Tiny Project Report

Instructor: Huynh Trung Hieu

Member: Nguyen Trong Huy - 10423046

Project Report: Linear System Solver and Evaluator

Introduction

The objective of this project is to design and implement an object-oriented C++ system that can solve linear systems using techniques from linear algebra. This includes handling regular square systems, symmetric positive definite systems, and non-square systems such as those encountered in data modeling.

The project is divided into two parts:

- **Part A:** Create reusable C++ classes to represent vectors, matrices, and various types of linear systems.
- **Part B:** Apply these classes to solve a real-world linear regression problem using the UCI CPU Performance dataset.

Through these parts, the project demonstrates how object-oriented programming (OOP) principles and numerical linear algebra can be combined to build modular, testable, and extensible computational tools.

GitHub Repository: `https://github.com/TrogHuy/tinyProjectFinal`

Overview of Implemented Classes

1. **Vector**
   Represents a dynamic array of doubles using 0-based indexing. Supports element access, dot product, scalar operations, and printing.

2. **Matrix**
   Implements a 2D array using 1-based indexing. Features include matrix operations (addition, multiplication, transpose), determinant and inverse for square matrices, and pseudo-inverse for non-square matrices.

3. **LinearSystem**
   A base class for solving square systems using Gaussian elimination. Accepts a square matrix A and a vector b and solves $Ax = b$.

   Code snippet:

Listing 1: Solving system using Gaussian elimination:

```

Vector LinearSystem::Solve() const {
    Matrix A(*mpA);
    Vector b(*mpb);
    Vector x(mSize); // result

     for (int i = 0; i < mSize; ++i) {
        // Pivoting (optional for now)
        assert(A(i + 1, i + 1) != 0.0);  // prevent divide-by-zero

        for (int k = i + 1; k < mSize; ++k) {
            double factor = A(k + 1, i + 1) / A(i + 1, i + 1);
            for (int j = i + 1; j <= mSize; ++j)
                A(k + 1, j) -= factor * A(i + 1, j);
            b[k] -= factor * b[i];
        }
    }

    // Back-substitution
    for (int i = mSize - 1; i >= 0; --i) {
        double sum = 0.0;
        for (int j = i + 1; j < mSize; ++j)
            sum += A(i + 1, j + 1) * x[j];

        x[i] = (b[i] - sum) / A(i + 1, i + 1);
    }

    return x;
}
```

4. **PosSymLinSystem**
   Inherits from LinearSystem, using the Conjugate Gradient method for symmetric positive definite matrices.

   Code snippet:

Listing 2: Solving PosSymLinSystem using Conjugate Gradient method

```

Vector PosSymLinSystem::Solve() const {
    int n = mpb->size();
    Vector x(n, 0.0);                 // Initial guess x0 = 0
    Vector r = *mpb;                  // Initial residual r0 = b - A x0 = b
    Vector p = r;                     // Initial direction
    Vector Ap(n);
    double rs_old = DotProduct(r, r);
```

```

    for (int k = 0; k < n; ++k) {
        Ap = (*mpA) * p;              // A pk
        double alpha = rs_old / DotProduct(p, Ap);
        for (int i = 0; i < n; ++i) {
            x[i] += alpha * p[i];
            r[i] -= alpha * Ap[i];
        }
        double rs_new = DotProduct(r, r);
        if (sqrt(rs_new) < 1e-10) break; // convergence
        for (int i = 0; i < n; ++i)
            p[i] = r[i] + (rs_new / rs_old) * p[i];
        rs_old = rs_new;
    }

    return x;
}
```

5. **NonSquareSystem**
   Solves $Ax = b$ where A is not square, using the Moore–Penrose pseudo-inverse computed as $(A^T A)^{-1} A^T$.

   Code snippet:

Listing 3: Solving NonSquareSystem using Moore-Penrose Pseudo-inverse

```

Vector NonSquareSystem::Solve() const {
    Matrix At = mpA->Transpose();
    Vector result(mCols, 0.0);

    if (mRows >= mCols) {
        // Over-determined system: x = (At * A)^(-1) At * b
        Matrix AtA = At * (*mpA);
        Matrix AtAInv = AtA.Inverse();
        Matrix pseudoInv = AtAInv * At;
        result = pseudoInv * (*mpb);
    } else {
        // Under-determined system: x = At (A At)^(-1) b
        Matrix AAt = (*mpA) * At;
        Matrix AAtInv = AAt.Inverse();
        Matrix pseudoInv = At * AAtInv;
        result = pseudoInv * (*mpb);
    }

    return result;
}
```

6. **DataLoader**
   Loads and parses the `machine.data` file from the UCI dataset, extracting the 6 input features as matrix A and the target PRP as vector b.

   Code snippet:

Listing 4: Loading machine.data file

```
void LoadDataFromFile(const string& filename, Matrix& A, Vector& b) {
    ifstream infile(filename);
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Failed to open file: " << filename << endl;
        return;
    }

    // First read: count rows
    string line;
    int rowCount = 0;
    while (getline(file, line)) {
        if (!line.empty()) rowCount++;
    }

    int numFeatures = 6;
    A = Matrix(rowCount, numFeatures);
    b = Vector(rowCount);

    // Second read: read values
    file.clear(); // enable second read
    file.seekg(0, ios::beg); // move back to beginning of file

    int row = 1;
    int index = 0;

    while (getline(infile, line)) {
        if(line.empty()) continue;

        stringstream ss(line);
        string token;
        int col = 0;
        int value;

        // Skip vendor name and model name
        getline(ss, token, ',');  // vendor name
        getline(ss, token, ',');  // model name

        // Read 6 features
        for (int j = 1; j <= numFeatures; ++j) {
```

```cpp
                getline(ss, token, ',');
                value = stoi(token);
                A(row, j) = value;
            }

            // PRP
            getline(ss, token, ',');  // PRP
            value = stoi(token);
            b[index] = value;

            row++;
            index++;
        }

        file.close();
}
```

7. **Evaluator**
   Provides functions to shuffle, split the dataset, compute RMSE, and remove outliers to improve model generalization.

   Code snippet:

Listing 5: Shuffle Function

```cpp
void shuffle(Vector &indices) {
    srand(time(0));
    for(int i = indices.size()-1; i > 0; i--) {
        int j = rand() % (i+1);
        swap(indices[i], indices[j]);
    }
}
```

Listing 6: Split data and compute RMSE

```cpp
void SplitTrainTest(const Matrix& A, const Vector& b,
                    Matrix& trainA, Vector& trainb,
                    Matrix& testA, Vector& testb,
                    double trainRatio=0.8) {
    int totalSamples = b.size();
    int numFeatures = A.getNumCols();
    int trainSize = static_cast<int>(totalSamples * trainRatio)
        ;
    int testSize = totalSamples - trainSize;

    // Shuffle indices
    Vector indices(totalSamples);
    for (int i = 0; i < totalSamples; ++i)
        indices[i] = i;
```

```
    shuffle(indices);

    // Allocate memory
    trainA = Matrix(trainSize, numFeatures);
    trainb = Vector(trainSize);
    testA = Matrix(testSize, numFeatures);
    testb = Vector(testSize);

    // Fill training set
    for (int i = 0; i < trainSize; ++i) {
        int idx = static_cast<int>(indices[i]);
        trainb[i] = b[idx];

        for (int j = 0; j < numFeatures; ++j) {
            trainA(i + 1, j + 1) = A(idx + 1, j + 1);
        }
    }

    // Fill test set
    for (int i = 0; i < testSize; ++i) {
        int idx = static_cast<int>(indices[trainSize + i]);
        testb[i] = b[idx];

        for (int j = 0; j < numFeatures; ++j) {
            testA(i + 1, j + 1) = A(idx + 1, j + 1);
        }
    }
}

double computeRMSE(const Vector& predicted, const Vector&
    actual) {
    assert(predicted.size() == actual.size());
    double sumSquaredError = 0.0;

    for(int i = 0; i < predicted.size(); i++) {
        double diff = predicted[i] - actual[i];
        sumSquaredError += diff * diff;
    }

    return sqrt(sumSquaredError / predicted.size());
}
```

Outlier removal explanation:

RemoveOutliers filters out samples where the target value exceeds a threshold. We chose the default max_threshold = 400, as over 98% of PRP values fall below this. This helps reduce the influence of extreme outliers. Lowering the threshold removes more noise but risks data loss, while raising it may inflate RMSE due to noisy points.

Function code:

Listing 7: Remove Outliers function

```cpp
void removeOutlierse(Matrix &A, Vector &b, double max_threshold=400) {
    int rows = A.getNumRows();
    int cols = A.getNumCols();

    int validCount = 0;
    for (int i = 0; i < b.size(); ++i) {
        if (b[i] <= max_threshold) {
            validCount++;
        }
    }

    Matrix filteredA(validCount, cols);
    Vector filteredb(validCount);

    int newRow = 1;  // 1-based indexing for Matrix
    for (int i = 0; i < b.size(); ++i) {
        if (b[i] <= max_threshold) {
            filteredb[newRow - 1] = b[i];
            for (int j = 1; j <= cols; ++j) {
                filteredA(newRow, j) = A(i + 1, j);
            }
            newRow++;
        }
    }
    A = filteredA;
    b = filteredb;
}
```

Applying the Classes to Part B

In Part B, the classes are used to build a linear regression model that predicts CPU performance (PRP) from features: MYCT, MMIN, MMAX, CACH, CHMIN, CHMAX.

Workflow:

1. Load the dataset using `LoadDataFromFile("machine.data", A, b);`

2. Split the data using `SplitTrainTest(A, b, trainA, trainb, testA, testb);`

3. Solve:

```cpp
NonSquareSystem model(&trainA, &trainb);
Vector x = model.Solve();
```

4. Predict: `Vector predicted = testA * x;`
5. Evaluate: `double rmse = ComputeRMSE(predicted, testb);`

Result:

Using an 80/20 train-test split across multiple runs, this is the range that result mostly varies in:

```
Before removing outlierse:
    RMSE on train set: 60 - 70
    RMSE on test set: 55 - 100

After removing outlierse:
    RMSE on train set: 30 - 38
    RMSE on test set: 25 - 55
```

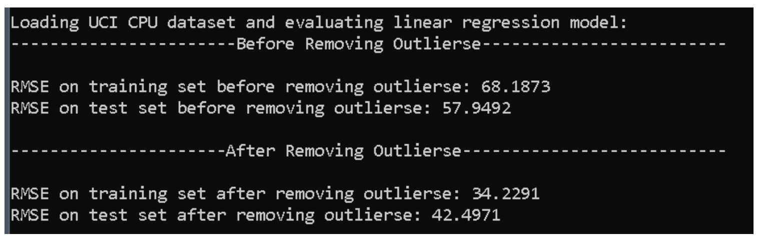

Figure 1: Sample Result

Conclusion

This project highlights the effectiveness of combining OOP and numerical techniques for linear regression tasks. The system is modular, extensible, and performs reasonably well on real-world data.

References

- `https://archive.ics.uci.edu/ml/datasets/Computer+Hardware`
- Golub, G. H., & Van Loan, C. F. (2013). *Matrix Computations.* JHU Press.
- Trefethen, L. N., & Bau, D. (1997). *Numerical Linear Algebra.* SIAM.