# CUSTOMER CHURN SYSTEM

Group: 2

Members:

- Fahim Tanvir
- Abul Hassan
- Ahmed Ali

Table of contents

## Introduction:

**Purpose:**

This project aims to develop a model that predicts customer churn for a telecom company. Customer churn refers to when the customer is willing to leave the company by canceling their services or subscription. This prediction model will be trained on historical customer data to measure customer loyalty. By identifying patterns in the customers behavior, the company can take steps to retain customers that have a chance of leaving and improve the business model.

**Problem Statement:**

The telecom company is currently experiencing high rates of customer turnover, leading to significant revenue loss. Retaining existing customers can be cheaper for the company than attracting new ones and also implementing changes within business depending on what sections get churned the most. Essentially, churn prediction is critical for the company's profitability. By implementing this system, the company will be able to take proactive measures to prevent churn.

After making the prediction, the system will analyze the data of customers who left. It will then provide suggestions to help change the behavior of at-risk customers, encouraging them to act more like their peers who stayed. It will also suggest to the company on what changes or improvements can be made using data of customers who churned and looking for correlations(for example, if a customer that churned is known for having a large monthly bill, changes within company systems will be made to allow extended payment plan or even discounts for non-churned customers).

This is the dataset we will be using:

https://www.kaggle.com/datasets/blastchar/telco-customer-churn

## Glossary:

- **Churn/Customer Churn**:
  The process in which the customer chooses to stop using the company's services. Basically the likelihood to terminate their subscriptions for the company becomes very high.
- **Retention**:
  Efforts that the company makes to keep existing customers from leaving by addressing their needs or providing incentives to stay.
- **Prediction model:**
  A machine learning algorithm designed to predict whether the customer will stay or not.
- **Customer Turnover**:
  The rate at which customers leave the company, measured over a specific period.
- **Behavioral Patterns**:
  Common characteristics are found in the (data) actions and decisions of customers, which we use in our algorithm to make the predictions.
- **Loyal Peers/Regulars**:
  A group of customers with similar characteristics or behavior patterns with the churned customer, but still with the company. We are trying to make the other customers behave like them.
- **Proactive Measures**:
  Steps taken by the company to prevent customer churn before it happens, based on the model's predictions.

**User requirements:**

### Predicting Customer Churn

- **Type**: Functional
  **Rationale**:
  The primary goal of this project is to predict whether a customer will leave the company or not. Accurate churn predictions allow the company to proactively address customer concerns, improve satisfaction, and retain valuable customers.

### Training the System for Customer Prediction

**Type**: Functional:
**Rationale:**

Before the system can make accurate predictions, it needs to be trained. Training involves teaching the system to recognize patterns in historical customer data. Imagine showing examples to a system, like teaching it what "high-risk" and "loyal" customers look like based on their past behaviors and characteristics. The Random Forest model learns from these examples so that when it encounters new customers, it can make informed decisions about whether they are at risk of leaving. This training step is essential to ensure the system works correctly and provides reliable predictions.

- **The system should have a high level of prediction accuracy. (system quality)**
  - **Type:** Non- Functional
  - **Rationale:** The system should be accurate, so it will be helpful to the company for knowing the targeted customers. If this isn't met, the system might fail as it won't recognize customers who are regulars or ones that will churn eventually.

- **The System must have the capability to handle large datasets**
  - **Type:** Non-Functional
  - **Rationale:** The system must be efficient enough to manage large datasets, especially ones in the telecom sector, supporting up to 16,000 customer records without performance issues.

- **The system must load and perform efficiently**
    - **Type:** Non-Functional
    - **Rationale:** Telecom companies typically deal with massive amounts of customer data. The system must be optimized for performance to handle this data efficiently without any mistakes or inefficiencies. If the system slows down or takes too long(more than 3 minutes), that is no good.

- **The system identifies at-risk customers and provides tailored recommendation plans**

- **Type**: Functional
  **Rationale**:
  This functionality is essential for enhancing customer retention by proactively addressing the needs of at-risk customers through customized recommendations, thereby improving overall satisfaction and loyalty.

## System Requirements:

**Functional requirements:**

**Functional Requirement #1: Churn Prediction**

**Description:** The system should **predict** whether the customer is leaving the company or not.

**Inputs:**

- **Customer data:** gender , SeniorCitizen, Partner, Dependents, tenure, PhoneService, MultipleLines

  OnlineSecurity , OnlineBackup, DeviceProtection, TechSupport, StreamingTV StreamingMovies, PaperlessBilling, MonthlyCharges, TotalCharges, Churn (Target)

**Source:**

- Customer database for a telecom company (e.g., Kaggle dataset).

**Outputs:**

- **Prediction Result**: Yes/No for churn.

**Destination:**

- Display on-screen.
- Export results to the dashboard for company managers.

**Precondition:**

- Customer data must be available in the dataset.

**Algorithm:**

The Random Forest model combines predictions from multiple decision trees to generate the final result. Each tree looks at different subsets of the customer data and provides an individual prediction. The final decision is based on majority voting, ensuring robust and accurate predictions

**Algorithm Explanation:**

1. Builds multiple decision trees using different subsets of data.
2. Combines the outputs of all the trees using majority voting (for classification).
3. Improves prediction accuracy and reduces overfitting through randomization.

**Working:**

1. **Input Data:**
    - Accept customer data as input.
2. **Prediction by Trees:**
    - Pass the input data through all trained decision trees.
    - Each tree makes an independent prediction (Yes/No).
3. **Combine Predictions:**
    - Use majority voting to determine the final prediction.

**Parameters for Random Forest:**

- **Number of trees (nTrees):** Default set to 10 but adjustable based on the dataset size.
- **Maximum tree depth (maxDepth):** Default set to 5 to avoid overfitting.
- **Minimum samples to split a node (minSamplesSplit):** Default set to 2.

**Functional Requirement #2: Usage of Random Forest**

**Description:**
The system should **train** a Random Forest model using historical customer data to predict churn.

**Inputs:**

- **Customer data:** gender ,SeniorCitizen, Partner, Dependents, tenure, PhoneService, MultipleLines OnlineSecurity , OnlineBackup, DeviceProtection, TechSupport, StreamingTV StreamingMovies, PaperlessBilling, MonthlyCharges, TotalCharges, Churn (Target)

**Source:**
Customer database for a telecom company (e.g., Kaggle dataset).

**Outputs:**
A trained Random Forest model.

**Destination:**

The trained model is stored in memory or saved to disk for future predictions.

**Precondition:**

Customer data must be preprocessed to handle missing values and validate its structure.

**Algorithm:** The Random Forest algorithm employs an ensemble approach by training multiple decision trees on varied subsets of the dataset.

**Algorithm Explanation:**

1. **Bootstrap Sampling**: Each decision tree is trained independently using a bootstrapped subset of the customer data, allowing each tree to learn unique patterns.
2. **Random Feature Selection**: At each decision node, the trees randomly select a subset of features to consider for splitting, enhancing diversity among the trees and reducing the risk of overfitting.
3. **Ensemble Prediction**: After training, the predictions from all individual trees are aggregated (usually by majority voting) to form a final, more accurate and robust prediction compared to any single tree.

**Working:**

Split the Dataset:

Divide the customer data into training (70%) and test (30%) datasets.

Then use the trained data for the random forest training

**Bootstrap Sampling:**

Randomly sample the training data (with replacement) to create unique subsets for each tree.

**Train Each Tree:**

Build each decision tree using its bootstrapped subset.

Randomly select a subset of features at each split to diversify the trees.

Continue splitting until stopping criteria are met (e.g., maxDepth or minSamplesSplit).

**Combine Trees:**

Add all trained trees to the Random Forest model.

**Parameters for Training Random Forest:**

- **Number of trees (nTrees):** Default set to 10 but adjustable based on dataset size.
- **Maximum tree depth (maxDepth):** Default set to 5 to avoid overfitting.

- **Minimum samples to split a node (minSamplesSplit):** Default set to 2 to ensure sufficient data in each node.

**Postcondition:**

- The Random Forest model is successfully trained and ready to predict customer churn.

**Postcondition:**

- A churn prediction result is generated and displayed.

**Functional Requirement #3: Proactive Retention System**

**Description:**

The system identifies high-risk customers and provides actionable retention strategies to improve customer loyalty and reduce churn.

**Inputs:**

High-risk customers identified by the Random Forest model.

**Source:**

Outputs from the Random Forest model (categorization of high-risk customers).

**Outputs:**

1. Detailed reports outlining behavioral similarities between high-risk and loyal customers.
2. Actionable retention strategies, such as personalized offers, targeted communication plans, and customer engagement activities.

**Destination:**

1. Reports are forwarded to the customer retention recommendation system.
2. Retention strategies are integrated into customer relationship management systems and shared with customer service teams.

**Precondition:**

1. High-risk customers must be identified by the Random Forest model.
2. The system must receive the customer data and give it a recommendation based on the customer's need.

**Algorithm:**

The system operates in two phases:

**Phase 1: Unloyal Customer Identification**

**Random Forest Analysis:**

- user input to predict customer churn using a Random Forest model. It collects customer attributes from a GUI, converts them into a numerical format, and stores them in an array. The attributes include gender, senior citizen status, and various service options, which are transformed into binary or numeric values as required. After preparing the data, the algorithm feeds it into the pre-trained Random Forest model to predict whether the customer is likely to churn. The result is displayed, and based on the prediction, specific retention strategies are suggested, tailored to the customer's profile.

## Phase 2: Retention Strategy Formulation

integrates insights from external analysis of customer data to identify key attributes that influence customer decisions to leave. This analysis helps pinpoint the most impactful factors such as service usage, billing practices, and customer support interactions. Based on these insights, the function offers customized retention strategies to address specific vulnerabilities and enhance satisfaction. For example, it suggests tailored plans for seniors, enhanced support for frequent help-seekers, and financial flexibility for those with high charges, all aimed at mitigating churn by aligning services more closely with customer needs identified through the analysis.

**Postcondition:**

1. Similarity reports and actionable retention strategies are developed.
2. Strategies are deployed to customer relationship management systems and shared with customer service teams for execution.

**Non-Functional Requirements**:

- **Non-Functional Requirement #1: Prediction Accuracy**

  **Description:** The system should have a high level of prediction accuracy (around 80%)
  Accuracy testing will be performed using cross-validation and testing with historical datasets.
  **Verification Method:** The system's prediction accuracy will be thoroughly set up using a dual approach. First one is cross-validation, which involves partitioning the data into multiple subsets to train and test the model across different segments. Second is by testing with historical or preceding datasets to see how well the model generalizes to new and previous conditions. This evaluation ensures the model is consistently accurate.

- **Non-Functional Requirement #2**: **Dataset Capacity**

  **Description:** The system should load large datasets at least 7,000 and be able to take the data of 16,000 customers
  **Verification Method:** To confirm the system's capacity to efficiently handle large datasets, load testing will be executed using datasets of increasing sizes, up to the maximum specified capacity of 16,000 customer records.

- **Non-Functional Requirement #3: Runtime Efficiency**

  **Description:** The system should be able to efficiently process predictions with a fast runtime, specifically aiming for under 5 seconds per prediction cycle
  **Verification Method:** The system's performance will be validated through runtime testing, where the time taken to process predictions on datasets customers is measured. This testing will be carried out using a benchmark dataset processed repeatedly to assess runtime consistency across multiple cycles. The system must consistently achieve a prediction runtime of under 5 seconds per cycle to meet our performance criteria.

## Architectural Design:

Major design considerations and rationales:

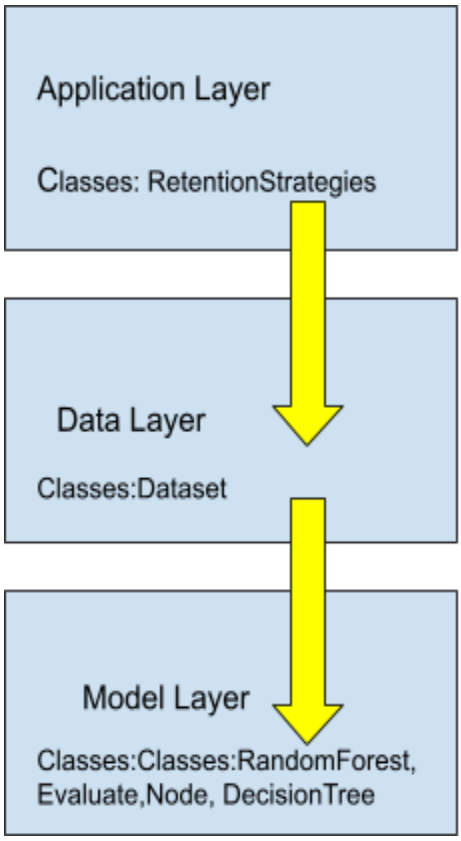
We need a design that can carefully and thoroughly traverse through customer data to predict how much at risk they are from churning from the company. High-risk and low-risk customers will be compared to monitor which sectors within the company need improvement. Missing data that will bridge the gap between information will be solved through imputation for numerical data and similarity analysis. Lastly, retentional strategies and customer churn risk as a final output.

With these in mind, we need a design that is thorough and good for maintenance. It is ideal for the system to be separated by functionality and purpose.

Architectural organization

The best choice is a layered architecture model. This was chosen as it will help keep our data organized and concise to ease our algorithm. Dependability of our system is maintained as each function is separated into one layer relevant to its task.

Architectural Diagram




This is where our human interaction starts and ends. Our basic UI and user inputs come from here

Where our data will be undergo predictional and unit testing along with having missing data accounted for. Final evaluation is sent to Application Layer

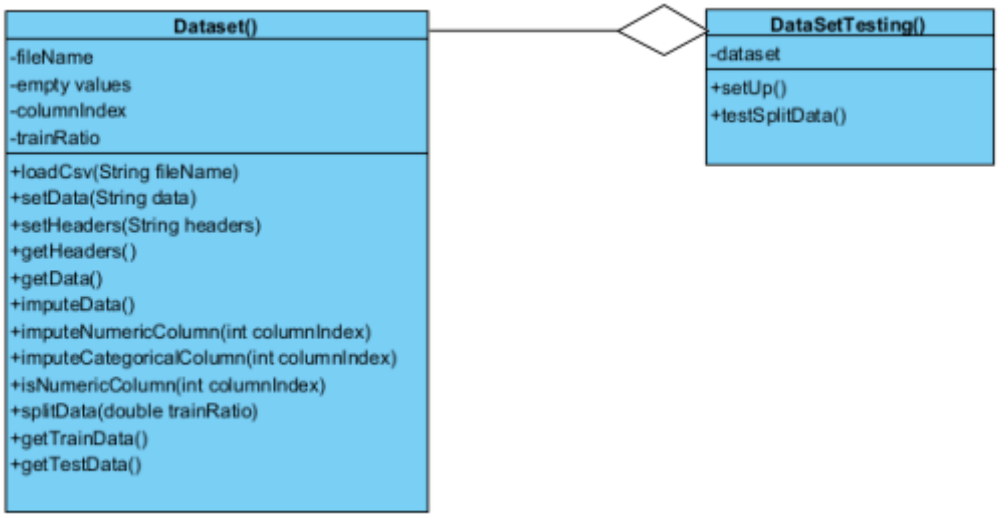
This is where all of our data for testing(specifically test cases), training(for Random Forest of course) and validation is stored.

---

## Structural Modelling:

These are of our classes and functions defined based on layer:

1. **Dataset Layer:**

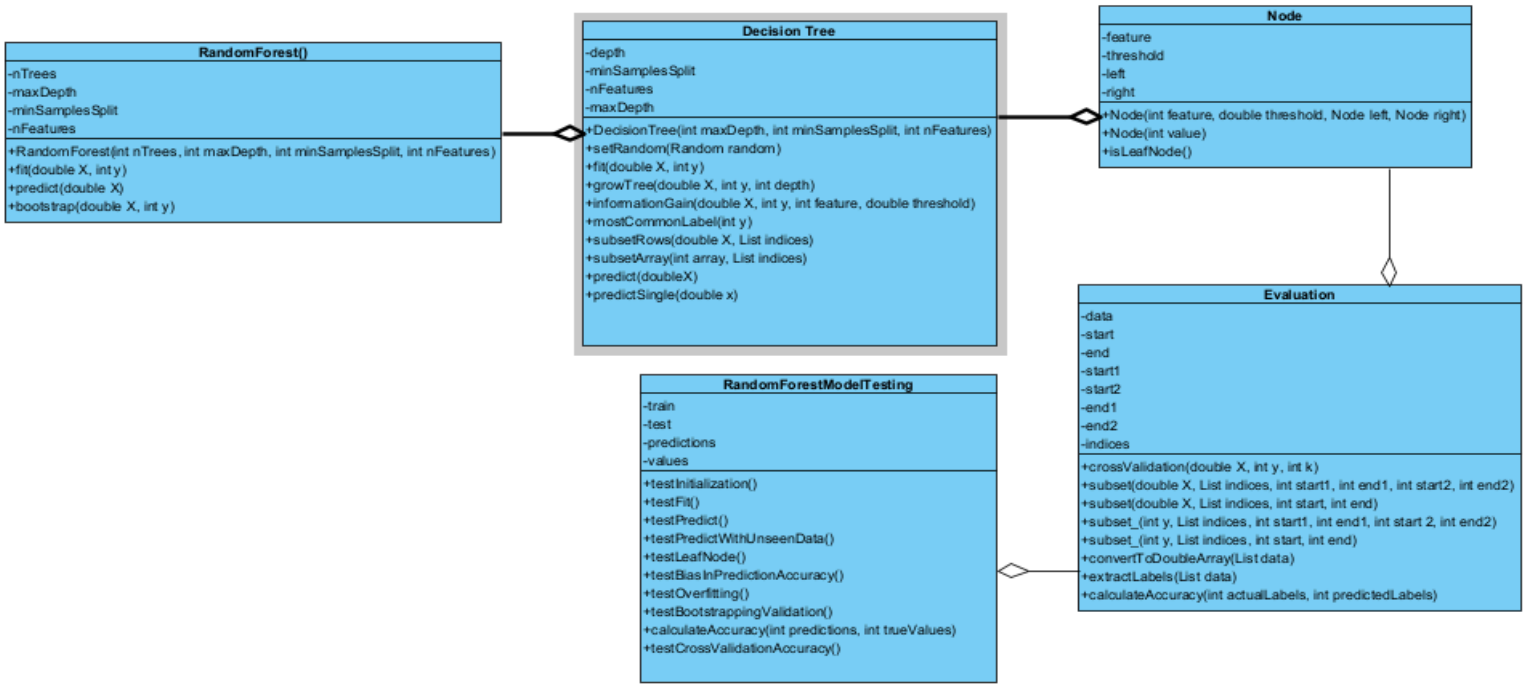


- Dataset: This class handles loading, splitting, and imputing data to prepare it for training and evaluation.

  Functions:

  - **loadCsv(String fileName):** Reads a CSV file and loads its content into the data list (a list of lists where each inner list represents a row of data) and headers (a list of column names).
  - **setData(List<List<String>> data)**: Sets the data (list of rows in our CSV file) for the dataset.
  - **setHeaders(List<String> headers)**: Sets the headers (list of column names) for the dataset.
  - **getHeaders():** Returns the list of column headers.
  - **getData():** Returns the entire dataset (list of rows specifically).
  - **imputeData()**: Handles missing data (empty values) in the dataset. For numeric columns, it replaces missing values with the mean of the column. For categorical columns, it replaces missing values with the mode (most frequent value) of the column.
  - **imputeNumericColumn(int columnIndex)**: Method imputes missing numeric values in a specific column by replacing them with the mean of the column.
  - **imputeCategoricalColumn(int columnIndex)**: Method imputes missing categorical values in a specific column by replacing them with the most frequent value (mode) of the column.
  - **isNumericColumn(int columnIndex)**: Checks whether a given column contains numeric data by attempting to parse the values as doubles.
  - **splitData(double trainRatio)**: Splits the dataset into training and testing  data is shuffled before splitting, and the training and testing data are saved to separate CSV files.
  - **getTrainData():** Returns the training data after the split.
  - **getTestData()**: Returns the testing data after the split.
- DatatsetTesting: This class prepares our datasets for unit testing.
  - **setUp()**: Initializes the dataSet object and prepares headers and data for testing. It creates a list of 100 rows, each containing values "1", "2", and "3", and sets it as the data for the dataset.
  - **testSplitData()**: Tests the splitData() method by splitting the dataset with a 70% training and 30% testing ratio. It then asserts that the training data size is 70 rows and the testing data size is 30 rows.
  - **testImputation()**: Tests the imputeData() method by preparing data with missing values. It sets headers for a numeric column, categorical column, and another numeric column. Then it imputes missing values.

## 2. Model Layer:



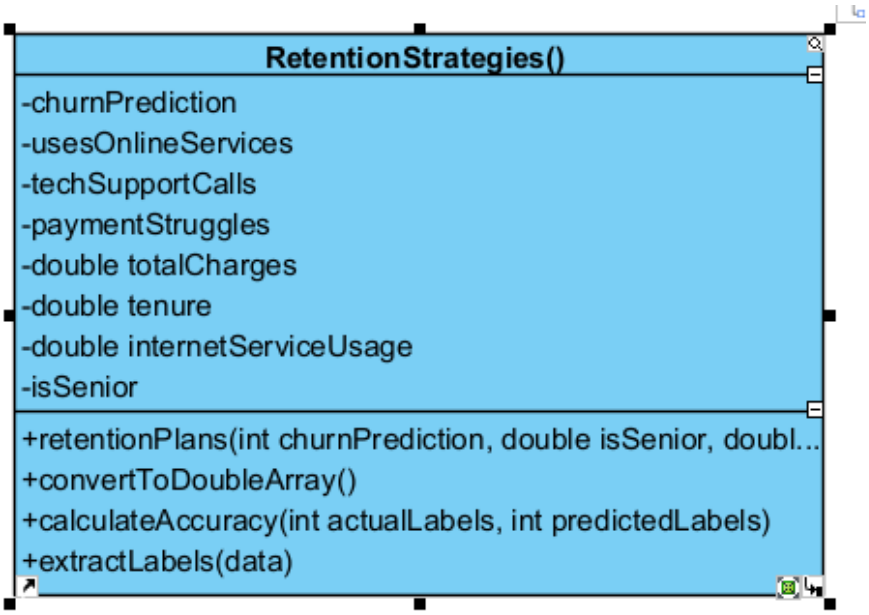

- Random Forest: As the name suggests, this class implements the Random Forest model.
    - **RandomForest(int nTrees, int maxDepth, int minSamplesSplit, int nFeatures)**: Constructor initializes the random forest with the number of trees, maximum depth, minimum samples required for a split, and the number of features to consider during tree splits.
    - **fit(double[][] X, int[] y)**: Trains the random forest by creating decision trees. Each tree is trained on a bootstrapped dataset (randomly sampled with replacement) from the input data X and corresponding labels y.
    - **predict(double[][] X)**: Predicts the labels for the dataset X by aggregating predictions from all decision trees using a majority voting mechanism.
    - **bootstrap(double[][] X, int[] y)**: Creates a bootstrapped dataset by randomly sampling rows (by replacing it) from the input dataset X and corresponding labels y. Returns the bootstrapped data as a map containing X and y.
- Decision Tree: This class is dedicated for the decision trees or splits created by Random Forest.
    - **DecisionTree(int maxDepth, int minSamplesSplit, int nFeatures)**: Constructor initializes the decision tree with the maximum depth, minimum samples required for a split, and the number of features to consider when splitting.
    - **setRandom(Random random)**: Sets the random number generator used for selecting features during tree growth.

    - **fit(double[][] X, int[] y)**: Trains the decision tree by building its structure based on the input dataset X and corresponding labels y.
    - **growTree(double[][] X, int[] y, int depth)**: Recursively builds the decision tree by splitting the dataset and assigning child nodes until reaching maximum depth or minimum sample size.
    - **informationGain(double[][] X, int[] y, int feature, double threshold)**: Calculates the information gain for splitting data on a specific feature at a given threshold.
    - **entropy(int[] y)**: Computes the entropy of an array of labels to measure impurity.
    - **mostCommonLabel(int[] y)**: Determines the most frequently occurring label in an array, used for leaf node predictions.
    - **subsetRows(double[][] X, List<Integer> indices)**: Extracts subsets of rows from the dataset X based on a list of indices.
    - **subsetArray(int[] array, List<Integer> indices)**: Extracts subsets of an array (e.g., labels y) based on a list of indices.
    - **predict(double[][] X)**: Predicts the labels for a dataset X by traversing the decision tree for each instance.
    - **predictSingle(double[] x)**: Predicts the label for a single instance by navigating the decision tree from the root to a leaf node.
- Node: This class is dedicated for the internal nodes or leaves of the decision tree(where each tree ends at and where our Random Forest is pathed).
    - **Node(int feature, double threshold, Node left, Node right)**: Constructor for internal nodes of the decision tree. It initializes the node with the feature to split on (feature), the threshold value for the split (threshold), and the left and right child nodes (left and right). The value is set to null because this is not a leaf node.
    - **Node(int value)**: Constructor for leaf nodes of the decision tree. It initializes the node with a specific label (value) and sets the feature to -1 and threshold to Double.NaN to indicate that this is a leaf. The left and right child nodes are set to null as leaf nodes do not have children.
    - **isLeafNode()**: A helper function that checks if the node is a leaf node. It returns true if the node has a value (it would have a leaf), and false if it is an internal node (if the value is null).
- Evaluation: This class performs calculations and accuracy testing for Random Forest mode.
    - **crossValidation(double[][] X, int[] y, int k)**:Performs k-fold cross-validation on the dataset X with labels y. The data is shuffled and divided into k folds, training and validating the model on different subsets of the data. Returns the average accuracy across all folds.
    - **subset(double[][] X, List<Integer> indices, int start1, int end1, int start2, int end2)**:Creates a subset of the dataset X by selecting rows based on provided

indices within specific ranges. Used during cross-validation to create training and validation subsets for each fold.
    - **subset(double[][] X, List<Integer> indices, int start, int end)**:
      Creates a subset of the dataset X by selecting rows within a range defined by start and end. Used to extract training or validation data in cross-validation.
    - **subset(int[] y, List<Integer> indices, int start1, int end1, int start2, int end2)**:
      Creates a subset of the label array y based on provided indices and specified ranges. Used during cross-validation to create corresponding label arrays for training and validation.
    - **subset(int[] y, List<Integer> indices, int start, int end)**:
      Creates a subset of the label array y by selecting labels within a range of indices. Used to create training or validation label arrays in cross-validation.
    - **convertToDoubleArray(List<List<String>> data)**:
      Converts a list of string lists (e.g., from CSV) into a double[][] array of features, excluding the last column (assumed to be the label).
      Prepares the features for training the model.
    - **extractLabels(List<List<String>> data)**: Extracts and returns the labels from the dataset as an int[] array, converting them from the last column of the data.
    - **calculateAccuracy(int[] actualLabels, int[] predictedLabels)**:
      Computes the accuracy by comparing predicted labels to actual labels.
      Returns the proportion of correct predictions.
- Random Forest Model Test: Class for testing and validating everything within our Model Layer
    - **testInitialization()**: Verifies that the DecisionTree object is correctly initialized and is not null after creation.
    - **testFit()**: Trains the DecisionTree using a small dataset and checks that the tree is successfully trained and not null. Further assertions could be added to check specific properties of the trained tree (e.g., root node).
    - **testPredict()**: Predicts labels for a dataset using a trained DecisionTree and compares the predicted labels to the true values, ensuring correct predictions.
    - **testPredictWithUnseenData()**: Verifies that the DecisionTree can correctly predict labels for new, unseen data after being trained on a previous dataset.
    - **testLeafNode()**: Tests the isLeafNode() method of the Node class, ensuring that the method correctly identifies leaf nodes and checks the node's value.
    - **testBiasInPredictionAccuracy()**: Tests the accuracy of the RandomForest model on two distinct groups. The accuracy for each group is calculated and verified to ensure the model is unbiased and performs well.
    - **testOverfitting()**: Evaluates the RandomForest model for overfitting by training it on a small dataset and ensuring that the predictions perfectly match the true values, indicating potential overfitting.

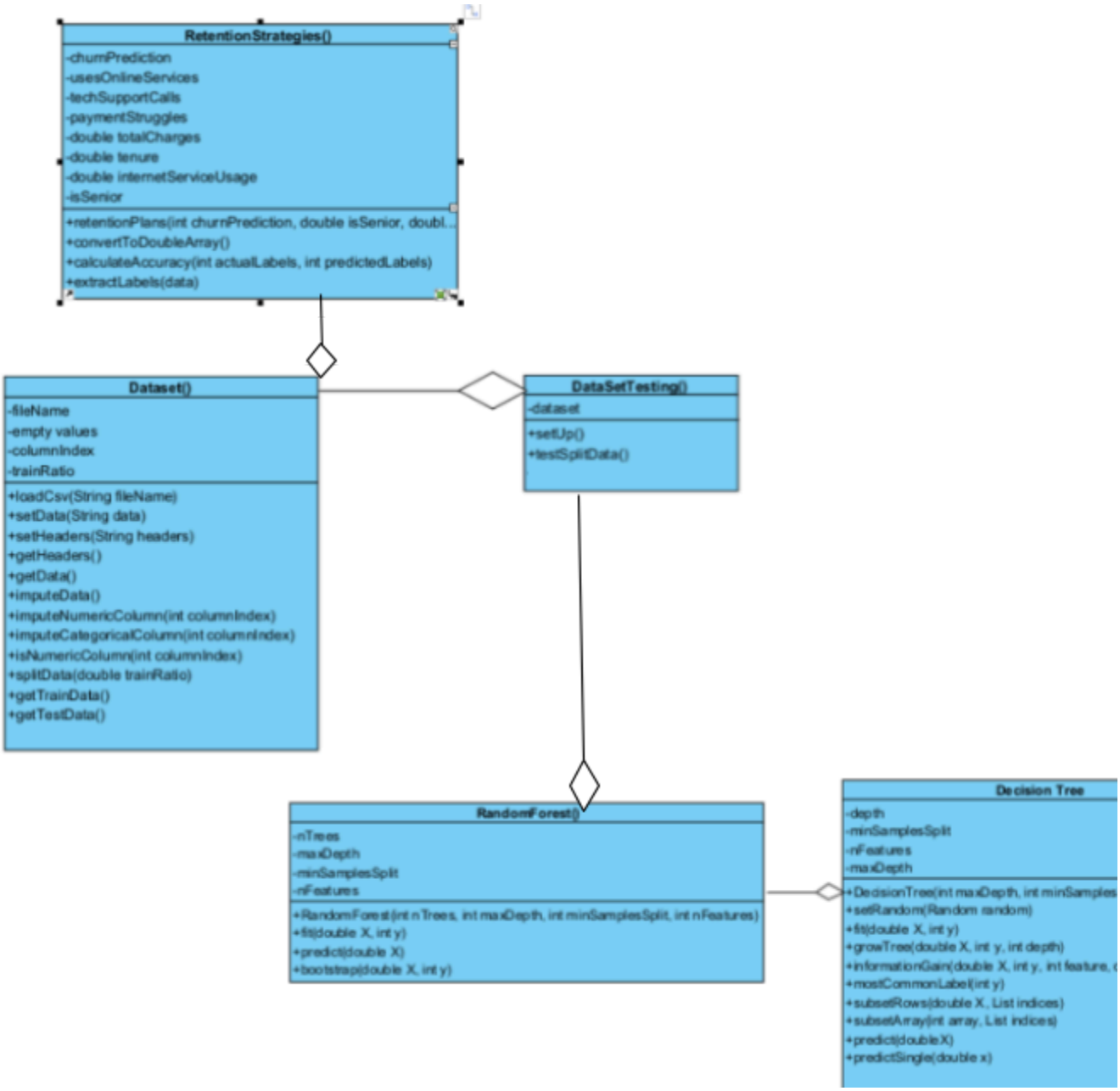
- **testBootstrappingValidation()**: Verifies the bootstrapping process in the RandomForest model by checking that the bootstrapped dataset (X and y) has the same length as the original dataset.
- **calculateAccuracy(int[] predictions, int[] trueValues)**: A helper method that calculates the accuracy of the model's predictions by comparing them to the true labels and returning the proportion of correct predictions.
- **testCrossValidationAccuracy()**: Verifies that the cross-validation method in the Evaluation class calculates a non-negative accuracy value when performing k-fold cross-validation on the training data.

### 3. Application Layer



- RetentionStrategies: First asks for user input for a customer, then generates outputs.
    - **retentionPlans(int churnPrediction, double isSenior, double usesOnlineServices, double paperlessBilling, double techSupportCalls, double paymentStruggles, double totalCharges, double tenure, double internetServiceUsage)**: Generates a retention strategy for a customer based on their predicted churn and various customer features. It checks conditions like seniority, tech support usage, payment struggles, and internet service usage to suggest a retention strategy (e.g., cheaper plans, support improvements, or payment plans).
    - **convertToDoubleArray(List<List<String>> data)**: Converts the input dataset (a list of lists of strings) into a double[][] array, excluding the last column (the labels).
    - **extractLabels(List<List<String>> data)**: Extracts the labels from the dataset (the last column) and returns them as an int[] array.
    - **calculateAccuracy(int[] actualLabels, int[] predictedLabels)**: Calculates the accuracy of the model by comparing the predicted labels to the actual labels. Returns the accuracy as a proportion of correct predictions.

Now with all layers together, our class diagram for the entire system:

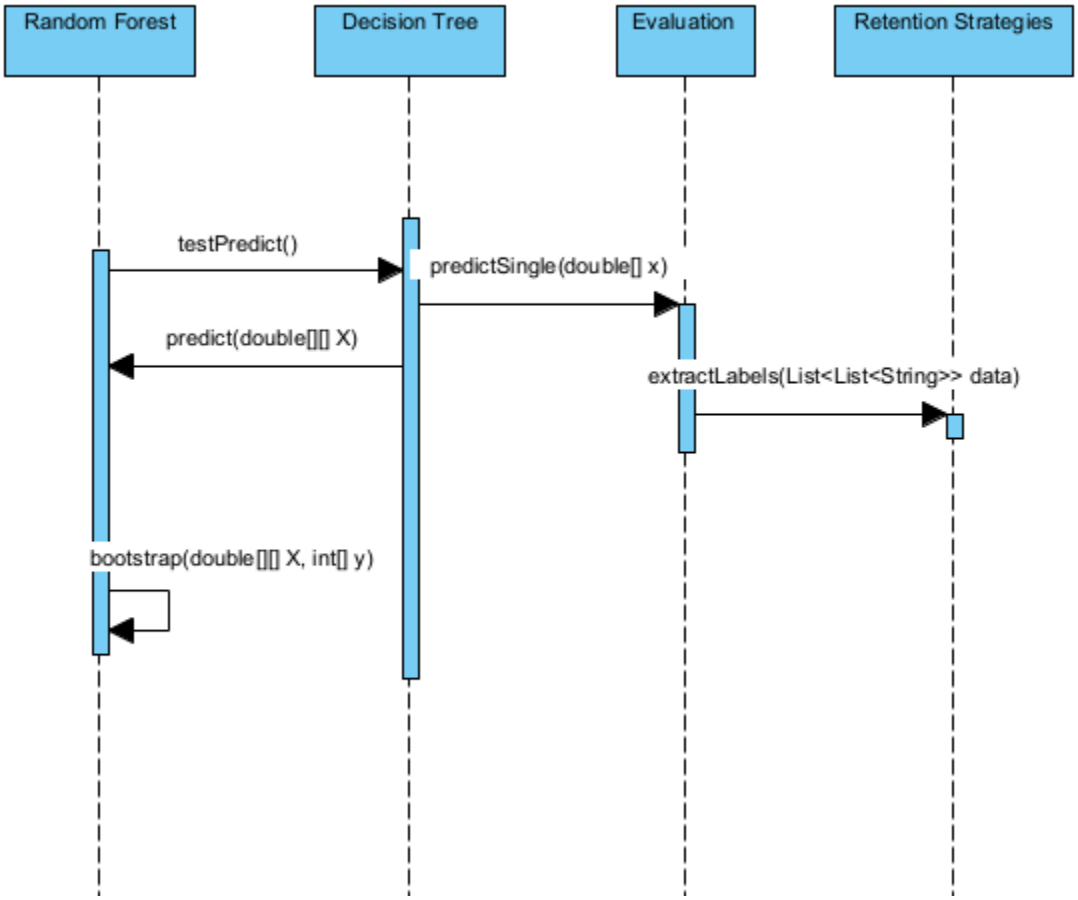



---

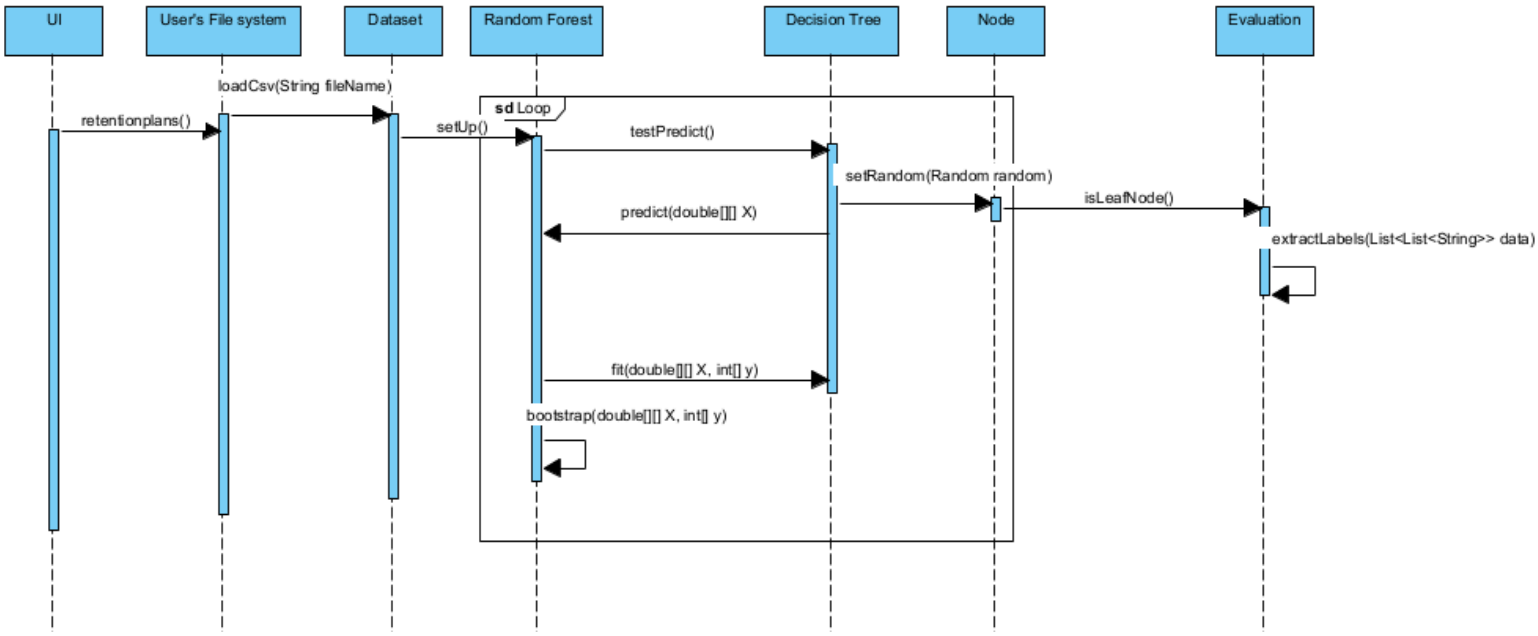
## Interaction Modeling:

Each sequence diagram present showcases how the system works and how it interacts with each object when it comes to our functional requirement.

**Functional requirement 1:** The system should predict whether the customer is leaving the company or not.
System will first train and test Random Forest and then do extensive prediction methods to predict customer churn, which is then sent to the Application layer.

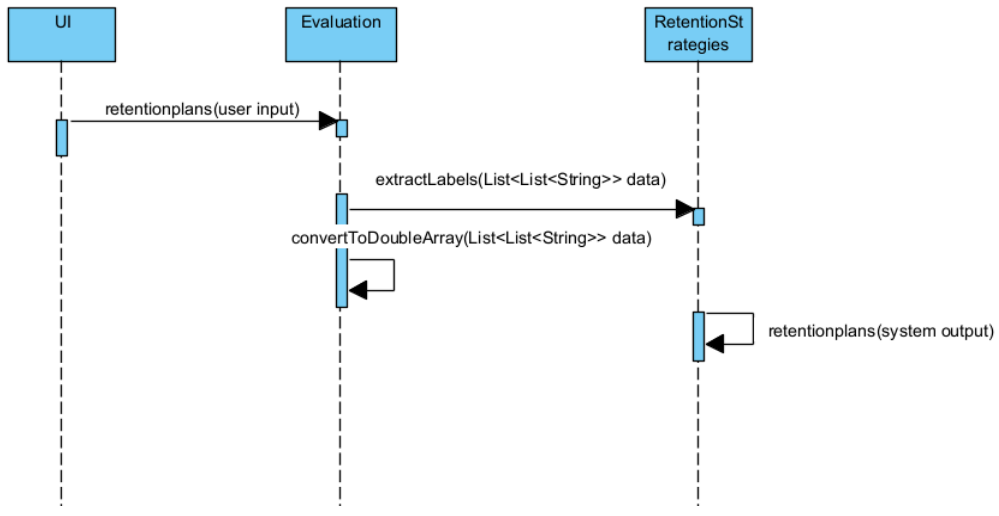


**Functional requirement 2:** The system should train a Random Forest model using historical customer data.
Expanding upon the first requirement, the test and train datasets found in the user's file system will be fed into the Random Forest model's prediction methods and then saved.




**Functional requirement 3 :**The system identifies high-risk customers and provides actionable retention strategies to improve customer loyalty and reduce churn.

Expanding upon the first 2 requirements, the user will input the customer data(hence why diagram states "user input" inside retentionplans method). Then through the model layer, it will



extract the data after it has undergone training and testing. Finally, actual system-generated retention strategies is send back to the UI.

---

## Programming environment

| Component | Version | Purpose |
|---|---|---|
| Java Development Kit (JDK) | Java 17 | For compiling and running the application |
| Eclipse IDE | Latest (2024) | Integrated Development Environment (IDE) |
| JUnit | V5 | Unit testing framework |
| Operating System | Windows/Linux/macOS | Compatible OS for development |

```
RandomForestProject
├── src
│   ├── ApplicationLayer
│   │   └── RetentionStrategies.java
│   ├── ModelLayer
│   │   ├── DecisionTree.java
│   │   ├── RandomForest.java
│   │   ├── Node.java
│   │   └── RandomForestModelTestCases.java
│   ├── DataLayer
│   │   ├── DataSet.java
│   ├── Evaluation.java
├── resources
│   ├── train_data.csv
│   ├── test_data.csv
├── lib
│   └── (Junit V5)
```

## Programming Languages

**. Java**

---

## External software packages/libraries and their roles in the development of the system

**JUnit 5**

- **Role**: For unit testing and validating the functionality of core components, including the DecisionTree and RandomForest classes.

**Eclipse IDE**

- **Role**: Integrated Development Environment for coding, debugging, and testing the Java application.

**Swing (javax.swing)**

- **Role**: Provides graphical user interface (GUI) components for the RetentionStrategies application.

**Java Collections Framework (java.util)**

- **Role**: For managing data structures such as List, Map, and Set used in data processing, bootstrapping, and tree construction.

**Dataset Overview: Data Description.**

- **Dataset Format:** The dataset consists of rows representing customer records and columns representing features (attributes) used to make predictions. Each row includes features like customer age, usage frequency, subscription type, and whether the customer churned or stayed.

**Size:** Total number of entries in the dataset: **7,044**

**Training and Testing Data**

- **Training Dataset**: Comprising **70%** of the total data, with **4,932** entries.
- **Testing Dataset**: Making up **30%** of the total data, with **2,112** entries.

**Data Attributes and Types**

- **Overview**: The dataset includes a mix of categorical and numerical data types used as features for predicting customer behavior.

**Categorical Data**

| Attribute | Type | Description |
|---|---|---|
| SeniorCitizen | Boolean | represent senior status |
| TechSupport | Boolean | Whether the customer has tech support |
| Dependents | Boolean | Whether the customer has dependents |
| PhoneService | Boolean | Whether the customer has phone service |
| MultipleLines | String | Type of multiple line services |
| InternetService | String | Type of internet service |
| OnlineSecurity | Boolean | Whether the customer uses online security |
| OnlineBackup | Boolean | Whether the customer uses online backup |
| DeviceProtection | Boolean | Whether the customer has device protection |

| | | |
|---|---|---|
| StreamingTV | Boolean | Whether the customer uses streaming TV |
| StreamingMovies | Boolean | Whether the customer uses streaming movies |
| Contract | String | Type of customer contract |
| PaperlessBilling | Boolean | Whether the customer uses paperless billing |

**Numerical data**

| Attribute | Type | Description |
|---|---|---|
| Tenure | Integer | Length of service in months |
| MonthlyCharges | Double | Monthly service charges |
| TotalCharges | Double | Total charges so far |

**Format for Training and Testing**

- **CSV Format**: Data is stored in comma-separated values (CSV) format, where each line corresponds to a customer record, and values are separated by commas.
- **Data Rows**: Following the header, each row contains the data for one customer.

**Data Splitting**

- **Training Dataset**: This dataset includes a larger portion of the total dataset (typically 70%). It is used to train the Random Forest model, allowing the decision trees to learn from the data.
- **Testing Dataset**: This dataset comprises the remaining portion of the data (typically 30%). It is used to test the trained model, providing an unbiased evaluation of its performance in predicting new, unseen data.

**Handling and Processing**

- **Data Loading**: The DataSet class handles loading of CSV files into memory.
- **Imputation**: Missing values in both categorical and numerical data are imputed — numerical columns by computing the mean and categorical columns by determining the mode.
- **Data Split**: A method in the DataSet class shuffles the complete dataset and then splits it into training and testing datasets based on the specified ratio.

---

**Test Cases design and code:**

## 1. Design for testSplitData Method

The **testSplitData** method verifies the dataset's ability to split into training and testing sets based on a predefined ratio. Specifically, it tests the functionality that allocates 70% of the data for training and 30% for testing, crucial for any data-driven application's training and validation stages.

**Code:**

```
@Test
public void testSplitData() {
   dataSet.splitData(0.70); // Split with 70% training, 30% testing
   assertEquals("Training data size should be 70", 70, dataSet.getTrainData().size());
   assertEquals("Testing data size should be 30", 30, dataSet.getTestData().size());
}
```

---

## 2. Design for testImputation Method

This method assesses the imputation mechanism designed to handle missing values in the dataset. It specifically tests for the replacement of empty strings in numeric and categorical columns with the mean of the numeric columns and the mode of the categorical columns. The design ensures that no data is left incomplete, which is vital for maintaining the quality and accuracy of data processing.

**Code:**

```
@Test
public void testImputation() {
    // Prepare headers and data with missing values
    List<String> headers = Arrays.asList("NumericColumn", "CategoricalColumn",
"AnotherNumericColumn");
    dataSet.setHeaders(headers);
    List<List<String>> data = new ArrayList<>();
    data.add(new ArrayList<>(Arrays.asList("", "A", "10"))); // Missing numeric value
    data.add(new ArrayList<>(Arrays.asList("20", "", "30"))); // Missing categorical value
    data.add(new ArrayList<>(Arrays.asList("", "A", ""))));    // Missing numeric values
    dataSet.setData(data);


    dataSet.imputeData(); //test imputation


    // Assertions for CategoricalColumn
    assertEquals("CategoricalColumn missing values should be replaced by the mode", "A",
dataSet.getData().get(1).get(1));
    // Assertions for AnotherNumericColumn
    assertEquals("AnotherNumericColumn missing values should be replaced by the mean",
"20.0", dataSet.getData().get(2).get(2));


    // Ensure no empty values remain
    for (List<String> row : dataSet.getData()) {
```

```
    for (String value : row) {
        assertFalse("No value should be empty after imputation", value.isEmpty());}
}
```

---

### 3. Design Explanation for testInitialization

This test ensures the DecisionTree object is instantiated correctly and is not null after creation. It verifies basic object instantiation with initial parameters like depth, min_samples_split, and min_samples_leaf.

**Code:**

```
@Test
public void testInitialization() {
    tree = new DecisionTree(5, 2, 2);
    assertNotNull(tree, "DecisionTree object should not be null after initialization")
```

---

### 4. Design Explanation for `testFit`

Tests the fit method of the DecisionTree to confirm it can train on provided datasets (X and y). It primarily checks whether the tree remains valid (not null) after attempting to fit the model, implying successful training initialization.

**Code**:

```
@Test
public void testFit() {
    double[][] X = {{1.0, 2.0}, {3.0, 4.0}, {5.0, 6.0}};
    int[] y = {0, 1, 0};
    tree.fit(X, y);
    assertNotNull(tree, "Tree should be trained successfully");
```

}

---

### 5.Design Explanation for testPredict

Ensures that the DecisionTree can make predictions on the dataset it was trained on, and those predictions match the actual labels. It validates the effectiveness of the training process.

Code:

```
@Test
public void testPredict() {
    double[][] X = {{1.0, 2.0}, {3.0, 4.0}, {5.0, 6.0}};
    int[] y = {0, 1, 0};
    tree.fit(X, y);
    int[] predictions = tree.predict(X);
    assertArrayEquals(y, predictions, "Predictions should match the actual labels");
}
```

---

### 6.Design Explanation for testLeafNode

Checks if the Node class correctly identifies a node as a leaf and matches its assigned value. This test is crucial for validating the leaf node behavior in decision trees.

**code**:

```
@Test
public void testLeafNode() {
    Node leafNode = new Node(1);
    assertTrue(leafNode.isLeafNode(), "Node should be identified as a leaf node");
    assertEquals(1, leafNode.value, "Leaf node value should match the expected label");
```

```
}
```

---

## 7. **Design Explanation for testBiasInPredictionAccuracy**

This test assesses the prediction accuracy of RandomForest on different groups within the dataset to detect any potential bias in model performance across diverse data subsets.

**Code**:

```
@Test
public void testBiasInPredictionAccuracy() {
    double[][] X = {{1.0, 2.0}, {1.0, 3.0}, {0.0, 1.0}, {0.0, 4.0}, {0.0, 6.0}};
    int[] y = {0, 0, 1, 1, 1};
    forest.fit(X, y);
    double[][] group0X = {{0.0, 1.0}, {0.0, 4.0}, {0.0, 6.0}};
    int[] group0Y = {1, 1, 1};
    int[] group0Predictions = forest.predict(group0X);
    double group0Accuracy = calculateAccuracy(group0Predictions, group0Y);
    double[][] group1X = {{1.0, 2.0}, {1.0, 3.0}};
    int[] group1Y = {0, 0};
    int[] group1Predictions = forest.predict(group1X);
    double group1Accuracy = calculateAccuracy(group1Predictions, group1Y);
    assertTrue(group0Accuracy > 0.8, "Group 0 should have high accuracy");
    assertTrue(group1Accuracy > 0.8, "Group 1 should have high accuracy");
}
```

---

8.**Design Explanation for testOverfitting**

This test checks for overfitting by ensuring that the RandomForest perfectly predicts the training data it was trained on, a common symptom of overfitting.

**Code**:

```
@Test
public void testOverfitting() {
    double[][] X = {{1.0, 2.0}, {2.0, 3.0}, {3.0, 1.0}, {5.0, 4.0}, {4.0, 6.0}};
    int[] y = {0, 0, 1, 1, 1};
    forest.fit(X, y);
    int[] predictions = forest.predict(X);
    assertTrue(Arrays.equals(predictions, y), "Overfitting Test should pass with perfect accuracy");
}
```

---

**9.Design Explanation for testBootstrappingValidation**

Tests the bootstrap method of RandomForest to ensure it generates a correctly sized bootstrapped sample of the dataset, crucial for effective random forest training.

**Code:**

```
@Test
public void testBootstrappingValidation() {
    double[][] X = {{1.0, 2.0}, {2.0, 3.0}, {3.0, 1.0}, {5.0, 4.0}, {4.0, 6.0}};
    int[] y = {0, 0, 1, 1, 1};
```

```
    Map<String, Object> bootstrappedData = forest.bootstrap(X, y);

    double[][] bootstrappedX = (double[][]) bootstrappedData.get("X");

    int[] bootstrappedY = (int[]) bootstrappedData.get("y");

    assertEquals(X.length, bootstrappedX.length, "Bootstrapped X should have the same length as input X");

    assertEquals(y.length, bootstrappedY.length, "Bootstrapped Y should have the same length as input Y");

}
```

---

10. **Design Explanation for testCrossValidationAccuracy**

Evaluates the cross-validation implementation in the Evaluation class to ensure it correctly computes a non-negative accuracy score across multiple folds, indicating the model's general performance consistency

Code

```
@Test

public void testCrossValidationAccuracy() {

    int k = 5;

    double result = Evaluation.crossValidation(X_train, y_train, k);

    assertTrue(result >= 0.0, "Cross-validation should calculate a non-negative accuracy");
```

---

## Unit Testing

## DataSetTest Class:

**Junit Testing**

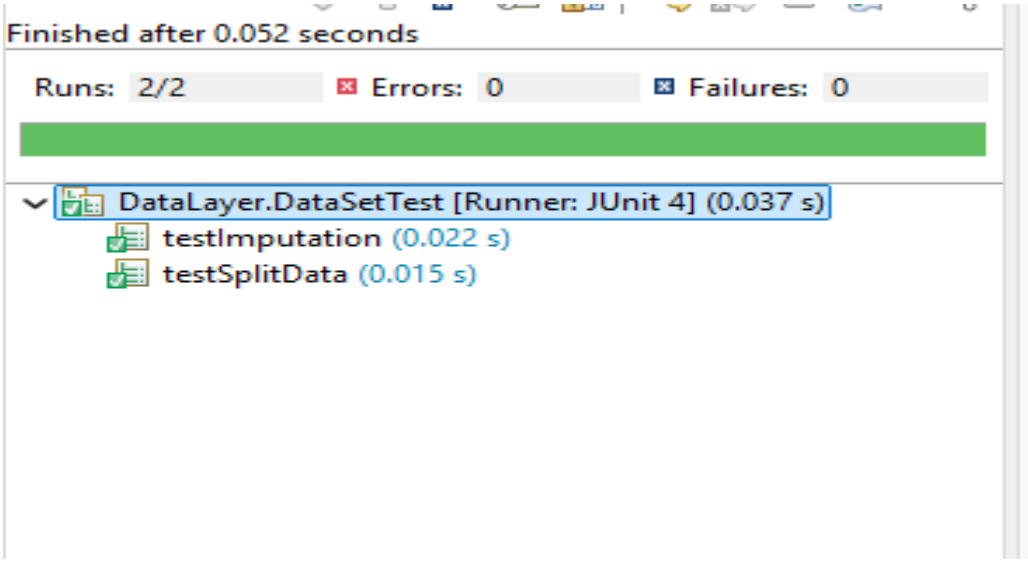
**1. Component Testing: DataSet - splitData()**

- **Purpose**: Tests whether the splitData() method correctly splits the dataset into training and testing datasets based on a specified ratio.
- **Input**:
    - **DataSet Initialization**: Initialized with predefined data.
    - **Split Ratio**: 0.70 (70% training, 30% testing).
- **Expected Outcome**: The method should properly allocate 70% of the entries to the training dataset and 30% to the testing dataset, maintaining dataset integrity and the specified proportions.
- **Assertion Method**:
    - **Training Data Size Check**: Asserts that the training dataset size is exactly 70% of the total dataset.
    - **Testing Data Size Check**: Asserts that the testing dataset size is exactly 30% of the total dataset.

**2. Component Testing: DataSet - imputeData()**

- **Purpose**: Tests the imputeData() method to ensure it correctly imputes missing values in both numerical and categorical columns based on the dataset.
- **Input**:
    - **DataSet Initialization**: DataSet is initialized with a small sample set containing missing values in different columns.
    - **Headers**: ["NumericColumn", "CategoricalColumn", "AnotherNumericColumn"]
    - **Data with Missing Values**:
        - First row: Missing numeric value.
        - Second row: Missing categorical value.
        - Third row: Multiple missing numeric values.
- **Expected Outcome**: All missing values should be correctly imputed. Numeric missing values should be replaced by the mean of their respective columns, and categorical missing values by the mode.
- **Assertion Method**:
    - **Categorical Imputation**: Asserts that missing values in the categorical column are replaced by the mode of existing values.

- **Numerical Imputation**: Asserts that missing values in numerical columns are replaced by the mean of existing values.
- **No Empty Values Check**: Asserts that there are no empty values in any row after imputation.




**<u>RandomForestModelTest Class:</u>**

**** This class combines all the test cases for the Model (Random Forest) Layer including (Decision tree class, Evaluation class, Random Forest class)**

**Decision tree test cases:**

**3. Component Testing: DecisionTree - testInitialization**

- **Purpose**: Verify that the DecisionTree instances are initialized correctly.
- **Input**: Instantiation of DecisionTree with parameters (5, 2, 2).
- **Expected Outcome**: Both instances should be created without throwing any exceptions.
- **Assertion Method**: **assertNotNull(tree, "DecisionTree object should not be null after initialization")**: This assertion checks if the tree object is successfully instantiated. A null object would indicate a failure in the constructor or initialization process, which could be due to misconfiguration or runtime issues that prevent proper object creation.

**4.Component Testing: DecisionTree - testFit**

- **Purpose**: Ensure the DecisionTree can be trained with specific data.
- **Input**: Training dataset X = [[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]] and labels y = [0, 1, 0].
- **Expected Outcome**: The tree should correctly process and fit the training data.

- **Assertion Method**: **assertNotNull(tree, "Tree should be trained successfully")**: After attempting to train the tree with specific data, this assertion verifies that the tree object still exists and isn't inadvertently altered or nullified during the training process. This confirmation is crucial to ensure that the tree maintains its integrity after being exposed to training data

**5.Component Testing: DecisionTree - testPredict**

- **Purpose**: Confirm that the DecisionTree correctly predicts labels for a trained dataset.
- **Input**: Same dataset as used for fitting: X = [[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]].
- **Expected Outcome**: Predicted labels should match the actual labels [0, 1, 0].
- **Assertion Method**: **assertArrayEquals(y, predictions, "Predictions should match the actual labels")**: This assertion directly compares the array of predictions made by the tree against the actual labels used during training. It's vital for validating the tree's ability to correctly process and predict data based on its training, ensuring model accuracy and effectiveness.

**6.Component Testing: DecisionTree - testLeafNode**

- **Purpose**: Verify the functionality of detecting a leaf node.
- **Input**: Explicit creation of a Node initialized as a leaf with a label 1.
- **Expected Outcome**: The node should be identified as a leaf and should store the correct label.
- **Assertion Method**: **assertTrue(leafNode.isLeafNode(), "Node should be identified as a leaf node") and assertEquals(1, leafNode.value, "Leaf node value should match the expected label")**: These assertions test both the structural and functional aspects of a leaf node. The first checks the node's status as a leaf, ensuring that the tree correctly recognizes terminal nodes. The second verifies that the leaf node's value (or label) is accurately stored and retrieved, which is essential for the final decision-making in prediction tasks.

---

**Random Forest Test Cases:**

**7.Component Testing: RandomForest - testBiasInPredictionAccuracy**

- **Purpose**: Evaluate prediction accuracy on distinct data groups to detect any bias.

- **Input**: Training data X = [[1.0, 2.0], [1.0, 3.0], [0.0, 1.0], [0.0, 4.0], [0.0, 6.0]], labels y = [0, 0, 1, 1, 1]. Groups tested for prediction: group0X = [[0.0, 1.0], [0.0, 4.0], [0.0, 6.0]], group1X = [[1.0, 2.0], [1.0, 3.0]].
- **Expected Outcome**: Both groups should be predicted with high accuracy, ideally above 80%.
- **Assertion Method**: **assertTrue(group0Accuracy > 0.8, "Group 0 should have high accuracy") and assertTrue(group1Accuracy > 0.8, "Group 1 should have high accuracy")**: These assertions check whether the prediction accuracy for different groups within the test dataset meets a high standard (e.g., 80% accuracy). Such checks are critical to ensure the model does not exhibit bias or underperformance on any segment of the data, affirming the model's fairness and effectiveness across varied data subsets.

**8.Component Testing: RandomForest - testOverfitting**

- **Purpose**: Assess if the RandomForest model is overfitting by testing on the training dataset.
- **Input**: Dataset X = [[1.0, 2.0], [2.0, 3.0], [3.0, 1.0], [5.0, 4.0], [4.0, 6.0]] and labels y = [0, 0, 1, 1, 1].
- **Expected Outcome**: The predictions should perfectly match the labels, indicating potential overfitting.
- **Assertion Method**: **assertTrue(Arrays.equals(predictions, y), "Overfitting Test should pass with perfect accuracy")**: This test is designed to identify potential overfitting by comparing the model's predictions on the training data against the actual training labels. A perfect match suggests that the model might be overly fitted to the training data, which could compromise its performance on new, unseen data.

**9.Component Testing: RandomForest - testBootstrappingValidation**

- **Purpose**: Verify the correctness and integrity of the bootstrap sampling process.
- **Input**: Original dataset X = [[1.0, 2.0], [2.0, 3.0], [3.0, 1.0], [5.0, 4.0], [4.0, 6.0]] and labels y = [0, 0, 1, 1, 1].
- **Expected Outcome**: The bootstrap samples should have the same length as the original dataset and represent a valid random sample.
- **Assertion Method**: **assertEquals(X.length, bootstrappedX.length, "Bootstrapped X should have the same length as input X") and assertEquals(y.length, bootstrappedY.length, "Bootstrapped Y should have the same length as input Y")**: These assertions ensure that the bootstrapping process correctly replicates the dataset in terms of size, and that the sampling process involves appropriate randomization. This verification is essential for the integrity of the training process, ensuring that each tree in the RandomForest is trained on a valid, randomly sampled subset, which helps improve

model robustness and generalization.

---

**Evaluation Class Test Cases**:

**10.Component Testing: - testCrossValidationAccuracy**

- **Purpose**: Evaluate the model's performance using cross-validation to ensure robustness.
- **Input**:
  double[][] X_train = {

  {1.0, 2.0},

  {3.0, 4.0},

  {5.0, 6.0},

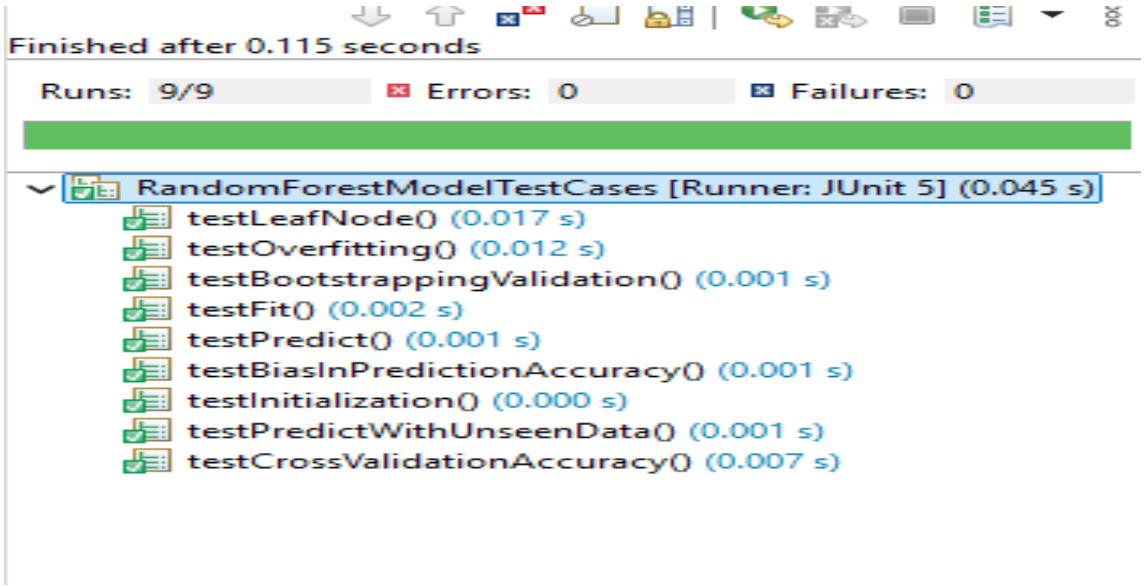
  {7.0, 8.0},

  {9.0, 10.0}

  };
  int[] y_train = {0, 1, 0, 1, 0};
  For this test case, k is set to 5.

- **Expected Outcome**: The cross-validation should return a non-negative accuracy score.
- **Assertion Method**: assertTrue is used to confirm that the calculated cross-validation accuracy is non-negative, ensuring that the model's performance evaluation is logically

correct and meaningful.




---

## SystemWise

### Functional Requirement #1: Churn Prediction

#### Test Case 1: Standard Prediction

- **Objective:** Verify that the system accurately predicts customer churn based on typical customer data.
- **Steps:**
    1. Input a standard set of customer data with known churn outcomes into the system.
    2. Execute the churn prediction using the Random Forest model.
- **Expected Result:** The system should predict churn accurately, matching the expected churn outcomes with an accuracy rate consistent with predefined performance metrics (about 80% accuracy).

#### Test Case 2: Handling of Incomplete Data

- **Objective:** Ensure the system handles incomplete input data gracefully and still makes a prediction.
- **Steps:**
    1. Input customer data with missing values for non-critical fields (e.g., MultipleLines, StreamingTV).
    2. Execute churn prediction.

- **Expected Result:** The system should provide a prediction using default or inferred values for missing data without crashing or throwing errors, and it should log or alert that predictions were made with incomplete data.

## Functional Requirement #2: Usage of Random Forest

### Test Case 1: Model Training and Accuracy

- **Objective:** Test the system's ability to train the Random Forest model and achieve an expected accuracy threshold.
- **Steps:**
    1. Divide a set of known customer data into training and testing datasets.
    2. Train the Random Forest model on the training set.
    3. Evaluate the model on the testing set.
- **Expected Result**: The model should achieve a minimum accuracy rate of 80% on the testing dataset, demonstrating robustness and effective learning from the training dataset.

### Test Case 2: Model Reusability

- **Objective:** Confirm that a trained model can be saved and reused for predictions.
- **Steps:**
    1. Train the Random Forest model with a specific dataset.
    2. Save the model to disk.
    3. Load the model from disk and make predictions on a new set of data.
- **Expected Result:** The loaded model should produce the same predictions on the new dataset as it did before being saved, confirming no loss in prediction accuracy or model integrity post-load.

## Functional Requirement #3: Proactive Retention System

### Test Case 1: Identification and Action on High-Risk Customers

- **Objective:** Verify the system's ability to identify high-risk customers and suggest appropriate retention strategies.
- **Steps:**
    1. Input customer data labeled as high risk by previous churn predictions.
    2. Execute the system's retention strategy recommendation process.
- **Expected Result:** The system should generate detailed reports outlining behavioral similarities among high-risk customers and propose specific actionable retention strategies tailored to each high-risk profile.

---

**Acceptance testing for non-functional requirements.**

## 1. High Level of Prediction Accuracy

**Objective**: Ensure the system maintains about 80% accuracy in predicting customer churn.

**How to Test**:

- **Test Preparation**: Obtain a dataset of 1,000 customer records from the last 12 months that includes a balanced number of churned (500) and retained (500) customers, ensuring it's reflective of diverse scenarios.
- **Execution**:
  - Feed the dataset into the system.
  - Record the system's churn predictions for each record.
- **Validation**:
  - Compare predictions against actual churn data.
  - Calculate accuracy, precision, recall, and F1-score.
- **Metrics to Measure**:
  - **Accuracy**: Correct predictions/Total predictions\text{Correct predictions} / \text{Total predictions}Correct predictions/Total predictions
  - **Precision**: True Positives/(True Positives+False Positives)\text{True Positives} / (\text{True Positives} + \text{False Positives})True Positives/(True Positives+False Positives)
  - **Recall**: True Positives/(True Positives+False Negatives)\text{True Positives} / (\text{True Positives} + \text{False Negatives})True Positives/(True Positives+False Negatives)
  - **F1-score**: 2×(Precision×Recall)/(Precision+Recall)2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})2×(Precision×Recall)/(Precision+Recall)
- **Passing Criteria**:
  - Accuracy ≥ 80%
  - Precision, Recall, F1-score ≥ 75%

---

### 2. Capability to Handle Large Datasets

**Objective**: Demonstrate the system's capability to efficiently manage and process a dataset with **7,044** customer records without performance degradation.

**How to Test**:

- **Test Preparation**: Use a synthetic dataset with **7,044** records, each having 18 attributes, to mimic real operational data in terms of size and complexity.
- **Execution**:
  - Load the dataset into the system.
  - Execute multiple queries to retrieve data, update records, and generate summary reports.
- **Validation**:
  - Monitor and record system response times and resource usage.
- **Metrics to Measure**:
  - **Response Time**: Time taken to complete each query and report.
  - **Resource Usage**: CPU, memory utilization during execution.
- **Passing Criteria**:
  - Average response time for queries ≤ 2 seconds.
  - CPU usage ≤ 70%, Memory usage ≤ 75% during operations.

---

### 3. System Performance and Efficiency

**Objective**: Confirm that the system can handle regular and peak operational loads efficiently without significant delays or errors.

**How to Test**:

- **Test Preparation**: Define typical daily tasks such as processing 500 customer updates, running 100 churn predictions, and generating 50 reports.
- **Execution**:
  - Conduct a baseline performance test by executing the defined tasks under normal load.
  - Perform a stress test by doubling the operations to simulate peak load.
- **Validation**:
  - Measure how the system copes with increased load and observe any performance bottlenecks or failures.

- **Metrics to Measure**:
    - **Task Completion Time**: Time taken to complete each task under normal and peak loads.
    - **System Stability**: Error rates, timeouts, or crashes are recorded during the stress test.
- **Passing Criteria**:
    - Task completion times under normal load should be less than 3 minutes.
    - Under peak load, task completion times should not exceed 5 minutes.
    - Error rates should remain below 1%, with no critical system failures.

# User's Guide

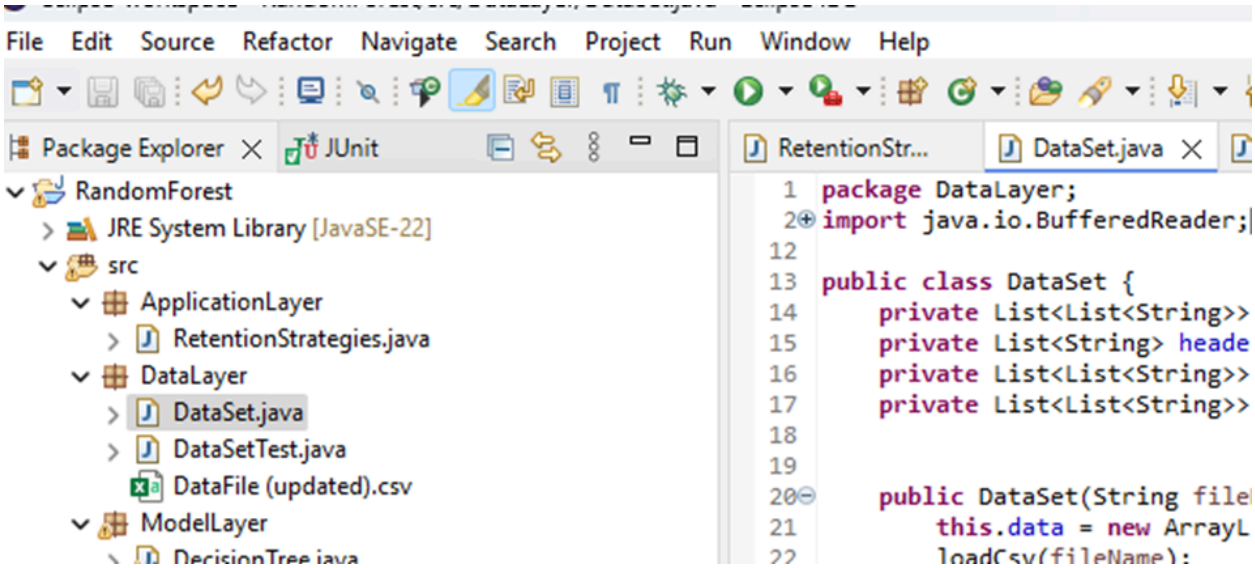
**1.** Go to the DataSet class in DataLayer. Run it for one time only to split the data into files

Note: you don't need to do it again anymore just for the first time you run the program.

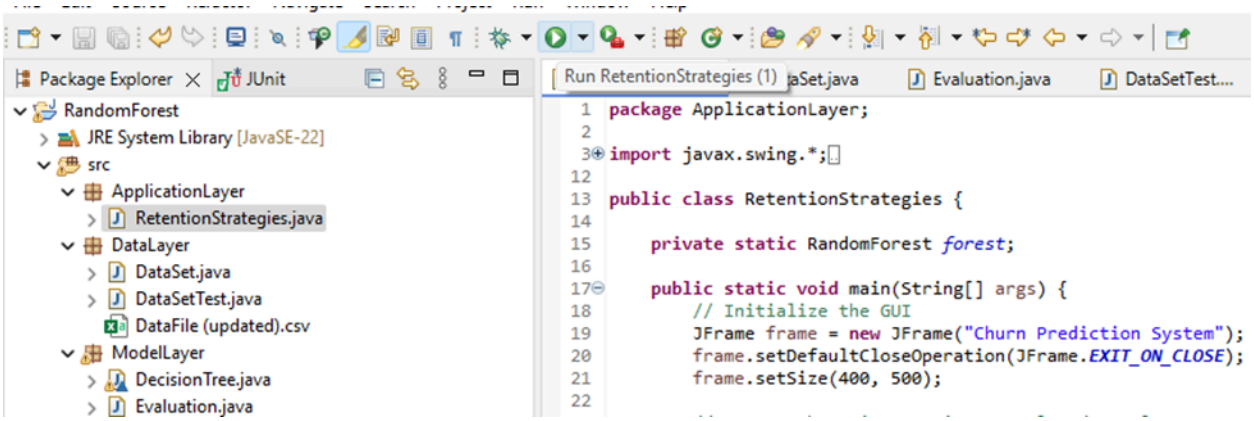
**After that it will be automatically saved in the new files

Note: you don't have to do it, it's just an extra step, to make sure the data is safe to use

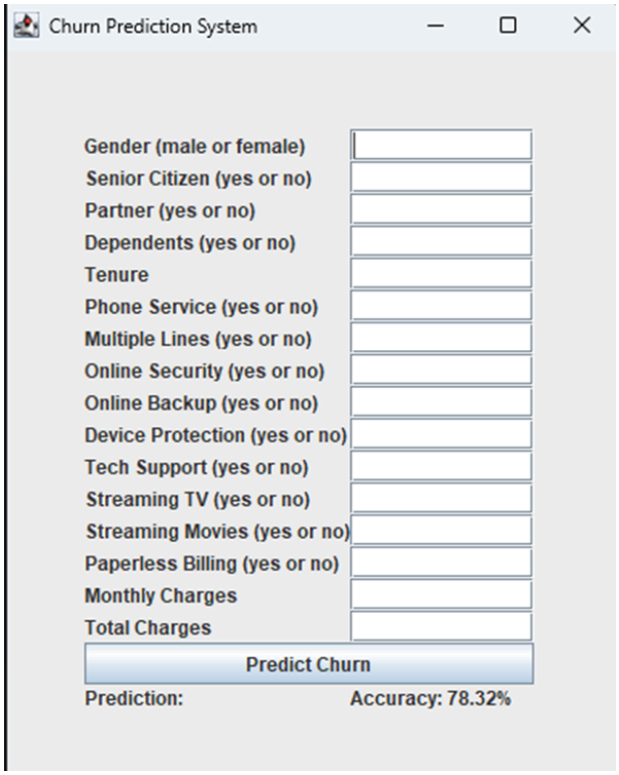
You can run the Application immediately without the need for this step



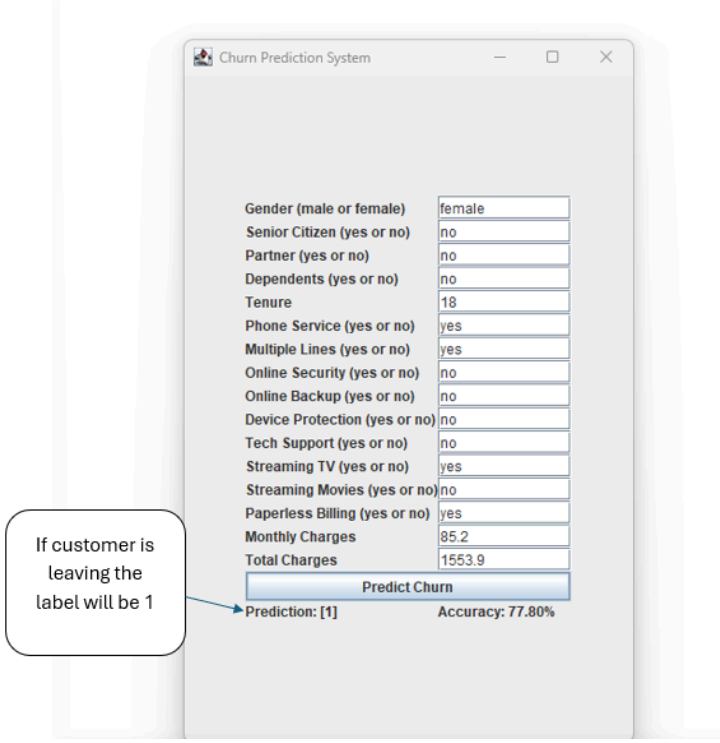

2. After splitting the data, go to the Application Layer and run the program



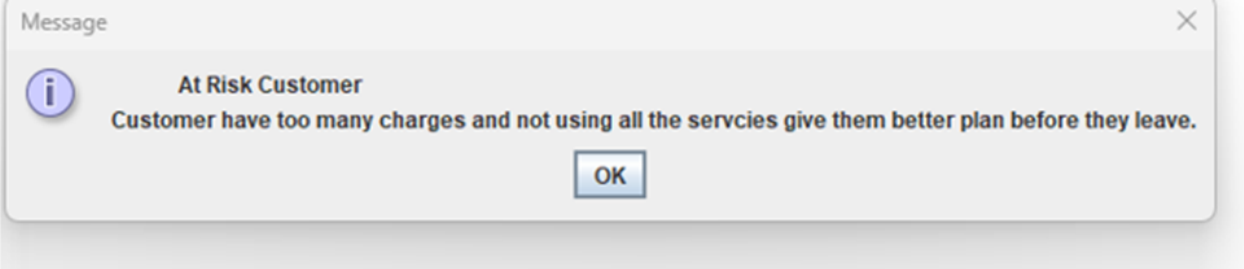
3. A screen will pop up




4. input the data of the customer you suspect will leave.



5. If the customer is leaving a customized message will pop up based on the customer's needs




6. you have the option to enter new customer data again or close the program.

---

**Planning For V 2.0**

For Version 2.0 of the app, we're making a change from using outside data analysis on most attributes that make the customers leave (provided in the Kaggle dataset) to using our own advanced machine learning method. This new method, called dot product similarity analysis, will help us understand and change the behavior of customers who might leave to be more like customers who stay and are loyal. This will also improve our accuracy as for Version 1.0, the accuracy ranges from 70-80, which is good but 90-100 is better.

In Version 1.0, the problem was that the system paid too much attention(weighted more than the other attributes) to the customer's monthly and total charges. This caused the recommendations to focus too much on these charges only ignoring the other issues. To fix this in Version 2.0, we are planning to use better techniques to make sure these charges don't affect the system's decisions too much by creating more functions to memorialize those 2 attributes. This will help make our predictions and suggestions better and more useful for keeping customers.

---

GitHub Location: https://github.com/letsdothis2003/Customer_Churn_System