-Classification.git
```

Then cd into the cpp_port derictory

```
cd cpp_port
```

Run the following command to get the application

```
g++ -std=c++17 -Wall -Wextra -O2 -o depression_classification main.cpp
```

Step 2: Run the C++ application

Run the following command to run the application

```
./depression_classification
```

10 B. Perform Unit Test

In the same directory as the C++ application mentioned above, run the following code in the terminal to get the application for unit test

```
g++ -std=c++11 unit_test.cpp -o test
```

10 C. Screenshot of a sample workflow and Unit test

```
❯ ./student_rf_scratch
Student Depression Risk Prediction (Random Forest from Scratch)
------------------------------------------------------------
Data loaded: 27851 samples, 90 features.
Training set: 22280, Test set: 5571

Training Random Forest model...
Training Random Forest with 50 trees.
Trained tree 10/50
Trained tree 20/50
Trained tree 30/50
Trained tree 40/50
Trained tree 50/50
Random Forest training complete.

--- Evaluating Model on Test Set ---
Test Accuracy: 83.7013%

Confusion Matrix (Test Set):
          Predicted 0   Predicted 1
Actual 0       1722           589
Actual 1        319          2941


--- Interactive Prediction (Single User History) ---


Make a new prediction? (yes/no or quit): yes
No past prediction history found.
```

```
Please enter student details (type 'quit' at any prompt):
Age (17-70): 22
Academic Pressure (1-5): 2
CGPA (0-10): 8
Study Satisfaction (1-5): 4
Suicidal Thoughts (0=No, 1=Yes): 0
Work/Study Hours per day (0-24): 8
Financial Stress (1-5): 3
Family History of Mental Illness (0=No, 1=Yes): 0
Sleep Ordinal (0:<5h, 1:5-6h, 2:7-8h, 3:>8h): 0
Gender (0=Female/Other, 1=Male): 1
Degree Type (0=Sci/Tech, 1=Non-Sci/Arts/Biz/Law): 0

--- Current Prediction Result ---
Predicted Risk: LOW (Likely Not Depressed)
Probability of High Risk (Depression): 0.100
Current prediction saved to history.

--- Recommendations ---
- **Continue to prioritize your well-being!**

**Regarding Sleep:**
  - Aim for a consistent schedule.
  - Create a relaxing bedtime routine.
```

```
Make a new prediction? (yes/no or quit): yes

--- Recent Prediction History (Last 5) ---
2025-05-11 15:39:00 - Risk: LOW (Probability of High Risk: 0.100)
-------------------------------------

Please enter student details (type 'quit' at any prompt):
Age (17-70): 22
Academic Pressure (1-5): 4
CGPA (0-10): 8
Study Satisfaction (1-5): 4
Suicidal Thoughts (0=No, 1=Yes): 1
Work/Study Hours per day (0-24): 10
Financial Stress (1-5): 3
Family History of Mental Illness (0=No, 1=Yes): 0
Sleep Ordinal (0:<5h, 1:5-6h, 2:7-8h, 3:>8h): 1
Gender (0=Female/Other, 1=Male): 1
Degree Type (0=Sci/Tech, 1=Non-Sci/Arts/Biz/Law): 0

--- Current Prediction Result ---
Predicted Risk: HIGH (Likely Depressed)
Probability of High Risk (Depression): 0.960
Current prediction saved to history.
```

```
--- Recommendations ---
- **It's important to reach out:** Consider speaking with a counselor or therapist.

**Regarding Academic Pressure (4/5):**
  - Break tasks down.
  - Practice time management.
**Regarding Work/Study Hours (10h/day):
  - Evaluate sustainability.
  - Schedule rest.

**IMPORTANT: Suicidal Thoughts:**
  - **Please reach out immediately for help (e.g., crisis hotline 988 in USA, counseling services).**
**Regarding Sleep:**
  - Aim for a consistent schedule.
  - Create a relaxing bedtime routine.

Make a new prediction? (yes/no or quit): no

Exiting program.
```

Unit Test

```
❯ g++ -std=c++11 unit_test.cpp -o test
❯ ./test
--- Running Unit Tests ---
Running UT_001_Gini_Balanced... PASS
Running UT_002_Gini_Pure... PASS
Running UT_003_Majority_Mixed... PASS
Running UT_004_FindBestSplit_Simple... PASS

--- Test Summary ---
Total Tests Run: 4
Tests Passed:    4
Tests Failed:    0

ALL TESTS PASSED!
```

11. Summary of Work Planned for Phase I

11 A. Complete Work (Phase I):

- Data Loading & Basic Preprocessing: C++ module to load the data.
- Decision Tree Implementation: Core logic for a decision tree classifier built from scratch, including Gini impurity calculation, best split finding, and recursive tree construction with parameters for max depth and min samples per leaf.
- Random Forest Implementation: Ensemble logic built from scratch, managing multiple decision trees, and feature subsampling.
- Model Training: Functionality to train the Random Forest model.
- Prediction Logic: Class prediction based on majority voting from trees in the forest.
- Probability estimation for the positive class based on the proportion of tree votes.
- Basic Evaluation: Calculation and display of test set accuracy and a confusion matrix.
- Simplified CLI: A command-line interface for user interaction.
- Prompts for simplified, more intuitive user inputs.
- Displays prediction results (risk category and probability).
- Rule-Based Recommendation Engine: Generates basic recommendations based on the predicted risk level and specific user input values
- Single-User History Tracking: Saves prediction timestamps, probabilities, and risk levels to a local text file (risk_history.txt) and displays recent history to the user.

11 B. Incomplete Work for Future Improvement (Beyond Phase I Scope):

- Advanced Hyperparameter Tuning in C++: While the Random Forest has tunable parameters, a systematic hyperparameter optimization framework (like GridSearchCV or Bayesian Optimization) was not implemented in C++. Parameters are currently set manually.
- Robust Input Validation in CLI: The CLI input validation is basic. More comprehensive checks for data types, ranges, and formats could be added.
- Advanced Feature Importance Calculation: A from-scratch implementation for calculating feature importance specific to this Random Forest was not developed.
- Cross-Validation within C++ Training: The current C++ training trains on the whole training split. Implementing k-fold cross-validation within the C++ training loop for more robust model parameter assessment was not part of this phase.
- More Sophisticated Recommendation Engine: The rule-based engine is simple; it could be expanded with more nuanced rules or draw from a larger knowledge base.
- Error Handling and Logging: While some basic error handling exists, a more comprehensive error handling and logging mechanism could be implemented.
- No GUI: The interface is CLI-only.

12. Planning for Phase II

Phase II could focus on enhancements and addressing some of the incomplete areas:

- GUI Development
- Enhanced Recommendation System
- Model Iteration and Improvement using Advanced Hyperparameter Tuning
- Feature Engineering & Selection
- Feedback Mechanism

13. Location for GitHub repository for code sharing

The repo can be accessed via the following url:

https://github.com/DiwBhat/Depression-Classification/cpp_port

Both the classifier and the unit test files are on the same directory that is obtained by following the above link

The classifier file name is **main.cpp**

The unit test file name is **unti_test.cpp**