​Chandravallika Murarisetty

murarisc@uwindsor.ca​

Weiming Zheng​

Zheng2c@uwindsor.ca​

Tausif Zaman

Zaman45@uwindsor.ca​

Saima Khatoon​

khatoons@uwindsor.ca

Aoqing Liu

Liu586@uwindsor.ca

Mobile Plan Search & Recommendation System​

Group No. 12​

Group name: All for on

# Mobile Plan Recommendation System

The Mobile Plan Search and Recommendation System is an advanced, feature-rich application designed to assist users in selecting the most suitable mobile data plans by evaluating various criteria such as data usage, cost, and plan features. This system aims to streamline the decision-making process for users by offering intelligent recommendations and advanced search capabilities.

**Key Features:**

* **Data and Cost-Based Recommendations**: Suggests plans tailored to user preferences, whether focused on high data limits or budget-friendly options.
* **Real-Time Search with Word Suggestions**: Provides dynamic word completion as users type, enhancing the search experience with real-time suggestions.
* **Spell Checking and Correction**: Automatically detects and suggests corrections for misspelled search terms to improve search accuracy.
* **Search Frequency Tracking**: Records and displays search frequency trends, helping users identify popular plans and frequently searched terms.
* **Page Ranking and Inverted Indexing**: Implements advanced ranking algorithms and inverted indexing for efficient retrieval and sorting of mobile plans based on relevance.
* **Advanced Page Searching**:
* **User Complaint Support**: Integrated support system allowing users to submit complaints, with email and mobile validation, and stores feedback securely for follow-up.
* **Interactive GUI Using Swing**: Provides an intuitive, event-driven interface using Java Swing components, including responsive tabs, buttons, and search panels for a seamless user experience.

# Contribution

main

└── java

└── com.mac.acc

├── datacomparison -- Chandravallika Murarisetty

│ ├── DataComparison -- Chandravallika Murarisetty

│ └── MobilePlans -- Chandravallika Murarisetty

├── features

│ ├── SearchBar – Tausif Zaman

│ ├── SearchFrequencyQuery – Weiming Zheng

│ ├── SearchFrequencyQueryFactory– Weiming Zheng

│ ├── SimpleSearchFrequencyQuery– Weiming Zheng

│ ├── SpellChecker – Tausif Zaman

│ └── WordCompletion – Aoqing Liu

├── recommendation – Aoqing Liu

│ ├── Package – Aoqing Liu

│ ├── PackageRecommender – Aoqing liu

│ ├── ScoredPackage – Aoqing Liu

│ └── UserPreferences – Aoqing Liu

search

├── engine– Weiming Zheng

│ ├── Document– Weiming Zheng

│ ├── Field– Weiming Zheng

│ └── SearchEngineFactory– Weiming Zheng

└── ui

├── Features - Saima Khatoon

├── HomeTab - Saima Khatoon

├── LogEntry– Weiming Zheng

├── MobilePlanRecommendationSystem - Saima Khatoon

├── PageRankingUI -- Chandravallika Murarisetty

├── RecommendationTab – Aoqing Liu

├── SearchBarUI - Saima Khatoon, Tausif Zaman

└── SupportTab – Saima Khatoon

resources

├── logback.xml– Weiming Zheng

├── mobile\_plans.csv -- Aoqing Liu

└── URLs.txt -- Chandravallika Mruarisetty

# Graphical User Interface

For GUI, we used Java Swing due to its robust features and flexibility. Swing provides a comprehensive set of lightweight components and tools that enabled us to build a responsive, user-friendly interface.

* **Java Swing for User Interface**:  
  Swing is utilized for creating an intuitive and interactive graphical user interface (GUI), offering a smooth and engaging user experience.
* **Desktop Application Suitability**:  
  Designed for desktop environments, Swing applications provide a stable and responsive interface ideal for standalone systems.
* **Cross-Platform Compatibility**:  
  Swing is platform-independent, ensuring consistent behavior across Windows, macOS, and Linux without code modifications.
* **Comprehensive GUI Components**:  
  Swing offers a rich set of components that are essential for modern UIs:
  + **JFrame :** A JFrame window is created to serve as the main container of the application.
    - The window is set to close when the user exits (setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE)).
    - The frame size is set to 1000x800 pixels.

Figure 1: Displaying the Home Page GUI

A screenshot of a computer

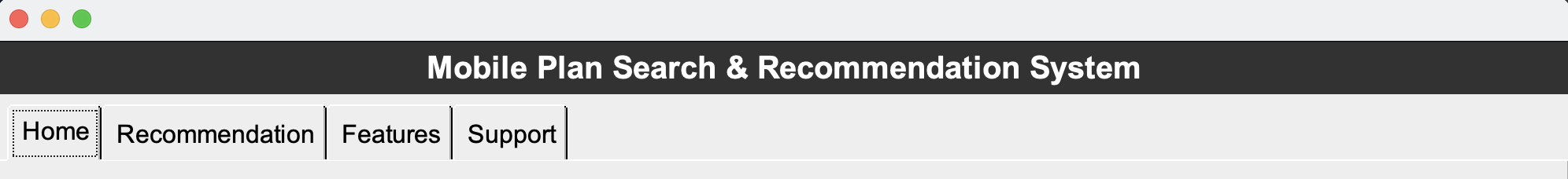
Description automatically generated

* + **JPanel:** For organizing components into structured layouts.
    - A JPanel is created for the title of the window.
  + **JTabbedPane:** A JTabbedPane is used to organize the application into tabs.

Four tabs are added to the JTabbedPane:

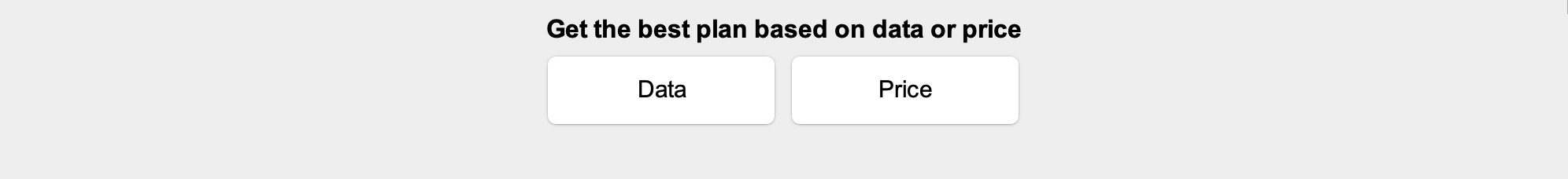
* + - **Home Tab**: Contains a custom panel (HomeTab).
    - **Recommendation Tab**: Contains a custom panel (RecommendationTab).
    - **Features Tab**: Contains a custom panel (Features).
    - **Support Tab**: Contains a custom panel (SupportTab).

Figure 2: Displaying the use of JTabbedPane, JPanel and JLabel



* + **JEditorPane:** For handling rich text and HTML content. Used in Home Tab:
    - A JEditorPane is created to display the recommended plan.
    - The content type is set to "text/html", allowing HTML content (such as links) to be displayed in the editor.
    - The editor is set to non-editable to prevent user modifications.
  + **JTextField:** For user input fields. Used in Recommendation Tab, Support Tab and Features Tab.
  + **JButton:** For interactive buttons. In Home Tab two Jbuttons: one for selecting a Data-based plan and one for selecting a Price-based plan is created.

Figure 3: Displaying the use of JButton.



* + **JLabel:** For displaying text and images.
    - A JLabel with the title "Mobile Plan Search & Recommendation System" is created for the title.
    - It is also used in Search Bar to display the search result.
* **Event-Driven Model for Interaction Handling**:  
  Swing’s event-driven model efficiently manages user interactions through various listeners:
  + **ActionListener**: Handles button clicks and other actionable events. In Home Tab ActionListener is used to respond to button clicks, allowing the user to get recommendations for mobile plans based on either data usage or price.
  + **HyperLink Listener**: Responds to hyperlink actions within text components. A HyperlinkListener is added to the editor pane in the Home Tab, so when the user clicks on a hyperlink within the displayed content, it opens the link in the default web browser.

Figure 4: Demonstrating the use of ActionListener when a button is clicked and HyperlinkListener in the Home Tab.

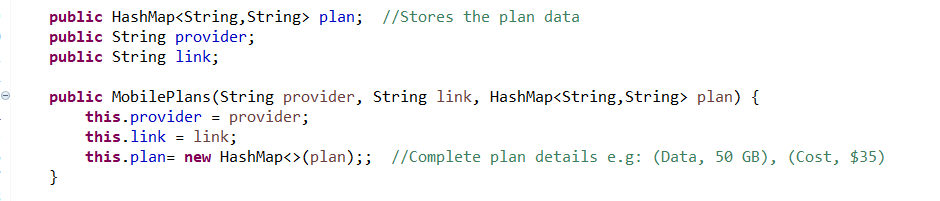
A computer screen shot of a program code

Description automatically generated

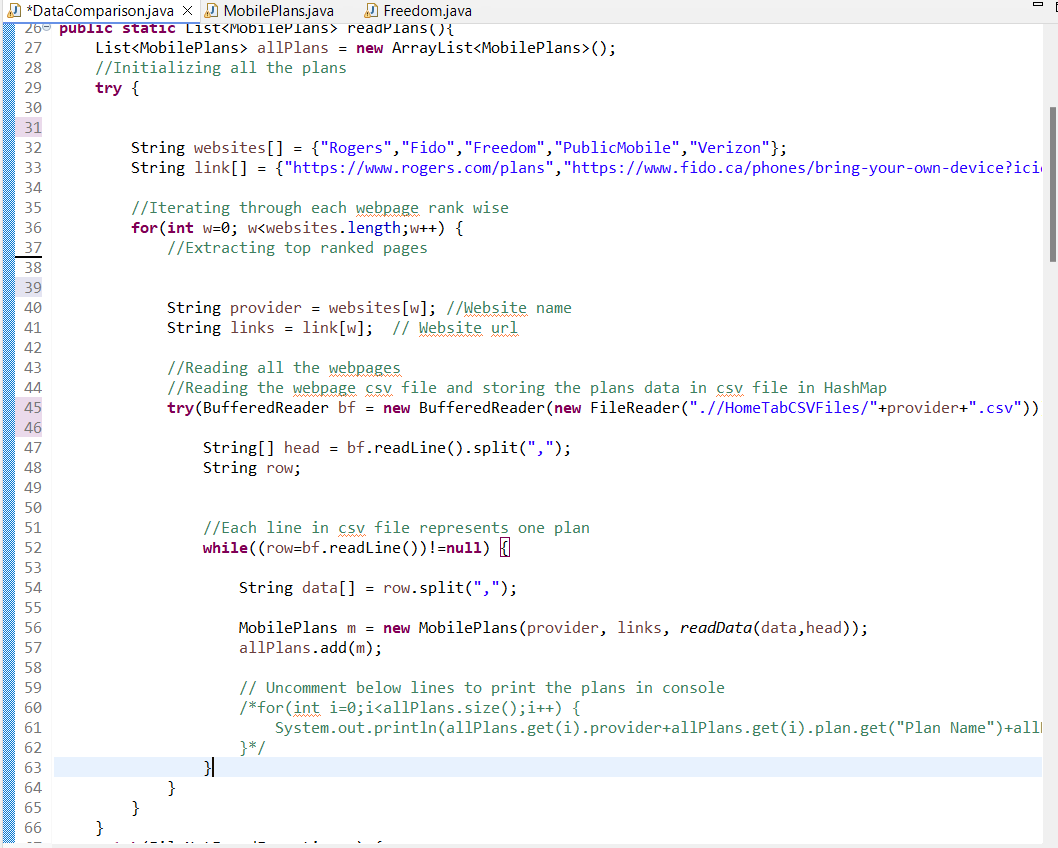
# Data Comparison

Data Comparison is based on the information crawled from websites and the user selection. User has two options one is Data and another one is Price. After selecting the desired option, the four mobile plans will be displayed to the user.

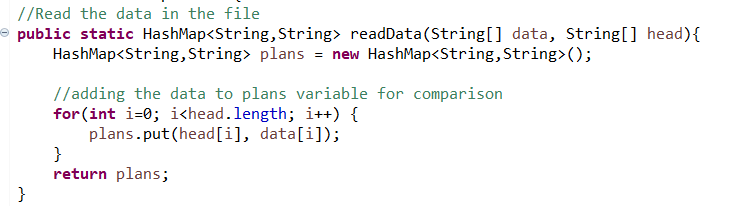
When user click the Data option, the page displays the top four plans based on data. Internally it calls the function comparePlansByData and when user clicks the Price option, the page displays the four plans based on Plan price and it internally calls the function comparePlansByCost. The both functions initially reads the data from all the csv files. Each row in csv file is stored in form of HashMap. Later listed all the plans using ArrayList. The datastructure for each plan:



The readPlans method reads all the plans from the csv files. And creates a list of plans using ArrayList.

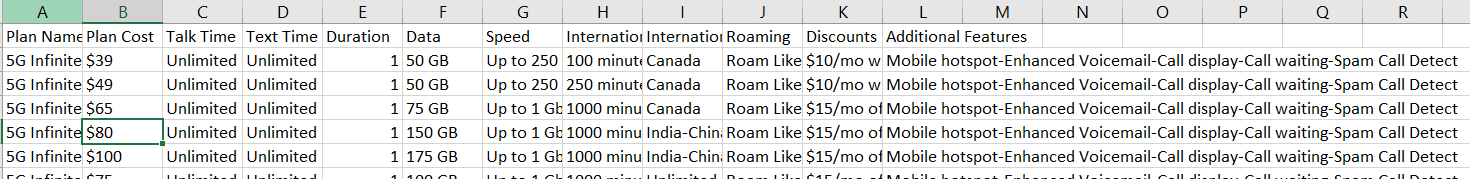


The readData function reads the data of the each plan present in CSV files and stores the data in HashMap. The HashMap contains <ColumnName, ColumnValue>.



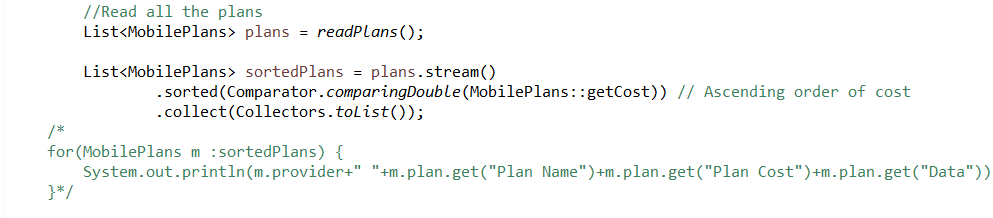
For e.g, the hashmap for the second plan in the below image is like

(“Plan Name”,” 5G Infinite” ),(“Plan Cost”,” $49 ”),(“Talk Time”,” Unlimited”),(“Text Time”,” Unlimited”),(“Duration”,”1”),(“Data”,” 50 GB”),(“Speed”,” Up to 250 Mbps”),(“International Minutes”,” 250 minutes”),(“International Countries”,”Canada”),(“Roaming”,” Roam Like Home e”),...

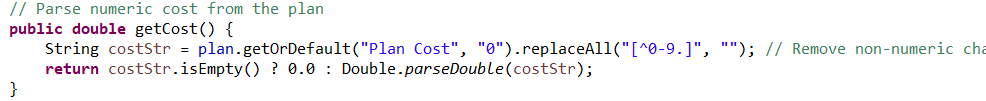


**By Price:**

Sorts the plans based on the price.

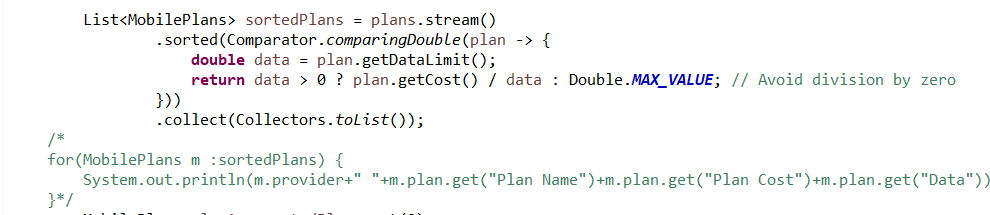


First processed the data using regular expression to get the acute cost of the plan

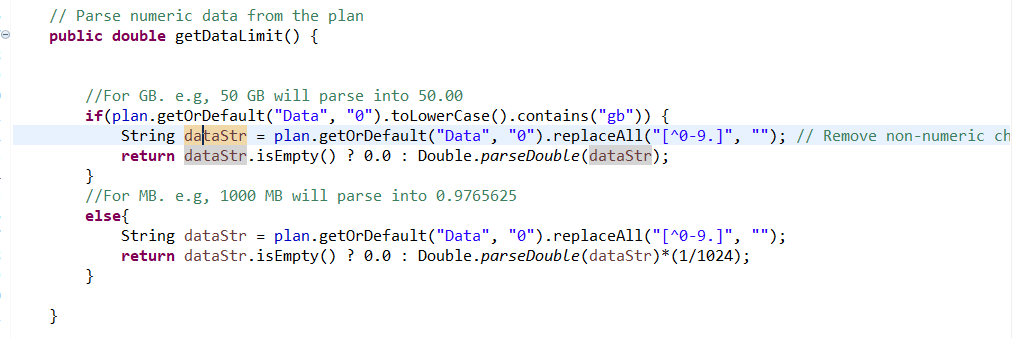


**By Data:**

Sorts the plans based on the Data by calculating the cost of each GB.

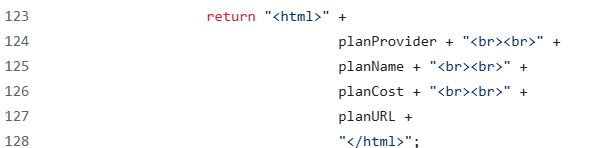


The getDataLimit method calculates the cost per each GB



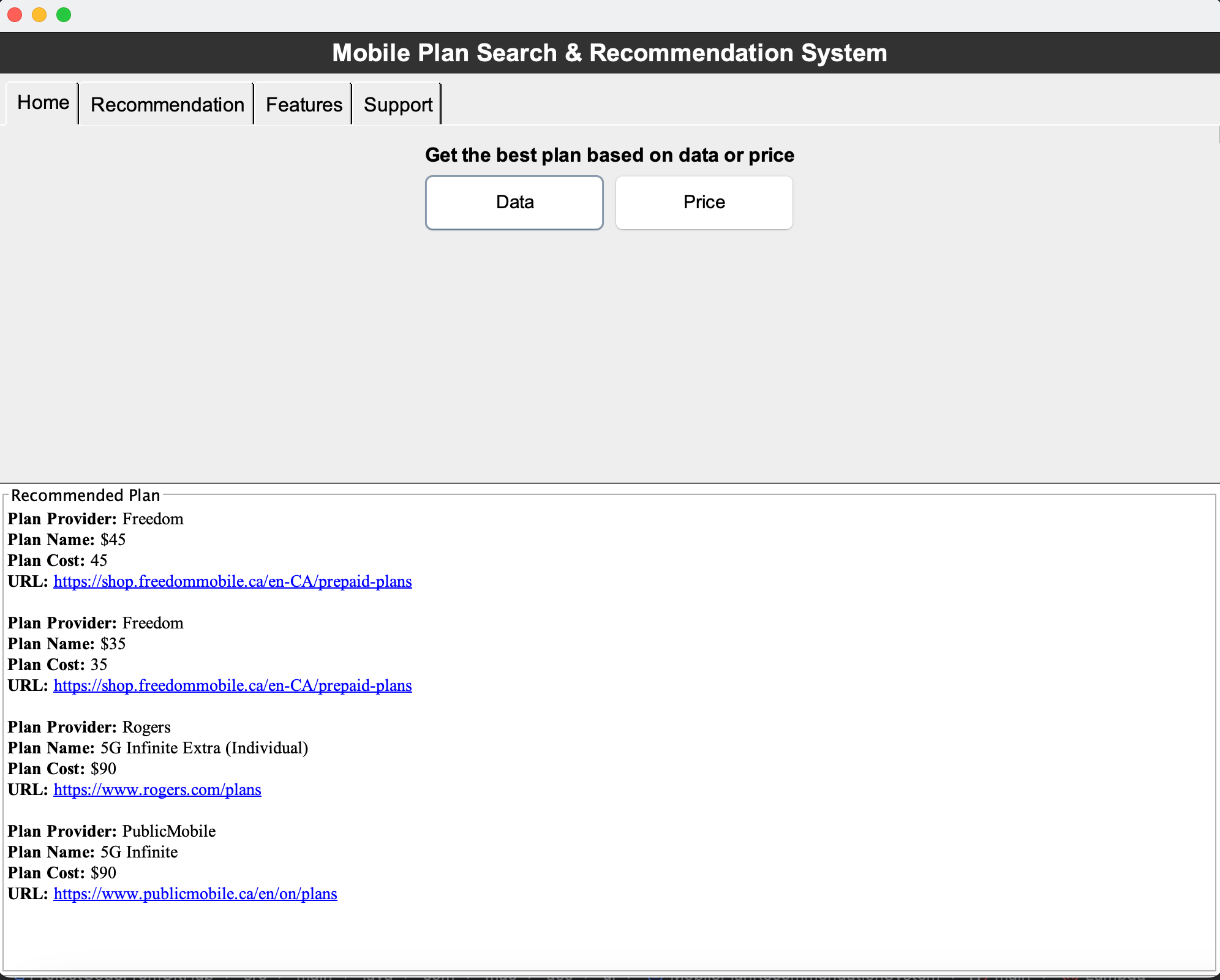
The top 4 plans are printed using HTML format to make more visible.





**Output screen:**

Comparing plans by Data

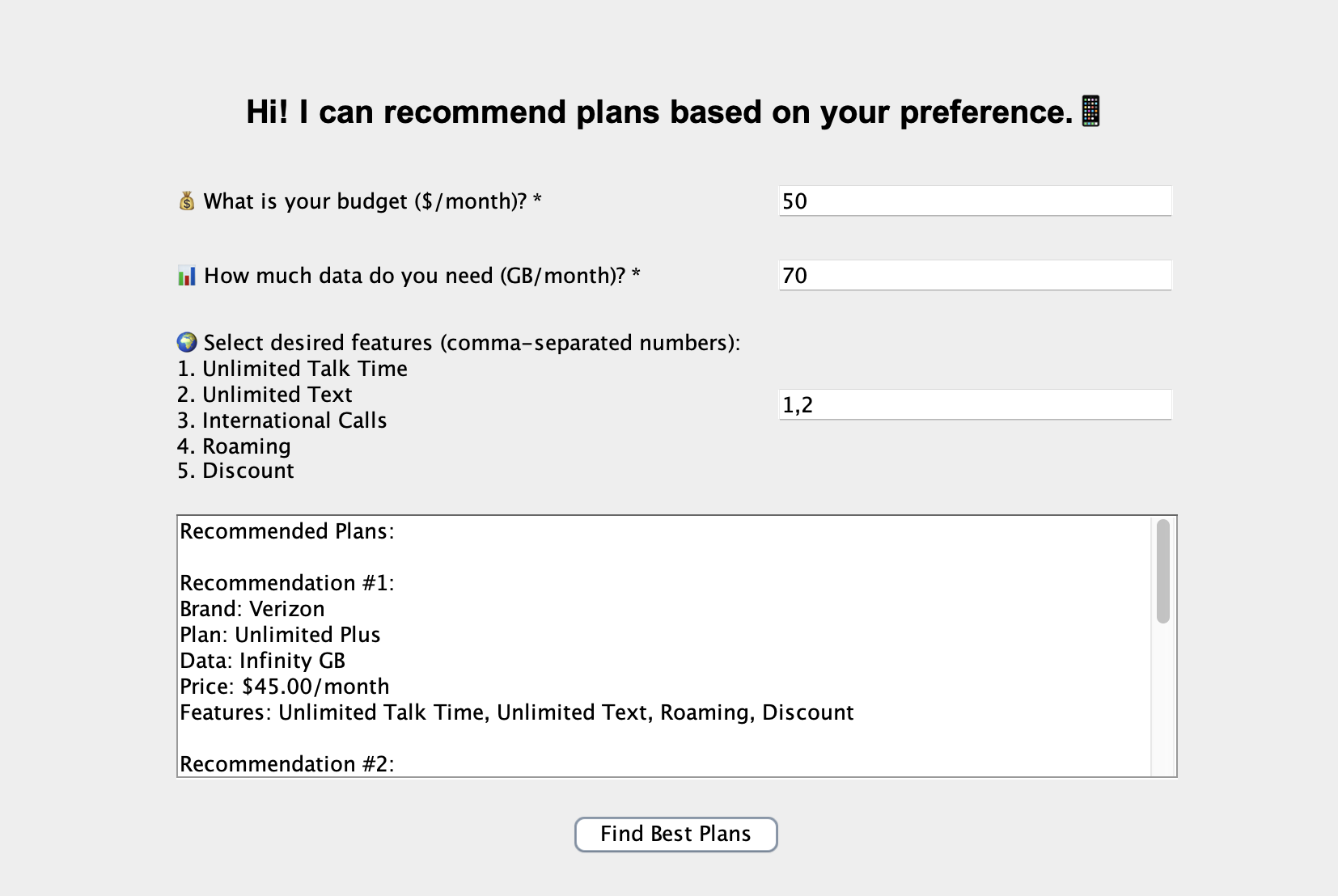


Comparing plans by Price



# Plan Recommendation

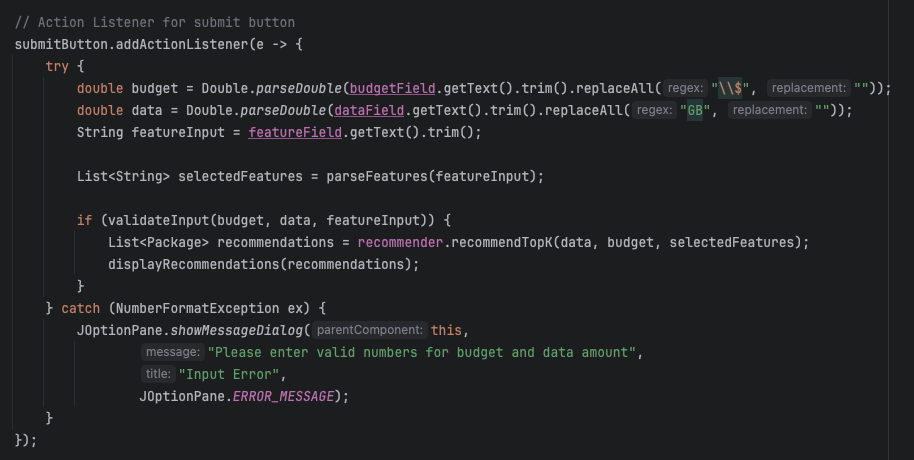
Plan recommendation is based on the information crawled from websites and the user’s preferences. Just put the budget, desired data and desired features, this function will give 3 best options. Through this function, user can find the best plan in their expectations.

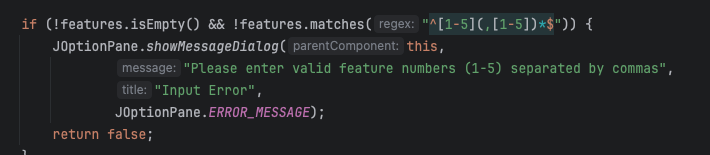


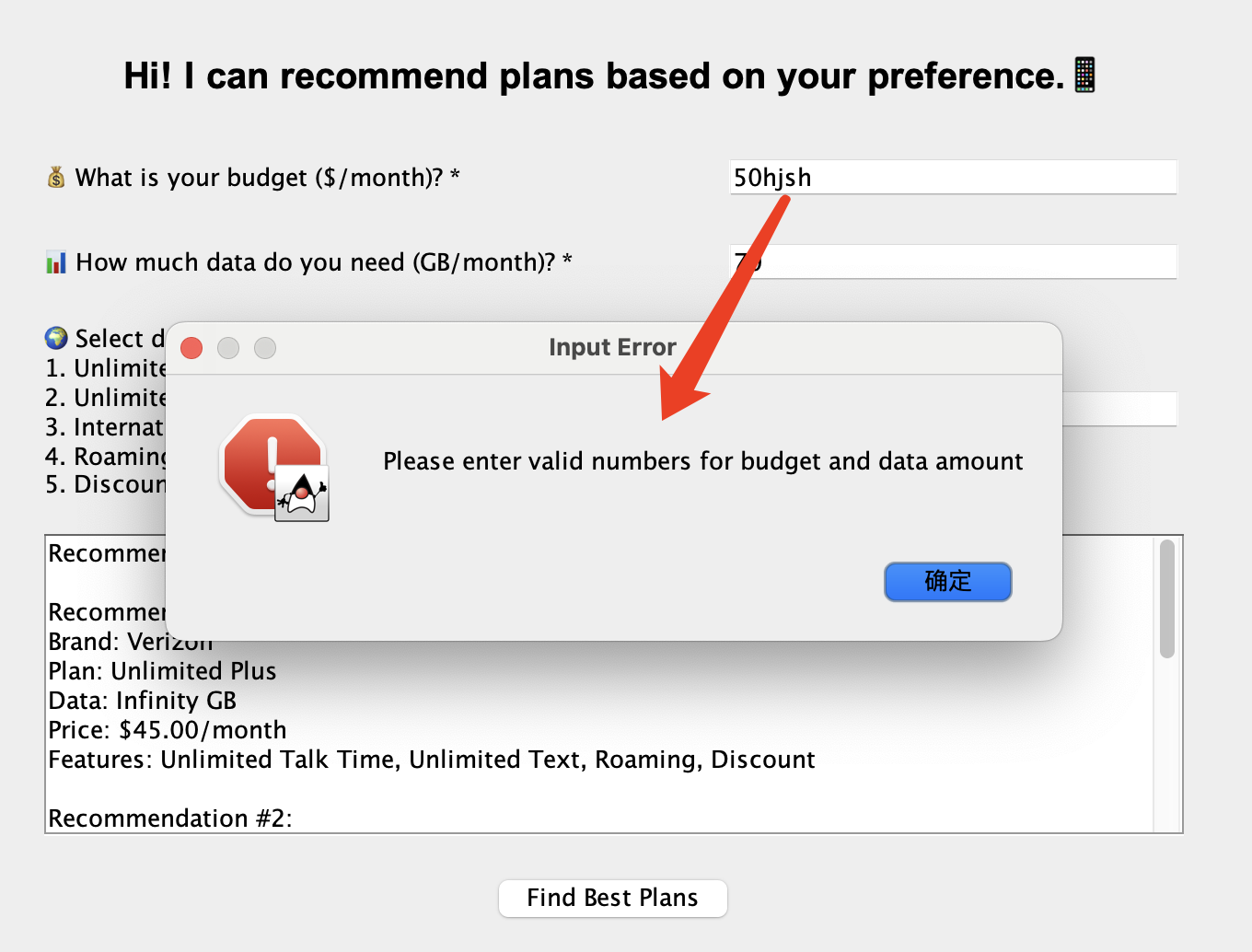
* Feature list (overview):

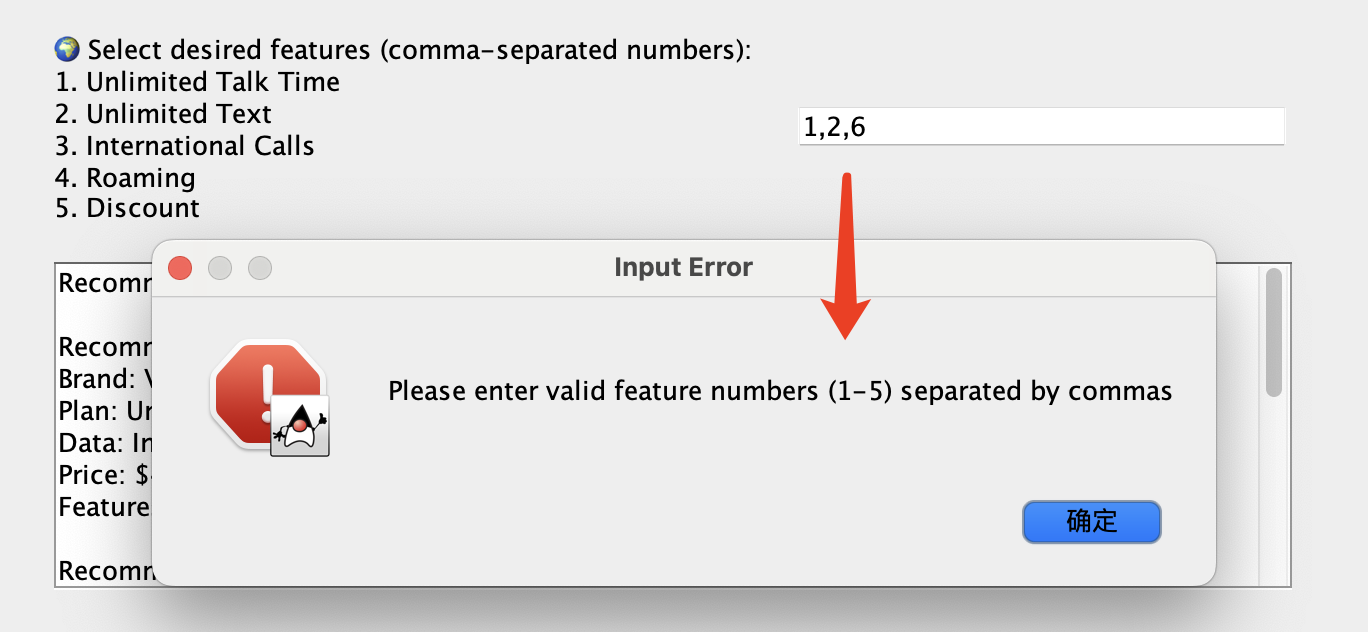
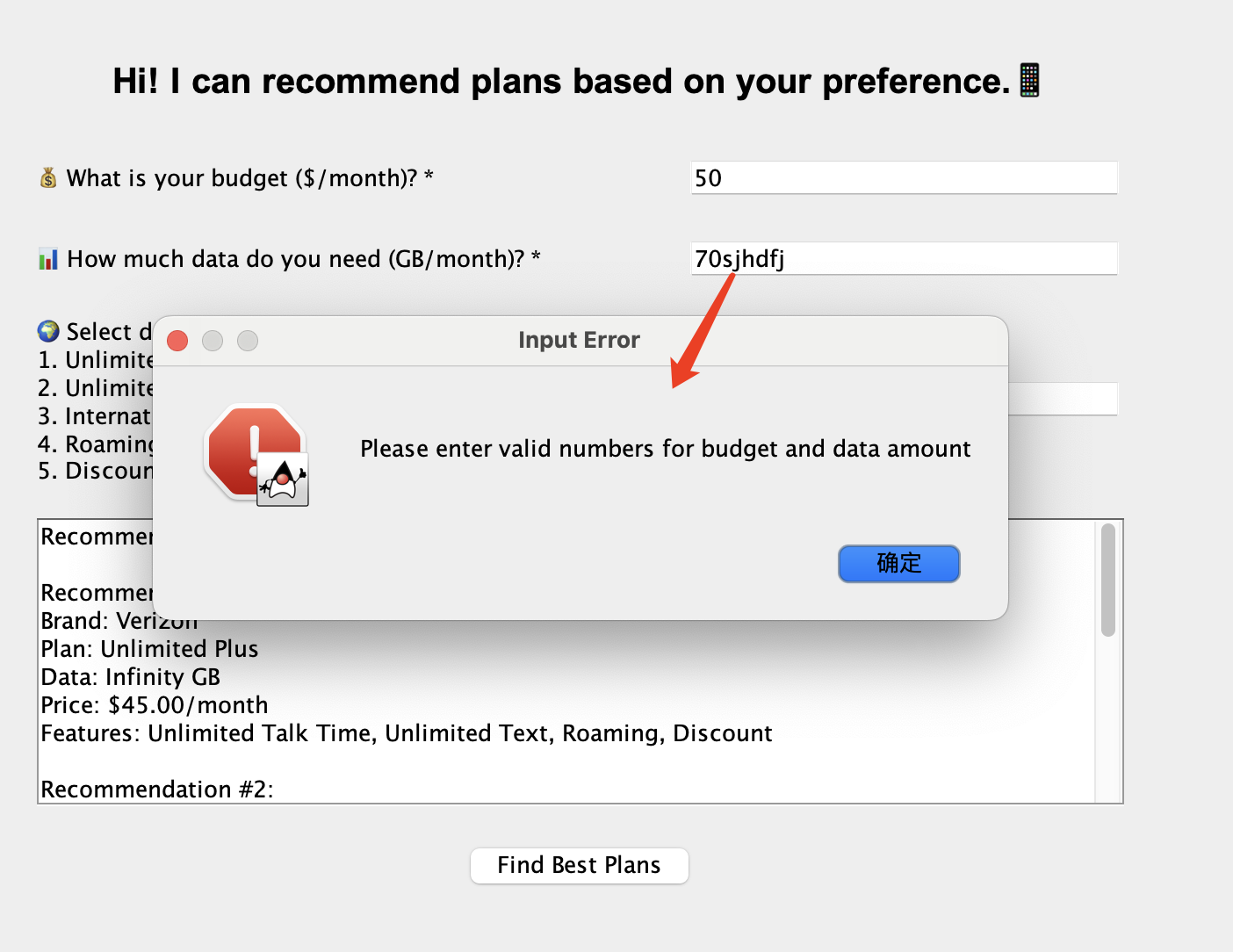
1. GUI interface (validate the info with **Regular Expressions**)
2. CSV file reading and processing(extract the info with **Regular Expressions**)
3. Score Algorithm

* Data Structure:
  + Hash map – store all the plan information
  + Priority queue – maintain the top 3 recommendations
* GUI interface:
  + Using swing to display the interface, we give the instructions of each question for what type info should be fill in and what is required(note with “ \* ” mark). At the same time, when click the “Find Best Plans” button, it will validate if the user input the invalid string or blank.
  + Validation1:Required field—budget. User can only input positive number with “$”(it can be placed both right or left side of the number, this mark is optional);
  + Validation2: Required field—data amount. User can only input zero or positive number with “GB |MB”(it can only be placed right side of the number, this mark is optional, default unit is GB);
  + Validation3: Optional field—features. We offered 5 features with serial number. User can only input number 1-5, different number must be separate with comma -- , ;

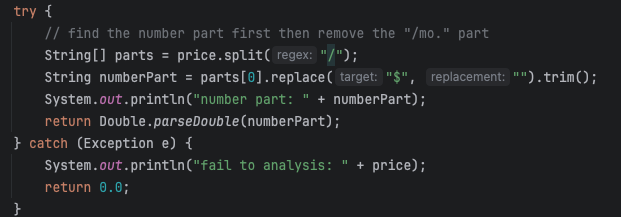
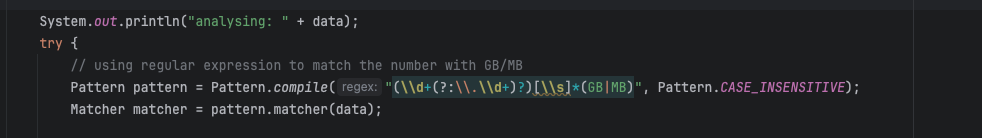
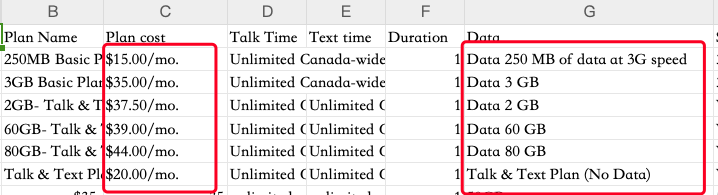








* CSV file reading and processing(extract the info with **Regular Expressions**)
  + This function reading csv file line by line, we only collect few columns to get the key information.
  + We also need to extract the key word from the raw scraped data (eg , “$15.00 / mo.”, “No data ”) . We need to trim all the unnecessary info and convert the special word into the number.



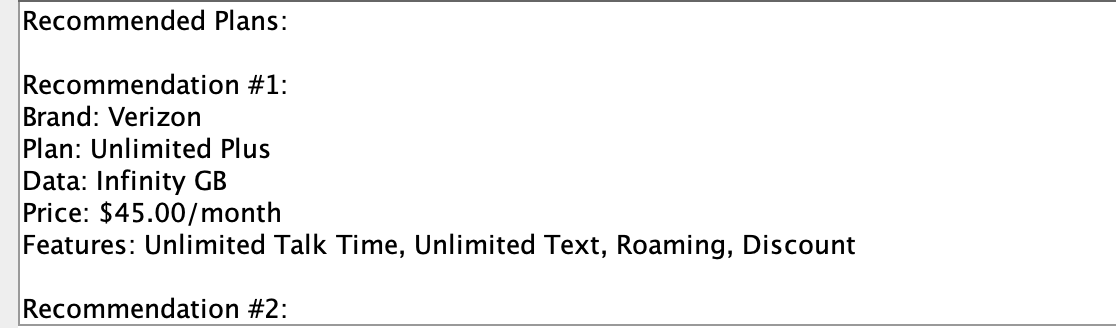
* Score Algorithms

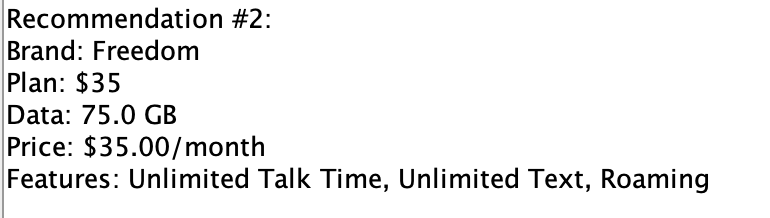
This algorithm decide every plans can get how many points.

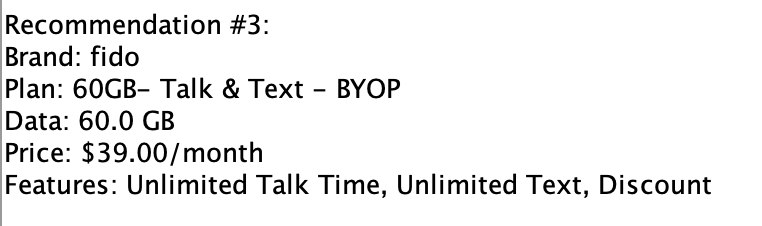
* + Step1. Pre-filter, if the plan’s cost is exceed the user’s budget or the data amout is lower than the 50% of the user’s preference, then it won’t be calculate.
  + Step2. Score calculation – price, every plan can get top to 3 points.

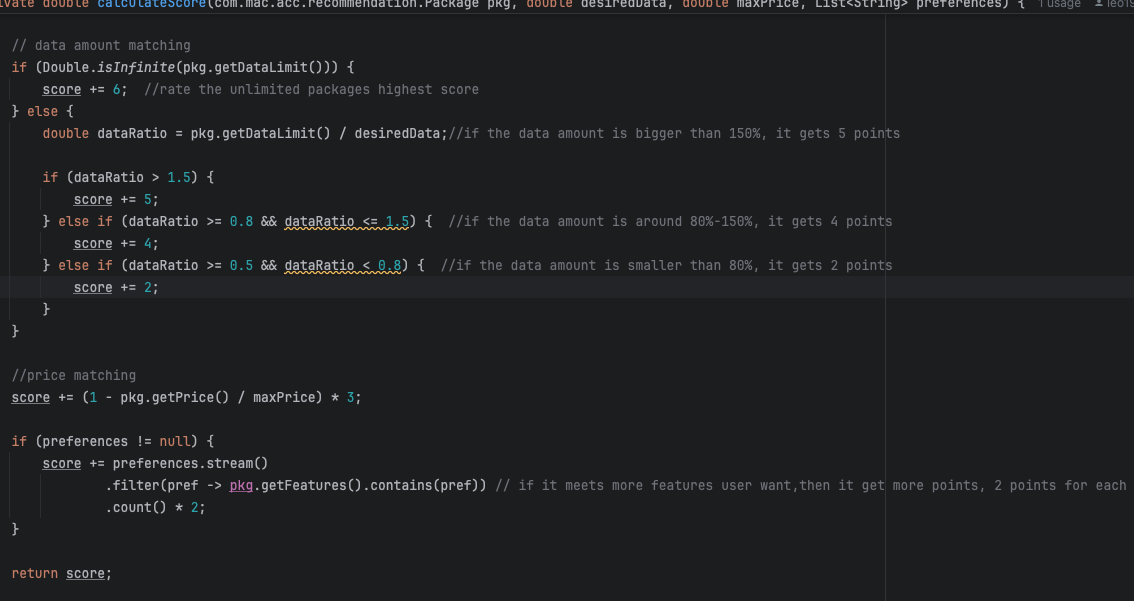
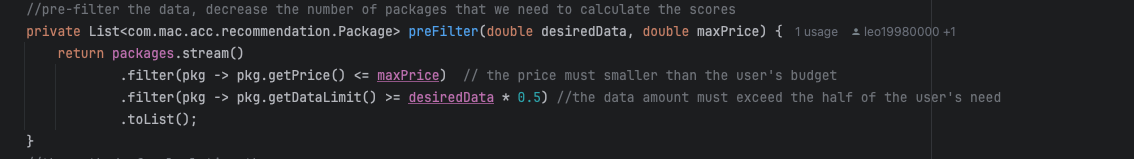
points = (1 – plan cost / budget) \* 3

* + Step3. Score calculation – data amount, every plan can get top to **6** points.
    - 1.unilimited data plan gets 6 points;
    - 2.plan cost >= 150%\* input, gets 5 points;
    - 3.150%\*input > plan cost >= 80%\* input, gets 4 points;
    - 4. 80%\*input > plan >= 50%\* input, cost,gets 2 points.
  + Step4. Score calculation – features, every plan can get top to 10 points. If the plan meet 1 feature that user wants, gets 2 points.
  + Step5. After adds all points up, we put it in the **priority queue** to maintain the top 3 highest plans, using this data structure can save time for sorting all the scores.





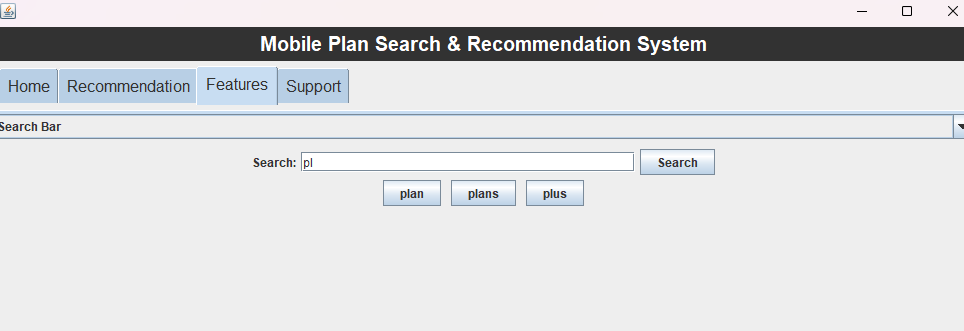




# Search Bar

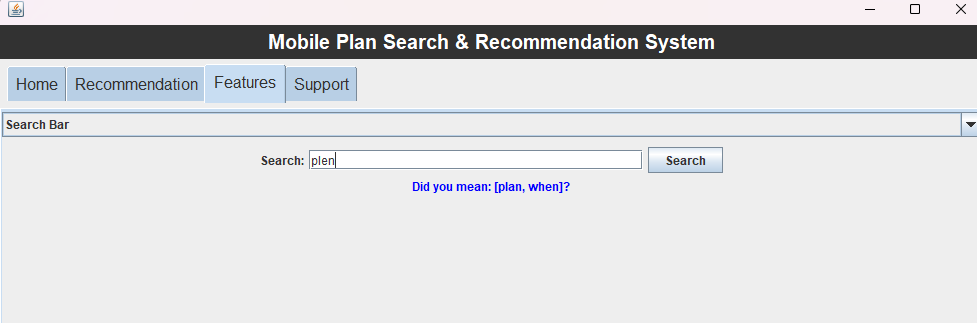
#### **1. Word Completions**

* Test autocompletion by providing a prefix.
  + **Action**: Call the method to retrieve completions for a given prefix (e.g., "app").
  + **Expected Output**: A list of suggested words starting with the prefix, like apple, application.



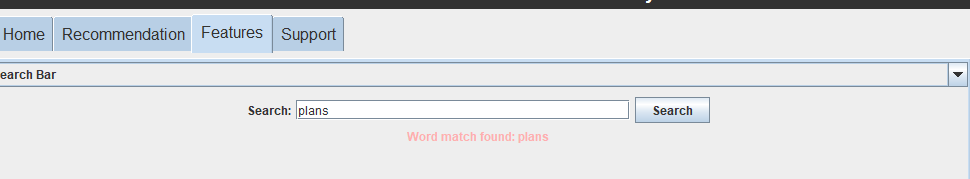
#### **2. Spell Corrections**

* Check for spelling suggestions when a word is misspelled.
  + **Action**: Provide a misspelled word and set the maximum edit distance (e.g., "aple", 2).
  + **Expected Output**: A list of suggested corrections, like apple, ample.



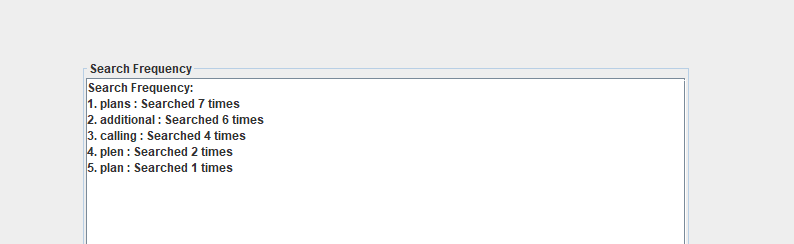
#### **3. Handle Search Logic**

* Simulate a search process:
  + **Steps in Search**:
    - Check if the input matches a word exactly.
    - Provide word completions if no exact match is found.
    - Suggest spelling corrections if there are no completions.
    - Display "No match found" if no results are available.
  + **Expected Output**: A message indicating the match type (e.g., Word Completions: [apple, application]).



#### **4. Track Search Frequency**

* View the most searched terms ranked by frequency.
  + **Action**: Retrieve the top searched terms after performing several searches.
  + **Expected Output**: A ranked list showing the terms and how many times they were searched.



### **Search Bar (Functions)**

### **Word Completion (getWordCompletions method)**

* Reads data from a CSV file (mobile\_plans.csv) line by line.
* Cleans the text by removing special characters and splitting it into individual words.
* Inserts the words into a **Trie data structure** (used for efficient prefix-based search).
* Suggests possible word completions for a given prefix by searching the Trie.

### **2. Spell Checker (findCorrections method)**

* Reads words from the same CSV file and inserts them into a **Spell Checker Trie**.
* If the input word exists in the Trie, it's considered correct, and no corrections are needed.
* If the word doesn't exist, it suggests similar words based on the maximum allowed edit distance (e.g., number of changes to transform one word into another).

### **3. Search Logic (handleSearch method)**

* Combines word completion, spell checking, and search frequency tracking:
  + **Step 1:** Checks for an exact match using word completion.
  + **Step 2:** If no exact match, shows possible word completions.
  + **Step 3:** If no completions, checks for spelling corrections.
  + **Step 4:** If no matches or corrections, informs the user.

### **4. Search Frequency Tracking (SearchFrequencyQuery object)**

* Tracks how often specific search queries are made.
* Updates the frequency whenever a search is performed.
* Retrieves the **top 10 most-searched queries** for display.

### **5. Display Most Frequent Searches (displaySearchFrequency method)**

* Formats and displays the most frequently searched terms along with their counts.

### **6. Word Completion as You Type**

* Monitors the search field using a **DocumentListener**:
  + Automatically shows word suggestions based on the input prefix.
  + Updates and displays suggestions dynamically as the user types or deletes characters.

### **7. Refreshing**

* Dynamically updates the panels (revalidate and repaint) to ensure new data is displayed correctly.

### **8. Event Listeners**

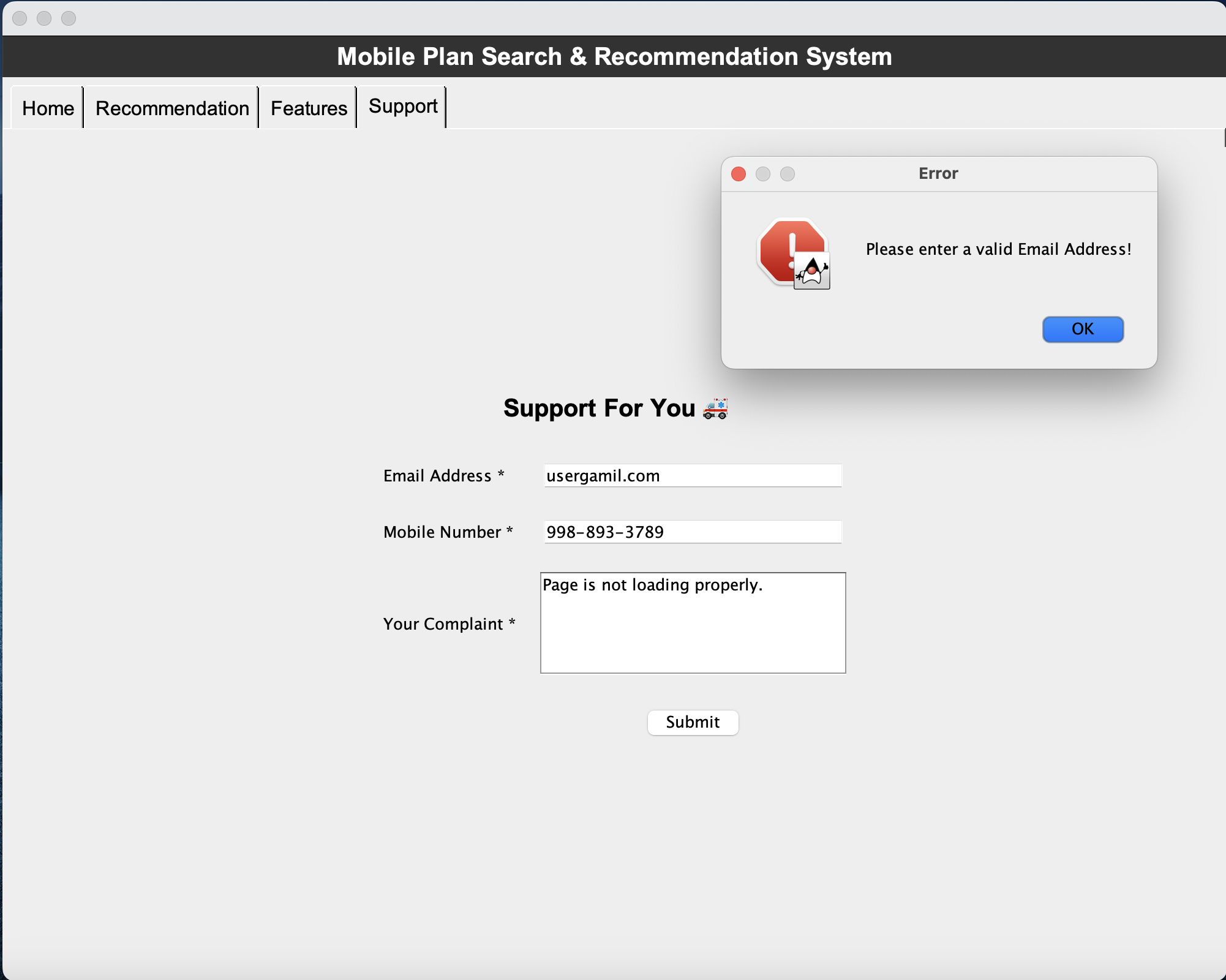
* Adds interactivity using:
  + **DocumentListener:** To handle typing in the search field.
  + **ActionListener:** To perform a search on button click.
  + **KeyListener:** To handle the Enter key for search

# Regular Expression​ Validation

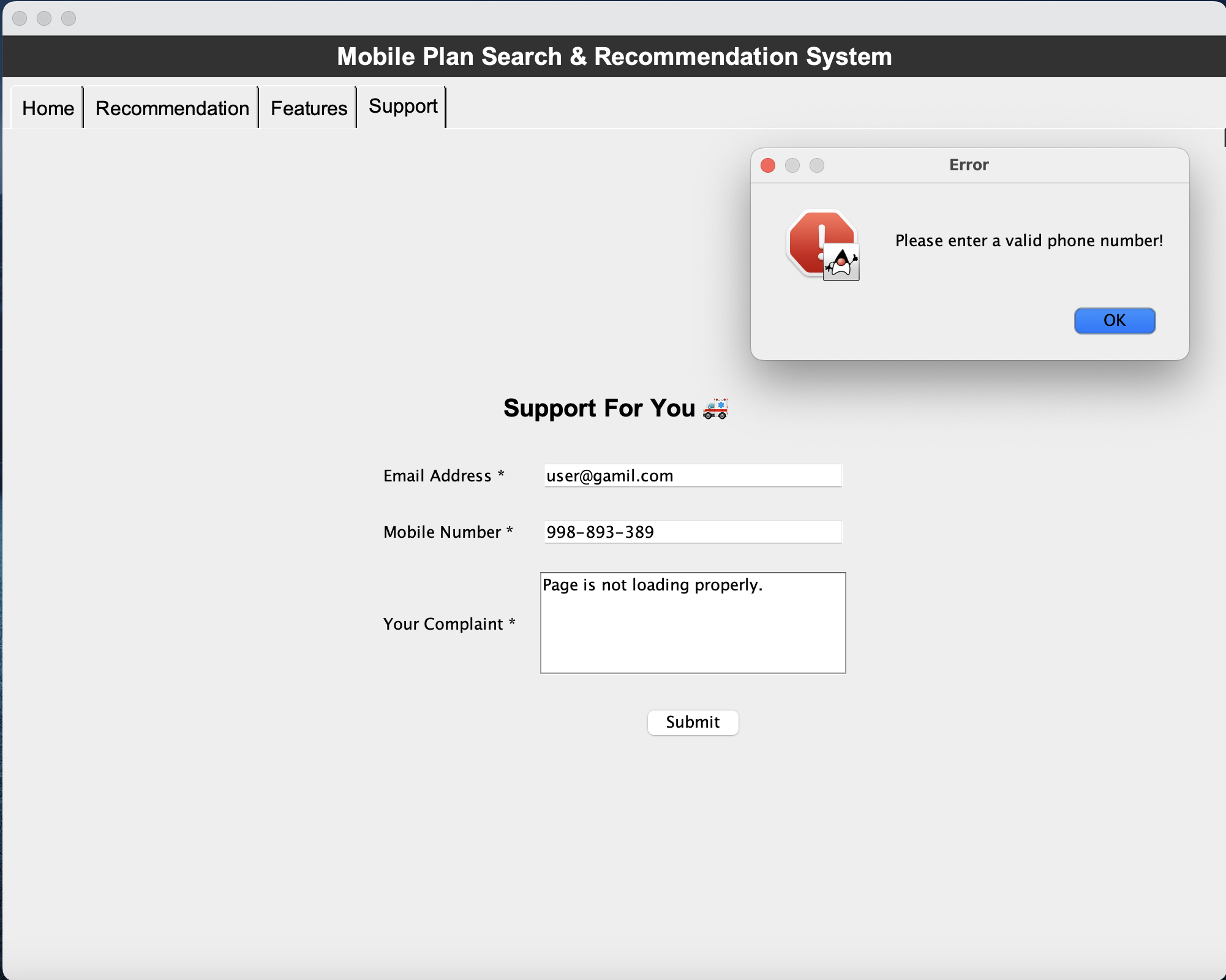
Support Page makes use of regular expression validation for both email addresses and mobile numbers, ensuring accurate data input.

**Support Page Features:**

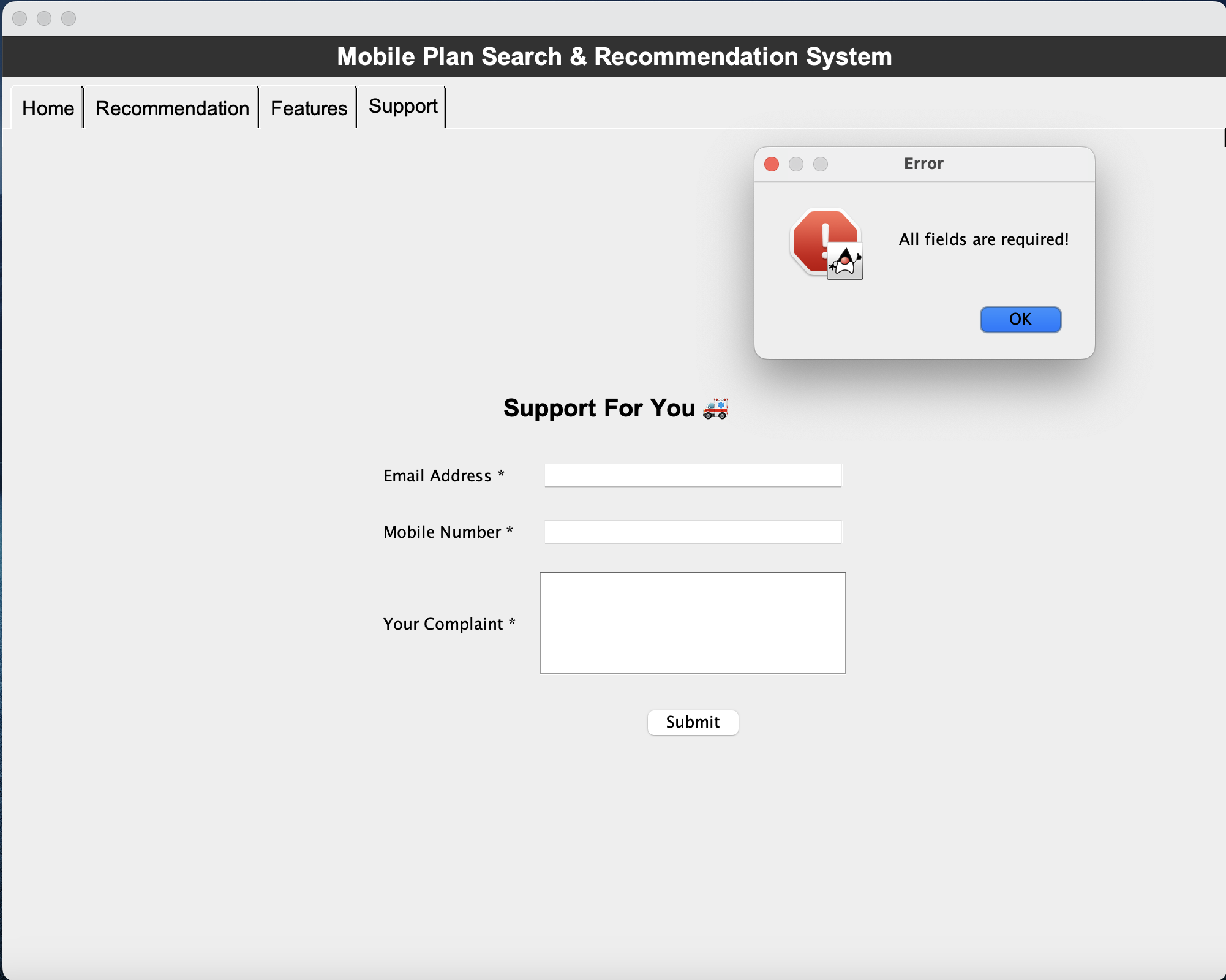
* **User Input Validation for Submitting Complaints**:
  + The system ensures that all user input is correctly validated before submission. This includes validating email addresses, phone numbers, and ensuring all required fields are filled.
* **Email Address Validation**:
  + Utilizes regular expressions (regex) to validate the format of the email address, ensuring it follows a standard format like [user@domain.com](mailto:user@domain.com).
  + Regular Expression for email address: [a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z]{2,}
  + Example formats include: [user123@example.com](mailto:user123@example.com) , [user.name+tag@example.co.uk](mailto:user.name+tag@example.co.uk) , [user@sub-domain.example.com](mailto:user@sub-domain.example.com) , [user\_name@example.net](mailto:user_name@example.net) , [user%email@example.io](mailto:user%25email@example.io) .



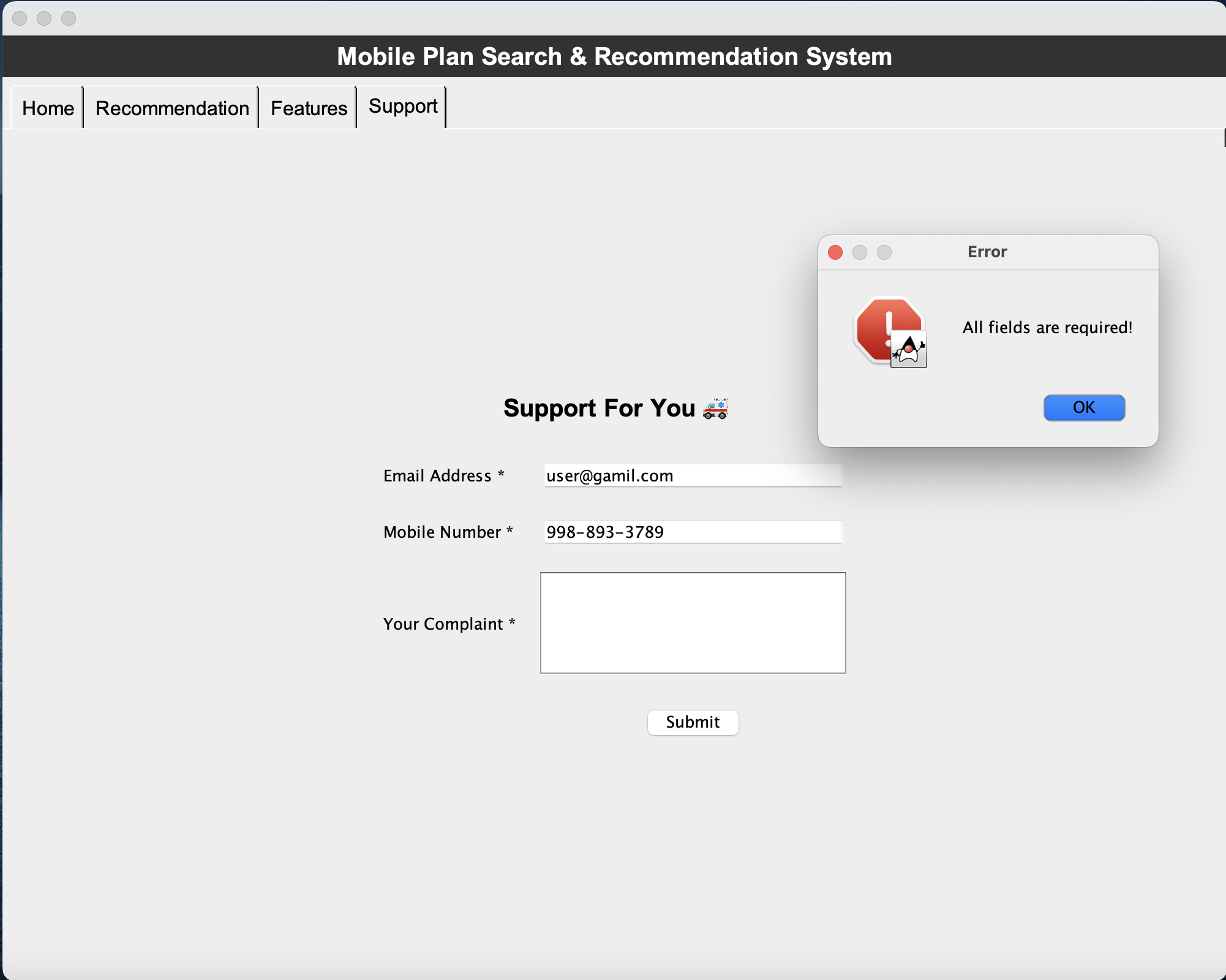
* **Mobile Number Validation**:
  + Regular expressions are also used to validate phone numbers, supporting various formats such as country codes, parentheses, and spaces.
  + Regular expression for phone number: ^(\+\d{1,2}\s?)?1?\-?\.?\s?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}$
  + Example formats include: 1234567890, +918234567890, +91 8234567890, (123) 456-7890, +1 123 456 7890, 1-800-555-1234
  + Allows separators like spaces, dashes, dots, or parentheses.



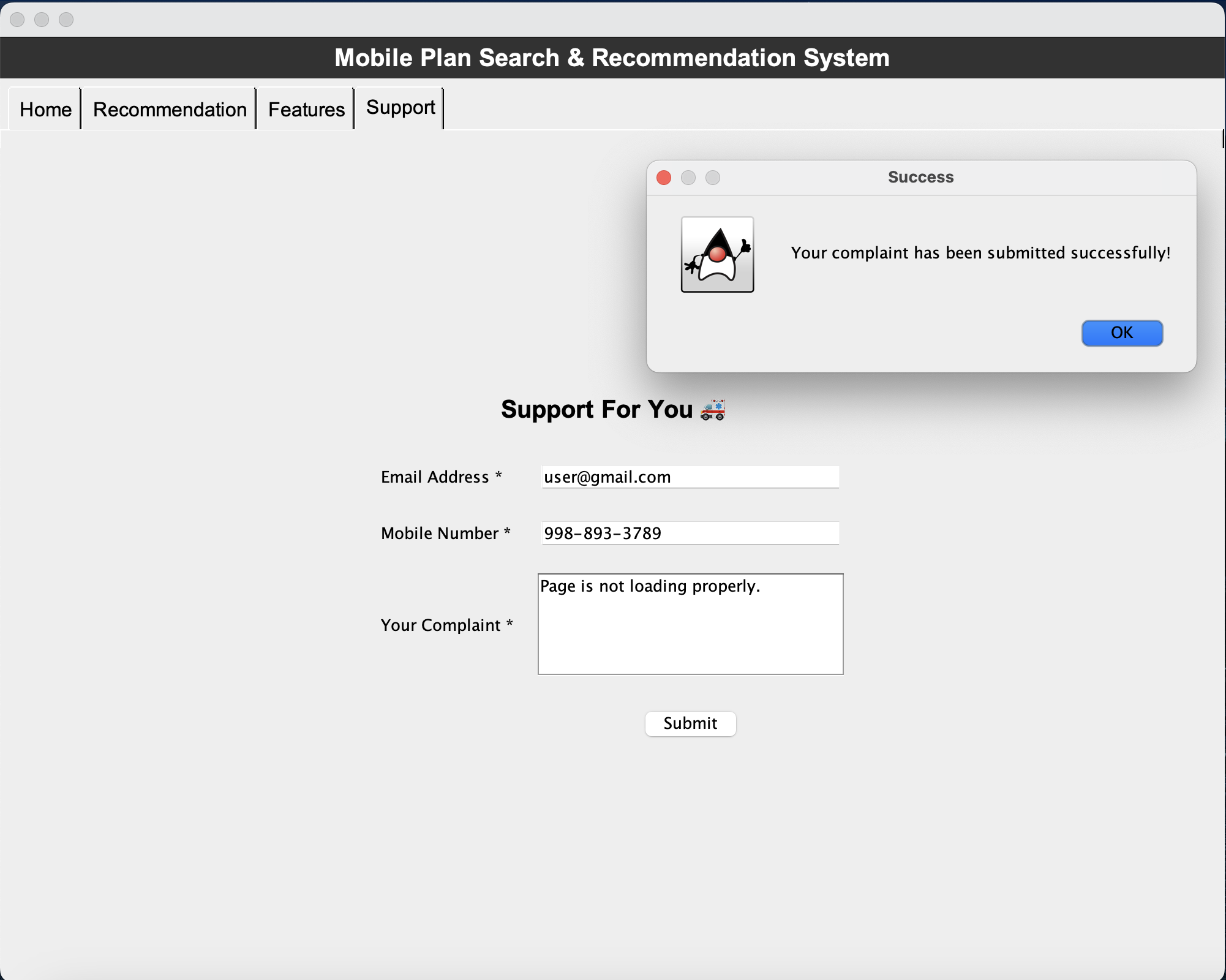
* **Empty Field Validation**:
  + Before submission, the system checks that all fields (email address, mobile number, and complaint) are filled out. If any of the required fields are empty, an error message is displayed.



* **Error Alerts**:
  + If the user fails to fill out any field or if the validation fails (e.g., incorrect email or phone number format), an alert is shown indicating the issue. This helpsx guide the user to correct any errors before submission.



* **Labeling Required Fields**:
  + The fields, such as "Email Address \*", are clearly labeled to indicate they are required, making the form user-friendly and easy to understand.
* **Storing Complaint Data**:
  + Once the form is successfully validated, the complaint data (email, phone number, and complaint text) is written to a file named complaints.txt. This stores the complaint information for further processing or record-keeping.

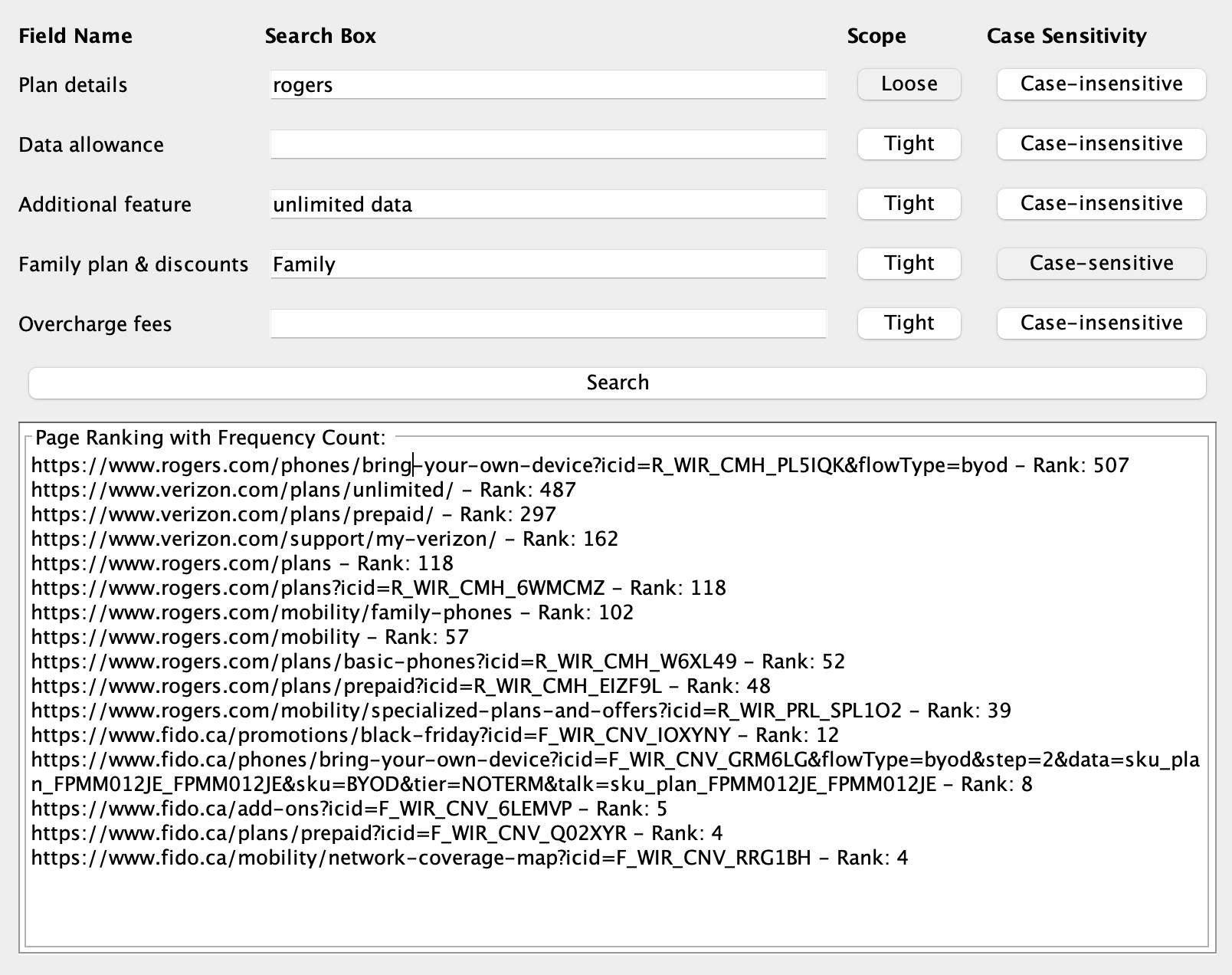


# Advanced Page Searching

## Overview and Design Philosophy

This section examines a specialized search engine component designed to process multi-field searches with varying match conditions. The system's key innovation lies in its ability to handle multiple search fields with different constraint levels, allowing for both mandatory and optional search criteria to be combined in a single query.

The search engine implements two types of field conditions: tight and loose. A tight field condition imposes a mandatory constraint - any document in the final result set must contain all words specified in tight fields. For example, if a search includes a tight field containing "unlimited data", every document in the results must contain both "unlimited" and "data". Loose fields, however, provide a mechanism to include additional relevant documents even if they don't satisfy all tight constraints. Documents containing words from loose fields will appear in the results even if they don't match all tight field criteria. This allows users to find documents that either strictly match their primary requirements (tight fields) or contain related terms of interest (loose fields).



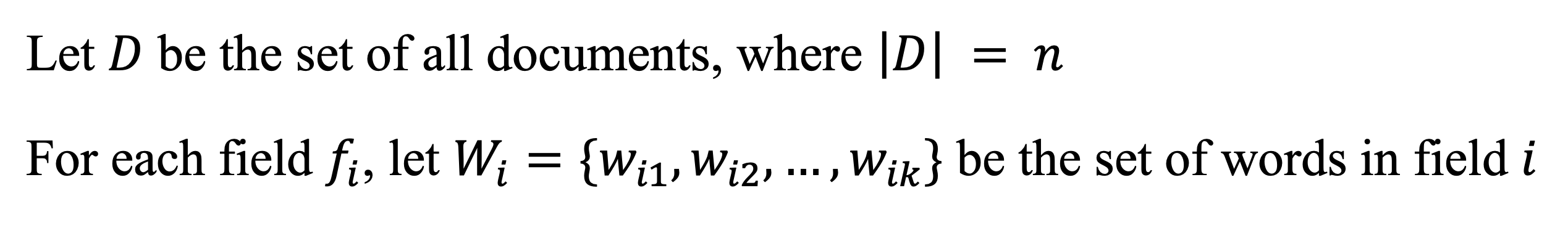
## System Architecture and Implementation

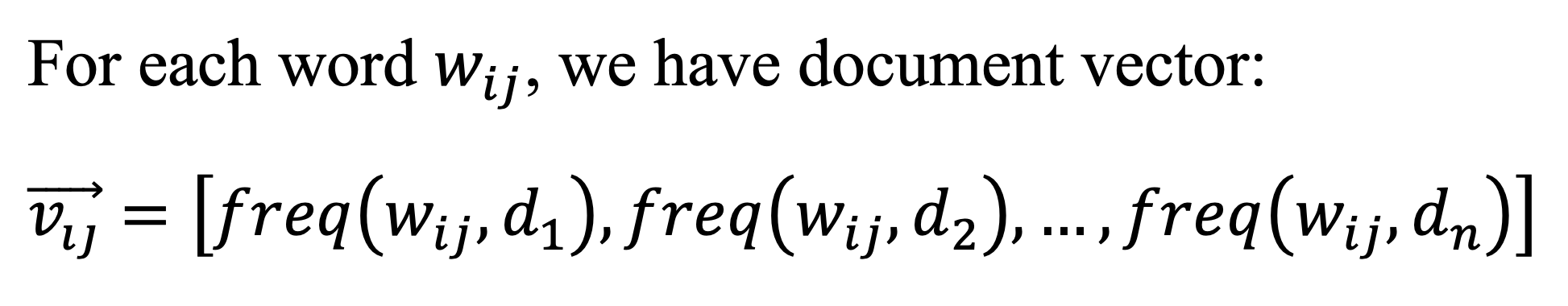
The search engine's architecture consists of an integrated pipeline that processes documents and queries through several stages. Initially, the document processor handles HTML retrieval and parsing, converting web content into processable text. This text feeds into a Trie-based inverted index, which maintains efficient word-to-document mappings with frequency information. The core search component then uses these indices along with specialized vector operations to compute document relevance while respecting the tight and loose field constraints.

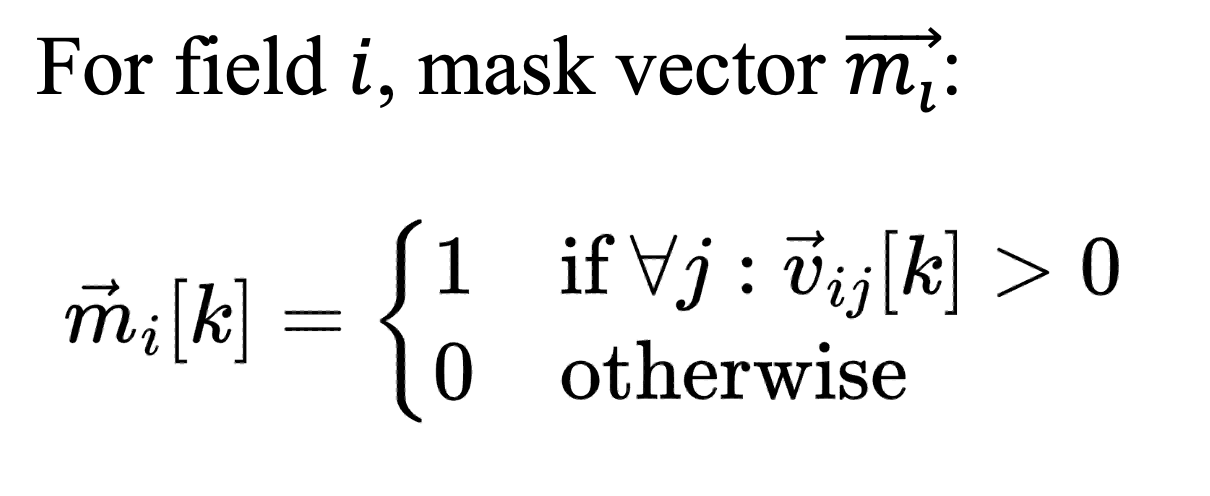
The implementation uses a dual-index approach, maintaining both case-sensitive and case-insensitive indices to support flexible matching options. Document processing employs JSoup for robust HTML handling, while the search algorithm utilizes vector operations to combine results from different fields according to their conditions.

## Mathematical Model and Search Algorithm

The search algorithm follows a rigorous mathematical model that combines vector operations to respect field conditions. The formal definitions are:

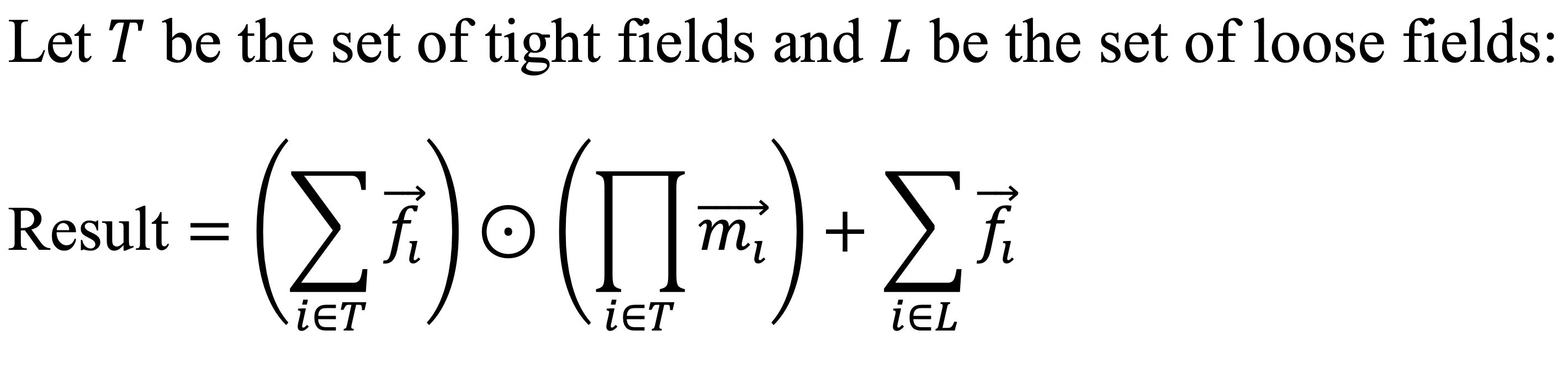






A math equation with numbers and symbols

Description automatically generated



This formula ensures that documents must match all tight field conditions while allowing additional matches from loose fields to contribute to the final relevance score.

## Performance and Error Handling

The search implementation balances algorithmic efficiency with constraint satisfaction. Trie operations maintain complexity for word length L, while vector operations scale linearly with document count N. The overall search complexity of for W total words ensures responsive performance even with multiple fields. Space complexity follows , scaling with both the character set size and document count.

Error handling encompasses document validation, search field validation, and constraint checking, with a structured exception hierarchy providing clear error identification and recovery paths.

## Conclusion

The search engine component successfully implements a flexible yet precise approach to multi-field document search. By distinguishing between tight and loose field constraints, it enables users to specify both mandatory criteria and optional relevant terms. The mathematical foundation ensures accurate results while the implementation provides practical features like case sensitivity options and relevance scoring based on both strict and relaxed matching criteria.