
Peer review: Jennifer Starling

SDS 385

David Puelz

October 18, 2016

CONTENTS

1	Comments on code	2
2	Cross-validation exercise	2
2.1	Comparing the solution paths	6

1 COMMENTS ON CODE

I enjoyed reading through Jennifer's work and code. It is very well-documented and easy to understand. Her understanding of the statistics in class is rock-solid, and her efforts on the assignments are tremendous (do I sound like Donald Trump?). I have no specific suggestions other than to continue to modularize your code (both Cpp and R) as much as possible. By having everything in functions, you can do so much with relatively little effort and the code becomes easy to debug.

2 CROSS-VALIDATION EXERCISE

I was most interested in comparing our cross-validation approaches. To do so, I modified your Cpp function to allow for training and testing the model after several passes of stochastic gradient descent. I removed some of your original comments and added some additional lines of code at the very beginning and end – I tried not to touch anything else! The first “train” observations of X are used to train the model, and the remaining observations are used to test the model.

Let me know if you have any questions regarding my modifications. I'll post the code on github as well.

```
1 List sparse_sgd_logit(MapMatd Xtall, VectorXd Y, VectorXd m, double step, int train, int ←
    npass, VectorXd beta0, float lambda){
2
3 // David modified here for Cpp cross-validation
4 // Training and test X!
5 int totalcols = Xtall.cols();
6 DaveMat X = Xtall.leftCols(train);
7 DaveMat Xtest = Xtall.rightCols(totalcols-train);
8 int nSamptest = Xtest.cols();
9
10 //-----
11 //INITIALIZE VALUES:
12 int p = X.rows(); //Number of features in X matrix.
13 int n = X.cols(); //Number of observations in X matrix.
14 int iter = 0; //Global iteration counter.
15 int j = 0; //Inner iterator row number.
16 double epsilon = 1E-6; //Constant for Adagrad numeric stability.
17
18 //Initialize vectors for beta, gradient, and doubles for Adagrad updates in sparse row loop.
19 VectorXd beta_hat(p); //Beta_hat vector, length p.
20 beta_hat = beta0; //Set beta_hat to initial beta input value.
21 VectorXd hist_grad(p); //Vector to hold running hist_grad. Will be updated piecewise in ←
    Sparse Row Loop.
22 VectorXd adj_grad(p); //Vector to hold adj_grad_j for each beta_hat_j.
23
24 //Initialize hist_grad values at 1E-3 for numerical stability.
25 for (int i=0;i<p;i++){
26 hist_grad(i) = 1E-3;
27 adj_grad(i) = 0;
28 }
29
30 double grad_j = 0; //Holds jth element of gradient. Do not need to store whole ←
    gradient at once.
31 //double adj_grad_j = 0; //Holds jth element of adj_grad for Adagrad. (In Sparse Row←
    Loop)
32
33 //Initialize elements to hold X, Y and m for a single observation (column).
34 SparseVector<double> Xi(n);
35 Xi=X.innerVector(0);
```

```

36
37 double Yi = Y(0);
38 double mi = m(0);
39 double wi = .5;           //wi will be a scalar, since calculating weights in inner vector.
40 double wi_exponent = 0; //Holds the exponential part of the wi update.
41
42 //Initialize vector (length p) to keep track of when each predictor updated, for lazy updates←
43 NumericVector last_updated(p,0.0);
44 double skip = 1; //Holds how many iterations since last update for a j row of ith col.
45
46 //Initialize vectors to hold log-likelihood and running avg neg log-likelihood.
47 double nll = 0;           //Holds avg neg loglikelihood for the current i obs.
48 NumericVector nll_ra(n*npass,0.0); //Stores running avg loglikelihood.
49
50 //Initialize variable to hold accumulated penalty for a beta_j.
51 double accum_l2_penalty = 0; //Holds accumulated penalty.
52 double gam = 0;           //Holds step*adj_grad_j, for use in calculating accumulated ←
    penalty.
53
54 //-----
55 //LOOPING THROUGH DATA SET:
56
57 //Loop 1: Loop over entire data set npass times.
58 for (int np=0; np < npass; ++np){
59
60 //Rcpp::Rcout << "npass:" << np << std::endl; //REMOVE LATER: Output start of each npass ←
    through data to R console.
61
62 //Loop 2: Loop through observations (cols) of X data set.
63 for (int i=0; i<n; ++i){
64
65 //Set up the single observation for the next stochastic iteration.
66 Xi = X.innerVector(i); //Select new X observation; the ith column of matrix X.
67 Yi = Y(i);             //Select new Y observation; the ith value of vector Y.
68 mi = m(i);             //Select new m observation; the ith value of sample size vector m.
69
70 //Update wi probability value. (w is scalar, since looking at one obs.)
71 wi_exponent = Xi.dot(beta_hat); //breaking out exponential term helps with efficiency.
72 wi = 1 / (1 + exp(-wi_exponent));
73
74 //Update neg loglikelihood and running average.
75 nll = -(Yi * log(wi + epsilon) + (mi - Yi) * log(1 - wi + epsilon));
76 if(iter > 0) {
77 nll_ra(iter) = (nll_ra(iter-1) * (iter-1) + nll) / iter;
78 }
79
80 //Loop 3: Loop through active feature values (rows) of X data set for ith obs (col).
81 for (InIterVec it(Xi); it; ++it){
82
83 //Set j to the feature (row) number.
84 j = it.index();
85
86 //-----
87 //STEP 1: Part 1 of Row Updates for ith Feature: Deferred (lazy) L2 penalty updates.
88 //This aggregates all of the penalty-only updates since last time a feature was updated.
89 //"Penalty-only" updates refers to the gradient not being updated except
90 //for adding the 2*lambda*beta penalty term.
91
92 //Cap maximum number of recursive updates at 5, for numeric stability.
93 //This works bc updates go to zero fairly quickly when lambda<1.
94 skip = iter - last_updated(j); //Number of iters since last update. (Skip=1 means updated ←
    last iter.)

```

```

95 if (skip > 5){ skip = 5;}
96 last_updated(j) = iter;          //Update the last_updated flag for all j's active in this iter.
97
98 //Calculate accum penalty. Based on recursion defined in my notes.
99 //NOTE: This is the gradient for minimizing the neg log-likelihood.
100 //See final note in recursion doc.
101 gam = step*adj_grad(j);
102 accum_l2_penalty = beta_hat(j) * ( (1 - pow(1+lambda*gam,skip)) / (1-lambda*gam) );
103
104 //Add accum l2 penalty to beta_hat_j before doing current iteration update.
105 beta_hat(j) -= accum_l2_penalty;
106
107 //-----
108 //STEP 2: Continue with updates for jth row in ith col.
109
110 //Calculate l2 norm penalty.
111 double l2penalty = 2*lambda*beta_hat(j);
112
113 //Update the jth gradient term. Note: it.value() looks up Xji for nonzero entries.
114 grad_j = (mi*wi-Yi) * it.value() + l2penalty;
115
116 //Update the jth hist_grad term for Adagrad.
117 //This is the running total of the jth squared gradient term.
118 hist_grad(j) += grad_j * grad_j;
119
120 //Calculate the jth adj_grad term for Adagrad.
121 adj_grad(j) = grad_j * invSqrt(hist_grad(j) + epsilon);
122
123 //Calculate the updated jth beta_hat term.
124 beta_hat(j) -= step*adj_grad(j);
125 }
126 ++iter; //Update global counter. (Counts each iteration.)
127 } //End Loop 2: Loop through observations (cols) of X data set.
128 } //End Loop 1: Loop over entire data set npass times.
129
130 //-----
131 //Loop 4: Loop over predictors to catch any last accumulated penalty updates
132 //for predictors not updated in last iteration.
133 for (int j=0; j<p; ++j){
134     //Using (iter-1) since last_updated indexes from 0, and n is based on counting rows from 1.
135     skip = (iter-1) - last_updated(j);
136
137     //Cap maximum number of recursive updates at 5, for numeric stability.
138     //This works bc updates go to zero fairly quickly when lambda<1.
139     if (skip > 5){ skip = 5;}
140
141     //Calculate accum penalty.
142     gam = step*adj_grad(j);
143     accum_l2_penalty = beta_hat(j) * ( (1 - pow(1+lambda*gam,skip)) / (1-lambda*gam) );
144
145     //Update beta_j's to add accum penalty.
146     beta_hat(j) -= accum_l2_penalty;
147 }
148
149
150 // Now, the big loop to test
151 double tally = 0;
152 double tally2 = 0;
153 double tally3 = 0;
154 double num1s = 0;
155 double num0s = 0;
156 int yhat;
157 for(int i = 0; i < nSamptest; i++)

```

```

158 {
159   SpVec Xsamp = Xtest.col(i);
160   double XB = Xsamp.dot(beta_hat);
161   double w = 1 / (1 + exp(-XB));
162   double y = Y(i+train);
163
164   if(w < 0.5)
165   {
166     yhat = 0;
167     if(y==0){ tally3 += 1; }
168   }
169   else
170   {
171     yhat = 1;
172     if(y==1){ tally2 += 1; }
173   }
174   tally += fabs(y-yhat);
175   if(y==0){ num0s += 1; }
176   else{ num1s +=1; }
177 }
178 double classrate = 1 - (tally/nSamptest);
179 double sensi = (tally2/num1s);
180 double speci = (tally3/num0s);
181
182
183 //-----
184 //Return function values.
185 return Rcpp::List::create(
186   _["n"] = n,
187   _["p"] = p,
188   _["iter"] = iter,
189   _["beta_hat"] = beta_hat,
190   _["classrate"] = classrate
191 ) ;
192
193 }

```

2.1 COMPARING THE SOLUTION PATHS

I ran your code as well as mine, choosing the first 80% of the observations as training data and the remaining 20% as testing data. In each case, the X matrix is scaled the same way. I ran the algorithm 10 passes through the training data and for 20 different λ values. The results are shown in the plot below.

