Computing Large Scale Linear Least Squares with SKiLLS

Zhen Shao

December 9, 2020

1 Overview

SKiLLS (SKetchIng-Linear-Least-Sqaures) is a C++ package for finding solutions to over-determined linear least square problems. SKiLLS uses a modern dimensionality reduction technique called sketching, and is particularly suited for large scale linear least squares where the number of measurements/observations is far greater than the number of variables.

Mathematically, SkiLLS solves

$$\min_{x \in \mathbb{R}^d} f(x) := \|Ax - b\|_2^2, \tag{1}$$

where $A \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$ given. The matrix A is allowed to be rank-deficient or nearly rank-deficient.

1.1 When to use SKiLLS

Problem (1) frequently appears as a sub-problem in many computational and data science problems, for example, in many non-linear/general function minimisation routine linearises the problem and iteratively solve problems of the form (1). In the following situations using SKiLLS yields significant computational advantage comparing to state-of-the-arts[1]:

1. A is dense, large, sufficiently over-determined with unknown rank.
2. A is dense, large, sufficiently over-determined with high coherence[2].
3. A is sparse, large, sufficiently over-determined with non-zero entries randomly distributed.
4. A is moderately sparse (e.g. one percent of entries are non-zero), large, moderately over-determined.

In the following situations using SKiLLS is competitive with state-of-the-arts

1. A is dense, large, sufficiently over-determined with full rank.
2. A is sparse, large, moderately over-determined.

For further details of computational advantages, see our paper. *Computing Large Scale Linear Least Squares with SKiLLS, C. Cartis and Z. Shao.*

[1]Blendenpik [1] for dense problems, SPQR [2] and Cholesky-preconditioned LSQR [4] for sparse problems

[2]The coherence of a matrix A is defined as the largest Euclidean norm of U, where U is defined in the reduced singular value decomposition of A.

2 Installing SKiLLS

2.1 Dependency

2.2 Installation instruction

2.3 Test the installation

3 Using SKiLLS

This is a libary of two/three routines etc....

3.1 Data structure

Maybe point out to the header file (or quote the header file).

Dense Vector *SKiLLS uses C++ class Vec for dense vectors. To create a* $n \times 1$ *vector from already existed data, take a pointer to the data and call Vec(n, *data).*

Dense Matrix *SKiLLS uses C++ class Mat for dense matrices. To create a* $n \times d$ *matrix from already existed data, take a pointer to the data and call Mat(n, d, *data). Basic matrix operations such as return a specific row, column, or matrix-matrix and matrix vector multiplications are implemented.*

Sparse Matrix *SKiLLS uses Compressed Column Data structure for sparse matrices and vectors. The sparse data structure is from SuiteSparse [3]. In particular, the cs_dl structure used as the input matrix format for sparse linear least squares is taken from the CXSparse package from SuiteSparse. It is a C structure with 7 fields:*

- *nzmax: maximum number of non-zeros.*
- *m: number of rows.*
- *n: number of columns.*
- **p: column indices (size nzmax)*
- **i: row indices, size nzmax.*
- **x: numerical values of non-zeros, size nzmax.*
- *nz: -1 for compressed column format.*

Sometimes we will also use the data structure cholmod_sparse for sparse matrix in compressed column format. It has a similar structures. For more information, see the header file or documentation of SuiteSparse.

Examples *We give two examples of constructing dense and sparse matrices from user given data.*

The below code snippet creates a dense matrix and a vector.

```
long m = 4;
long n = 2;
double data_A[8] = {0.306051, 1.53078,1.64493, 1.61322,
          0.2829,0.474476, 0.586278, 0.610202};

double data_b[4] = {0.0649338,0.845946, 0.0164085,0.247119};

Mat_d A(m,n,data_A);
Vec_d b(m,data_b);
```

The below code snippet creates a sparse matrix.

```
long m = 4;
long n = 3;
long nnz = 5;

long col[4] = {1, 2, 4, 6};
long row[5] = {1, 1, 3, 2, 4};
double val[5] = {2.0, 3.0, 1.0, 4.0, 5.0};
long col_0[4];
long row_0[5];

// convert to zero based indices
for (long i=0; i<n+1; i++){
 col_0[i] = col[i] 1;
}

for (long i =0; i<nnz; i++){
 row_0[i] = row[i] 1;
}

cholmod_common Common, *cc;
cc = &Common;
cholmod_l_start(cc);
cc >print = 4;

cholmod_sparse* A;
A = cholmod_l_allocate_sparse(
m, n, nnz, true, true, 0, CHOLMOD_REAL, cc);

A >p = col_0;
A >i = row_0;
A >x = val;

cholmod_l_print_sparse( A, "A", cc);

cholmod_l_finish(cc);
```

3.2 Dense problems

To compute a solution of a linear least square problem where the matrix A is dense, a call of the following form should be made.

ls_dense_hashing_blendenpik(A, b, x, rank, flag, it, gamma, k, abs_tol, rcond, it_tol, max_it, debug, wisdom):

seperate into input, output and options .... typical correct values of these parameters

- *A is an input of derived type containing the $\mathbb{R}^{m\times n}$ matrix.*
- *b is an input of derived type containing the $\mathbb{R}^{m}$ vector.*
- *x is an output of derived type containing the $\mathbb{R}^{n}$ vector solution on exit.*
- *rank is an output of derived type containing a scaler, detected rank of the matrix A.*
- *flag indicates LSQR convergence.*
- *it indicates LSQR iteration count.*
- *gamma is the over-sampling ratio used in sketching.*
- *k is number of non-zeros per column in the hashing matrix.*

- *abs_tol is absolute residual tolerance, the algorithm will immediately return if it founds a vector x with $\|Ax - b\| \leq$ abs_tol.*
- *rcond a parameter used in column pivoted QR to determine rank. After a column pivoted QR factorization of the matrix $SAP = QR$, diagonal entried of R less than rcond is treated as zero and the whole corresponding column in the matrix Q is discarded.*
- *it_tol is tolerance used in the preconditioned LSQR algorithm for LSQR convergence.*
- *max_it is maximum iteration allowed in the preconditioned LSQR algorithm.*
- *debug is a flag. If debug=1 then the solver will print some additional information.*
- *wisdom is a flag. If wisdom=1 then the user is expected to have provided a wisdom file for the Discrete Cosine Transform used in sketching.*

If the matrix A is known to be of full numerical rank, then a different routine can be used which is slightly faster:

ls_dense_hashing_blendenpik_noCPQR(A, b, x, rank, flag, it, gamma, k, it_tol, max_it, debug, wisdom):

The arguments usage is the same as above, but we don't need the rank-detection parameter rcond, and the shortcut related to abs_tol is not implemented.

Example *The following program solves an example of problem* (1) *with given A and b.*

The matrix A is created by

```
#include <iostream>
#include "Config.hpp"
#include "SpMat.hpp"
#include "Random.hpp"
#include <math.h>
#include "RandSolver.hpp"
#include "cs.h"
#include <stdio.h>
#include "SuiteSparseQR.hpp"
#include "cholmod.h"
#include "lsex_all_in_c.h"
#include "IterSolver.hpp"
#include <sys/timeb.h>
#include "bench_config.hpp"
#include "blendenpik.hpp"

int main(int argc, char **argv)
{
        // create data
        long m = 4;
        long n = 2;
        double data_A[8] = {0.306051, 1.53078,1.64493, 1.61322,
         0.2829,0.474476, 0.586278, 0.610202};

        double data_b[4] = {0.0649338,0.845946, 0.0164085,0.247119};

        Mat_d A(m,n,data_A);
        Vec_d b(m,data_b);

        // parameters
        long k = NNZ_PER_COLUMN;
        double gamma = OVER_SAMPLING_RATIO;
        long max_it = MAX_IT;
        double it_tol = IT_TOL;
        double rcond = 1e 10;
```

```
    int wisdom = 0;
    double abs_tol = ABS_TOL;

    // storage

    long it;
    int flag;
    long rank;
    double residual;
    Vec_d x(A.n());

    // solve
    ls_dense_hashing_blendenpik(
    A, b, x, rank, flag, it, gamma, k, abs_tol,
    rcond, it_tol, max_it, debug, wisdom);

    std::cout << "A is: "<< A << std::endl;
    std::cout << "b is: "<< b << std::endl;
    std::cout << "solution is: "<< x << std::endl;

    // compute residual
    A.mv('n', 1, x, 1, b);
    std::cout << "Residual of ls_blendenpik_hashing is: "<< nrm2(b) << std::endl;
    std::cout << "Iteration of ls_blendenpik_hashing is: " << it << std::endl;

}
```

3.3 Sparse problems

To compute a solution of a linear least square problem where the matrix A is sparse, a call of the following form should be made.

*ls_sparse_spqr(*A, b, x_ls_qr, rank, flag, it, gamma, k, abs_tol, ordering, it_tol, max_it, rcond, peturb, debug):*

- *A is an input of derived type containing the $\mathbb{R}^{m\times n}$ matrix.*
- *b is an input of derived type containing the $\mathbb{R}^{m}$ vector.*
- *x is an output of derived type containing the $\mathbb{R}^{n}$ vector solution on exit.*
- *rank is an output of derived type containing a scaler, detected rank of the matrix A.*
- *flag indicates LSQR convergence.*
- *it indicates LSQR iteration count.*
- *gamma is the over-sampling ratio used in sketching.*
- *k is number of non-zeros per column in the hashing matrix.*
- *abs_tol is absolute residual tolerance, the algorithm will immediately return if it founds a vector x with $\|Ax-b\|\leq$ abs_tol.*
- *ordering is a parameter used in to select fill-reduced ordering strategy in sparse QR factorization.*
- *it_tol is tolerance used in the preconditioned LSQR algorithm for LSQR convergence.*
- *max_it is maximum iteration allowed in the preconditioned LSQR algorithm.*
- *rcond is a parameter used in sparse rank-revealing QR to determine rank.*
- *perturb is a potential pivot perturbation if R_{11} returned by the sparse QR factorization is ill-conditioned.*
- *debug is a flag. If debug=1 then the solver will print some additional information.*

4 Reference

References

[1] *H. Avron, P. Maymounkov, and S. Toledo. Blendenpik: Supercharging LAPACK's Least-Squares Solver.* SIAM Journal on Scientific Computing*, 32(3):1217–1236, Jan. 2010.*

[2] *T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization.* ACM Transactions on Mathematical Software (TOMS)*, 38(1):8, Nov. 2011.*

[3] *T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal Multithreaded Rank-Revealing Sparse QR Factorization.* ACM Transactions on Mathematical Software (TOMS)*, 38(1), Dec. 2011.*

[4] *J. Scott and M. Tůma. HSL_MI28: An Efficient and Robust Limited-Memory Incomplete Cholesky Factorization Code.* ACM Transactions on Mathematical Software*, 40(4):1–19, June 2014.*