ALGORITHMS AND PARALLEL COMPUTING

Augmented Matrix Factorization with Explicit Labels for Recommender Systems

CODE DOCUMENTATION

Lara Bombardieri Andrea Brandoli Matteo Limberto

1 INTRODUCTION

This document provides a brief guide to compile and run the source code implementing the recommendation algorithm denominated *Augmented Matrix Factorization with Explicit Labels for Recommender Systems*. This code has been developed by our group in the framework of the *Algorithms and Parallel Computing* class held by Professor Cremonesi at Politecnico di Milano. The paper describing the aforementioned algorithm was provided to us.

2 COMPILATION

A compiler supporting the C++11 standard is required in order to compile the source code. We used the latest versions of the `g++` and `clang++` compilers.

2.1 EXTERNAL LIBRARIES

The code is built upon the ARMADILLO library (`http://arma.sourceforge.net`) which is used to manage matrices and linear-algebra operations. This library is in turn built upon the BLAS and LAPACK linear-algebra libraries. These three libraries must be installed before the compilation of the project is initiated.

2.2 Compilation using *CMake*

We used the *CMake* tool in order to generate the *MakeFile* for our code and the *CMakeLists.txt* is provided in the code folder. This was particularly useful since in our team we were not using the same platform.

To compile, just type

```
cmake <path-to-folder-containing-CMakeLists>
```

and then

```
make
```

We recommend not to compile in the same folder where the source code is located.

2.3 Manual compilation

Compilation can also be performed manually. In this case the following order of dependency shall be respected during the process :

- `my_utils.cpp`
- `compute_v.cpp   proj_gradient.cpp   amf.cpp`
- `main.cpp`

2.4 Other settings

By default a lot of print-outs are visualised on screen so that the execution can be closely followed by the user. The output can be heavily reduced by defining the `NDEBUG` preprocessor macro at compile time. To do so, just add the `-DNDEBUG` compiler flag.

By defining the `AMFTIME` preprocessor macro (with the same procedure as above) the time elapsed during the execution of some time-wise critical tasks is tracked.

By defining the `AMFOPENMP` preprocessor macro parallelisation within the *OpenMP* framework is enabled. The operations which are time-wise most critical have been parallelised so that execution speed is drastically increased. Please, remember to set the number of threads you wish to use via the `OMP_NUM_THREADS` environment variable before running the executable.

An additional increase in speed can be obtained by using parallelized versions of the *BLAS* library, although we didn't experiment this solution.

3 Execution

The main task performed by our code is to solve the minimisation problem described in the paper that was given to us. The final output consists of three matrices (namely U, H, V) that allow us to perform recommendation.

In particular, the rating given to item j from user i is predicted as

$$R = (UHV)_{ij}$$

3.1 Input matrices and parameters

We must give as an input to the executable the user-rating matrix and the item-content matrix. The format in which they must be given is a text file where each row contains two integer indexes (row and column index) and a value corresponding to these indexes. These three values are separated by a whitespace.

In the `MATLAB` sub-folder a script which extracts a suitable input file from a `.mat` matrix file is provided.

The parameters are stored in the `parameters.txt` file which can be modified by the user. This file is then given as an input to the solver.

3.2 Running the solver

To perform recommendation we need to solve a complex optimisation problem. This is done in the `main_solve.cpp` main file.

The program creates an instance of the solver class, imports the user-rating and item-content matrices along with the parameters and eventually runs the optimisation procedure. The resulting matrices are then exported using the Armadillo binary file format.

Once the code compiled, to run the solver just type

```
./amf_out <path-to-URM-matrix> <path-to-ICM-matrix> <path-to-parameters-file>
```

The results we're looking for are then exported according to the following filename

```
amf_<n-latent-factors>_<lambda>_<name-matrix>
```

where `n-latent-factors` and `lambda` are set in the parameters file.

3.3 Tuning the solver

Since the optimal parameters are unknown to the user, a tuning procedure has been implemented so that they can be easily identified. This is done in the `main_tuning.cpp` main file.

An additional validation user-rating matrix is given as an input to the solver. At every iteration, predictions of the ratings are evaluated for the elements of the validation URM and the goodness of the estimations is evaluated via *RMSE* (root mean-square error). These values are then stored on a log-file.

The path to the validation user-rating matrix must be added as an input to the executable.

3.4 Getting Top-N recommendations

In order to evaluate the goodness of the recommendation system, *Top N* predictions must be computed for every user once the solving procedure is completed. This is done in the `main_get_topN.cpp` main file.

The results from the solving procedure are loaded in memory and then the matrix containing the *Top N* recommendations for every user is computed and, optionally, exported in a `.txt` file.

The executable must be run as in the solving procedure. Once this has been done, *precision* and *recall* can be evaluated by running the script provided in the `MATLAB` sub-folder.

4 Description of the AMF class

The core of the code is encapsulated into a class denominated AMF, which stores the matrices as attributes and contains all the methods needed to solve the optimisation problem, which eventually consists of solving three convex optimisation subproblems via projected-gradient methods.

4.1 Constructors

```
AMF();

AMF(std::string URM_Tr_filename, std::string ICM_filename, std::string
    param_filename);
```

Two constructors have been implemented, the first one being the default constructor (which creates a basically empty instance). The latter instead requires as parameters the paths to the user-rating matrix, item-content matrix and parameters file respectively.

4.2 Public methods : solvers

```
void solve();

void solve_With_Log();

void solve_For_Tuning(std::string mfilename);

void solve_For_Tuning_With_Log(std::string mfilename);

arma::umat get_TopN_Recommendation(arma::uword N, bool export_to_file = true);
```

The first two methods of the above list solve the optimisation problem and compute the resulting matrices (U, H, V) needed to perform recommendation. The `solve_With_Log()` method evaluates the objective function at every gradient sub-iteration, hence it is significantly slower than the first one but allows for detailed tracking of the objective function.

The `solve_For_Tuning()` method additionally evaluates the error with respect to a validation matrix, as explained in section 3.

The last method extracts a matrix containing *Top N* recommendations for every user.

4.3 Public methods : input / output

```
inline void set_lambda(double lambda) { lambda_ = lambda; }
inline void set_n_max_iter(unsigned int n) { n_max_iter_ = n; }
inline void set_n_latent_factors(arma::uword r) { r_ = r; }
inline void set_n_max_iter_gradient(unsigned int n) { n_max_iter_gradient_=n;}
inline void set_toll_gradient(double toll) { toll_gradient_ = toll; }
inline double get_lambda() { return lambda_; }
inline unsigned int get_n_max_iter() { return n_max_iter_; }
inline arma::uword get_n_latent_factors() { return r_; }
inline unsigned int get_n_max_iter_gradient() { return n_max_iter_gradient_; }
inline double get_toll_gradient() { return toll_gradient_; }
inline double get_gradient_step() { return gradient_step_; }
inline void print_ICM(){ICM_.print("ICM =");}
inline void print_URM_Tr(){URM_Tr_.print("URM =");}
void export_Results();
void import_Results();
```

Most of these methods are quite self-explanatory. We wish to draw your attention to the last two methods of the list, which respectively export and import the U, H, V matrices in *ARMA* binary file format.