Nelson Tran

SID# 862332246

Email: ntran189@ucr.edu

March 20, 2025

Project 2 for CS 170 Winter 2025, with Dr. Eamonn Keogh.

All code is original, except:

1) I used predefined subroutines in the standard C++ library, such as:
    - All subroutines used from **iostream**, for standard input and output.
    - All subroutines used from **fstream**, for file handling.
    - All subroutines used from **sstream**, for string stream operations.
    - All subroutines used from **vector**, for using vectors (type of set).
    - All subroutines used from **string**, for string operations.
    - All subroutines used from **cmath**, for mathematical functions.
    - All subroutines used from **limits**, for handling infinite values.
    - All subroutines used from **algorithm**, for the “find” function.
2) I used the C++ 11, 14, 17, 20, 23, and 26 Documentation for trivial structure and implementation. This is the URL to the aforementioned C++ versions’ table of contents: https://en.cppreference.com/w/.
3) I consulted the "Machine Learning" lecture slides, and the "Project_2_Briefing", "Project2_Winter_2025", and "Project_2_sample_report" prompt slides by Dr. Eamonn Keogh, in addition to notes annotated from his lectures, for feature selection implementation, such as computing the Euclidean distance.

This report will go over the second assigned project in Dr. Eamonn Keogh's Introduction to Artificial Intelligence (CS170) course taken at the University of California, Riverside during the academic quarter of Winter 2025. The full code for this assignment is included at the bottom of this report. In this project, we are tasked with performing a feature selection on two datasets using forward selection search and backward elimination search.

In Figure 1, we see the result of running forward selection on CS170_Small_Data__36.txt, which is the small file assigned to me.

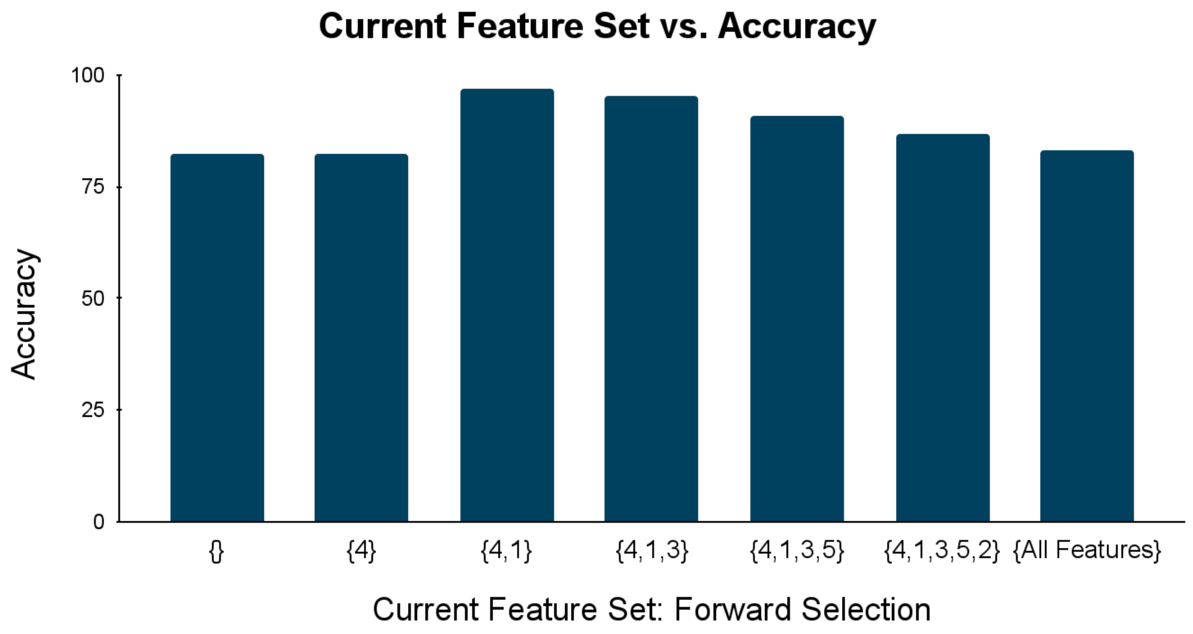

Figure 1: Accuracy of increasingly large subsets of features discovered by forward selection.

At the beginning of the search, we have no features (as shown with {}) so I reported the default rate, which was 82.4%. Adding feature '4' surprisingly did not change the accuracy, keeping it at 82.4%, but then adding feature '1' gave us a noticeably higher accuracy of 96.8%. Now if we add feature '3', the accuracy decreases by a small amount, to 95.4%. The minor decrease suggests that the subset of features ‘4’ and ‘1’ may be the set of useful features in this dataset. From here, each feature being added to our current set reduces the accuracy, until we have the full set of features, which gives us an accuracy of 83.2%.

Next, as shown in Figure 2, I ran backward elimination on CS170_Small_Data__36.txt.

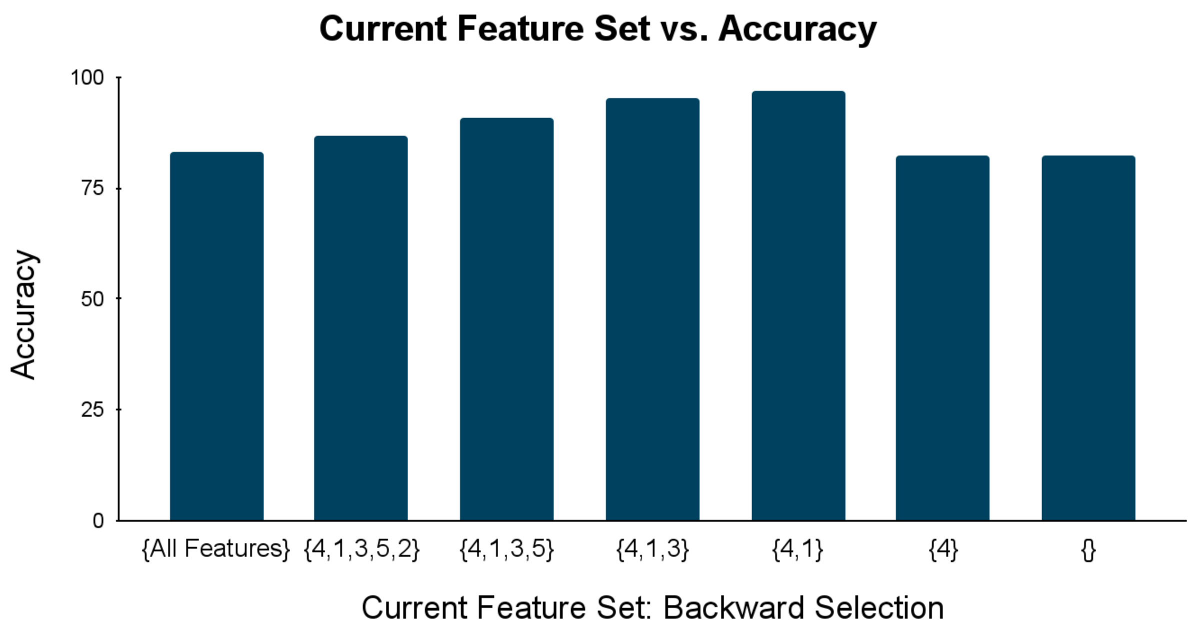

Figure 2: Accuracy of increasingly small subsets of features discovered by backward selection.

At the beginning of the search, we have all the features in our subset, which has an accuracy of 83.2%. That accuracy is the same accuracy as the last tested subset in forward selection, which makes sense considering that they are the same subset from the same set. Similarly, the last tested subset with no features has an accuracy of 82.4%, which is the same accuracy as the first tested subset in forward search. Now going back to the beginning with the subset with all features, we remove feature ‘6’, increasing our accuracy to 86.8%. From here, we keep removing features with each one increasing our accuracy until we get our highest accuracy of 96.8% with features ‘1’ and ‘4’ left in our subset. Removing feature ‘1’ drastically reduces our accuracy to 82.4%. And similar to what occurred in forward search, removing feature ‘4’ surprisingly does not affect our accuracy, keeping it at 82.4% for our empty set as aforementioned. Interestingly, backward elimination is just as accurate as forward selection for this particular problem. Also, both searches show that features '1' and '4' are really good features when together in the same feature subset.

Conclusion For Small Dataset: I believe that features ‘1’ and ‘4’ are the best features for this problem. There is very weak evidence that feature ‘3’ might be useful, but that needs further investigation. If we deploy this {1,4} model, I believe the accuracy will be about 96.8%.

Now, we turn our focus to the more challenging dataset I was tasked with investigating. In Figure 3, we see the result of running forward selection on CS170_Large_Data__28.txt, which is the large file assigned to me.

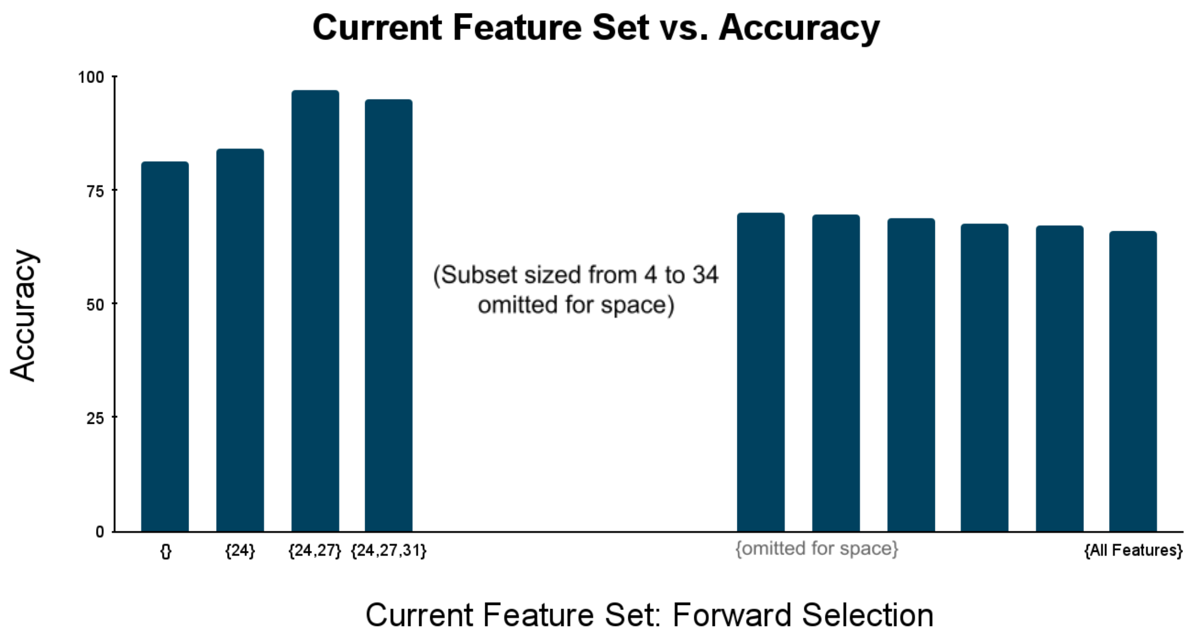

Figure 3: Accuracy of increasingly large subsets of features discovered by forward selection.

At the start of the search, we have no features (as shown with {}) so I reported the default rate, which was 81.2%. Adding feature '24' slightly increased the accuracy to 84.3%, and then adding feature '27' significantly increased the accuracy to 96.9%. From here and onwards, each feature being added to our current set reduces the accuracy, until we eventually have the full set of features that gives us an accuracy of 65.9%. Next, as shown in Figure 4, I ran backward elimination on CS170_Large_Data__28.txt.

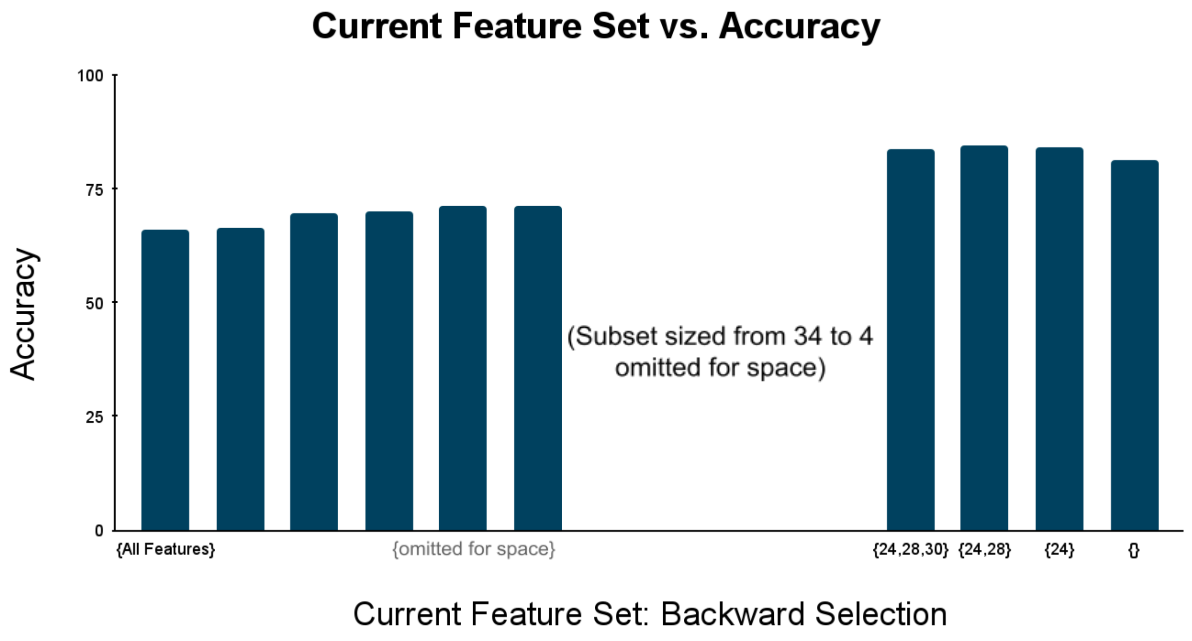

Figure 4: Accuracy of increasingly small subsets of features discovered by backward selection.

At the start of the search, we have all the features in our subset, which has an accuracy of 65.9%. Just like with the small dataset, the first tested subset with all the features in the large dataset having the same accuracy as the last tested subset in forward selection makes sense considering that they are the same subset from the same set. Similarly, the last tested subset with no features has an accuracy of 81.2%, which is the same accuracy as the first tested subset in forward search. Starting from the beginning, we remove features one at a time, with each feature being removed slightly increasing the accuracy little by little. This trend continues until we get our highest accuracy of 84.5% with features ‘24’ and ‘28’. Removing feature ‘28’ barely decreases the accuracy down to 84.3%, and finally, removing the last feature ‘24’ gives us the aforementioned accuracy of 81.2% for the empty subset. For this problem, backward elimination is not as accurate as forward selection. However, because it does find feature '24' as its best feature subset, it gives us confidence that feature '24' is a really good feature in the large dataset.

Conclusion For Large Dataset: I believe that features ‘24’ and ‘27’ are the best features for this problem. There is very weak evidence that feature ‘31’ might be useful, but that needs further investigation. If we deploy this {24,27} model, I believe the accuracy will be about 96.9%.

Computational Effort for Search: I implemented the search in C++ (version 17) on Visual Studio Code, and ran all experiments on a laptop with an Intel Core i7-1255U and 16 gigs of main memory. In table 1, I report the running time for the four searches I conducted.

	Small dataset (6 features, 500 instances)	Large dataset (40 features, 1000 instances)
Forward Selection	0.8 seconds	15.6 minutes
Backward Elimination	0.8 seconds	13.7 minutes

Table 1: Runtimes of forward selection search and backward elimination search on both datasets.

Note that for the large dataset, backward elimination was a little faster than forward selection. Overall, both methods provided pretty consistent results in identifying the most relevant features for the datasets.

Below I show a single trace of my algorithm. I am *only* showing Forward Selection on the small dataset.

```
Welcome to Nelson Tran’s Feature Selection Algorithm.
Type the name of the file to test: CS170_Small_Data__36.txt

Type the number of the algorithm you want to run.
     1) Forward Selection
     2) Backward Elimination

1

This dataset has 6 features (not including the class attribute), with 500 instances.
Running nearest neighbor with all 6 features, using "leaving-one-out" evaluation, I get an
accuracy of 83.2%
Beginning search.
     Using feature(s) {1} accuracy is 73.2%
     Using feature(s) {2} accuracy is 70.6%
     Using feature(s) {3} accuracy is 75%
     Using feature(s) {4} accuracy is 82.4%
     Using feature(s) {5} accuracy is 70%
     Using feature(s) {6} accuracy is 73%
Feature set {4} was the best, accuracy is 82.4%
     Using feature(s) {1,4} accuracy is 96.8%
     Using feature(s) {2,4} accuracy is 85%
     Using feature(s) {3,4} accuracy is 81%
     Using feature(s) {5,4} accuracy is 83.4%
     Using feature(s) {6,4} accuracy is 82%
Feature set {1,4} was the best, accuracy is 96.8%
     Using feature(s) {2,1,4} accuracy is 93.2%
     Using feature(s) {3,1,4} accuracy is 95.4%
     Using feature(s) {5,1,4} accuracy is 91.8%
     Using feature(s) {6,1,4} accuracy is 91%
(Warning, Accuracy has decreased! Continuing search in case of local maxima)
Feature set {3,1,4} was the best, accuracy is 95.4%
     Using feature(s) {2,3,1,4} accuracy is 89.4%
     Using feature(s) {5,3,1,4} accuracy is 90.8%
     Using feature(s) {6,3,1,4} accuracy is 87.6%
(Warning, Accuracy has decreased! Continuing search in case of local maxima)
Feature set {5,3,1,4} was the best, accuracy is 90.8%
     Using feature(s) {2,5,3,1,4} accuracy is 86.8%
     Using feature(s) {6,5,3,1,4} accuracy is 85.6%
(Warning, Accuracy has decreased! Continuing search in case of local maxima)
Feature set {2,5,3,1,4} was the best, accuracy is 86.8%
     Using feature(s) {6,2,5,3,1,4} accuracy is 83.2%
Finished search!! The best feature subset is {1,4}, which has an accuracy of 96.8%
```

Code: Below is my code for this project.

URL to my Code: https://github.com/Grymrose/feature_selection/blob/main/Feature_Selection.cpp/

Feature_Selection.cpp

```
#include <iostream>      // For standard input and output.
#include <fstream>       // For file handling.
#include <sstream>       // For string stream operations.
#include <vector>        // For using vectors.
#include <string>        // For string operations.
#include <cmath>         // For mathematical functions (e.g., sqrt, pow).
#include <limits>        // For handling infinite values.
#include <algorithm>     // For find function.

using namespace std;

int ALGORITHM_TYPE;     // Global variable to store the user's algorithm choice.

// Function declarations.
void input_data_from_file(vector<vector<double>>*, string);
void feature_search_forward(vector<vector<double>>);
void feature_search_backward(vector<vector<double>>);
double leave_one_out_cross_validation(vector<vector<double>>, vector<int>, int);

int main() {
    ios::sync_with_stdio(0);        // Fast input and output.

    vector<vector<double>> data;    // 2D vector to store dataset values.

    // Prompt user to enter the dataset filename.
    cout << "Welcome to Nelson Tran’s Feature Selection Algorithm." << '\n'
         << "Type the name of the file to test: ";
    string file_name; getline(cin, file_name);
    cout << '\n';
    input_data_from_file(&data, file_name);

    // Prompt user to select the algorithm type.
    cout << "Type the number of the algorithm you want to run." << '\n'
         << "     1) Forward Selection" << '\n'
         << "     2) Backward Elimination" << '\n' << '\n';
    cin >> ALGORITHM_TYPE;
    cout << '\n';

    // Execute the selected algorithm.
    if (ALGORITHM_TYPE == 1) {
        feature_search_forward(data);
    }
    else if (ALGORITHM_TYPE == 2) {
        feature_search_backward(data);
    }

    return 0;
}

// Reads dataset from a file and stores it into a 2D vector.
void input_data_from_file(vector<vector<double>>* data, string file_name) {
    ifstream file(file_name);               // Open file.
    if (!file) {
        cerr << "Error: Could not open file!" << endl;
        return;
    }

    string line_in_file;
    while (getline(file, line_in_file)) {   // Reads file line by line.
        istringstream iss(line_in_file);    // Converts line into stream of inputs.
        vector<double> line;
        double num_from_file;
```

```cpp
        while (iss >> num_from_file) {      // Extract numbers from the line.
            line.push_back(num_from_file);
        }

        if (!line.empty()) {
            data->push_back(line);          // Data gets updated with a line of numbers.
        }
    }

    file.close();
    return;
}

// Forward Selection Algorithm: Adds one feature at a time.
void feature_search_forward(vector<vector<double>> data) {
    vector<int> full_set_of_features;
    for (int i = 1; i < data[0].size(); i++) {  // Create a full set with all features.
        full_set_of_features.push_back(i);
    }

    // Tell user information about the data.
    cout << "This dataset has " << data[0].size() - 1 << " features (not including the class attribute), with " << data.size() << " instances." << '\n'
         << "Running nearest neighbor with all " << data[0].size() - 1 << " features, using \"leaving-one-out\" evaluation, I get an accuracy of "
         << leave_one_out_cross_validation(data, full_set_of_features, data[0].size()) * 100 << "%" << '\n'
         << "Beginning search." << '\n';

    double best_accuracy = -1;                  // Store the final best accuracy.
    vector<int> best_set_of_features;           // Store the final best feature subset found.
    vector<int> current_set_of_features;        // Store the current best feature subset found.

    for (int i = 1; i <= data[0].size() - 1; i++) {
        int feature_to_add_at_this_level = -1;
        double accuracy = 0;                    // Store the current accuracy.
        double best_so_far_accuracy = -1;       // Store the current best accuracy.

        // Iterate over features to determine the best one to add.
        for (int k = 1; k <= data[0].size() - 1; k++) {
            if (find(current_set_of_features.begin(), current_set_of_features.end(), k) == current_set_of_features.end()) {
                // Only consider adding if not already added.
                accuracy = leave_one_out_cross_validation(data, current_set_of_features, k);

                // Tell user what features are used in this subset and its accuracy.
                cout << "     Using feature(s) {" << k;
                for (int j = 0; j < current_set_of_features.size(); j++) {
                    cout << ',' << current_set_of_features[j];
                }
                cout << "} accuracy is " << accuracy * 100 << '%' << '\n';

                // Update current best accuracy and feature set if improved.
                if (accuracy > best_so_far_accuracy) {
                    best_so_far_accuracy = accuracy;
                    feature_to_add_at_this_level = k;
                }
            }
        }

        current_set_of_features.insert(current_set_of_features.begin(), feature_to_add_at_this_level);

        // Update the final best accuracy and feature set if improved.
        if (best_so_far_accuracy > best_accuracy) {
            best_accuracy = best_so_far_accuracy;
            best_set_of_features = current_set_of_features;
        }
        else {
            if (i < data[0].size() - 1) {
                cout << "(Warning, Accuracy has decreased! Continuing search in case of local maxima)" << '\n';
            }
        }

        // Tell user what the current best subset and accuracy is.
        if (i < data[0].size() - 1) {
            cout << "Feature set {" << current_set_of_features[0];
            for (int k = 1; k < current_set_of_features.size(); k++) {
                cout << ',' << current_set_of_features[k];
            }
            cout << "} was the best, accuracy is " << best_so_far_accuracy * 100 << '%' <<'\n';
        }
    }

    // Tell user what the final best subset and accuracy is.
    cout << "Finished search!! The best feature subset is {" << best_set_of_features[0];
    for (int i = 1; i < best_set_of_features.size(); i++) {
        cout << ',' << best_set_of_features[i];
    }
    cout << "}, which has an accuracy of " << best_accuracy * 100 << '%' <<'\n';

    return;
}

// Backward Selection Algorithm: Removes one feature at a time.
void feature_search_backward(vector<vector<double>> data) {
    // Tell user information about the data.
    cout << "This dataset has " << data[0].size() - 1 << " features (not including the class attribute), with " << data.size() << " instances." << '\n'
         << "Running nearest neighbor with all " << data[0].size() - 1 << " features, using \"leaving-one-out\" evaluation, I get an accuracy of "
         << leave_one_out_cross_validation(data, {}, data[0].size()) * 100 << "%" << '\n'
         << "Beginning search." << '\n';

    double best_accuracy = -1;                  // Store the final best accuracy.
    vector<int> best_set_of_features;           // Store the final best feature subset removed.
    vector<int> current_set_of_features;        // Store the current best feature subset removed.

    for (int i = 1; i <= data[0].size() - 1; i++) {
        int feature_to_add_at_this_level = -1;
        double accuracy = 0;                    // Store the current accuracy.
        double best_so_far_accuracy = -1;       // Store the current best accuracy.

        // Iterate over features to determine the best one to remove.
        for (int k = 1; k <= data[0].size() - 1; k++) {
            if (find(current_set_of_features.begin(), current_set_of_features.end(), k) == current_set_of_features.end()) {
                // Only consider removing if not already removed.
                accuracy = leave_one_out_cross_validation(data, current_set_of_features, k);

                // Tell user what features are used in this subset and its accuracy.
                cout << "     Using feature(s) {";
                int j;
                for (j = 1; j <= data[0].size() - 1; j++) {
                    if ((k != j) && (find(current_set_of_features.begin(), current_set_of_features.end(), j) == current_set_of_features.end())) {
                        cout << j; j++; break;
                    }
                }
                while (j <= data[0].size() - 1) {
                    if ((k != j) && (find(current_set_of_features.begin(), current_set_of_features.end(), j) == current_set_of_features.end())) {
                        cout << ',' << j;
                    }
                    j++;
                }
                cout << "} accuracy is " << accuracy * 100 << '%' << '\n';

                // Update current best accuracy and feature set if improved.
                if (accuracy > best_so_far_accuracy) {
                    best_so_far_accuracy = accuracy;
                    feature_to_add_at_this_level = k;
                }
            }
        }

        current_set_of_features.insert(current_set_of_features.begin(), feature_to_add_at_this_level);

        // Update the final best accuracy and feature set if improved.
        if (best_so_far_accuracy > best_accuracy) {
            best_accuracy = best_so_far_accuracy;
            best_set_of_features = current_set_of_features;
        }
        else {
            if (i < data[0].size() - 1) {
                cout << "(Warning, Accuracy has decreased! Continuing search in case of local maxima)" << '\n';
            }
        }

        // Tell user what the current best subset and accuracy is.
        if (i < data[0].size() - 1) {
            cout << "Feature set {";
            int k;
            for (k = 1; k <= data[0].size() - 1; k++) {
                if (find(current_set_of_features.begin(), current_set_of_features.end(), k) == current_set_of_features.end()) {
                    cout << k; k++; break;
                }
            }
            while (k <= data[0].size() - 1) {
                if (find(current_set_of_features.begin(), current_set_of_features.end(), k) == current_set_of_features.end()) {
                    cout << ',' << k;
                }
                k++;
            }
            cout << "} was the best, accuracy is " << best_so_far_accuracy * 100 << '%' <<'\n';
        }
    }

    // Tell user what the final best subset and accuracy is.
    cout << "Finished search!! The best feature subset is {";
    int i;
    for (i = 1; i <= data[0].size() - 1; i++) {
        if (find(best_set_of_features.begin(), best_set_of_features.end(), i) == best_set_of_features.end()) {
            cout << i; i++; break;
        }
    }
    while (i <= data[0].size() - 1) {
        if (find(best_set_of_features.begin(), best_set_of_features.end(), i) == best_set_of_features.end()) {
            cout << ',' << i;
        }
        i++;
    }
    cout << "}, which has an accuracy of " << best_accuracy * 100 << '%' <<'\n';

    return;
}

// Evaluates accuracy using Leave-One-Out Cross Validation.
double leave_one_out_cross_validation(vector<vector<double>> data, vector<int> current_set, int feature_to_add) {
    current_set.push_back(feature_to_add);          // Add the selected feature to the current set.

    // Determine whether the current set is what should be added or be removed depending on the search algorithm.
    for (int i = 0; i < data.size(); i++) {
        for (int j = 1; j < data[i].size(); j++) {
            if ((ALGORITHM_TYPE == 1) && (find(current_set.begin(), current_set.end(), j) == current_set.end())) {
                data[i][j] = 0;
            }
            else if ((ALGORITHM_TYPE == 2) && !(find(current_set.begin(), current_set.end(), j) == current_set.end())) {
                data[i][j] = 0;
            }
        }
    }

    int number_correctly_classified = 0;

    for (int i = 0; i < data.size(); i++) {
        vector<double> object_to_classify(data[i].begin()+1, data[i].end());    // Features.
        double label_object_to_classify = data[i][0];                           // Object's label.

        // Variables to track the nearest neighbor.
        double nearest_neighbor_distance = numeric_limits<double>::infinity();
        int nearest_neighbor_location = numeric_limits<int>::infinity();
        double nearest_neighbor_label = -1;

        // Iterate through the dataset to find the nearest neighbor.
        for (int k = 0; k < data.size(); k++) {
            if (k != i) {            // Leave one out.
                double distance = 0.0;

                // Compute Euclidean distance between object_to_classify and current data point.
                for (int j = 1; j < data[k].size(); j++) {
                    distance += pow(object_to_classify[j - 1] - data[k][j], 2);
                }
                distance = sqrt(distance);

                // Update nearest neighbor if a closer one is found.
                if (distance < nearest_neighbor_distance) {
                    nearest_neighbor_distance = distance;
                    nearest_neighbor_location = k;
                    nearest_neighbor_label = data[nearest_neighbor_location][0];
                }
            }
        }

        if (label_object_to_classify == nearest_neighbor_label) {
            number_correctly_classified++;
        }
    }

    return static_cast<double>(number_correctly_classified) / data.size();      // Accuracy.
}
```