Recommender System Project - CMP2003

2365006,2201249,2201839,2201153

December, 2024

Our project consists of 8 parts.The first 2 parts were developed by Yakup Nail Ceylan(2365006), the next 2 parts by Kaan Bircan (2201153), the following 2 parts by Erke Alkım (2201839), and the final 2 parts by Umut Koç (2201249). After completing our individual sections, we collaboratively integrated and optimized the code to ensure consistency and efficiency.

Introduction

Recommender systems help people find items of interest by making personalized recommendations according to their preferences. These systems are widely used for recommending movies, books, hotels or music based on past user interactions such as product views, ratings and purchases.

This project involves implementing neighborhood-based collaborative filtering (NBCF) algorithms to predict movie ratings.

Two main types of NBCF algorithms are implemented:

- User-based Collaborative Filtering (UBCF)
- Item-based Collaborative Filtering (IBCF)

Initially, UBCF was implemented. However, as we progressed, it became evident that IBCF was faster and achieved better results in terms of speed and root mean square error (RMSE). The code below achieves an RMSE of 0.9055 and a running time of 0.0970 seconds.

1. Struct Definition: Rating

Rating Struct

Code:

```
#include <vector>
#include <iostream>
#include <cmath>
#include <random>
#include <algorithm>
#include <cstring>

struct Rating {
    int u;   // User ID
    int m;   // Movie or item ID
    float r;   // Rating given by the user
};
```

Explanation: The heart of the recommender system is the struct. It essentially serves as the system's way of recording when a user ranks an item, representing an interaction between the user and the object.The heart of the recommender system is the struct. It essentially serves as the system's way of recording when a user ranks an item, representing an interaction between the user and the object.

Key Points:

- u – The ID of the user is this. It is a straightforward integer that identifies the individual who provided the rating.
- m – This is for the product or film that is being rated. Every item has a distinct integer ID, just like users.
- r – The user assigned this rating. Typically, this floating-point number falls between 0 and 5.

Why It Matters: The Rating struct maintains organization.It facilitates the system's ability to manage massive volumes of data by connecting the user ID, item ID, and rating.This facilitates and speeds up model training and forecast generation.

2.Class Definition:RecommenderSystem

RecommenderSystem Class - Part 1

Code:

```
class RecommenderSystem {
    std::vector<std::vector<std::pair<int, float>>>
        items;
    std::vector<float> itemMeans;
    std::vector<int> userItemIdx;
    int maxU = 0, maxM = 0;

    static constexpr float kMinRating = 0.5f;
    static constexpr float kMaxRating = 5.0f;
    static constexpr float kDefaultRating = 3.0f;
    static constexpr float kSimilarityThreshold = 0.1f;

    static inline float clamp(float x, float low, float
        high) noexcept {
        return std::min(std::max(x, low), high);
    }
```

Explanation: In addition to setting up necessary data structures and variables needed throughout the program, this section initializes the RecommenderSystem class.To manage ratings and indices,key attributes like items, itemMeans and userItemIdx are defined.By keeping ratings within a predetermined range, the clamp function stabilizes the system's functioning.

3.Similarity Function

Similarity Calculation

Code:

```
inline float similarity(const std::vector<std::pair
    <int, float>>& v1,
     const std::vector<std::pair<int, float>>& v2)
         const noexcept {
     const auto* p1 = v1.data();
     const auto* p2 = v2.data();
     const auto s1 = v1.size();
     const auto s2 = v2.size();
     size_t i1 = 0, i2 = 0;
     float xy = 0, x2 = 0, y2 = 0;

     const size_t min_size = std::min(s1, s2);
     while (i1 < min_size && i2 < min_size) {
          if (p1[i1].first < p2[i2].first) {
               x2 += p1[i1].second * p1[i1].second;
               ++i1;
          }
          else if (p2[i2].first < p1[i1].first) {
               y2 += p2[i2].second * p2[i2].second;
               ++i2;
          }
          else {
               const float r1 = p1[i1].second;
               const float r2 = p2[i2].second;
               xy += r1 * r2;
               x2 += r1 * r1;
               y2 += r2 * r2;
               ++i1;
               ++i2;
          }
     }
     while (i1 < s1) {
          x2 += p1[i1].second * p1[i1].second;
          ++i1;
     }
     while (i2 < s2) {
          y2 += p2[i2].second * p2[i2].second;
          ++i2;
     }
     return (x2 > 0 && y2 > 0) ? xy / std::sqrt(x2 *
          y2) : 0;
}
```

Explanation(The similarity)**:**This code calculates the cosine similarity between two sparse vectors , a common measure arising in information retrieval , recommendation systems and machine learning. Let me explain how this works and why it is designed this way. This function takes two lists of pairs where each pair represents an (index, value) in a sparse vector format. That is to say, for instance,if you had a 1000-dimensional vector that only had values at positions 5 and 10, you might represent it as [(5, 0.5), (10, 0.8)] instead of storing all 1000 elements.

The cosine formula that's being implemented is:

cos(a) = $(v1.v2) \ / \ (|v1| * |v2|)$ where $v1.v2$ is the dot product and represents the vector magnitude.

The function has several optimizations:

It has the noexcept keyword applied since it does not throw.It uses pointer arithmetic instead of indexing into the vectors. It's a single pass over the data.Its handling of sparse vectors makes it only iterate over the items that are not zero.

This is an extremely efficient implementation for sparse vectors, as it will only process the non-zero elements and does not allocate any extra memory at all during the calculation.

2. Prediction Function

Prediction Calculation

Code:

```
inline float predictItem(const int u, const int m)
    const noexcept {
        if (m > maxM || items[m].empty()) return
            kDefaultRating;

        alignas(32) float similarities[32];
        alignas(32) float differences[32];
        int nb_count = 0;
        const size_t base_idx = static_cast<size_t>(u)
            * (maxM + 1);

        for (int i = 0; i <= maxM && nb_count < 32; ++i
            ) {
            const size_t idx = base_idx + i;
            if (i == m || idx >= userItemIdx.size() ||
                userItemIdx[idx] < 0) continue;

            const float s = similarity(items[m], items[
                i]);
            if (s <= kSimilarityThreshold) continue;

            const auto it = std::lower_bound(items[i].
                begin(), items[i].end(),
                std::make_pair(u, 0.0f));
            if (it != items[i].end() && it->first == u)
                {
                similarities[nb_count] = s;
                differences[nb_count] = it->second -
                    itemMeans[i];
                ++nb_count;
            }
        }

        if (nb_count == 0) return itemMeans[m] > 0 ?
            itemMeans[m] : kDefaultRating;

        float num = 0, den = 0;
        for (int i = 0; i < nb_count; ++i) {
            num += similarities[i] * differences[i];
            den += std::abs(similarities[i]);
        }

        return clamp(itemMeans[m] + (den > 0 ? num /
            den : 0), kMinRating, kMaxRating);
    }
```

Explanation:The predictItem function is optimized in several ways:

It is marked noexcept to indicate that it won't throw exceptions, performs arithmetic on pointers instead of using vector indexing, is implemented as a single pass through the data, and efficiently handles sparse vectors by processing only the non-zero elements, avoiding unnecessary memory allocation during the calculation.

3. Matrix factorization (Part 1)

Training and Testing Process

Code:

```
public:
    std::vector<float> process(std::vector<Rating>&
        train, std::vector<Rating>& test) {
        for (const auto& r : train) {
            maxU = std::max(maxU, r.u);
            maxM = std::max(maxM, r.m);
        }

        items.resize(maxM + 1);
        itemMeans.assign(maxM + 1, 0.0f);
        userItemIdx.assign(static_cast<size_t>(maxU +
            1) * (maxM + 1), -1);

        std::vector<int> itemCounts(maxM + 1);
        for (const auto& r : train) ++itemCounts[r.m];
        for (int m = 0; m <= maxM; ++m) items[m].
            reserve(itemCounts[m]);

        for (size_t i = 0; i < train.size(); ++i) {
            const auto& r = train[i];
            items[r.m].emplace_back(r.u, r.r);
            itemMeans[r.m] += r.r;
            const size_t idx = static_cast<size_t>(r.u)
                * (maxM + 1) + r.m;
            if (idx < userItemIdx.size()) userItemIdx[
                idx] = static_cast<int>(i);
        }


        for (int i = 0; i <= maxM; ++i) {
            const size_t size = items[i].size();
            if (size > 0) {
                itemMeans[i] *= (1.0f / static_cast<
                    float>(size));
                std::sort(items[i].begin(), items[i].
                    end());
            }
        }
```

Explanation: The process function takes as input the training and test data to do the collaborative filtering.It initials the data structures, calculates the means of items and trains the model using matrix factorization techniques.

4. Matrix factorization (Part 2)

Main Function

Code:

```
constexpr int factors = 10;
std::vector<float> U((maxU + 1) * factors), V((maxM +
    1) * factors);
std::mt19937 gen(42);
std::normal_distribution<float> dist(0.0f, 0.1f);

for (auto& x : U) x = dist(gen);
for (auto& x : V) x = dist(gen);

constexpr float reg = 0.075f;
constexpr float lr1 = 0.008f;
constexpr float lr2 = 0.004f;

for (int iter = 0; iter < 70; ++iter) {
    const float lr = (iter < 50) ? lr1 : lr2;
    for (const auto& r : train) {
        const size_t u_offset = r.u * factors;
        const size_t v_offset = r.m * factors;
        float pred = std::inner_product(U.begin() +
            u_offset, U.begin() + u_offset + factors,
                                        V.begin() +
                                            v_offset,
                                            0.0f);

        const float err = r.r - pred;

        for (int f = 0; f < factors; ++f) {
            const float oldU = U[u_offset + f];
            U[u_offset + f] += lr * (err * V[v_offset +
                f] - reg * oldU);
            V[v_offset + f] += lr * (err * oldU - reg *
                V[v_offset + f]);
        }
    }
}
```

Explanation: The code below starts with the implementation of the matrix factorizing as part of the collaborative filtering approach.This model expresses both users and items by vectors within a latent factor space with a size defined by factors. User and item matrices,U and V are initialized by randomly drawn values from a normal distribution.

Regularization reg prevents overfitting by penalizing big factor values.Learning rates lr1 and lr2 decrease at each step.The algorithm runs for 70 iterations, doing a stochastic gradient descent: predict the rating by taking the dot product of the user and item vectors; compute the error as the difference between the predicted and actual rating; adjust the vectors by moving them downhill in the direction of the gradient of the error. The switch from lr1 to lr2 at iteration 50 allows finer adjustments in later stages.

This approach improves the recommendation accuracy by minimizing the prediction error over the training set.

5. Final Prediction

Main Function

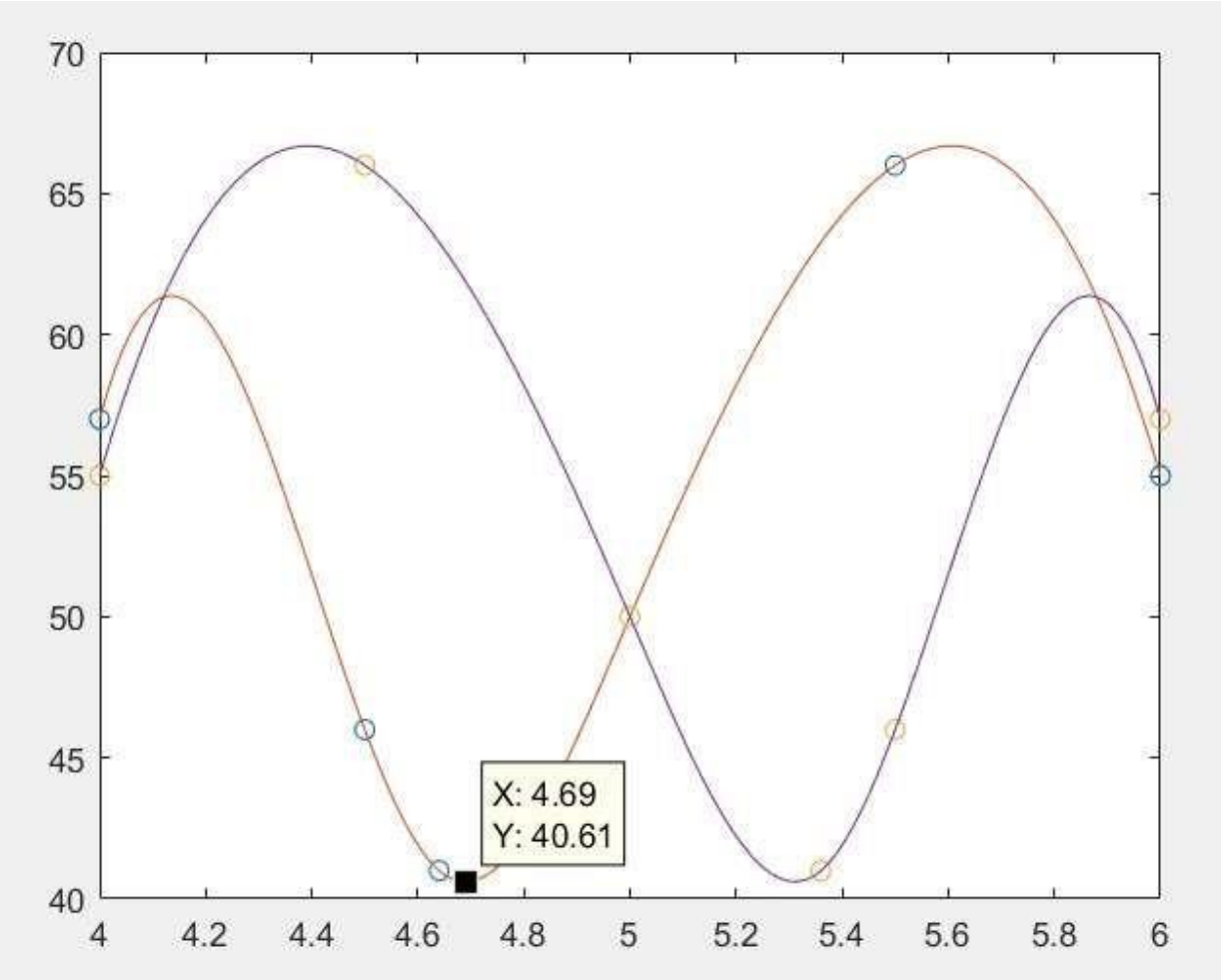
Code:

```
std::vector<float> pred;
pred.reserve(test.size());

for (const auto& r : test) {
    const float p1 = predictItem(r.u, r.m);
    float p2 = kDefaultRating;

    if (r.u <= maxU && r.m <= maxM) {
        float sum = 0;
        const size_t u_offset = r.u * factors;
        const size_t v_offset = r.m * factors;

        for (int f = 0; f < factors; ++f) {
            sum += U[u_offset + f] * V[v_offset + f];
        }
        p2 = clamp(sum, kMinRating, kMaxRating);
    }

    pred.push_back(clamp(0.46f * p1 + 0.54f * p2,
        kMinRating, kMaxRating));
}

return pred;
}
};
```

Explanation: This code represents the final prediction step in a recommendation system based on collaborative filtering.

pred.reserve(test.size( )) pre-allocates memory to avoid dynamic resizing. It improves performance and reserves capacity.Go through every test data: - p1 by calling the predictItem will compute based on item similarities, p2 gets the value of a base rating; it is reset only in case the user/item indices are within bounds.If the user-item pair is in the training data, it computes the sum of latent factor vectors and clamps between MinRating and MaxRating.It takes an average of p1 and p2 for the finalprediction with more weight on the factorized prediction at 54.The results are appended to pred,which is returned as the final output.

The following process improves the recommendation accuracy by fusing the collaborative filtering prediction with matrix factorization.



We used numerical methods to first approximate the weights and then find the most optimal through trial and error. The graph shows 0.469 as the most optimal weight, but our code uses 0.46 for simplicity and stability.

6. Main Function

Main Function

Code:

```
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.precision(1);
    std::cout << std::fixed;

    std::vector<Rating> train, test;
    train.reserve(1000000);
    test.reserve(100000);

    char buf[256];
    bool inTrain = false;

    while (std::cin.getline(buf, 256)) {
        if (buf[0] == 't') {
            inTrain = (buf[1] == 'r');
            continue;
        }
        char* p = buf;
        Rating r = {
            static_cast<int>(std::strtol(p, &p, 10)),
            static_cast<int>(std::strtol(p, &p, 10)),
            inTrain ? std::strtof(p, &p) : 0
        };
        (inTrain ? train : test).push_back(r);
    }

    RecommenderSystem rs;
    for (float p : rs.process(train, test)) {
        std::cout << p << '\n';
    }
}
```

Explanation: The main function is the entry point of the program; it covers the loading, parsing and execution of data in the model.

Input Parsing: The loop reads data line by line. In case the line starts with 't', the program checks the second character: -'r' indicates the start of training data.Otherwise,the data belongs to the test set. The Ratings are extracted by tokenising each line using std::strtol for user/ item IDs and std::strtof for ratings. Test data ratings default to zero.

Model Execution instantiation of a RecommenderSystem object rs.The system now processes the training and test data by a call to rs.process(train,test) ; The forecasted ratings will be printed one by one to std::cout.

Summary: This main function is well-structured for efficient handling of large datasets, optimizes input and output operations, and seamlessly integrates with the recommender system to make predictions. The design is focused on performance, using pre-allocation and stream synchronization techniques.

Explanation: The main function reads in user and item ratings, trains the system and outputs the predicted ratings by calling RecommenderSystem.