SDS 385: Homework 4

G. Paulon

October 7, 2016

Problem 1. Improving SGD for logistic regression

(A) We first implement the stochastic gradient descent coupled with backtracking for logistic regression. This algorithm uses minibatches of the data in order to pick the correct step size for a certain number of iterations. The step size is refreshed every k iterations, which leads to a natural decay over time. The code is reported in Listing A.1. The results of this algorithm applied to the cancer dataset are not as promising as the results obtained with the adaptive gradient descent. For this reason, in the following we will discuss an efficient implementation and a big-data application only of the second algorithm.

(B) The Adaptive Gradient Descent has been implemented. Let us describe the updating rules for this algorithm, which uses a diagonal approximation of the Hessian at every iteration. Let us first recall that the negative log-likelihood of the i^{th} data point is given by

$$\begin{aligned} l_i(\alpha, \boldsymbol{\beta}) &\propto -y_i \log(w_i) - (m_i - y_i)\log(1 - w_i) - \sum_{j=1}^{p} |\beta_j| \\ &= -m_i \log(1 - w_i) - y_i \log\left(\frac{w_i}{1 - w_i}\right) - \sum_{j=1}^{p} |\beta_j| \\ &= m_i \log(1 + \exp(\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta})) - y_i(\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta}) - \sum_{j=1}^{p} |\beta_j| \end{aligned}$$

where we added a penalization term that forces sparsity in the problem. Let us remark that we split the intercept and the other covariates. This is done for notation purposes, as the intercept is not penalized in the last term of the sum.

The gradient with respect to α of this contribution is

$$\begin{aligned} \nabla_\alpha l_i(\alpha, \boldsymbol{\beta}) &\propto m_i \frac{e^{\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta}}}{1 + e^{\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta}}} - y_i \\ &= \hat{y}_i - y_i, \end{aligned}$$

where

$$\hat{y}_i = m_i \frac{e^{\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta}}}{1 + e^{\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta}}}.$$

The gradient with respect to $\boldsymbol{\beta}$ of the individual negative log-likelihood is

$$\begin{aligned} \nabla_{\beta_j} l_i(\alpha, \boldsymbol{\beta}) &\propto m_i \frac{x_{ij} e^{\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta}}}{1 + e^{\alpha + \boldsymbol{x}_i^T \boldsymbol{\beta}}} - x_{ij} y_i - \lambda \frac{\beta_j}{|\beta_j|} \\ &= x_{ij}(\hat{y}_i - y_i) - \lambda \frac{\beta_j}{|\beta_j|}. \end{aligned}$$

Adagrad still has a base learning rate η, but this is multiplied with the elements of a vector $\boldsymbol{g}$, which is given by

$$g_j^{(T)} = \sum_{t=1}^{T} \left(\nabla_{\beta_j} l_i(\alpha, \boldsymbol{\beta}^{(t)})\right)^2.$$

The vector $\boldsymbol{g}$ is monotonically increasing and it can be interpreted as the historical gradient, since it adds at every iteration the square of the gradient contribution. The update step is

$$\beta_j^{(t+1)} = \beta_j^{(t)} - \frac{\eta}{\sqrt{g_j^{(T)}}} \nabla_{\beta_j} l_i(\alpha, \boldsymbol{\beta}^{(t)})$$

and as one can see, the step size decays naturally over time.

The code is illustrated in Listing A.2.

The algorithm has been applied to the usual dataset containing a cancer classification problem. The speed-up is remarkable if compared to the same version of the algorithm implemented in R language. The convergence results are displayed in Figure 1.

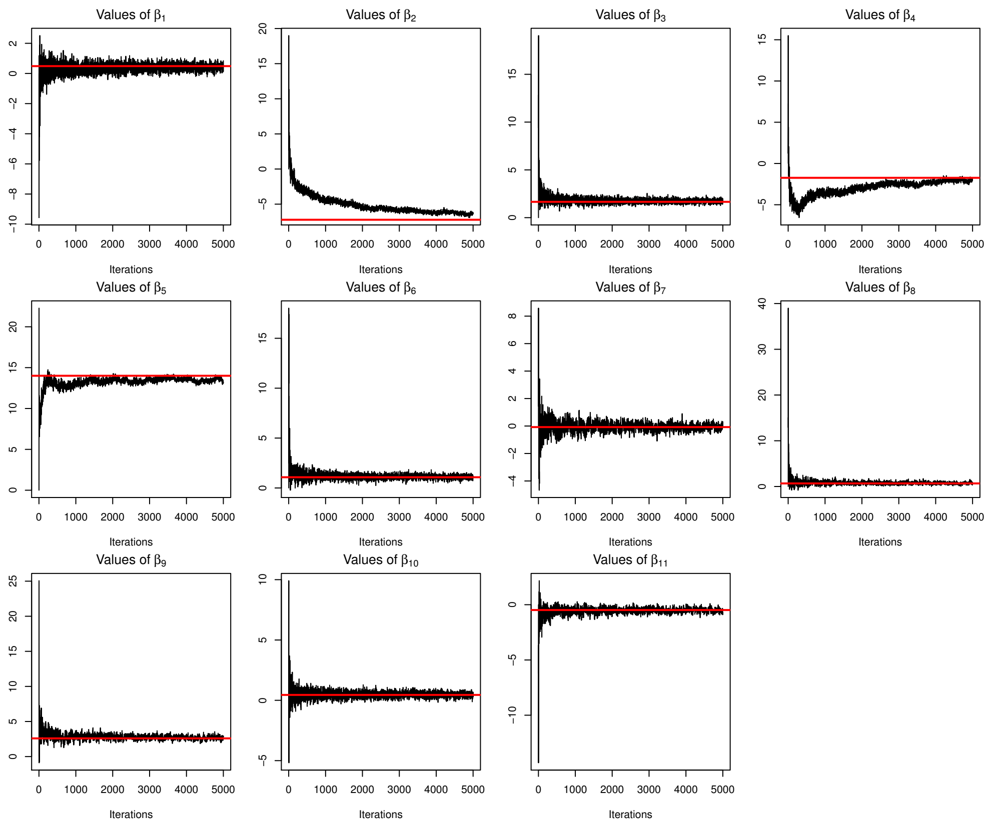

Figure 1: Results of the Adagrad method. The red lines denote the results obtained with Newton method.

Problem 2. Putting it all together on some biggish data

The algorithm has been improved by adding support for sparse matrices and it has been applied to a large (and sparse) dataset whose goal is to detect malicious URLs in web traffic.

The optimal value of the regularizer parameter λ has been chosen by 5-fold cross validation. The estimate of the test error (prediction accuracy) in this case is around 98.0%, which is a remarkable result.

	$\lambda = 0$	$\lambda = 10^{-10}$	$\lambda = 10^{-9}$	$\lambda = 10^{-8}$	$\lambda = 10^{-7}$	$\lambda = 10^{-6}$	$\lambda = 10^{-5}$	$\lambda = 10^{-4}$
Mean	0.9856	0.9856	0.9855	0.9853	0.9841	0.9787	0.9724	0.9626
Std. dev.	0.0006	0.0005	0.0005	0.0006	0.0008	0.0014	0.0019	0.0028

Table 1: Cross-validation test error estimates for several values of the regularization parameter λ.

The details of the algorithm are explained in Listing A.2. However, we discuss the main features of this implementation. In order to make the algorithm as efficient as possible, we added:

- support for **sparse matrices** X: in fact, computing the dot product $\boldsymbol{x}_i^T\boldsymbol{\beta}$ at every iteration can be efficiently done only by ignoring all the zero elements of $\boldsymbol{x}_i^T$;
- **column-major format** for the matrix X. Since C++ stores matrices in a column-major format (that is, contiguous cells are stored in memory following the vertical direction), we extract at each iteration the vector of the i^{th} observation by the column i of X. To do so, the original X matrix has been transposed before the algorithm starts;
- use of the **iterators**. Iterators are efficient pointers that allow to loop very quickly over matrices and vector (most of all, if they are sparse) by pointing to the addresses of the non-empty cells;
- **lazy update** of the penalization when the parameter $\boldsymbol{\beta}$ has not been updated for some iterations. In particular, in this version of the implementation we chose to use a L^1 penalization because it is more likely to fit well our problem, which has around 2000000 features, by shrinking some features towards 0. Let us suppose that for k consecutive iterations (that is, for k consecutive observations) the feature β_j is not updated because $x_{ij} = 0$. The gradient contribution of the negative log-likelihood is 0 and so β_j is not updated. However, we should still update the parameter because of the penalty term, which is never 0. In order to avoid to do this computation for every iteration (which would compromise the computational efficiency of having a sparse X) we update the cumulative penalty for β_j only when is updated.

  If k updates have been skipped from step $t+1$ to step $t+k$ (that is, $\beta_j^{(t)}$ is the last one updated and $\beta_j^{(t+k+1)}$ is the first one to be updated again), then:

$$\beta_j^{(t+1)} = \beta_j^{(t)} - \frac{\eta}{\sqrt{g_j^{(T)}}}\lambda\,\mathrm{sign}(\beta_j^{(t)})$$

$$\beta_j^{(t+2)} = \beta_j^{(t+1)} - \frac{\eta}{\sqrt{g_j^{(T)}}}\lambda\,\mathrm{sign}(\beta_j^{(t+1)}) = \beta_j^{(t)} - \frac{\eta}{\sqrt{g_j^{(T)}}}2\lambda\,\mathrm{sign}(\beta_j^{(t)})$$

$$\vdots$$

$$\beta_j^{(t+k)} = \beta_j^{(t)} - \frac{\eta}{\sqrt{g_j^{(T)}}}k\lambda\,\mathrm{sign}(\beta_j^{(t)})$$

where we assumed that β_j does not change its sign during the k iterations, and that the historical gradient $g_j^{(T)}$ is constant as well. Afterwards, $\beta_j^{(t+k+1)}$ is updated with the usual rule. In order to make our assumptions more reasonable,

$$\beta_j^{(t+k)} = \text{sign}(\beta_j^{(t)}) \left(0; |\beta_j^{(t)}| - \frac{\eta k \lambda}{\sqrt{g_j^{(T)}}} \right)_+ ,$$

which consists in setting $\beta_k^{(t+k)}$ to 0 if it would change sign during the lazy update.

Appendix A

R and C++ code

```r
SGD.linesearch <- function(y, X, mi, beta0, maxiter = 100000, tol = 1E-8){
  # Function for the gradient descent of the logit model coupled with backtracking
  # ------------------------------------------------------------------------------
  # Args:
  #   - y: response vector (length n)
  #   - X: matrix of the features (n*p)
  #   - mi: vector of the number of trials, always 1 in the logit framework (length n)
  #   - beta0: initial regression parameters (length p)
  #   - maxiter: number of maximum iterations the algorithm will perform if not converging
  #   - tol: tolerance threshold to convergence
  # Returns:
  #   - ll: values of the log-likelihood for every iteration
  #   - beta: final optimal regression parameters
  #   - alpha: selected step size for every iteration
  # -------------------------------------------------
  N <- dim(X)[1]
  P <- length(beta0)

  # Select the updating time and the size of the minibatches
  train.epoch <- floor(maxiter/1000)
  size.minibatch <- floor(N/10)

  # Sample the data points to calculate the gradient
  idx <- sample(1:N, maxiter, replace = TRUE)
  # Sample the indexes for the minibatch
  minibatches <- matrix(sample(1:N, 1000*size.minibatch, replace = TRUE), 1000, size.minibatch)
  idx.minibatch <- minibatches[1, ]

  betas <- array(NA, dim=c(maxiter, length(beta0)))
  betas[1, ] <- beta0 # Initial guess

  av.ll <- array(NA, dim = maxiter)
  ll <- log.lik(betas[1, ], y[idx[1]], matrix(X[idx[1],], nrow=1), mi[idx[1]])
  av.ll[1] <- ll

  for (iter in 2:maxiter){

    # Choose the direction according to the gradient of the single individual idx[i]
    grad <- grad.loglik(betas[iter-1,], y[idx[iter]], matrix(X[idx[iter], ],nrow=1), mi[idx[iter
        ]])
    dir <- - grad
    # if we need to update the optimal step
    if ((iter == 2) | (iter %% train.epoch == 0)){
      idx.minibatch <- minibatches[ceiling(iter/train.epoch), ]
      grad.minibatch <- grad.loglik(betas[iter-1, ], y[idx.minibatch], X[idx.minibatch, ], mi[idx
          .minibatch]) / length(idx.minibatch)
      alpha <- optimal.step(betas[iter-1, ], -grad.minibatch, grad.minibatch, y[idx[iter]],
          matrix(X[idx[iter], ], nrow=1), mi[idx[iter]])
    }

    # Update Beta parameters
    betas[iter, ] <- betas[iter-1, ] + alpha * dir
    # Compute log-likelihood and average log-likelihood
    ll <- log.lik(betas[iter, ], y[idx[iter]], matrix(X[idx[iter],], nrow=1), mi[idx[iter]])
    av.ll[iter] <- ((av.ll[iter-1])*(iter-1) + ll)/iter

    # Convergence check
    if (abs(av.ll[iter-1] - av.ll[iter]) / (av.ll[iter-1] + 1E-10) < tol){
      av.ll <- av.ll[1:iter]
      betas <- betas[1:iter, ]
      break;
    }

    else if (iter == maxiter & abs(av.ll[iter-1] - av.ll[iter])/(av.ll[iter-1] + 1E-10) >= tol){
      break;
    }
  }
  return(list("ll" = av.ll, "beta" = betas))
}
```

Listing A.1: Function implementing the SGD coupled with backtracking.

```cpp
#define ARMA_64BIT_WORD
// [[Rcpp::depends(RcppEigen)]]
#include <RcppEigen.h>
#include <iostream>
#include <algorithm>
#include <string>

using namespace Rcpp;

//using Eigen::Map;
//using Eigen::MatrixXd;
//using Eigen::MatrixXi;
using Eigen::VectorXd;
//using Eigen::VectorXi;
using Eigen::SparseVector;

typedef Eigen::MappedSparseMatrix<double>  MapMatd;
// typedef Map<MatrixXi>  MapMati;
// typedef Map<VectorXd>  MapVecd;
// typedef Map<VectorXi>  MapVeci;


// Function to compute the sign of a generic type
template <typename T> int sgn(T val) {
  return (T(0) < val) - (val < T(0));
```

```
}


// [[Rcpp::export]]
SEXP Ada_Grad(MapMatd& X, VectorXd& y, VectorXd& m, VectorXd& beta0, double eta = 1.0, unsigned
    int npass = 1, double lambda = 0.0, double weight = 0.01){
  // - X:      design matrix stored in sparse column-major format. That is, we transposed the
      matrix outside the function (in R).
  //           The features for the observation i is stored in column i
  // - y:      response vector
  // - m:      vector of sample sizes
  // - beta0:  initial guess values for beta
  // - eta:    master step-size
  // - npass:  number of times we go over the dataset
  // - lambda:    penalization of the Lasso regression (L1 penalty)
  // - weight:    weight for the computation of the negative log-likelihood (high weight gives
      importance at every single contribution, small weight smoothes the
  //        negative log-likelihood function).

  unsigned int N = X.cols();
  unsigned int P = X.rows();

  // x is the generic column of X that will be extracted at each iteration
  SparseVector<double> x(P);

  // Initialize parameters
  double psi, epsi, yhat, delta, h;
  double this_grad = 0.0;
  double newbeta, penalty;
  unsigned int j,k;

  // Initialize parameters alpha and beta
  double alpha = 0.0;
  VectorXd beta(P);

  // Initialize historical gradients (cumulative)
  double hist_int = 0.0; // historical gradient of the intercept term
  VectorXd hist_grad(P);
  for (int j = 0; j < P; j++){
    hist_grad(j) = 1e-3;
    beta(j) = beta0(j);
  }

  // How long has it been since the last update of each feature?
  NumericVector last_update(P, 0.0);

  // negative log likelihood for assessing fit
  double loglik_avg = 0.0;
  NumericVector loglik_vec(npass*N, 0.0);


  // Outer loop: number of passes over data set
  k = 0; // global interation counter
  for(unsigned int pass = 0; pass < npass; pass++) {

    // Loop over each observation (columns of X)
    for(unsigned int i = 0; i < N; i++) {
```

```cpp

      // Compute linear predictor and yhat
      x = X.innerVector(i);          // efficient way to extract a column from the matrix X
      psi = alpha + x.dot(beta);       // the function .dot() automatically ignores the zero elements of x
      epsi = exp(psi);
      yhat = m(i) * epsi/(1.0 + epsi); // MLE of y

      // (1) Update log-likelihood weighted average
      loglik_avg = (1.0 - weight) * loglik_avg + weight * (m(i) * log(1 + epsi) - y(i) * psi);
      loglik_vec(k) = loglik_avg;

      // (2) Update intercept
      delta = y(i) - yhat;                   // GRADIENT with respect to the intercept (the intercept component)
      hist_int += delta * delta;             // update historical gradient of the intercept
      alpha += (eta/sqrt(hist_int)) * delta; // compute the update of the intercept

      // (3) Update beta: iterate over the ACTIVE features for this instance
      for (SparseVector<double>::InnerIterator it(x); it; ++it) {

        // the .index() function extracts the x index of iterator currently pointing at x
        j = it.index();

        // STEP (a): aggregate all the penalty-only updates since the last time we updated this feature.
        // This is a form of lazy updating in which we approximate all the "penalty-only" updates at once.
        double skip = k - last_update(j);       // how many penalization updates did we skip for this feature?
        h = sqrt(hist_grad(j));
        penalty = skip * eta/h;           // penalize the skip previous updates

      // We can find the corresponding beta applying the penalty. Let us remark that if beta has changed sign, we set it to 0. This is reasonable
        beta(j) = sgn(beta(j)) * fmax(0.0, fabs(beta(j)) - penalty*lambda);
      // For the L2 penalty, we can change this line to: 2*lambda*skip*beta(j), which is an approximation because beta is changing at each iteration

        // Update the last-update vector
        last_update(j) = k;

        // STEP (b): Now we compute the regular update for this observation.
        this_grad = delta*it.value();

        // update the historical gradient for this feature
        hist_grad(j) += this_grad*this_grad;

        h = sqrt(hist_grad(j));
      penalty = eta/h;
      newbeta = beta(j) + penalty * this_grad;
        beta(j) = sgn(newbeta) * fmax(0.0, fabs(newbeta) - penalty * lambda);
      }
      k++; // increment global counter
    }
  }

  // (4) Update last penalties:
  // At the very end, apply the accumulated penalty for the variables we haven't touched recently
     . We still have to penalize some features that we did not update
  // during the last "skip" iterations.
  for (int j = 0; j < P; j++) {
    double skip = k - last_update(j);
    h = sqrt(hist_grad(j));
    penalty = skip*eta/h;
    beta(j) = sgn(beta(j))*fmax(0.0, fabs(beta(j)) - penalty*lambda);
  }

  return List::create(Named("alpha") = alpha,
                      Named("beta") = beta,
                 Named("loglik") = loglik_vec);
}
```

Listing A.2: Function implementing Adagrad algorithm exploiting C++'s library Eigen.