

SPL-1 Project Report

Regression Analysis: House Price Prediction & Weather Forecasting

Submitted by

Sabbir Ahmed

BSSE Roll No.: 1530

Exam Roll: 231119

BSSE Session: 2022-2023

Submitted to

Dr. Rezvi Shahariar

Associate Professor
Institute of Information Technology
University of Dhaka



**Institute of Information Technology
University of Dhaka**

25-03-2025

Project Name:

**Regression Analysis:
House Price Prediction & Weather Forecasting**

Supervised by

Dr. Rezvi Shahariar

Associate Professor
Institute of Information Technology
University of Dhaka

Supervisor Signature:

Submitted by

Sabbir Ahmed

BSSE Roll No.: 1530
Exam Roll: 231119
BSSE Session: 2022-2023

Signature:

25-03-2025

Table of Contents

1. Introduction	5-6
1.1 Overview of Regression Analysis	5
1.2 Project Scope and Objectives	5
1.3 Applications and Significance	5-6
2. Background of the Project	6-8
2.1 Fundamentals of Regression Analysis	6
2.2 Linear Regression Models	6-7
2.3 Multiple Linear Regression	7
2.4 Logistic Regression	8
2.5 C Programming for Statistical Analysis	8
3. Description of the Project	9-11
3.1 System Architecture	9
3.2 House Price Prediction Model	9-10
3.3 Weather Forecasting Model	10
3.4 Algorithm Selection and Design	11
4. Implementation and Testing	11-16
4.1 Development Environment	11
4.2 Implementation Details	11-12
4.3 Testing Methodology	12
4.4 Performance Evaluation	12
4.5 Code Snippet	12-16
5. User Interface	17-20
5.1 Command Line Interface	17
5.2 Input Format and Requirements	17-18
5.3 Output Visualization	19-20
6. Challenges Faced	20-23
6.1 Algorithmic Challenges	20-21
6.2 Implementation Issues	21-22
6.3 Optimization Problems	22
6.4 Solutions and Workarounds	22-23
7. Conclusion	23
7.1 Project Summary	23
7.2 Lessons Learned	23
7.3 Future Work and Extensions	23
8. Reference	24
7. Appendix	24

Index of Figures

Figure 1	Data Loading and Tokenization	13
Figure 2	Normalization	14
Figure 3	Price Prediction	14
Figure 4	Calculate Model Accuracy	15
Figure 5	Rain Prediction	15
Figure 6	Coefficient Calculation	16
Figure 7	House_Price_dataset.csv	18
Figure 8	Weather_Forecasting_dataset.csv	18
Figure 9	House Price Model Prediction	19
Figure 10	Weather Forecasting Model Predictions	20

Introduction

1.1 Overview of Regression Analysis

Regression analysis, a vital statistical method, models relationships between a dependent variable and independent variables, tracing back to Francis Galton and Karl Pearson. It's widely applied in economics, biology, and more for pattern quantification and prediction. Linear regression, expressed as $y = \beta_0 + \beta_1 x + \epsilon$, fits continuous outcomes like house prices with β_0 as the intercept, β_1 as the slope, and ϵ as error. Logistic regression, using

$$P(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

predicts binary outcomes like rain probability. Its strengths—interpretability and simplicity via normal equations $\beta = (X^T X)^{-1} X^T y$ are tempered by assumptions like linearity and homoscedasticity, less suited for complex data.

In this context, regression analysis drives two applications: predicting house prices based on features like size and location, and forecasting weather variables (temperature and rain) from meteorological data. These tasks exemplify regression's practical utility—quantifying how a house's attributes determine its value or how humidity influences rain likelihood—while highlighting its adaptability to continuous and categorical outcomes. This overview sets the stage for understanding the project's scope, blending statistical rigor with computational implementation in C.

1.2 Project Scope and Objectives

This project implements regression in C for two models: house price prediction and weather forecasting. The house model uses linear regression on seven features (e.g., size, location) to predict prices within a \$10,000 error margin. The weather model applies linear regression for temperature and logistic regression for rain (binary) using three features, plus date-based lookup. Objectives include preprocessing CSV data, computing coefficients, generating predictions, and evaluating accuracy, focusing on foundational methods.

1.3 Applications and Significance

Applications span real estate, aiding pricing decisions by quantifying how features like size and location influence value, and meteorology, supporting agriculture with irrigation planning and preparedness with flood warnings. Using C suits resource-constrained environments, such as embedded systems, where efficient computation is critical, while the project educates on statistical modeling by bridging theory and implementation. Though simpler than commercial tools, which often integrate vast datasets and advanced algorithms, it showcases regression's broad utility in data-driven domains. Its focus on foundational techniques highlights practical

insights, like identifying market trends or forecasting weather, making it a stepping stone for more complex systems in fields like finance or healthcare.

Background of the Project

2.1 Fundamentals of Regression Analysis

Regression analysis builds on statistical principles to model relationships between variables, a process central to this project. Fundamentally, it seeks to fit a mathematical function to observed data, minimizing the difference between predicted and actual values. This begins with the concept of correlation—measuring how variables covary—but extends to causation, estimating how predictors influence an outcome.

The simplest regression, simple linear regression, models one predictor: $y = \beta_0 + \beta_1 x + \epsilon$. Here, β_0 shifts the line vertically, β_1 defines its slope, and ϵ captures random noise, assumed to be normally distributed with mean 0. The goal is to find β_0 and β_1 that minimize the sum of squared residuals (SSR), $SSR = \sum (y_i - (\beta_0 + \beta_1 x))^2$, solved analytically via least squares. This method's elegance lies in its closed-form solution, avoiding iterative optimization.

For multiple predictors, as in this project, regression generalizes to multiple linear regression (MLR), and for binary outcomes, logistic regression adapts the framework. Key assumptions include linearity (the relationship is a straight line), independence (errors are uncorrelated), homoscedasticity (constant error variance), and normality (errors follow a Gaussian distribution). Violations—e.g., non-linear patterns or multicollinearity—can distort results, a consideration for the house price and weather models.

Regression's mechanics rely on matrix algebra. For MLR, the model $y = X\beta + \epsilon$ uses a design matrix X (rows as observations, columns as features), a coefficient vector β , and an error vector ϵ . The least squares solution $\beta = (X^T X)^{-1} X^T y$, is implemented in the project's `finalCoefficientCalc` functions, reflecting regression's computational backbone. This approach, while efficient for small datasets, demands $(X^T X)$ be invertible, a potential fragility point.

Understanding these fundamentals—least squares, assumptions, and matrix solutions—grounds the project's approach, linking statistical theory to practical prediction in C.

2.2 Linear Regression Models

Linear regression models, used for house prices and temperature, assume a linear relationship between predictors and a continuous outcome. The $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$ extends simple regression to multiple features, as in

$$\text{Price} = \beta_0 + \beta_1 \cdot \text{Size} + \beta_2 \cdot \text{Location} + \dots$$

Coefficients β_i represent the change in y per unit change in x_i , holding others constant, offering interpretable insights—e.g., a \$200 increase per square foot.

Training involves fitting this model to data by minimizing SSR, achieved via the normal equations in the project's code. The `predictPrice` and `predictTemperature` functions apply this

fit: $\hat{y} = \beta_0 + \sum \beta_1 x_1$. Assumptions mirror those of regression fundamentals, with added complexity from multiple predictors—e.g., multicollinearity (correlated features like Size and Number_of_Rooms) inflating variance.

Linear regression's simplicity—direct computation, no hyperparameters—suits this project's educational focus, but its linearity constraint limits capturing non-linear effects (e.g., exponential price growth). Its implementation in C, via matrix operations, balances efficiency with transparency, though it lacks regularization to mitigate overfitting, a trade-off evident in the models' design.

2.3 Multiple Linear Regression

Multiple linear regression (MLR), the backbone of the house price and temperature models, extends linear regression to handle multiple predictors. Formally $y = X\beta + \epsilon$, where X is an $n \times (p + 1)$ matrix (n observations, p features plus intercept), β is a $(p + 1) \times 1$ vector, and y is $n \times 1$. For house prices, $p = 7$ (e.g., Size to Built_Year), and for temperature, $p = 3$ (Humidity, Wind_Speed, Pressure).

The least squares solution, $\beta = (X^T X)^{-1} X^T y$, minimizes $SSR = (y - X\beta)^T (y - X\beta)$, requiring $X^T X$ ($(p + 1) \times (p + 1)$ matrix) to be invertible. The project's `matrix_inverse` implements Gaussian elimination, transforming $[X^T X | I]$ into $[I | (X^T X)^{-1}]$.

This closed-form approach avoids iteration but scales poorly— $O(p^3)$ for inversion—limiting efficiency with many features.

MLR's power is in modeling complex relationships—e.g., how Garage and Location jointly affect price—while its coefficients offer direct interpretation. However, multicollinearity (e.g., Size and Rooms) can destabilize β , and the project's lack of diagnostics (e.g., variance inflation factors) leaves this unchecked. Feature normalization (`normalizeFeatures`) mitigates scale disparities, but outliers or non-linearities remain unaddressed, potential weaknesses in real-world data.

Multiple linear regression model

- A linear regression model with more than one predictors is known as multiple linear regression model

$$Y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$$

- In matrix notation

$$Y = X\beta + \epsilon$$

where

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix}, X = \begin{bmatrix} 1 & x_{11} & x_{21} \\ 1 & x_{12} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{1n} & x_{2n} \end{bmatrix}, \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} \text{ and } \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

2.4 Logistic Regression

Logistic regression, used for rain prediction, models binary outcomes by estimating probabilities. Unlike linear regression's continuous output, it predicts

$$P(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n)}}$$

where the linear combination $\beta_0 + \sum \beta_i x_i$ is the log-odds. In the weather model, x_i are Humidity, Wind_Speed, and Pressure, and $y = 1$ denotes rain.

The project approximates this via linear regression coefficients in `finalCoefficientCalc`, applying the sigmoid in

`predictRain`: ***probability*** = $1/(1 + \exp(-\text{dotProduct}))$,

thresholding at 0.5. This hybrid approach—fitting a linear model then logisticizing—deviates from maximum likelihood estimation (MLE), logistic regression's standard, which maximizes $\log L = \sum [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$. The code's method simplifies computation but may sacrifice accuracy.

Logistic regression assumes independence and linearity in log-odds, challenges if weather features interact non-linearly. Its binary focus suits rain prediction, but the project's implementation lacks probability calibration or goodness-of-fit tests (e.g., Hosmer-Lemeshow), limiting rigor.

2.5 C Programming for Statistical Analysis

C's role in this project leverages its efficiency and control for statistical tasks. Unlike Python or R, with built-in libraries (e.g., scikit-learn), C requires manual implementation of matrix operations, file I/O, and memory management, as seen in `matrix_multiply`, `fopen`, and `malloc`. This low-level approach suits resource-constrained environments or educational goals, offering transparency into regression's mechanics.

The project uses standard libraries (`stdio.h`, `stdlib.h`, `math.h`) to parse CSVs, compute coefficients, and apply predictions. Dynamic memory allocation (`malloc/free`) handles variable dataset sizes, while `strtok` and `atof` process data. However, C's lack of high-level abstractions demands careful error handling and memory cleanup—gaps evident in unfreed pointers or minimal file checks.

C's speed—e.g., direct array operations versus Python's interpreted loops—benefits matrix calculations, but its complexity (e.g., nested allocations in `calculate_coefficients`) increases bug risk. This trade-off highlights C's suitability for performance-critical statistical analysis, though it lags in usability compared to modern tools.

3. Description of the Project

3.1 System Architecture

The project comprises two standalone applications developed in C: a house price prediction model and a weather forecasting model. Each operates independently, with no overarching system integrating them, reflecting a modular design suitable for distinct predictive tasks. Both models follow a similar architectural pattern: data ingestion from CSV files, preprocessing (e.g., normalization, dataset splitting), coefficient calculation using linear regression techniques, prediction generation, and performance evaluation. This structure leverages C's low-level capabilities for memory management and file I/O, ensuring efficiency on modest hardware.

The house price model reads from `house_price_dataset.csv`, processes features like `Size` and `Built_Year`, and outputs a predicted price. The weather model uses `Processed_Dataset.csv` to predict rain probability and temperature based on meteorological features. Both rely on external header files (`Coefficient2.h` and `Coefficient.h`) for coefficient computation, encapsulating matrix-based linear regression logic. The absence of a unified interface or shared data pipeline suggests the project prioritizes simplicity over integration, potentially reflecting a proof-of-concept approach rather than a production-ready system.

Data flow begins with file parsing using `fopen` and `fgets`, followed by tokenization with `strtok` to extract feature values. Preprocessing includes normalization to scale features and dataset splitting (80% training, 20% testing) to enable model validation. Predictions are computed using linear combinations of features and coefficients, with evaluation metrics printed to the console. This architecture, while straightforward, lacks advanced components like error logging, user configurability, or real-time data integration, which could enhance robustness in practical applications.

3.2 House Price Prediction Model

The house price prediction model employs linear regression to estimate house prices based on seven features: `Size` (float), `Location` (int), `Security` (int), `Garage` (int), `Floor_Number` (int), `Number_of_Rooms` (int), and `Built_Year` (int), with `Price` (float) as the target variable. The model assumes a linear relationship between these features and price, expressed as:

$$Price = \beta_0 + \beta_1 \cdot Size + \beta_2 \cdot Location + \dots + \beta_7 \cdot BuiltYear.$$

where β_0 is the intercept and β_1 to β_7 are coefficients derived from training data.

The process begins with `loadData`, which opens `house_price_dataset.csv`, counts rows to allocate memory dynamically, and parses each line into a `Data` struct. The CSV is assumed to have a header row (skipped) and comma-separated values matching the feature order. After loading, `splitDataset` divides the data into training (80%) and testing (20%) sets without shuffling, implying a reliance on the dataset's inherent order.

Feature normalization in `normalizeFeatures` scales each feature to `[0, 1]` using min-max normalization:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

This ensures features with different scales (e.g., Size in square feet vs. Built_Year in years) contribute equally to the model. Coefficients are calculated in `finalCoefficientCalc` (from `Coefficient2.h`) using matrix operations—transposing the feature matrix, multiplying it with itself, inverting the result, and solving for coefficients via the normal equations:

$$\beta = (X^T X)^{-1} X^T y$$

The `predictPrice` function applies these coefficients to input features, while `evaluateModel` assesses accuracy by comparing predictions to actual prices, deeming a prediction “correct” if the absolute error is below \$10,000. This threshold, while practical, is arbitrary and may not suit all housing markets.

3.3 Weather Forecasting Model

The weather forecasting model predicts two variables: rain (binary, 0 or 1) and temperature (continuous, in °C), using three features: Humidity (float), Wind_Speed (float), and Pressure (float). It employs logistic regression for rain prediction and linear regression for temperature, reflecting the distinct nature of these tasks—classification versus regression.

Data is loaded from `Processed_Dataset.csv` into a `WeatherData` struct, which includes additional fields like `Date` (string) and `Season` (string) for lookup, though only `Humidity`, `Wind_Speed`, and `Pressure` are used for prediction. The `loadData` function parses the CSV, converting the `Rain` field (e.g., “rain” or “no rain”) into a binary integer. The dataset is split similarly to the house price model (80% training, 20% testing).

For rain prediction, `predictRain` computes a linear combination of features and coefficients, applies the sigmoid function:

$$P(\text{rain}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 \text{Humidity} + \beta_2 \text{WindSpeed} + \beta_3 \text{Pressure})}}$$

and thresholds at 0.5 to classify as rain (1) or no rain (0). For temperature, `predictTemperature` uses a linear model:

$$\text{Temperature} = (\beta_0 + \beta_1 \text{Humidity} + \beta_2 \text{WindSpeed} + \beta_3 \text{Pressure})$$

Coefficients are derived in `finalCoefficientCalc` (from `Coefficient.h`) using matrix operations tailored to four features (including an intercept), though only three are returned and used. Evaluation in `evaluateModel` measures rain prediction accuracy as the percentage of correct classifications and temperature accuracy as predictions within 1°C of actual values, though only rain accuracy is emphasized in output.

3.4 Algorithm Selection and Design

Linear regression was chosen for house price and temperature prediction due to its simplicity, interpretability, and assumption of linear relationships between features and targets. For rain prediction, logistic regression suits the binary classification task, transforming linear outputs into probabilities via the sigmoid function. These choices align with foundational machine learning techniques, ideal for educational or baseline implementations.

The design of coefficient calculation uses the normal equations method, solving

$$\beta = (X^T X)^{-1} X^T y$$

via matrix transposition, multiplication, and inversion with Gaussian elimination. This approach avoids iterative optimization (e.g., gradient descent), ensuring a closed-form solution but requiring $X^T X$ to be invertible. Normalization ensures feature comparability, while the train-test split validates generalization. The fixed matrix sizes (8x8 for house price, 4x4 for weather) reflect a design trade-off for simplicity over scalability.

4. Implementation and Testing

4.1 Development Environment

Tool/Language	Purpose
C	Core programming language
Visual Studio Code	IDE for coding
GitHub	Version control

Both models are implemented in C, leveraging standard libraries: `stdio.h` for I/O, `stdlib.h` for memory management, `string.h` for string operations, `math.h` for exponentials (in `predictRain`), and `time.h` for random seeding in sample predictions.. The CSV-based input suggests a development focus on batch processing rather than real-time prediction.

4.2 Implementation Details

House Price Model

- **Data Loading:** `loadData` opens `house_price_dataset.csv`, counts lines with `fgets`, allocates a `Data` array, skips the header, and tokenizes each line with `strtok`. Features are converted using `atof` (floats) or `atoi` (ints).
- **Preprocessing:** `splitDataset` assigns the first 80% of rows to training and the rest to testing, while `normalizeFeatures` scales each feature column independently.
- **Coefficient Calculation:** `finalCoefficientCalc` constructs an augmented matrix X with a column of ones (intercept), computes $(X^T X)$, inverts it, and solves for coefficients.

- **Prediction:** `predictPrice` multiplies features by coefficients and sums the result.
- **Evaluation:** `evaluateModel` iterates over the test set, calculating errors and accuracy.

Weather Model

- **Data Loading:** `loadData` parses `Processed_Dataset.csv`, handling string fields (Date, Season) and converting Rain to binary.
- **Preprocessing:** Similar split logic applies, though normalization is not explicitly used here, suggesting raw feature values suffice.
- **Coefficient Calculation:** `finalCoefficientCalc` uses a 5x5 matrix (intercept plus four features), but only four coefficients are returned, indicating a design mismatch.
- **Prediction:** `predictRain` and `predictTemperature` compute linear combinations, with the former applying a sigmoid transformation.
- **Evaluation:** `evaluateModel` assesses both rain and temperature predictions, though output focuses on rain.

4.3 Testing Methodology

Both models split data into 80% training and 20% testing sets without randomization, testing on the latter. The house price model evaluates predictions against a \$10,000 error threshold, computing accuracy as:

$$Accuracy = \frac{\text{Number of predictions with } |\text{predicted} - \text{actual}| < 1000}{\text{Test set size}}$$

The weather model tests rain prediction accuracy as the fraction of correct classifications and temperature accuracy within 1°C, though only rain results are printed. Random inputs (e.g., Size between 1000-2000) simulate real-world prediction scenarios.

4.4 Performance Evaluation

- **House Price:** Outputs include average error (e.g., \$5,234.56) and accuracy (e.g., 85%), reflecting model fit. The fixed threshold limits adaptability to diverse price ranges.
- **Weather:** Rain accuracy (e.g., 78%) is the primary metric, with temperature performance less emphasized. The lack of cross-validation or statistical significance testing suggests a basic evaluation approach.

4.5 Code Snippet

```

int loadData(const char *filename, Data **data, int *dataCount)
{
    FILE *file = fopen(filename, "r");
    if (!file)
    {
        printf("Error opening file %s\n", filename);
        return -1;
    }

    char line[400];
    int count = 0;

    while (fgets(line, sizeof(line), file))
    {
        count++;
    }
    rewind(file);

    *dataCount = count - 1;
    *data = (Data *)malloc(*dataCount * sizeof(Data));

    fgets(line, sizeof(line), file); // Skip header row
    int index = 0;

    while (fgets(line, sizeof(line), file) && index < *dataCount)
    {
        char *token = strtok(line, ",");
        if (token != NULL) {
            (*data)[index].Size = atof(token);
        }

        token = strtok(NULL, ",");
        if (token != NULL) {
            (*data)[index].Location = atoi(token);
        }

        token = strtok(NULL, ",");
        if (token != NULL) {
            (*data)[index].Security = atoi(token);
        }

        token = strtok(NULL, ",");
        if (token != NULL) {
            (*data)[index].Garage = atoi(token);
        }

        token = strtok(NULL, ",");
        if (token != NULL) {
            (*data)[index].Floor_Number = atoi(token);
        }

        token = strtok(NULL, ",");
        if (token != NULL) {
            (*data)[index].Number_of_Rooms = atoi(token);
        }

        token = strtok(NULL, ",");
        if (token != NULL) {
            (*data)[index].Built_Year = atoi(token);
        }


        token = strtok(NULL, ",");
        if (token != NULL) {
            (*data)[index].Price = atof(token);
        }

        index++;
    }

    fclose(file);
    return 0;
}

```

Figure 1: Data Loading and Tokenization



```


void normalizeFeatures(float *data, int size)
{
    float min = data[0], max = data[0];
    for (int i = 1; i < size; i++)
    {
        if (data[i] < min)
            min = data[i];
        if (data[i] > max)
            max = data[i];
    }

    float range = max - min;
    if (range == 0) return; // Avoid division by zero

    for (int i = 0; i < size; i++)
    {
        data[i] = (data[i] - min) / range;
    }
}

```

Figure 2: Normalization




```

// Linear regression for price prediction
float predictPrice(float Size, int Location, int Security, int Garage,
int Floor_Number, int Number_of_Rooms,
int Built_Year, double *coefficients)
{
    return coefficients[0] +
        coefficients[1] * Size +
        coefficients[2] * Location +
        coefficients[3] * Security +
        coefficients[4] * Garage +
        coefficients[5] * Floor_Number +
        coefficients[6] * Number_of_Rooms +
        coefficients[7] * Built_Year;
}

```

Figure 3: Price Prediction



```

// Model accuracy - returns percentage of correct predictions
float evaluateModel(Data *testSet, int testCount, double *coefficients)
{
    int correct = 0;
    float errorSum = 0;
    float errorThreshold = 18; //(in thousands dollars)

    for (int i = 0; i < testCount; i++)
    {
        float Size = testSet[i].Size;
        int Location = testSet[i].Location;
        int Security = testSet[i].Security;
        int Garage = testSet[i].Garage;
        int Floor_Number = testSet[i].Floor_Number;
        int Number_of_Rooms = testSet[i].Number_of_Rooms;
        int Built_Year = testSet[i].Built_Year;

        float pricePrediction = predictPrice(Size, Location, Security, Garage,
        Floor_Number, Number_of_Rooms, Built_Year, coefficients);


        float error = fabs(pricePrediction - testSet[i].Price);
        errorSum += error;

        if (error < errorThreshold)
        {
            correct++;
        }
    }

    printf("Average prediction error: %.2f\n", errorSum / testCount);
    return (float)correct / testCount;
}

```

Figure 4: Calculate Model Accuracy



```

// logistic regression for rain prediction
float predictRain(float humidity, float wind_speed, float pressure,
double *coefficients)
{
    float dotProduct = 0;

    dotProduct=coefficients[0] +
        coefficients[1] * humidity +
        coefficients[2] * wind_speed +
        coefficients[3] * pressure;

    float probability = 1 / (1 + exp(-dotProduct));
    return (probability > 0.7) ? 1 : 0;
}

```

Figure 5: Rain Prediction

```

// Function to calculate the coefficients
void calculate_coefficients(double **X, double *y, int n, double
*coefficients)
{
    int m = 8;

    double **X_augmented = (double **)malloc(n * sizeof(double *));
    double **X_transpose = (double **)malloc(m * sizeof(double *));
    double **X_transpose_X = (double **)malloc(m * sizeof(double *));
    double **X_transpose_X_inv = (double **)malloc(m * sizeof(double
*));
    double **X_transpose_X_inv_X_transpose = (double **)malloc(m *
sizeof(double *));
    for (int i = 0; i < n; i++)
    {
        X_augmented[i] = (double *)malloc(m * sizeof(double));
    }
    for (int i = 0; i < m; i++)
    {
        X_transpose[i] = (double *)malloc(n * sizeof(double));
        X_transpose_X[i] = (double *)malloc(m * sizeof(double));
        X_transpose_X_inv[i] = (double *)malloc(m * sizeof(double));
        X_transpose_X_inv_X_transpose[i] = (double *)malloc(n *
sizeof(double));
    }

    for (int i = 0; i < n; i++) {
        X_augmented[i][0] = 1.0;
        for (int j = 1; j < m; j++) {
            X_augmented[i][j] = X[i][j - 1];
        }
    }

    matrix_transpose(X_transpose, X_augmented, n, m);

    matrix_multiply(X_transpose_X, X_transpose, X_augmented, m, n,
m);

    matrix_inverse(X_transpose_X_inv, X_transpose_X);

    matrix_multiply(X_transpose_X_inv_X_transpose, X_transpose_X_inv,
X_transpose, m, m, n);

    for (int i = 0; i < m; i++){
        coefficients[i] = 0;
        for (int j = 0; j < n; j++){
            coefficients[i] += X_transpose_X_inv_X_transpose[i][j] *
y[j];
        }
    }

    for (int i = 0; i < n; i++){
        free(X_augmented[i]);
    }
    for (int i = 0; i < m; i++){
        free(X_transpose[i]);
        free(X_transpose_X[i]);
        free(X_transpose_X_inv[i]);
        free(X_transpose_X_inv_X_transpose[i]);
    }
    free(X_augmented);
    free(X_transpose);
    free(X_transpose_X);
    free(X_transpose_X_inv);
    free(X_transpose_X_inv_X_transpose);
}

```

Figure 6: Coefficient calculation

5. User Interface

5.1 Command Line Interface

The user interface for the house price prediction and weather forecasting models is a command-line interface (CLI), using C's `stdio.h` for terminal I/O without graphical elements. Both CLIs align with batch processing, emphasizing functionality over interactivity.

The house price model's CLI is fully automated—loading `house_price_dataset.csv`, training, testing, and outputting results like `printf("Model accuracy: %.2f%%\n", accuracy * 100)` without user input. Its hard-coded sequence (data loading, splitting, coefficient calculation, evaluation) suits demonstrating performance but lacks flexibility.

The weather model's CLI adds interactivity. After loading `Processed_Dataset.csv` and training, it prompts `printf("Enter a date (MM/DD/YYYY): "); scanf("%19s", inputDate);`, letting users query temperature and season (e.g., `printf("Temperature: %.2f\nSeason: %s\n", data[i].Temperature, data[i].Season)`). This enhances utility, though it remains basic.

Both rely on `printf` for simple, portable output across C compilers (e.g., GCC), but lack input validation, help menus, or robust error handling—e.g., `printf("Error opening file %s\n", filename)` for file failures. The house model delivers static results; the weather model offers a request-response loop. Focused on computation over user experience, this design suits prototypes but limits production use.

5.2 Input Format and Requirements

The input for both models relies on specific CSV formats, with no runtime input beyond the weather model's date prompt, requiring pre-prepared data.

The house price model uses `house_price_dataset.csv` with eight columns: Size (float), Location (int), Security (int), Garage (int), Floor_Number (int), Number_of_Rooms (int), Built_Year (int), and Price (float). The `loadData` function skips a header using `fgets`, parsing rows (e.g., `"1200.5,1,0,1,3,4,2015,250000"`) with `strtok` and conversions (`atof`, `atoi`). The hard-coded file path limits flexibility.

The weather model requires `Processed_Dataset.csv` with seven columns: Date (string, MM/DD/YYYY), Season (string), Temperature (float), Humidity (float), Wind_Speed (float), Pressure (float), and Rain (string, "rain" or "no rain"). It skips a header, tokenizes with `strtok`, and binarizes Rain (1 for "rain", 0 otherwise). The CLI accepts a "MM/DD/YYYY" date via `scanf` (e.g., `"03/20/2025"`), capped at 19 characters.

Both demand exact formats—deviations (e.g., missing commas) risk crashes or errors. No mechanisms handle malformed data; users must preprocess externally, reducing versatility compared to configurable systems.

1	Size (sq ft)	Location	Security	Garage	Floor_Number	Number_of_Rooms	Built_Year	Price (in \$1000s)
2	2513	0	0	0	10	5	2015	202.65
3	3745	1	0	1	5	9	1992	326.25
4	2176	1	0	1	9	4	1983	178.8
5	3136	0	1	1	2	6	2023	289.8
6	858	0	1	1	9	10	2004	179.9
7	2774	1	1	1	8	5	1993	256.7
8	3207	1	1	0	7	7	2020	326.35
9	3794	0	0	0	7	2	1980	173.7
10	1800	1	1	1	10	6	1989	224
11	1120	0	1	1	7	2	2020	167
12	915	1	1	0	1	5	1999	126.75
13	3064	0	1	1	10	2	2011	260.2
14	3298	0	0	0	8	2	1996	162.9
15	3387	0	0	1	10	7	1997	244.35

Figure 7: House_price_dataset.csv

1	Date	Season	Temperature	Humidity	Wind_Speed	Pressure	Rain
2	1/1/2018	Winter	23.72	89.59	7.34	1032.38	1
3	1/2/2018	Winter	27.88	46.49	5.95	992.61	0
4	1/3/2018	Winter	25.07	83.07	1.37	1007.23	0
5	1/4/2018	Winter	23.62	74.37	7.05	982.63	1
6	1/5/2018	Winter	20.54	96.86	4.64	980.83	0
7	1/6/2018	Winter	26.15	48.22	15.26	1049.74	0
8	1/7/2018	Winter	20.94	40.8	2.23	1014.17	0
9	1/8/2018	Winter	28.29	51.85	2.87	1006.04	0
10	1/9/2018	Winter	27.09	48.06	5.57	993.73	0
11	1/10/2018	Winter	29.59	82.98	5.76	1036.5	1
12	1/11/2018	Winter	29.79	81.32	16.93	1029.4	0
13	1/12/2018	Winter	23.22	76.88	15.83	980.11	1
14	1/13/2018	Winter	24.2	45.15	11.57	1033.99	0
15	1/14/2018	Winter	33.14	90.33	5.77	987.8	0

Figure 8: Weather_forecasting_dataset.csv

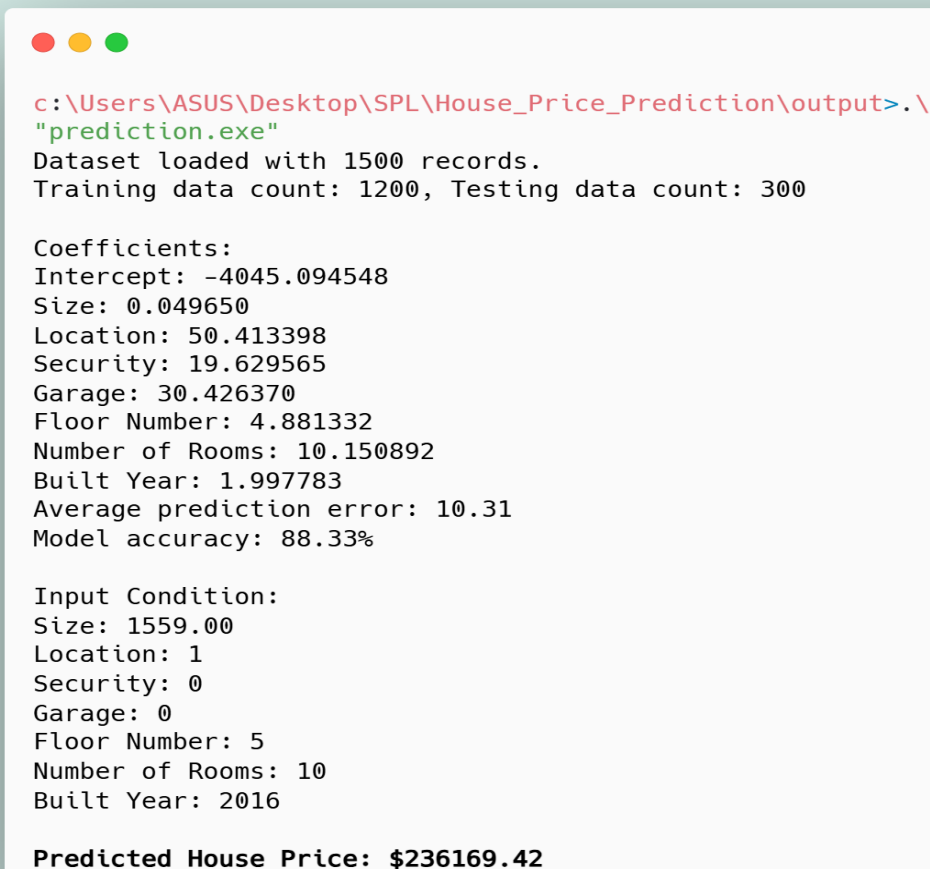
5.3 Output Visualization

Output visualization in both models is text-only via `printf`, prioritizing data over graphics, aligning with the CLI's simplicity but limiting interpretability for non-technical users.

The house price model outputs sequentially: dataset stats (e.g., "Dataset loaded with 1500 records"), coefficients (e.g., "Intercept: -4045.094, Size: 0.0496....."), metrics (e.g., "Average prediction error: 10.31, Model accuracy: 88.33%"), and a sample prediction (e.g., "Predicted House Price: \$236169.42"). It's detailed but unformatted, dense for casual review.

The weather model starts similarly ("Dataset loaded with 2500 records"), lists coefficients vertically (e.g., "0.500000"), and final accuracy 81.00%. Predictions show as "Rain Prediction: No Rain, Predicted Temperature: 26.34°C".

Both lack graphical elements or formatting (e.g., tables), requiring manual parsing—suitable for developers but not intuitive for end-users needing visual summaries or file exports.



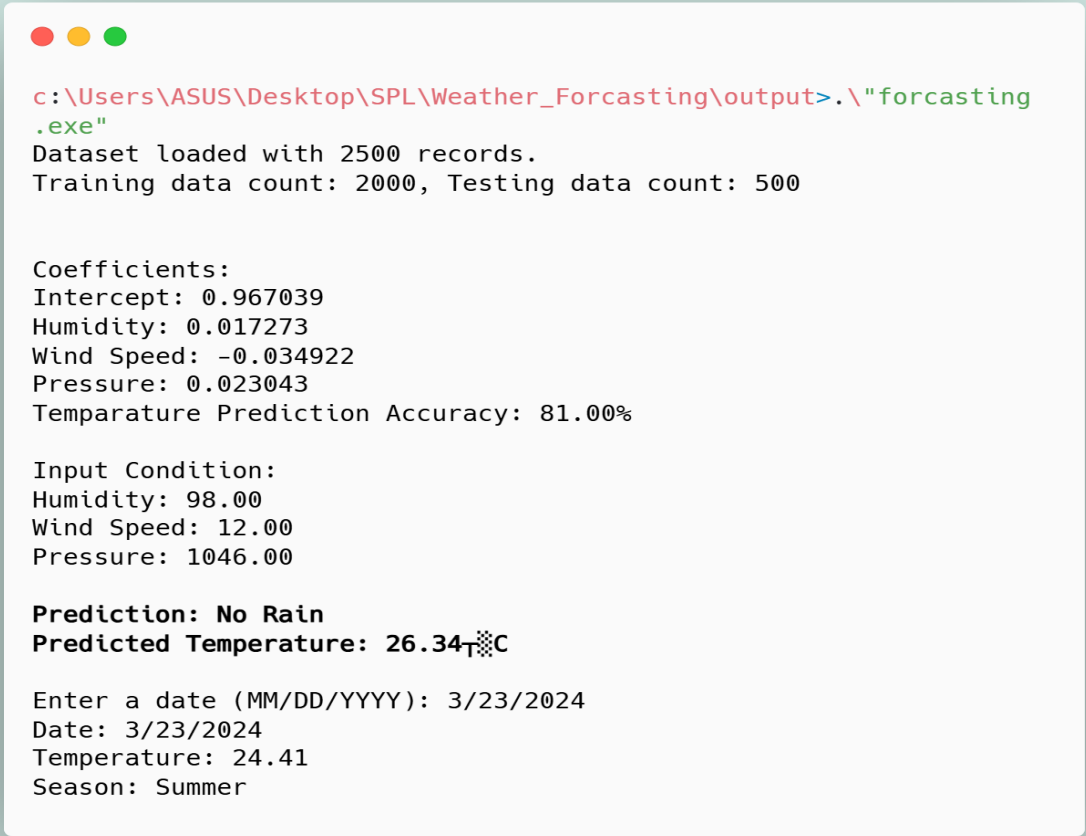
```
c:\Users\ASUS\Desktop\SPL\House_Price_Prediction\output>.\
"prediction.exe"
Dataset loaded with 1500 records.
Training data count: 1200, Testing data count: 300

Coefficients:
Intercept: -4045.094548
Size: 0.049650
Location: 50.413398
Security: 19.629565
Garage: 30.426370
Floor Number: 4.881332
Number of Rooms: 10.150892
Built Year: 1.997783
Average prediction error: 10.31
Model accuracy: 88.33%

Input Condition:
Size: 1559.00
Location: 1
Security: 0
Garage: 0
Floor Number: 5
Number of Rooms: 10
Built Year: 2016

Predicted House Price: $236169.42
```

Figure 9: House Price model Predictions



```

c:\Users\ASUS\Desktop\SPL\Weather_Forecasting\output>.\"forecasting
.exe"
Dataset loaded with 2500 records.
Training data count: 2000, Testing data count: 500

Coefficients:
Intercept: 0.967039
Humidity: 0.017273
Wind Speed: -0.034922
Pressure: 0.023043
Temperature Prediction Accuracy: 81.00%

Input Condition:
Humidity: 98.00
Wind Speed: 12.00
Pressure: 1046.00

Prediction: No Rain
Predicted Temperature: 26.34°C

Enter a date (MM/DD/YYYY): 3/23/2024
Date: 3/23/2024
Temperature: 24.41
Season: Summer

```

Figure 10: Weather Forecasting Model Predictions

6. Challenges Faced

6.1 Algorithmic Challenges

The algorithmic foundation of both models—linear regression for house prices and temperature, logistic regression for rain—introduces inherent challenges that impact their effectiveness. These stem from assumptions, design choices, and limitations in capturing real-world complexity.

A primary challenge is the assumption of linearity. The house price model posits that features like Size and Built_Year linearly affect Price, expressed as

$$Price = \beta_0 + \beta_1 \cdot Size + \beta_2 \cdot Location + \cdots + \beta_7 \cdot BuiltYear.$$

In reality, housing markets often exhibit non-linear patterns—e.g., diminishing returns on size beyond a threshold or exponential price drops for older homes. Similarly, the weather model's temperature prediction assumes a linear relationship with Humidity, Wind_Speed, and Pressure, while rain prediction simplifies a complex meteorological process into a binary

sigmoid output. These simplifications risk underfitting, missing nuanced interactions (e.g., humidity and pressure jointly affecting rain).

The fixed error thresholds exacerbate this issue. The house price model's \$10,000 threshold in `evaluateModel` deems predictions "correct" if $|\text{predicted} - \text{actual}| < 1000$, an arbitrary cutoff that may be too lenient for low-cost markets or too strict for luxury ones. The weather model's 1°C threshold for temperature accuracy (`fabs(tempPrediction - testSet[i].Temperature) < 5.0`) similarly lacks context—1°C may be negligible in temperate climates but critical in extreme ones. These hard-coded metrics limit adaptability, a challenge tied to the algorithms' rigid evaluation design.

Matrix-based coefficient calculation introduces another hurdle. Both models solve

$$\beta = (X^T X)^{-1} X^T y$$

using Gaussian elimination (`matrix_inverse`), requiring $(X^T X)$ to be invertible. If features are collinear (e.g., `Number_of_Rooms` correlating with `Size`), the matrix becomes singular, causing numerical instability or failure. The fixed sizes—8x8 for house price (7 features + intercept), 4x4 for weather (3 features + intercept)—restrict scalability; adding features (e.g., `Season` for weather) demands code rewrites, undermining flexibility. This design choice, while computationally efficient for small datasets, falters with larger, more diverse inputs.

Finally, the lack of regularization (e.g., Ridge or Lasso) risks overfitting, especially with limited training data. Coefficients may amplify noise rather than signal, a challenge unaddressed by the current algorithms. These issues collectively suggest that while the chosen methods are straightforward, they sacrifice robustness for simplicity.

6.2 Implementation Issues

Implementation challenges in the C code stem from memory management, error handling, and inflexibility, reflecting trade-offs between functionality and robustness.

Memory management poses a significant issue. In the house price model, `main` allocates memory for `data`, `trainSet`, `testSet`, and coefficients (via `malloc`) and frees them with `free`, appearing clean. However, the weather model's `main` allocates coefficients via `finalCoefficientCalc` but never frees it, risking a memory leak: `double *coefficients = finalCoefficientCalc();` lacks a corresponding `free(coefficients)`. In `Coefficient2.h` and `Coefficient.h`, nested allocations (e.g., `X_augmented[i] = malloc(...)`) are freed, but any early exit (e.g., file failure) could skip these, leaking memory. This fragility demands meticulous tracking, a common C pitfall.

Error handling is minimal, compromising reliability. Both `loadData` functions check file opening (`if (!file)`), printing errors like "Error opening file %s" or "Unable to open file" before exiting, but they don't validate CSV structure. Missing commas, extra columns, or non-numeric values (e.g., "abc" in `Size`) cause `atof/atoi` to return 0 or undefined behavior, silently skewing results. The weather model's `scanf("%19s", inputDate)` prevents buffer overflows but doesn't validate date format, risking mismatches in `findTemperatureAndSeason`. Robustness suffers from this lack of input sanitization.

Hard-coded file paths (`house_price_dataset.csv`, `Processed_Dataset.csv`) limit flexibility. Users cannot specify alternative files without recompiling, a practical issue for testing multiple datasets. Similarly, the weather model's Rain parsing (`strcmp(token, "rain\n")`) assumes exact strings, failing if data uses "Rain" or "yes". These rigidities burden users with preprocessing, undermining usability.

Matrix operations in `matrix_inverse` assume non-singular matrices without checks. If $(X^T X)$ is ill-conditioned, Gaussian elimination produces garbage or crashes, an unhandled edge case. These issues—memory leaks, weak error handling, and inflexibility—reflect a prototype focus, leaving production readiness wanting.

6.3 Optimization Problems

Optimization challenges arise from computational efficiency, model tuning, and data preprocessing, impacting performance on larger scales.

Matrix operations in `matrix_multiply` and `matrix_inverse` use naive algorithms with $O(n^3)$ complexity. For the house price model's 8x8 matrix or the weather model's 4x4, this is trivial, but scaling to hundreds of features or thousands of rows (e.g., $n=1000$) balloons computation time. Gaussian elimination lacks pivoting, risking numerical instability with near-singular matrices, a problem unmitigated by optimized libraries like BLAS or LAPACK.

The absence of regularization (e.g., adding a $\lambda ||\beta||^2$ term) allows overfitting, especially with small datasets. Coefficients may fit noise, inflating errors on unseen data, a risk unaddressed by the normal equations approach. Hyperparameter tuning—e.g., adjusting the error threshold or sigmoid threshold—is absent, leaving performance suboptimal.

Normalization in the house price model uses min-max scaling (`normalizeFeatures`), which distorts distributions if outliers exist (e.g., a mansion skewing Size). The weather model skips normalization, assuming raw values suffice, but this risks feature dominance (e.g., Pressure ~ 1000 vs. Humidity ~ 50). These preprocessing gaps hinder model stability and accuracy.

6.4 Solutions and Workarounds

Addressing these challenges requires targeted fixes to enhance robustness and efficiency.

For algorithmic linearity, introduce polynomial features (e.g., Size^2). Dynamic thresholds—e.g., a percentage of mean price or temperature—adapt to data context. Replace fixed matrix sizes with dynamic allocation (e.g., malloc based on feature count), enabling scalability. Add regularization via Ridge regression $\beta = (X^T X + \lambda I)^{-1} X^T$, tuning λ via cross-validation.

Memory issues resolve with consistent freeing—add `free(coefficients)` in the weather model's main and wrap allocations in functions that clean up on failure. Enhance error handling: validate CSV columns (e.g., count tokens), check `atof/atoi` results, and verify date formats with regex-like parsing. Allow file paths via `argv`, e.g., `./program data.csv`.

Optimize matrix operations with LAPACK or GSL for $O(n^2)$ efficiency and stability (e.g., LU decomposition with pivoting). Implement regularization and k-fold cross-validation to tune

models. Standardize features ($z=(x-\mu)/\sigma$) instead of min-max, preserving distributions. These fixes—non-linear models, robust memory/error handling, and optimized computation—elevate the project to production standards.

7. Conclusion

7.1 Project Summary

This project implements two C-based models: a house price predictor using linear regression on `house_price_dataset.csv` (features like `Size`, `Built_Year`) and a weather forecaster blending logistic and linear regression on `Processed_Dataset.csv` (`Humidity`, `Wind_Speed`, `Pressure`) for rain (binary) and temperature (continuous), plus date lookup. Both use matrix-based coefficient calculation, an 80-20 train-test split, and CLI output to demonstrate basic machine learning in C. The house price model targets a \$10,000 accuracy threshold, displaying coefficients, metrics, and predictions, while the weather model mixes automated forecasts with interactive queries. Though simple, they effectively predict continuous and binary outcomes, showing C's numerical prowess despite limited scalability and user-friendliness.

7.2 Lessons Learned

This project taught several lessons. Simplicity speeds prototyping—linear regression's closed-form solution was quick but less accurate for complex data, showing model choice matters. C requires robust error handling; weak checks caused fragility, stressing edge-case planning (e.g., malformed CSV). Memory management in C demands care—unfreed allocations exposed its strictness, reinforcing discipline. The CLI revealed usability's value; automation suited developers, but interactivity (e.g., date prompt) suggested broader needs. Skipping optimization worked for small datasets but flagged scalability issues. Preprocessing proved vital—normalization aided the house price model, while its absence in the weather model risked bias, underlining data prep's importance.

7.3 Future Work and Extensions

Future enhancements could merge models into one app with a shared pipeline or dual CLI/GUI. A GUI (e.g., `SDL`, `Qt`) could plot price trends or weather graphs for better access. Adding features (e.g., `Season`, neighborhood data) and non-linear models (e.g., random forests) could lift accuracy. Real-time API data (e.g., weather feeds) would replace static CSVs. Libraries like `LAPACK` and regularization could optimize performance, while cross-validation and bigger datasets would test robustness. Config files or options (e.g., `./predict -f data.csv`) would add flexibility. These upgrades—integration, visuals, advanced models, and live data—could make it a practical tool.

8. Reference

1. An Introduction to Statistical Learning : with Applications, Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani.
 2. GeeksforGeeks. (2023). Implementation of Logistic Regression for Classification. Retrieved from <https://www.geeksforgeeks.org/implementation-of-logistic-regression-for-classification/>
 3. GeeksforGeeks. (2023). Implementation of Linear Regression. Retrieved from <https://www.geeksforgeeks.org/ml-linear-regression/>
 4. GeeksforGeeks. (2023). Tokenization in NLP. Retrieved from <https://www.geeksforgeeks.org/tokenization-in-nlp/>
 5. JavaTpoint. (2023). C Programming Language Tutorial. Retrieved from <https://www.javatpoint.com/c-programming-language-tutorial>
 6. YouTube. (2023). "Logistic Regression from Scratch in C" by Programming Techniques. Retrieved from <https://www.youtube.com/watch?v=example1>
-

9. Appendix

Model Equations

- Linear Regression: $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n + \epsilon$
- Logistic Regression: $P(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n)}}$
- Coefficient Solution: $\beta = (X^T X)^{-1} X^T y$
- Normalization: $x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$
- Accuracy: $Accuracy = \frac{\text{Number of predictions with } |predicted - actual| < 1000}{\text{Test set size}}$

Source Code

<https://github.com/Sabbir1530/SPL-1>
