Machine Learning with Large Dataset Assignment 3: Logistic Regression Using Stochastic Gradient Descent

Due Thursday, Oct. 13, 2016 via Autolab

Out Thursday, Sept. 29, 2016

Policy on Collaboration among Students

These policies are the same as were used in Dr. Rosenfeld's previous version of 10601 from 2013. The purpose of student collaboration is to facilitate learning, not to circumvent it. Studying the material in groups is strongly encouraged. It is also allowed to seek help from other students in understanding the material needed to solve a particular homework problem, provided no written notes are shared, or are taken at that time, and provided learning is facilitated, not circumvented. The actual solution must be done by each student alone, and the student should be ready to reproduce their solution upon request. The presence or absence of any form of help or collaboration, whether given or received, must be explicitly stated and disclosed in full by all involved, on the first page of their assignment. Specifically, each assignment solution must start by answering the following questions in the report:

- Did you receive any help whatsoever from anyone in solving this assignment? Yes / No. If you answered 'yes', give full details: _____ (e.g. "Jane explained to me what is asked in Question 3.4")
- Did you give any help whatsoever to anyone in solving this assignment? Yes / No. If you answered 'yes', give full details: _____ (e.g. "I pointed Joe to section 2.3 to help him with Question 2".

Collaboration without full disclosure will be handled severely, in compliance with CMU's Policy on Cheating and Plagiarism. As a related point, some of the homework assignments used in this class may have been used in prior versions of this class, or in classes at other institutions. Avoiding the use of heavily tested assignments will detract from the main purpose of these assignments, which is to reinforce the material and stimulate thinking. Because some of these assignments may have been used before, solutions to them may be (or may have been) available online, or from other people. It is explicitly forbidden to use any such sources, or to consult people who have solved these problems

before. You must solve the homework assignments completely on your own. I will mostly rely on your wisdom and honor to follow this rule, but if a violation is detected it will be dealt with harshly. Collaboration with other students who are currently taking the class is allowed, but only under the conditions stated below.

1 Important Note

As usual, you are expected to use Java for this assignment. It could take hours to run your experiments. Start early.

Yulan Huang (yulanh@andrew.cmu.edu) and Jingyuan Liu (jingyual@andrew.cmu.edu) are the contact TAs for this assignment. Please post clarification questions to Piazza, and the instructors can be reached out at the following eamil address: 10605-Instructors@cs.cmu.edu.

2 Background: SGD for Logistic Regression

One fairly simple way (and extremely scalable way) to implement logistic regression is stochastic gradient descent.

In the lecture we followed Charles Elkan's notes[1], which are for a binary classification task. We estimate the probability p that an example $\mathbf{x} = \langle x_1, \ldots, x_d \rangle$ is positive in the log-odds form:

$$\log \frac{p}{1-p} = \alpha + \sum_{j=1\ldots d} \beta_j x_j \tag{1}$$

If we assume there is a "bias feature" x_0 that is true for every example, then you can simplify and drop the α, leaving just the β_j's to estimate. Therefore

$$p = \frac{\exp(\beta^T \mathbf{x})}{1 + \exp(\beta^T \mathbf{x}))}. \tag{2}$$

It's convenient to consider examples of the form $\mathbf{x}, y$ where $y = 0$ or $y = 1$. The log of the conditional likelihood for example is example will be $LCL(\mathbf{x}, y) = \log p$ if $y = 1$ and $LCL(\mathbf{x}, y) = \log(1 - p)$ if $y = 0$, where p is computed as in Eq. **??**. With a little calculus you can show that for a positive example,

$$\frac{\partial}{\partial \beta_j} LCL(\mathbf{x}, y) = \frac{1}{p} \frac{\partial}{\partial \beta_j} p$$

and for a negative example,

$$\frac{\partial}{\partial \beta_j} LCL(\mathbf{x}, y) = \frac{1}{1-p}(-\frac{\partial}{\partial \beta_j} p)$$

[1] `http://cseweb.ucsd.edu/~elkan/250B/logreg.pdf`

and that

$$\frac{\partial}{\partial \beta_j} p = p(1-p)x_j$$

and putting this together we get that if $y = 1$

$$\frac{\partial}{\partial \beta_j} LCL(\mathbf{x}, y) = (1-p)x_j$$

and if $y = 0$ then

$$\frac{\partial}{\partial \beta_j} LCL(\mathbf{x}, y) = -px_j$$

so in either case

$$\frac{\partial}{\partial \beta_j} LCL(\mathbf{x}, y) = (y-p)x_j \tag{3}$$

So an update to the β's that would improve most LCL would be along the gradient—i.e., for some small step size λ, let

$$\beta_j = \beta_j + \lambda(y-p)x_i$$

Notice that if $x_i = 0$ then β_j is unchanged.

So this leads to this algorithm, which is very fast (assuming you have enough memory to hash all the parameter values).

1. Initialize a hashtable B
2. For $t = 1, \ldots, T$
   - For each example $\mathbf{x}_i, y_i$:
     - For each non-zero feature of $\mathbf{x_i}$ with index j and value x_j:
       * If j is not in B, set $B[j] = 0$.
       * Set $B[j] = B[j] + \lambda(y-p)x_i$
3. Output the parameters $\beta_1, \ldots, \beta_d$.

The time to run this is $O(nT)$, where n is the total number of non-zero features for each example and T is the number of iterations.

3 Efficient regularized SGD

Logistic regression tends to overfit when there are many rare features. One fix is to penalize large values of β, by optimizing, instead of LCL, some function such as $LCL - \mu \sum_{j=1}^{d} \beta_j^2$. Here μ controls how much weight to give to the penalty term. The update for β_j becomes

$$\beta_j = \beta_j + \lambda((y-p)x_i - 2\mu\beta_j)$$

or equivalently

$$\beta_j = \beta_j + \lambda(y - p)x_i - \lambda 2\mu\beta_j$$

Experimentally this greatly improves overfitting - but unfortunately, this makes the computation much more expensive, because now *every* β_j needs to be updated, not only the ones that are non-zero.

The trick to making this efficient is to break the update into two parts. One is the usual update of adding $\lambda(y - p)x_i$. Let's call this the "LCL" part of the update. The second is the "regularization part" of the update, which is to replace β by

$$\beta_j = \beta_j - \lambda 2\mu\beta_j = \beta_j \cdot (1 - 2\lambda\mu)$$

So we could perform our update of β_j as follows:

- Set $\beta_j = \beta_j \cdot (1 - 2\lambda\mu)$
- If $x_j \neq 0$, set $\beta_j = \beta_j + \lambda(y - p)x_i$

Following this up, we note that we can perform m successive "regularization" updates by letting $B_j = B_j \cdot (1 - 2\lambda\mu)^m$. The basic idea of the new algorithm is to not perform regularization updates for zero-valued x_j's, but instead to simply keep track of how many such updates would need to be performed to update β_j, and perform them only when we would normally perform "LCL" updates (or when we output the parameters at the end of the day).

Here's the final algorithm (for more detail, see "Lazy sparse stochastic gradient descent for regularized multinomial logistic regression", Bob Carpenter[2])

[suggestion: Current lazy regularized SGD algorithm is an approximation to dense regularization. To make it not an approximation, we need to fix two points for the previous write-up algorithm. The original algorithm is shown below:]

1. Let $k = 0$, and let A and B be empty hashtables. A will record the value of k last time $B[j]$ was updated.
2. For $t = 1, \ldots, T$
   - For each example $\mathbf{x}_i, y_i$:
   - Let $k = k + 1$
     - For each non-zero feature of $\mathbf{x_i}$ with index j and value x_j:
       * If j is not in B, set $B[j] = 0$.
       * If j is not in A, set $A[j] = 0$.

[2] `http://lingpipe.files.wordpress.com/2008/04/lazysgdregression.pdf`

* Simulate the "regularization" updates that would have been performed for the $k - A[j]$ examples since the last time a non-zero x_j was encountered by setting

$$B[j] = B[j] \cdot (1 - 2\lambda\mu)^{k-A[j]}$$

* Set $B[j] = B[j] + \lambda(y - p)x_i$
* Set $A[j] = k$

3. For each parameter $\beta_1, \ldots, \beta_d$, set

$$B[j] = B[j] \cdot (1 - 2\lambda\mu)^{k-A[j]}$$

4. Output the parameters $\beta_1, \ldots, \beta_d$.

[suggestion: To make it not an approximation, we have to fix two points:

- Before executing lazy gradients update, we need to loop over non-zero entries to do regularization first. This is because in dense algorithm, the regularization is executed right after gradients update for every example. Therefore, we need to execute previous regularization before current example gradients update.
- After one epoch loop of data, we need to do the remaining regularization for all weights, and clear the clock dict. This is because for next epoch, the learning rate λ is different.

The new algorithm is shown below:]

1. Let $k = 0$, and let A and B be empty hashtables. A will record the value of k last time $B[j]$ was updated.
2. For $t = 1, \ldots, T$
   - Clear table B if B is not empty, Calculate λ based on current epoch
   - For each example $\mathbf{x}_i, y_i$:
   - Let $k = k + 1$
     - For each non-zero feature of $\mathbf{x_i}$ with index j and value x_j:
       * If j is not in B, set B[j] = 0
       * If j is not in A, set A[j] = 0
       * Simulate the "regularization" updates that would have been performed for the $k - A[j] - 1$ examples since the last time a non-zero x_j was encountered by setting

$$B[j] = B[j] \cdot (1 - 2\lambda\mu)^{k-A[j]-1}$$

        * set A[j] = k
    - Calculate p based on regularized B[j]
    - For each non-zero feature of $\mathbf{x_i}$ with index j and value x_j:
        * Simulate the "regularization" updates for current example by setting

$$B[j] = B[j] \cdot (1 - 2\lambda\mu)$$

        * Gradients update by setting $B[j] = B[j] + \lambda(y - p)x_i$
- After one epoch, For each parameter $\beta_1, \ldots, \beta_d$, set

$$B[j] = B[j] \cdot (1 - 2\lambda\mu)^{k - A[j]}$$

3. Output the parameters $\beta_1, \ldots, \beta_d$.

The learning rate λ is often decreased over time. On the t-th sweep through the data, set $\lambda = \frac{\eta}{t^2}$. we used $\eta = 0.5$. (Sometimes λ is also scaled by $1/n_e$, where n_e is the number of examples.) We also used a value of $2\mu = 0.1$.

When running stochastic gradient descent, it is usual to randomize the order of examples, and scale the feature values so that they are comparable (if they are not already binary). However, randomization is not trivial to do for a large dataset. We recommend implementing SGD as a process that streams once through a data stream, with the number of examples n_e being passed in separately as a command-line argument so that the algorithm is aware of what the current value of t is. Then write a separate module that will input a file of examples and then stream the individual examples out in approximately random order.

Hint: to randomly sort a file in linux you can do

```
shuf <text_file>
```

[suggestion: To sort it in mac system, you can install "gshuf" and use it like "shuf"]

To fit our model to the memory of a desktop we will use the hash trick discussed in class: map every word to a features id in the range 0 N (allowing collisions), where N is the dictionary size.

Hint: to convert a string to an id between 0 and N you can do something like

```
int id = word.hashCode() % N;
if (id<0) id+= N;
```

4 Task

[suggestion: Actually, this is not a very good dataset to use for this homework. For this homework, we use 17 binary classifier. As a result, the training of all classes are heavily biased. Most of the example for each class is negative. So a baseline, majority guess can achieve about 92% accuracy, while a correctly implemented algorithm can achieve about 96%. This will cause some problem when grading. Students can easily sacrifice accuracy score to achieve higher performance score. Maybe we should try a dataset with less label, or just implement a 17-class classifier.]

For this assignment, we will be using the same dataset as homework 1b and homework 2. The data appears at `/afs/cs.cmu.edu/project/bigML/dbpedia_16fall/`

The data format is the same as for Assignment 1. The three columns are separated by tab, and the documents are preprocessed so that there are no tab in the body.

To reduce the experimental load we will only use the abstract data sets for this assignment. Again, they are in increasing size so that you can debug your code on smaller data. The files that start with *abstract* include Wikipidia text.

```
abstract.full.test
abstract.full.train
abstract.small.test
abstract.small.train
abstract.tiny.test
abstract.tiny.train
```

In contrast to the previous assignments, we are going to treat the multilabel classification problem as multiple independent binary classification tasks. This means we are going to train **17** binary classifiers. Ideally, to obtain perfect performance (which is often hard in practice), you will need to correctly predict all labels associated with the target document.

Hint: when using "sigmoid" function for classification, you can refer to Part 5 Deliverables Question 1, using the "safe sigmoid" to avoid overflow.

To reduce the experimental load, fix the number of training iterations (scans of data sets) to 20 for all data sets.

5 Deliverables

For your reference, we tested the running time, using a full score code, with following command:

```
time cat train | java -Xmx128m LR 10000 0.5 0.1 1 44925 test > log
```

To understand the meaning of the parameters, please go to the **Autolab Implementation Details** part. We tested it on a Mac with 8GB memory and 2.6 GHZ I5 Core. To achieve good scores on Autolab, your code should run less than 20 seconds to finish 1 iteration on the **small** dataset.

You should implement the algorithm by yourself instead of using any existing machine learning toolkit. You should upload your code (including all your function files) to the HW3: SGD link in Autolab.

Also you should submit a **report** in pdf format via the HW3: Report link in Autolab, which should solve the following questions:

1. Show values of overall likelihood function for each iteration when training with the **small** data set having dictionary size 10000 and $\mu = 0.1$. The objective function is defined as the sum of all **17** classes $\sum_i \sum_c LCL_c(\mathbf{x}^i, y^{c,i})$. Here c is the label id and i is the document id.

   Hint: in order to prevent overflow when calculating p as defined in equation (**??**) you can use a special version of sigmoid function as the following:

   ```
   static double overflow=20;
   protected double sigmoid(double score) {
      if (score > overflow) score =overflow;
      else if (score < -overflow) score = -overflow;
      double exp = Math.exp(score);
      return exp / (1 + exp);
   }
   ```

   [suggestion: You can try to ask students to draw the average LCL per example per class. Average LCL is easier to identify whether the results are in a reasonable range, since we can transfer it to probability. Also we can mention this trick for students, so they can debug their implementations in future]

2. Show accuracy curves for the **small** data set, using 20 iterations, with varying regularization parameter $\mu = 0, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 0.2, 0.3, 0.5, 1$ and fixed dictionary size 10^4, and discuss. [suggestion: Some students say the figure requirement is a little big confusing. You can add this line: "Please set the μ as x-axis to draw the figure"]

3. Show accuracy curves for the **small** data set, using 20 iterations, with varying dictionary sizes $D = 10, 100, 10^3, 10^4, 10^5, 10^6$ and the best μ values you found in the previous step, and discuss. [suggestion: Some students say the figure requirement is a little big confusing. You can add this line: "Please set the D as x-axis to draw the figure"]

4. Currently we are using the L2-Norm regularization to penalize large values of β, which means, in the context of learning as optimizations, instead of optimizing LCL, we are trying to maximizing the objective function as

$$LCL - \mu \sum_{j=1}^{d} \beta_j^2 \tag{4}$$

Here μ controls how much weight to give to the penalty term. Besides the L2-Norm, the L1-Norm, also known as Lasso, is also very commonly used. For more details, you can refer to:

`https://en.wikipedia.org/wiki/Lasso_(statistics)`.

Basically, the optimization objective function for Lasso regression is

$$LCL - \mu \sum_{j=1}^{d} |\beta_j| \tag{5}$$

Please design an efficient lazy regularized SGD optimization algorithm for Lasso regression (Lazy regularizer for L1-Norm). Also please discuss about the differences of L1-Norm and L2-Norm in penalizing β. You do not need to implement this algorithm, just give pseudo code or detailed steps. [suggestion: Please clarify that the gradients for the regularization when weight = 0 is 0]

5. Currently we are using an "on-line" style SGD in training, which means we approximated the true gradients over the whole corpus by gradients at a single example. This is easy to imeplement and especially efficient with lazy regularizer for **sparse** data.

Besides the "on-line" style, we can also do optimization in a "mini-batch" style, which is to approximate true gradients with more than one training examples. This means we will aggregate gradients from several training examples before we updating β. Usually this can result in smoother convergence, as the gradient computed at each step uses more training examples. For more details, you can refer to:

`https://en.wikipedia.org/wiki/Stochastic_gradient_descent`

Suppose we are using L2-Norm regularized logistic regression, please design a lazy regularizied SGD in "mini-batch" style. You can assume the "mini-batch" size is 200, which means you will aggregate gradients for 200 training examples and then update β. You do not need to implement this algorithm, just give pseudo code or detailed steps. Also please discuss if "mini-batch" style lazy regularizer is supposed to be as efficient as "on-line" style lazy regularizer.

Hint: the "on-line" style lazy regularizer is efficient because it takes advantages of the sparsity of a single training example. If the feature space of a single training example is not sparse, "on-line" style lazy regularizer will not be efficient.

6. Answer the questions in the collaboration policy on page 1.

Autolab Implementation Details

Your logistic regression program LR.java should be able to run without the out-of-memory issue, using the follow example command:

```
for((i=1;i<=20;i++));
do shuf trainData.txt;
done | java -Xmx128m LR 10000 0.5 0.1 20 1000 testData.txt
```

Here, the first argument 10000 is the vocabulary size. Second argument 0.5 is the initial value of the learning rate. The third argument 0.1 is the regularization coefficient. The fourth argument 20 is the max iteration (# of passes through data). The fifth argument is the size of the training dataset (which allows you to determine the starting point of a new pass). The command produces output in the following format (**one-line-per-test-example**):

```
label1<tab>p_label1,label2<tab>p_label2,label3<tab>p_label3,...
```

Here, binary classifiers and their prediction scores are separated by comma. For each of the classifier, for example, in the first entry, `label1` refers to the name of the classifier, and `p_label1` is the posterior probability of the label given the observation. If the posterior is larger than or equal to 0.5, then it means your classifier predicts a positive output for this target label in this document. A tab is used to separate the label name and its posterior.

You should tar the following items into **hw3.tar** and submit to the homework 3 assignment via Autolab:

- LR.java
- and all other auxiliary functions you have written

Tar the files directly using `tar -cvf hw3.tar *.java`. Do **NOT** put the above files in a folder and then tar the folder. You do not need to upload the saved temporary files. Please make sure your code is working fine on `unix.andrew.cmu.edu` machines before you submit.

Also, please do not forget to submit your report "hw3.pdf" via the HW3: Report link in Autolab.

6 Submission

You must submit your homework through Autolab via the HW3: SGD. In this homework, we provide an additional tool called HW3-validation:

- HW3: validation: You will be notified by Autolab if you can successfully finish your job on the Autolab virtual machines. Note that this is **NOT** the place you should debug or develop your algorithm. All development should be done on **unix.andrew.cmu.edu** machines. This is basically an Autolab debug mode. There will be **NO** feedback on your **performance** in this mode. You have unlimited amount of submissions here. To avoid Autolab queues on the submission day, the validation link will be closed 24 hours prior to the official deadline. If you have received a score of 1000, this means that you code has passed the validation.
- HW3: SGD. This is where you should submit your validated final submission. You have a total of **10 possible unpenalized submissions** after which each additional submission will be penalized by an additional 2 percent. Your performance will be evaluated, and feedback will be provided immediately. Late submissions will lead to a 50% penalty, and we do not accept submissions later than 2 days.
- HW3: Report. This is where you should submit your report.

7 Grading

The total grade of this assignment is **100 points**. You will be graded based on the **memory usage (25 points)** and **runtime (25 points)** for your code, and your **final test performance (20 points)**. Note that each pass in SGD is randomized, and there will be sampling variance when measuring your memory and runtime information. You may contact us to use your highest Autolab submission grade after homework is due. The **report (30 points)** will be graded manually.