Machine Learning with RcppML

Zach DeBruine

2021-05-03

Contents

Chapter 1

Installation

Install the RcppML R package from github:

```
devtools::install_github("zdebruine/RcppML")
# or the development branch
devtools::install_github("zdebruine/RcppML-dev")
```

Installing the R package also provides access to the C++ header library. You can easily load the C++ files into the `inst/include` directory of your R package by adding the following to your `Makevars` and `Makevars.win` file:

```
PKG_LIBS = $(LAPACK_LIBS) $(BLAS_LIBS) $(FLIBS)
PKG_CPPFLAGS =  -I../inst/include/ -I src/RcppML
```

Chapter 2

Introduction

RcppML is an R package for machine learning with a focus on linear models, matrix factorizations, and clustering. Priorities are ease of use, fast performance, and simple new algorithms that avoid heuristics and hyperparameters.

Functions in RcppML use new algorithms to solve common problems on very large sparse matrices:

- Fast **non-negative least squares** (NNLS) by recursive active set selection/coordinate descent (RASS/CD)
- Tractable methods for **L0-regularization** of least squares solutions
- Extremely fast **matrix factorization** by in-place alternating least squares, fast stopping criteria, and random starts
- **Rank-2 factorization** (i.e. faster than *irlba* SVD) for approximating the second singular vector
- **Rank-1 factorization** for finding the first singular vector
- New alternating **divisive/agglomerative clustering** algorithm for extremely sensitive classification of samples

2.1 R package

The R package features high-level functions that adaptively call C++ subroutines. The basic functions in the RcppML toolkit are:

- `solve()`: least squares solutions with refinement by coordinate descent, optional non-negativity constraints and regularizations (L0, L1, L2, RL2)
- `project()`: solve for one side of a linear model in the form `A = WH`, either `W` or `H`.

- `factorize()`: decompose sparse matrix `A` into `WDH` where `D` is a diagonal matrix, using either a random start at a given rank or an exact incremental rank-expansion factorization (REMF)
- `cluster()`: fast divisive or divisive/agglomerative clustering of large sparse matrices

Each function is heavily overloaded with specialized C++ subroutines, which are accessible through the low-level Rcpp header library.

2.2 C++ library

The subroutines in the R functions are exposed as a C++ header library which can be used in any Rcpp application. There are three C++ classes:

- `RcppML::CoeffMatrix` holds the symmetric positive definite matrix giving coefficients of a linear system (a), a pre-conditioned Cholesky decomposition, and member functions for solving least squares equations. `RcppML::CoeffMatrix2` is a specialized class for 2-variable systems.
- `RcppML::MatrixFactorization` holds matrices A, W, D, and H corresponding to an $A = WDH$ decomposition, and member functions for projecting W given H (or vice versa), fitting the factorization by alternating least squares from a random start, or fitting the factorization using a new rank-expansion method that gives robust results. `RcppML::MatrixFactorization2` is a specialized class for rank-2 decompositions.
- `RcppML::ClusterModel` holds matrices A and member functions for clustering methods, stopping criteria, and processing results.

Each of these functions receive a chapter in this book in which the theory is outlined, the R function is described , and are heavily overloaded with specialized subroutines.

2.2.1 Design Priorities

RcppML is intended to be used firstly as an R package and secondarily as an RcppEigen header library:

1. An Rcpp-specific `RcppML::SparseMatrix` class using `double` type only rather than `Eigen::SparseMatrix<T>`. To facilitate zero-copy by-reference read-only access to R sparse matrix objects, a CSC sparse matrix class is used that is built from Rcpp base types `NumericVector` and `IntegerVector`. Note that R SEXP types only support `double`, not

`float`. The `RcppML::SparseMatrix` class is designed to operate very similarly to `Eigen::SparseMatrix<T>` and therefore the terms are nearly interchangeable.

2. Templating for `double` and `float` computation. Note that R SEXP types do not support `float`, and therefore the `RcppML::SparseMatrix` class can only be represented in double type because it is built from Rcpp `NumericVector` and `IntegerVector`. However, `float` operations can be significantly faster, especially in the solver module and thus are supported via templating.

2.2.2 Motivation

The original need for this project stems from fundamental questions in single-cell biology. Many single-cell analysis pipelines use PCA and graph-based Louvain clustering for learning cell types and gene expression programs. However, single-cell gene expression count data is non-negative and suffers from dropout, thus methods should be used which support imputation and non-negativity constraints. NMF is a great tool for this task, but unfortunately, current NMF algorithms are too slow. There is a great need for faster memory-efficient factorization and clustering methods.

Chapter 3

Solving Systems of Equations

Like `base::solve()`, RcppML solves the general system of equations for x:

$$a * x = b$$

a *is a symmetric positive definite dense matrix of dimensions* n x n b is a vector of length n

The solver can support:

- Non-negative constraints
- L0/L1/L2/RL2 regularization
- Refinement of any solution by coordinate descent

3.1 The solve() R function

The `solve()` R function is an adaptive interface to many specialized subroutines. Type `?solve` to view detailed documentation for the function.

```
# generate a random system of equations
X <- matrix(runif(100), 10, 10)
y <- runif(10)
a <- t(X) %*% X
b <- t(X) %*% y

x <- solve(a, b)                                    # unconstrained solution
```

```
x <- solve(a, b, nonneg = TRUE)                              # non-negative solution
x <- solve(a, b, nonneg = TRUE, L0 = 2, L0_method = "exact")  # L0-regularized solutio
x <- solve(a, b, nonneg = TRUE, maxit = 0)                   # NNLS with no coordinat
```

3.2 C++ class: CoeffMatrix

The `RcppML::CoeffMatrix<T>` class holds the coefficient matrix for the linear system (a) and contains member functions for solving linear systems.

Let's look at the **constructor** for the `CoeffMatrix<T>` class:

```
template <typename T>
RcppML::CoeffMatrix(Eigen::Matrix<T, -1, -1>& a, bool nonneg = false, int L0 = -1,
                    T L1 = 0, T L2 = 0, T RL2 = 0, int maxit = 100, T tol = 1e-8,
                    std::string L0_method = "ggperm") :
```

The symmetric positive definite dynamically-sized `a` matrix must always be specified, but otherwise the default parameters give an unconstrained and unregularized solution with coordinate descent refinement.

At the time of construction, L2/RL2 regularizations are applied to `a` and a Cholesky (LLT) decomposition is computed. This means that solutions can be found for any vector `b` taking advantage of the preconditioned decomposition:

```
//[[Rcpp::export]]
Eigen::Matrixxd::CoeffMatrix(Eigen::MatrixXd& a, Eigen::MatrixXd& b){
  RcppML::CoeffMatrix<double> model(a);
  for(int i = 0; i < b.cols(); ++i)
    x.col(i) = model.solve(b.col(i));
  return x;
}
```

3.3 Unconstrained solutions

Unconstrained least squares solutions are found by Cholesky (LLT) decomposition of `a` and forward/backward substitution on the Lower/Upper triangle using the Eigen library. Cholesky decomposition is exceptionally fast, but suffers from rounding errors in large solutions.

The solution is refined using gradient descent to correct for any rounding errors during solving from the Cholesky decomposition. This is useful for ill-conditioned or large systems. Generally, Cholesky + Coordinate Descent is

faster than using a method like QR, BCDSVD, or Jacobi SVD to get a better quality solution.

R usage:

```
x <- solve(a, b, maxit = 100, tol = 1e-8)
```

3.4 Non-negative constraints

Non-negative least squares (NNLS) is commonly solved using active set methods, first introduced by Lawson and Hanson. * The "**active set**" is a group of indices constrained to zero. * Conversely, a "**feasible set**" is a group of indices not constrained to zero.

The Lawson/Hanson algorithm has long been considered the gold standard for NNLS. It begins by adding a single variable to the feasible set based on the residuals of an all-zero solution, and repeats the process, recomputing the gradient and adding/removing variables with each iteration.

More recently, algorithms have been proposed such as the "TNT-NN" method which take the reverse approach, beginning with an unconstrained least squares equation, setting negative values to zero, calculating an unconstrained solution on only non-zero values, setting any negative values to zero, and repeating the process until the unconstrained solution on non-zero values from the previous solution are nonnegative. The "TNT-NN" algorithm then uses heuristics to attempt to add variables back to the active set.

However, rather than using heuristics, we adopt the first part of the "TNT-NN" approach and instead use coordinate descent to refine the solution. The result is an exceptionally fast procedure without heuristics. Often, coordinate descent is not needed and convergence is called after the first iteration.

R usage:

3.5 Coordinate Descent

Sequential coordinate descent is used to refine solutions. This algorithm has been adapted from the NNLM Rcpp package published by Xihui Lin, 2020.

The measure of tolerance is the distance between coefficients in consecutive iterations relative to the average value of coefficients. Thus, for a given iteration i and coefficient j, the tolerance is:

$$tol(xj) = 2\frac{|xj_i - xj_{i-1}|}{xj_i + xj_{i-1}}$$

The overall tolerance is the maximum across all coefficients xj.

As a rule of thumb, `maxit` should be set to 10 times the rank of the system, and `tol` should be set between 1e-6 and 1e-9 depending on the precision of the solution needed (if using `float` type, tolerances beyond 1e-6 are often unachievable).

R usage:

```
x <- solve(a, b, x_init = x, nonneg = T, maxit = 100, tol = 1e-8)
```

Note that by default `x_init = NULL` and thus the coordinate descent refinement cycle is initialized with the Cholesky/substitution or (in the non-negative case) RASS solution. Pre-conditioning coordinate descent with these methods tends to be much faster even if one has a decent guess for initial x.

Rcpp usage:

```
//[[Rcpp::export]]
void cd_nnls_example(Eigen::MatrixXd& a, Eigen::MatrixXd& b, Eigen::MatrixXd& x){
  RcppML::CoeffMatrix<double> a(a, nonneg = true, tol = 1e-6, maxit = 100);
  a.solve(b, x);
  // "x" is updated in the R environment
}
```

3.6 Convex Regularizations

Three convex regularizations are supported:

- **L1/LASSO** is subtracted from `b`. Useful for encouraging sparsity both in least squares equations and matrix factorization.
- **L2/Ridge** is added to the diagonal of `a`. Convexly shrinks values towards zero.
- **RL2/Trough** is added to off-diagonal of `a`. Maximizes independence between signals (i.e. variables which are more strongly connected are explained more by a single coefficient).

What we refer to as RL2, or "Reciprocal L2", is also referred to in the literature as "Pattern Extraction" or "angular" regularization.

R usage:

```
x <- solve(a, b, L1 = 0.1 * sum(b), L2 = 0.1 * mean(diag(a)))
```

Note that regularization penalties are not scaled to the distribution of the coefficients. Generally, it is useful to scale L1 to the sum of `b` and L2/RL2 to the mean or sum of the diagonal of `a`.

Rcpp usage:

The `CoeffMatrix<T>` class constructor receives arguments for L1, L2, and RL2. Penalties on `a` (L2/RL2) are added prior to computing the preconditioned LLT decomposition. Penalties on `b` are added at the time of solving.

The example below shows how to use L1 and L2 parameters, with a little twist for a homebrew elastic net implementation:

```
//[[Rcpp::export]]
Eigen::MatrixXd solve_elnet(const Eigen::MatrixXd& a, const Eigen::VectorXd& b, const double lamb
  RcppML::CoeffMatrix<double> a(a, L1 = lambda, L2 = 1 - lambda);
  return a.solve(b);
}
```

3.7 L0 Regularization

L0-regularization is attractive sparse learning tool because it guarantees definite sparsity. Compare to L1, which modifies `b` (but not `a`) to introduce sparsity, or L2 which modifies `a` (but not `b`) to encourage shrinkage to zero. L0, however, modifies neither `a` or `b` but seeks a sparse solution in `x`. The non-zero coefficients in an L0-regularized solution are generally different than an L1-regularized solution, and so is the interpretation.

However, L0 problems are NP-hard and not even weakly convex. Exact solutions are found only by an exhaustive search of all active set permutations. The number of solutions for any given rank k is $k!(k-1)!$ which means solutions are impractical above no more than rank 15.

3.7.1 L0 = 1 Regularization

The number of possible solutions for any L0 = 1 problem is simply equal to the rank of the system. Thus, all possible solutions may be enumerated, losses calculated, and the solution with the lowest error selected.

3.7.2 General L0 Regularization

Exact L0 solutions may be found, but above rank 15 or so they may cause terrible errors and/or eat up your processor.

However, consider an unregularized solution with a feasible set of values not bound to zero and an active set of values bound to zero. Now consider an algorithm by which the cardinality of this unregularized solution may be reduced by 1. This algorithm might be repeated until the desired cardinality is achieved.

1. **Convex reduction** (noperm): Reduce the cardinality by calculating the increase in error should each non-zero coefficient be set to zero, and then setting to zero the coefficient that increased error least.
2. **Exhaustive permutation** (eperm): Take the solution in #1 and consider every possible swap of indices between the feasible set and active set (i.e. setting a non-zero value to zero and a zero value to non-zero), measuring error, and keeping the solution that gives the least error. Repeat the process until no permutation gives error less than the previous iteration.
3. **Gradient-guided permutation** (ggperm): Take the solution in #1 and calculate the residuals gradient should any single feasible set value be removed. For each gradient calculated, add the variable with the maximum residual to the feasible set, calculate error, and keep if error is less than the original solution. Repeat the process until no permutation gives error less than the previous iteration.

Note that method #1 generally gives intuitive results, but not optimal L0 solutions. Method #2 gives near-exact solutions, but is hardly tractable above rank 100. Method #3 scales to much larger ranks (i.e. rank 500) and achieves nearly the same results as method #2, and thus is the default method.

R usage:

```
x <- solve(a, b, L0 = 1)
x <- solve(a, b, L0 = 10, L0_method = "ggperm")
```

Rcpp usage:

```
//[[Rcpp::export]]
Eigen::VectorXd solve_L0_1(Eigen::MatrixXd& a, const Eigen::VectorXd& b){
  RcppML:CoeffMatrix<double> a(a, L0 = 5, L0_method = "ggperm");
  return a.solve(b);
}
```

Note that existing implementations of L0 regularization in the literature (i.e. broken ridge/L2) do not achieve exact or near-exact L0-regularization solutions. Rather, they are intuitive methods for sparse learning which may be valuable in their own right.

3.8 Rank-2 Solutions

Exact rank-2 solutions.

Chapter 4

Linear Model Projection

4.1 The project() R function

4.2 The MatrixFactorization() class

4.3 Projecting W

4.4 Projecting H

4.5 Rank-2 projections

Chapter 5

Matrix Factorization

Chapter 6

Clustering

6.1 Bipartitioning with Rank-2 AMF

6.2 Divisive Clustering

6.3 Agglomerative Clustering

6.4 Alternating Division/Agglomeration

6.5 Stopping Criteria