Weiyi Ng
Brian Reschke
CS294-1
February 24, 2012

Programming Assignment 2: Regression Sentiment Model

Overview
Our objective was to develop a regression classifier to capture book review sentiments on Amazon.com. We experimented with regularization methods, penalty functions and optimization algorithms before settling on a Ridge-Regression model. The Ordinary Least Squares (OLS) estimator was employed; the cost function minimum was found using a Stochastic Gradient Descent algorithm. Ten-fold testing showed that our classifier had an AUC of 0.841 with a 1% lift factor.

Data and Summary Statistics
The data for the project, collected by Blitzer, Dredze, and Pereira (2007), consist of 975,194 book reviews appearing on Amazon.com. Reviews considered in our analysis had received a score of 1, 2, 4, or 5, with 5 being highly positive. We used a simple categorization of \{1,2\} as negative and \{4,5\} as positive.

Our instructors generously tokenized the data in advance of this assignment. Given the data size, we prepared a sparse matrix representation of review features by processing roughly one-twenty-fifth of the data at a time. For each sub-segment of the data, we identified the text of reviews that appeared between the hashes corresponding to ‘<review_text>’ and ‘</review_text>’. The first step of our process extracted the first 40 hashes of each review, yielding a 975,194 x 40 matrix. We used the unique function in MATLAB to obtain an index of the 545,355 unique reviews. Another step created a bag-of-words representation of each review in a sparse (#reviews x number of features) matrix. We constrained features to the 2,500 most frequent tokens in all reviews. Then, we reduced the resulting 975,194 x 2,500 matrix to unique rows (reviews) using the index obtained from step 1. In all, our ‘X’ matrix (before adding a column of 1’s for the intercept) was 545,355 x 2500.

On average, reviews contained 111.94 tokens (that is, tokens from the top 2500), with a standard deviation of 128.35 (suggesting high skewness). The majority of reviews were positive (88.1%), with an average of 110.98 tokens per review (126.54). Negative reviews (11.9%) had 119.13 tokens on average (standard dev. 140.85).

Regression and Optimization
The linear regression model was estimated using the Ordinary Least Squares (OLS) estimator. Due to possible non-full rank conditions (multi-collinearity), regularization was required. We explored two different regularization methods that involve the introduction of a penalty term to the cost equation: L2 (Ridge) and L1 (Lasso) regularization. In the case of Lasso-Regularization, the penalty term was explored in three ways: a naive implementation, a “clipping” penalty factor and a cumulative penalty factor (Tsuruoka et al., 2009). We find that the RSS (sum of squared residuals) and performance of these two regularization methods are comparable; neither conveys a marked advantage over the other.

The minimum of the cost function was found using two different optimization algorithms. Firstly, a regular steepest descent algorithm was explored. This was deployed with an Armijo Rule step-size. This is an adaptive step-size structured to vary the step-size in order to maximize the descent gain and prevent “steps over valleys.” An excerpt of the Armijo step-size loop is shown below:
while (ared1 < alpha*pred && (m < 50) )

    %Armijo step size
    gamma = gamin^m;
    %display (gamma)

    %grad = LassoPrime0(Bini,Y,X,lambda);
    grad = RidgePrime(Bini,Y,X,lambda);
    Bnew = Bini + (gamma)*grad;
    %C2 = LassoCost0(Bnew,Y,X,lambda);
    C2 = RidgeCost(Bnew,Y,X,lambda);

    %calculate actual reduction
    ared = C1 - C2;
    %calculate predicted reduction
    pred = gamma*sqrt(grad.'*grad);

    m = m+1;
end

As evidenced, above, the step-size gamma is modified by a power of m (0 < gamma < 1) for each loop if the actual reduction in cost is less than the scaled step-gradient prediction (ared < alpha*pred); in the case of stepping over valleys, the actual reduction will be negative. After each Armijo rule implementation, m is reset to 1. The Armijo rule has the added benefit that the step-size is now “auto-tuned.” In a sense it absorbs the manual tuning efforts into computational expenses in the form of additional inner loops (the Cost and Gradient functions have to be re-computed for each failed Armijo step-size).

While proving to be the most reliable, the classic steepest descent method is heavily computationally intensive, involving multiplication of massive matrices at each step. To solve this problem, we implemented the Stochastic Gradient Descent (SGD) method. This method passes through the entire training matrix, selecting blocks randomly for Cost and Gradient evaluations.

Consider that each iteration involving the evaluation of cost (scaled to the multiplication of the training matrix with the coefficient vector), we estimated the flops by considering two operations: Xb and (Y – Xb)ᵀ(Y-Xb). “Xtrain” considers the entire training vector (436280 rows); “Xin” considers a 5000 row block of training vectors. The sparse matrix multiplication flop counter by Tom Minka was employed.¹

```
Dominating flops per iteration for Steepest Descent Method.
>> flops_spmul(Xtrain,b1): ans = 56942970
>> flops_spmul((Ytrain-Xtrain*b1).',(Ytrain-Xtrain*b1)): ans = 872559
```
```
Dominating flops per iteration for Stochastic Gradient Descent (5000 block size).
>> flops_spmul(Xin,b1): ans = 662354
>> flops_spmul((Yin-Xin*b1).',(Yin-Xin*b1)): ans = 9999
```

We experimented with this optimization method, implementing the Armijo adaptive step algorithm; we find this addition to eliminate a substantial amount of random error (evidenced by eye balling cost-iteration curves) without too much additional computation cost. Another advantage of SGD is that the algorithm applies to both the evaluation of L1 and L2 regularization cost functions.

Certain adaptations are necessary to implement the Armijo rule in a SGD setting. The first is that a “break” measure is needed. Due to random sampling, it is likely that the gradient derived from the random block will have no actual reduction. Thus, actual reduction will always be less than the predicted reduction because the actual reduction is negative. The while loop implementation needs an additional condition to terminate such cases. The upshot is that in such cases, the stochastic increase in

¹ http://research.microsoft.com/en-us/um/people/minka/software/lightspeed/
cost function is minimized as the step-size will be very small. The down shot is that convergence criterion for our SGD implementation is trickier.

Figure 1 shows an initial example of the optimization using SGD in Ridge-Regression. As evidenced, the stochastic “noise” of the optimization processed is minimized by the Armijo adaptive step routine. However, due to infinite loop termination (certain random samples will “climb uphill”; the Armijo loop algorithm breaks at \( m = 50 \)), certain stochastic behavior is still evidenced.

![Figure 1. Sum of Squares Residuals (Cost Function) vs. Iterations](image)

To ascertain convergence, all stochastic optimization algorithms are immediately followed by a steepest descent optimization implementation. We find that after the former, the steepest descent optimization ascertains that an optimum is reached; it converges in less than 2 iterations. We also find that for our training set, convergence is established after \( \sim 4 \) passes, with marginal gains from further passes. As such, the algorithm is ran for a maximum of 10 passes or convergence, whichever comes first. Ultimately, we obtain weights reported in Table 1 below. Our intercept is approximately 0.6.

An initial test of the algorithms reveals that the weights derived make qualitative sense. Table 1 shows the top 20 ranked estimated weights from both Lasso and Ridge regularized estimation. Both regularization methods featured similar words for the top 20 ranks; the ordering is slightly different. Regardless, a first glance reveals that the words do indeed correspond to those that we would expect from positive and negative sentiments. These orderings are rather intuitive: many of the top negative words contain or stem from the term ‘no’ (e.g., “don’t”, “nothing”, “not”, “doesn’t”), while many of the top positive words are (more hyperbolic) synonyms of the word “good” (e.g., “great”, “excellent”, “best”). It is interesting that the word “author” is among the top negative tokens; this suggests that ‘negative’ books warrant more discussion of creator’s weaknesses than of the content of the books themselves. Also of note is the word “my” in the top good list—the reviewers may deploy more first-person, possessive perspectives in reviews of ‘good’ books.

<table>
<thead>
<tr>
<th>Lasso - Negative</th>
<th>Ridge - Negative</th>
<th>Lasso – Positive</th>
<th>Ridge - Positive</th>
</tr>
</thead>
</table>

Table 1. Greatest magnitude weighting coefficients
We also experimented with Tsuruoka et al.'s discussion of Lasso-Regularization penalty factor variants (Tsuruoka, 2009). Two are attempted: a “clipping” method and a cumulative penalty method. The issue addressed here is the notion that while L1 regularization drives the co-efficients to zero, the naïve implementation using SGD optimization does not achieve the step-wise selection. Tsuruoka et al. proposes alternative implementation of the regularization. We discuss this implementation in Appendix A.

### Regularization Penalty Tuning

We tune $\lambda$, the regularization constant and explore its impact on AUC scores. A “ten-fold” evaluation was applied in this manner: 80% of the data set was randomly employed as training data. 10% of the data set was randomly selected as tuning data and the final 10% was selected as training data. All tuning was done on the uni-gram training sets.

We find that changing $\lambda$ affects the rate of convergence of the algorithm, but at the expense of overall error. Table 2 below shows the respective errors and AUC scores for $\lambda$ values and regression types. We calculated AUC scores using the Ridge-Regression model.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>0.05</th>
<th>0.005</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lasso</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>0.7239375052</td>
<td>0.6855460682</td>
</tr>
<tr>
<td><strong>Ridge</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>0.7985959991</td>
<td>0.6821944002</td>
</tr>
<tr>
<td><strong>AUC</strong></td>
<td>0.55</td>
<td>0.7034</td>
</tr>
</tbody>
</table>

We therefore employ a $\lambda$ of 0.005 for our coefficients. Employing the test set, we obtain the AUC plot as below:
The AUC for the test set is calculated to be 0.841; the 1% lift score is calculated to be 10.68.

**Discussion**

We explored several regularized regression classifiers. We are pleased that our Lasso and Ridge regression provide corroborating results. We did attempt to implement bigrams; however, for the scope of this assignment, the parameter tuning was not completed on time to achieve comparable results.

Ultimately, there are many methods to improve upon our findings, both in terms of classifier performance and regression optimization performance. With regard to the former, bigram and trigram implementation, as discussed, with the necessary parameter tuning of optimization algorithms will give us a richer description, and ideally more accurate classification coefficient weights. With regard to the latter, the Tsuruoka et al. algorithm should be tuned; not only does it improve the performance by decreasing the size factor, it also automates a feature selection that will possibly improve classifier performance.

**References**


Appendix A: Alternative Lasso Penalty Functions

Alternative penalty functions were experimented with for the Lasso-regularization. Unfortunately, for the purposes of this assignment, these alternative functions require a higher level of tuning and implementation.

In the first instance (stochastic1half.m) clipping was used – in this case, the penalty function is modified.

```matlab
% Operate one weight
grad = LassoPrime(Brel,Yin,Xrel,lambda);
Brel = Brel + gamma*grad;

% Apply Penalty
for m = 1: relength
    z = Brel;
    if (Brel(m) > 0)
        Brel(m) = max(0, Brel(m) - (RegC/RegN)*gamma);
    elseif (Brel(m) < 0)
        Brel(m) = min(0, Brel(m) + (RegC/RegN)*gamma);
    end
end
```

Here we see that the new weight \( B_{rel} \) is compared to the penalty function; it is zeroed conditionally by force. In the second method, this penalty function is modified to be *cumulative*.

```matlab
u = u + gamma*(RegC/RegN);
...

% Operate one weight
grad = LassoPrime(Brel,Yin,Xrel,lambda);
Brel = Brel + gamma*grad;

% Apply Penalty
for m = 1: relength
    z = Brel;
    if (Brel(m) > 0)
        Brel(m) = max(0, Brel(m) - (u+qrel(m)));
    elseif (Brel(m) < 0)
        Brel(m) = min(0, Brel(m) + (u-qrel(m)));
    end
end
```

```matlab
%update inputs and weight
q(irel) = qrel + (Brel - z);
```

The algorithm behaved as expected, though the regularization factor now has an additional role of zeroing the coefficients. Tuning the regularization constant \( RegC \) becomes especially important. While we did not successfully manage to find the optimal constant for this method, we do show an example of how the algorithm zeroes the coefficients in the following figure.
The data are freely available at [http://www.cs.jhu.edu/~mdredze/datasets/sentiment/](http://www.cs.jhu.edu/~mdredze/datasets/sentiment/). Tokenized data may be available from John Canny at UC Berkeley.