SDS 385: Exercises 4 - Better Online Learning

October 13, 2016

Professor Scott

Spencer Woody

My implementation of SGD with AdaGrad for detecting malicious URLs in web traffic utilizes both R and C++. R is used for reading in the data, calling the function for SGD, plotting, and performing cross validation, while C++ is used to export a function to R. The Rcpp library is used to bridge C++ and R, and the Eigen library is used in C++ to perform matrix algebra observation. My code in C++ takes in the data, a default stepsize, an ℓ^2 (ridge) penalization parameter, and a number of times to parse through the data. It returns an intercept term, the estimated coefficients of the covariates, and a trace plot of the exponential moving average (EMA) of the negative binomial likelihood. Instead of sampling an observation randomly, I just sweep through the entire dataset in one epoch at a time, i.e., reading every observation sequentially.

There are three loops contained within my C++ code. The outermost loop is for each epoch, in which all the data is passed through once. The next loop loops through all the columns (observations) of the X feature matrix. The innermost loop parses through only the *active* features of X for a given observation. This is how the sparsity of the X feature matrix is leveraged. It takes my computer (a 2014 MacBook Air with 1.6 GHz Intel Core i5 processor) about 25 seconds to run on this whole dataset.

The intercept term in ridge regression is not penalized, so my code updates it for every observation without penalty.

Because the innermost loop only parses through active features, there will be many times that a specific coefficient β_j will be passed over, but even though these intermediate steps are skipped, there is still a penalization of β_j for every iteration. Therefore, in order to keep leverage over the sparse nature of the feature matrix we must utilize *lazy updating* whereby we approximate the ridge penalization accumulated over the length of consecutive iterations when β_j is passed over.

To see why this is neccessary, let's take a closer look at ridge regression, where we optimize the objective function

$$\tilde{l}(\beta) = l(\beta) + \frac{1}{2}\lambda||\beta||_2^2, \tag{1}$$

where $l(\beta) = -X^T(y - mw)$ is the binomial negative log likelihood. The gradient of this objective function is

$$\nabla\tilde{l}(\beta) = \nabla l(\beta) + \lambda\beta \tag{2}$$

Our vector of coefficients β is updated at each step as follows

$$\begin{aligned} \beta^{(k+1)} &= \beta^{(k)} - \gamma^{(k)}\nabla\tilde{l}(\beta^{(k)}) && (3)\\ &= \beta^{(k)} - \gamma^{(k)}\left(\nabla l(\beta^{(k)}) + \lambda\beta^{(k)}\right). && (4) \end{aligned}$$

Element-wise this may be written as

$$\beta_j^{(k+1)} = \beta_j^{(k)} - \gamma_j^{(k)}\left(\nabla l(\beta^{(k)})_j + \lambda\beta_j^{(k)}\right), \tag{5}$$

where

$$\gamma_j^{(k)} = \frac{\eta}{\sqrt{G_j}}\nabla\tilde{l}(\beta^{(k)})_j, \tag{6}$$

and G_j is the sum of squares of historical jth components of gradients and η is the default (first) AdaGrad step size. Whenever the jth feature of a given obersvation is zero, when the innermost loop of my code passes over it, the gradient at that feature is zero and the update is now

$$\beta_j^{(k+1)} = \beta_j^{(k)} - \gamma_j^{(k)}\lambda\beta_j^{(k)} = \left(1 - \gamma_j^{(k)}\lambda\right)\beta_j^{(k)} \tag{7}$$

Suppose there are m times that we skip the jth feature. We will need to subtract from β_j the accumulated penalties which are incurred before updating the cofficient as usual. The lazy update is

$$\beta_j^{\Delta} = \sum_{i=1}^{m} (1 + \gamma_j^{(k)} \lambda)^{i+1} \beta_j^{(k)} = \frac{1 - \left(1 + \gamma_j^{(k)} \lambda\right)}{1 - \gamma_j^{(k)} \lambda} \tag{8}$$

where k is the previous iteration during which β_j was updated. Once we touch β_j again, we first subtract this acc Figures 1 and 2 show traceplots. Both plots converge, suggesting that a global minimum is reached.

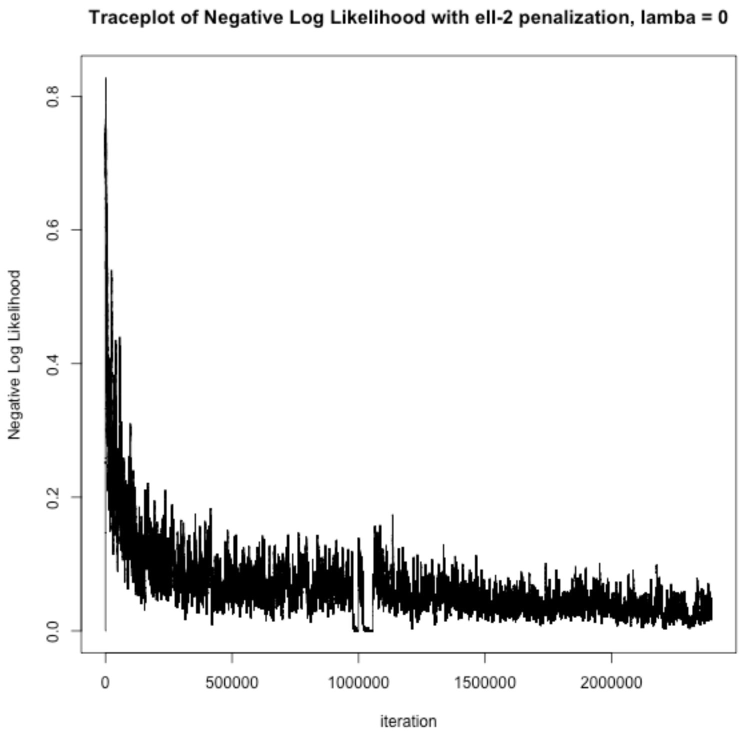

Figure 1: Traceplot of EMA of negative log likelihood for no penalization

Finally, I use cross-validations to tune the penalization parameter λ. I ran five-fold cross-validation in R, which was slow to predict the testing data because of the matrix operations needed to be done, so I ran cross-validation on a select series of values for the parameter. Figure 3 below compares results of cross validation across varying values of λ. Sensitivity is the true positive rate for predicting out-of-sample outcomes, specificity is the true negative rate, and accuracy is the total proportion of outcomes predicted successfully. Without penalization, the model has pretty impressive predictive power, with all three measures above 0.99. Even a small penalty like 10^{-6} reduces predictive power.

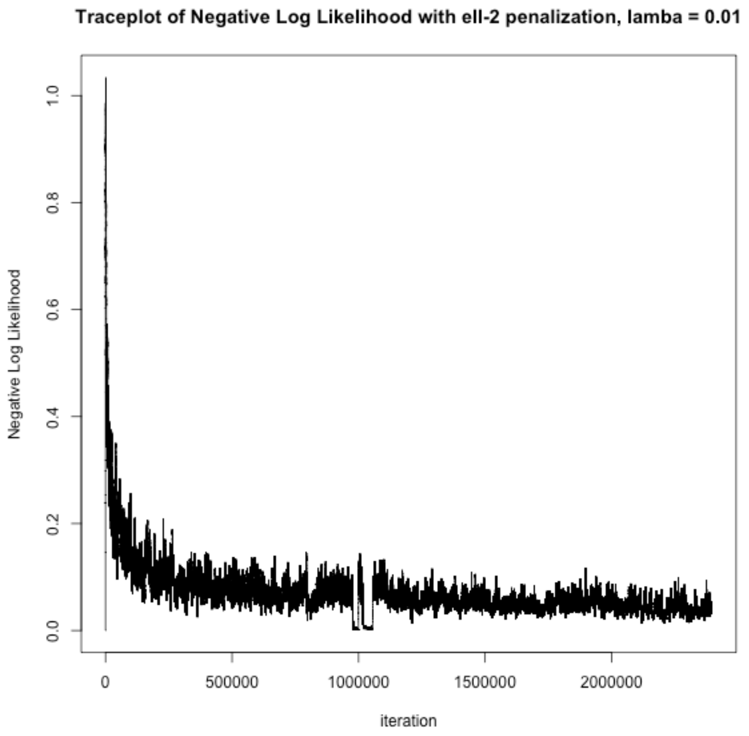

Figure 2: Traceplot of EMA of negative log likelihood for penalization with λ = 0.01

λ	Sensitivity	Specificity	Accuracy
0	0.9901595	0.9934549	0.9923656
10^{-6}	0.9897778	0.9936025	0.9923385
10^{-4}	0.9888078	0.9930379	0.9916390
10^{-2}	0.9726841	0.9922197	0.9857616

Figure 3: Five-fold cross-validation results for out-of-sample prediction

R script e4.R

```
############################################################
######### Created by Spencer Woody on 22 Sep 2016 #########
############################################################

library(Matrix)
library(Rcpp)
library(RcppEigen)

# sourceCpp("fastsgd.cpp")

sourceCpp("fastsgdridge.cpp")

# Note that file "url_Xt.rds" is already in column—oriented mode
tX <- readRDS('url_Xt.rds')
y <- readRDS('url_y.rds')

n <- length(y)
p <- nrow(tX)

M <- rep(1, n)
beta0 <- rep(0, p)

#sgd1 <- sgdcppr(tX, y, M, stepsize0 = 2, numpasses = 1, beta0, lambda = 0.0, edecay = 0.001)

sgd2 <- sgdcppr(tX, y, M, stepsize0 = 2, numpasses = 1, beta0, lambda = 0.01, edecay = 0.001)

png("nlltrace1.png", height = 580, width = 580)
plot(sgd2$logliktrace,
    type = "l",
    xlab = "iteration",
    ylab = "Negative Log Likelihood",
    main = "Traceplot of Negative Log Likelihood with ell-2 penalization, lamba = 0.01")
dev.off()


# Goodness of fit

dotprod <- crossprod(tX, sgd2$beta)
yhats <- 1 / (1 + exp(-dotprod))

dotprod2 <- X %*% sgd1$beta

sens <- sum((y == 1) * (yhats > 0.5)) / sum(y==1) # true positive
spec <- sum((y == 0) * (yhats < 0.5)) / sum(y==0) # true negative

# Cross validation
numbins <- 5
jumble <- sample(1:n, n, replace = F)
bin.indices <- split(jumble, cut(1:n, numbins))

# sens.vec <- rep(NA, numbins)
# spec.vec <- rep(NA, numbins)
# Create lists for holding bins of data
tX.list <- list()
y.list  <- list()
M.list  <- list()


for (k in 1:numbins) {
    indices.k <- bin.indices[[k]]

    tX.list[[k]] <- tX[, indices.k]
    y.list[[k]]  <- y[indices.k]
    M.list[[k]]  <- M[indices.k]
    print(k)
}

lambdas <- c(0, 0.000001, 0.0001, 0.01)
sens.mat <- matrix(rep(NA, numbins * length(lambdas)), nrow = length(lambdas))
spec.mat <- matrix(rep(NA, numbins * length(lambdas)), nrow = length(lambdas))
accu.mat <- matrix(rep(NA, numbins * length(lambdas)), nrow = length(lambdas))

for (i in 1:length(lambdas)) {
    lambda.i <- lambdas[i]
    for (j in 1:numbins) {
        indices.j <- c(1:numbins)[-j]

        tX.tr <- do.call(cbind, tX.list[ indices.j ])
        y.tr  <- do.call(cbind,  y.list[ indices.j ])
        M.tr  <- do.call(cbind,  M.list[ indices.j ])

        sgd.j <- sgdcppr(tX.tr, y.tr, M.tr,
                 stepsize0 = 2, numpasses = 1, beta0, lambda = lambda.i, edecay = 0.001)

        beta.j <- sgd.j$beta

        tX.te <- tX.list[[ j ]]
        y.te  <-  y.list[[ j ]]
        M.te  <-  M.list[[ j ]]

        tXbeta.j <- crossprod(tX.te, beta.j)
        yhats.j <- 1 / (1 + exp(-tXbeta.j  - sgd.j$alpha))

        sens.mat[i, j] <- sum((y.te == 1) * (yhats.j > 0.5)) / sum(y.te==1)
        spec.mat[i, j] <- sum((y.te == 0) * (yhats.j < 0.5)) / sum(y.te==0)
        accu.mat[i, j] <- sum((y.te == 1) * (yhats.j > 0.5) + (y.te == 0) * (yhats.j < 0.5)) / length
        sprintf("%i out of %i bins cycled through", j, numbins)
    }
    sprintf("i = %i out of %i lambdas cross-validated.", i, length(lambdas))
}

sens.vec <- apply(sens.mat, 1, mean)
spec.vec <- apply(spec.mat, 1, mean)
accu.vec <- apply(accu.mat, 1, mean)
```

```
print(lambdas)
print(sens.vec)
print(spec.vec)
print(accu.vec)


# > print(lambdas)
# [1] 0e+00 1e−06 1e−04 1e−02
# > print(sens.vec)
# [1] 0.9901595 0.9897778 0.9888078 0.9726841
# > print(spec.vec)
# [1] 0.9934549 0.9936025 0.9930379 0.9922197
# > print(accu.vec)
# [1] 0.9923656 0.9923385 0.9916390 0.9857616
```

R script e4.cpp

```
// Spencer Woody, 05 Oct 2016

#include <RcppEigen.h>
#include <algorithm>    // std::max

using namespace Rcpp;
using Eigen::Map;
using Eigen::MatrixXd;
using Eigen::LLT;
using Eigen::Lower;
using Eigen::MatrixXi;
using Eigen::Upper;
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;

// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::export]]
SEXP sgdcppr(MapMatd X, VectorXd y, VectorXd M, double stepsize0, int numpasses,
            VectorXd beta0, double lambda = 0.1, double edecay = 0.01) {
                // X is the design matrix, with each column an observation
                // y is the vector of responses
                // M is the vector of sample sizes (just ones in case of logistic regression)
                // stepsize0 is the default (first) step size for AdaGrad
                // numpasses is the number of times to cycle through the whole dataset
                // beta0 is the initial guess for beta
                // lambda is the ell-2 penalization parameter
                // edecay is the weight assigned to the new update in EMA of likelihood

                ///////////////
                // SECTION 1 // Declare Variables
                ///////////////

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

                // initialize parameters
                // x is used for each observation (column) in X
                // yhat is predicted y value
                // ydelta is difference between current y and yhat
                // dotprod is x_iˆT * beta, edotprod is eˆdotprod
                // gsqrt is used for sqrt of historical SS of gradients
                SparseVector<double> x(P);
                int j, k; // iteration counters
                double dotprod, edotprod, yhat, ydelta, gsqrt;


                // w_hat is used for an initial guess for te intercept, alpha
```

```
        // alpha is estimated intercept term
        // grad_j is the gradient at the jth feature in X at current observation
        // g0sq is the square of the gradient for the intercept term
        // gam is AdaGrad step from previous time beta(j) is updated
        // sum_penalty is accumulated penalty for lazy updating
        double w_hat = y.sum() / M.sum();
        double alpha = log(w_hat);
        double grad_j = 0.0;
        double g0sq = 0.0;
        double gam;
        double sum_penalty = 0;

        // Init beta
        // Init Gsq, vector for AdaGrad (1e-3 added for numerical stability)
        // adj_grad is gradient divided by sqrt(Gsq)
        VectorXd beta(P);
        VectorXd Gsq(P);
        VectorXd adj_grad(P);
        for (int j = 0; j < P; j++) {
            Gsq(j) = 1e-3;
            adj_grad(j) = 1e-7;
            beta(j) = beta0(j);
        }

        // Vector for last time feature j is updated (for lazy updating)
        NumericVector last_update(P, 0.0);

        // Hold current EMA of loglikelihood avg, and vector for storing historic nll
        double loglikavg = 0.0;
        NumericVector logliktrace(numpasses*N, 0.0);

        // Global iteration counter
        k = 0;

        ///////////////
        // SECTION 2 // Perform SGD with AdaGrad
        ///////////////


        // LOOP # 1
        // Loop for number of passes to go through
        for(int pass = 0; pass < numpasses; pass++) {

            // LOOP # 2
            // Outer loop over all observations (i.e. columns in design matrix)
            for(int i = 0; i < N; i++) {

                // For yhat, predicted value of y at current estimate of beta
                x = X.innerVector(i); // grab ith observation of x (but only
                                      // the active features)
                dotprod = alpha + x.dot(beta);
                edotprod = exp(dotprod);
                yhat = M[i] * edotprod / (1 + edotprod);
                // Update EMA of loglik, add it to vector
                loglikavg = (1 - edecay) * loglikavg + edecay * (M[i]*log(1 + edotprod) - y[
                logliktrace[k] = loglikavg;

                // Update intercept alpha
                ydelta = y[i] - yhat;
                g0sq += ydelta * ydelta;

                alpha += (stepsize0 / sqrt(g0sq)) * ydelta;

                // LOOP # 3
                // Inner loop; Parse over only active features within this observation
                for (SparseVector<double>::InnerIterator it(x); it; ++it) {

                    // Grad index of feature
                    j = it.index();

                    // STEP 1
                    // Lazy update for accumulated penalty

                    // number of iterations since last update
                    //Cap maximum number of recursive updates at 5, for numeric stability.
                    double skip = k - last_update(j);
                    if (skip > 5){skip = 5;}
                    last_update(j) = k; // update last_update with current global iterator

                    // Add up penalties
                    gam = stepsize0 * adj_grad(j);
                    sum_penalty = beta(j) * ( (1 - pow(1 + lambda * gam, skip)) / (1 - lambda

                    // Subtract penalties from beta(j)
                    beta(j) -= sum_penalty;

                    // ell-2 penalty for gradient
                    double ell2penalty = lambda * beta(j);

                    // Compute gradient for this feature at this observation
                    grad_j = - (ydelta * it.value()) + ell2penalty;

                    // Update Gsq vector for this feature
                    Gsq(j) += grad_j * grad_j;

                    // Compute adjusted gradient (i.e., AdaGrad)
                    gsqrt = sqrt(Gsq(j));
                    adj_grad(j) = grad_j / gsqrt;

                    // Update beta
                    beta(j) -= adj_grad(j) * stepsize0;
                }
                k++;
            }
```

```

        }
        // Penalize betas we haven't touched recently
        for (int j=0; j<P; ++j){
            //Using (k-1) since last_updated indexes from 0, and n is based on counting rows
            double skip = (k-1) - last_update(j);
            if (skip > 5){  skip = 5;}

            //Calculate accum penalty.
            gam = stepsize0*adj_grad(j);
            sum_penalty = beta(j) * ( (1 - pow(1 + lambda * gam, skip)) / (1 - lambda * gam))

            //Update beta_js to add accum penalty.
            beta(j) -= sum_penalty;
        }

        // Finally, return alpha, beta, and loglikelihood
        return List::create(Named("alpha") = alpha,
                            Named("beta")  = beta,
                            Named("logliktrace") = logliktrace);
}
```