IML HW3

Implement PCA in C/C++ (using Eigen Library)

1. To generate the dataset, we can use the following code. It will generate different csv files with random numbers as per the requirement. We can then choose to use these CSV files.

```
#include <iostream>
#include <fstream>
#include <string>
#include "Eigen/Dense"

using namespace std;
using namespace Eigen;
void saveDatasetToCsv(MatrixXd dataset, string filename) {
    ofstream file(filename);
    if (file.is_open()) {
        for (int i = 0; i < dataset.rows(); i++) {
            for (int j = 0; j < dataset.cols(); j++) {
                file << dataset(i, j);
                if (j < dataset.cols() - 1) {
                    file << ",";
                }
            }
            file << endl;
        }
        file.close();
    }
}
int main() {
    int dimensions[5][2] = {{100, 100}, {1000, 1000}, {10000, 10000},
{10000, 50000}, {50000, 50000}};
    for (int i = 0; i < 5; i++) {
        int m = dimensions[i][0];
        int n = dimensions[i][1];
        MatrixXd dataset = MatrixXd::Random(m, n);
        string filename = "dataset_" + to_string(m) + "x" + to_string(n) +
".csv";
        saveDatasetToCsv(dataset, filename);
    }
    return 0; }
```

2. PCA is a statistical technique used to reduce the dimensionality of high-dimensional datasets while retaining most of the variability in the data. Let's divide the code into smaller parts and explain each segment:
    a. This section includes the necessary libraries for the code, such as iostream (for input and output), fstream (for file input and output), string (for string operations), and the Eigen library (for matrix and linear algebra operations).

```
//include libraries
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include "Eigen/Dense"
#include "Eigen/Eigenvalues"
```

    b. This line allows us to use the Eigen library in our code without having to specify the namespace every time.

```
//use namespace
using namespace Eigen;      // to use the Eigen library
```

    c. This function generates a random dataset of size m by n using the MatrixXd data type provided by the Eigen library. It first initializes a matrix dataset of size m by n, and then fills it with random numbers using the MatrixXd::Random method. Finally, it returns the generated matrix.

```
//generate random dataset
Eigen::MatrixXd generateRandomDataset(int m, int n) {
    Eigen::MatrixXd dataset(m, n);                  // m rows, n columns
    dataset = Eigen::MatrixXd::Random(m, n);        // fill with random
numbers
    return dataset;                                 // return the matrix
}
```

    d. This function performs PCA on the input dataset dataset and returns the numComponents principal components. The first step is to center the data around the mean, which is done by subtracting the mean of each column from each element in that column. Then, the covariance matrix is calculated as the dot product of the centered dataset and its transpose. The covariance matrix is then decomposed into its eigenvalues and eigenvectors using the EigenSolver class provided by the Eigen library. Finally, the function returns the numComponents rightmost columns of the eigenvectors, corresponding to the principal components.

```
//pca stuff
Eigen::MatrixXd performPCA(Eigen::MatrixXd dataset, int numComponents) {
    Eigen::MatrixXd centered = dataset.rowwise() -
dataset.colwise().mean();         // center the data around the mean -
normalize the data
    Eigen::MatrixXd cov = centered.adjoint() * centered;
// calculate the covariance matrix
    Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> eig(cov);
// calculate the eigenvalues and eigenvectors
    Eigen::MatrixXd eigenVectors = eig.eigenvectors();
// get the eigenvectors
    return eigenVectors.rightCols(numComponents);
// return the rightmost columns of the eigenvectors - because these are
the ones with the highest variance - the principal components
}
//main function
int main()
{
    int m = 100;
    int n = 100;
    int numComponents = static_cast<int>(n * 0.1);
// 10% of the number of columns of the dataset
    MatrixXd dataset = MatrixXd::Random(m, n);
// generate a random dataset

    auto start = std::chrono::high_resolution_clock::now();
    MatrixXd principalComponents = performPCA(dataset, numComponents);
// perform PCA on 0.1% of the dataset
    auto stop = std::chrono::high_resolution_clock::now();

    std::cout << "Principal components:\n" << principalComponents <<
std::endl;     // print the principal components

    auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
    std::cout << "Time taken by function: " << duration.count() << "
microseconds" << std::endl;

    return 0;
}
```

For the 100x100 dataset, we will run both our implementation of PCA and sklearn.decomposition.pca, and compare the runtime.

Runtime (PCA Implementation in C++): **164797 microseconds**
Runtime (sklearn implementation): **58947.801590 microseconds**

We can say that sklearn implementation is 2.8 times faster than our pca implementation.

Related files:

1. generateData.cpp
2. Pca.cpp
3. pca.py
4. dataset_100x100
5. dataset_1000x1000