SDS 385: Homework 4

G. Paulon

October 2, 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

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

The code is illustrated in 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.

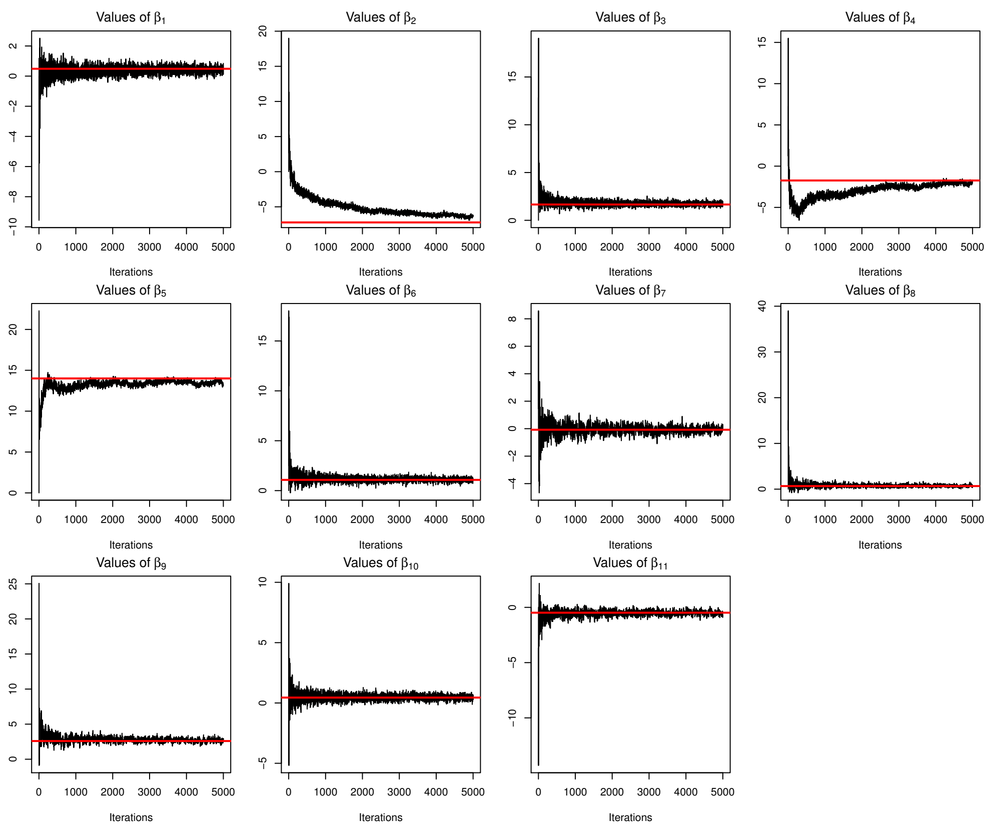

Figure 1: Results of the Adagrad method. The red lines denote the results obtained with Newton method.

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.

Appendix A

R 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.

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

using namespace Rcpp;
using Eigen::VectorXd;
using Eigen::SparseVector;

typedef Eigen::MappedSparseMatrix<double>  MapMatd;

// 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 discount = 0.01){
  // X is the design matrix stored in sparse column-major format
  // i.e. with features for case i stores in column i
  // y is the response vector
  // m is the vector of sample sizes
```

```

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

// x is the sparse vector that, at each iteration, will contain the columns of X
SparseVector<double> x(P);

// Initialize parameters
double psi0, epsi, yhat, delta, h;
double this_grad = 0.0;
double mu, gammatilde;

// Initialize parameters alpha and beta
double alpha = 1.0;
VectorXd beta(P);

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

// Vector denoting 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 nll_avg = 0.0;
NumericVector nll_tracker(npass*N, 0.0);

unsigned int k = 0; // global interation counter
unsigned int j; // j is an index that will denote the positions of the nonzero elements in x
for(unsigned int pass = 0; pass < npass; pass++) {

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

    // Form linear predictor and E(Y[i]) from features
    x = X.innerVector(i);
    psi0 = alpha + x.dot(beta);
    epsi = exp(psi0);
    yhat = m[i] * epsi/(1.0 + epsi);

    // (1) Update nll average
    nll_avg = (1.0 - discount) * nll_avg + discount * (m[i] * log(1 + epsi) - y[i] * psi0);
    nll_tracker[k] = nll_avg;

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

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

      // Which feature is this?
```

```
        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);
        h = sqrt(hist_grad(j));
        gammatilde = skip * eta/h;
        beta(j) = sgn(beta(j)) * fmax(0.0, fabs(beta(j)) - gammatilde*lambda);

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

        // STEP (b): Now we compute the update for this observation.
        // gradient of negative log likelihood
        this_grad = delta*it.value();

        // update adaGrad scaling for this feature
        hist_grad(j) += this_grad*this_grad;

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

  // (4) Update last penalties:
  // At the very end, apply the accumulated penalty for the variables we have not touched
      recently
  for (int j = 0; j < P; j++) {
    double skip = k - last_update(j);
    h = sqrt(hist_grad(j));
    gammatilde = skip*eta/h;
    beta(j) = sgn(beta(j))*fmax(0.0, fabs(beta(j)) - gammatilde*lambda);
  }

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

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