University of Washington
Department of Computer Science and Engineering / Department of Statistics

CSE 547 / Stat 548 Machine Learning (Statistics) for Big Data

Homework 1
Spring 2015

Issued: Thursday, April 2, 2015 **Due:** Thursday, April 16, 2014

Suggested Reading: Assigned Readings in Case Study I (see website).
Instructions: The homework consists of two parts: (i) Problems 1.1 to 1.3 cover theoretical and analytical questions and (ii) Problem 1.4 covers data analysis questions. Completed assignments should be submitted via Catalyst on the due date **before** the start of class. Refer to the course webpage for policies regarding collaboration and extensions.

Problem 1.1

(Source: KM Exercise 8.6) **Elementary properties of l_2 regularized logistic regression[24 points + 1 bonus]**
Consider minimizing

$$J(\mathbf{w}) = -l(\mathbf{w}, \mathcal{D}_{\text{train}}) + \lambda \|\mathbf{w}\|_2^2$$

where

$$l(\mathbf{w}, \mathcal{D}) = \sum_j \ln \mathbf{P}(y^j | \mathbf{x}^j, \mathbf{w})$$

is the log-likelihood on data set $\mathcal{D}$, for $y^j \in \{-1, +1\}$. Determine whether the following statements are true or false. Briefly explain.

(a) [3.5 points] With $\lambda > 0$ and the features x_k^j linearly separable, $J(\mathbf{w})$ has multiple locally optimal solutions.

(b) [3.5 points] Let $\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} J(\mathbf{w})$ be a global optimum. $\hat{\mathbf{w}}$ is typically sparse (has many zero entries).

(c) [3.5 points] If the training data is linearly separable, then some weights w_j might become infinite if $\lambda = 0$.

(d) [3.5 points] $l(\hat{\mathbf{w}}, \mathcal{D}_{\text{train}})$ always increases as we increase λ.

(e) [3.5 points] $l(\hat{\mathbf{w}}, \mathcal{D}_{\text{test}})$ always increases as we increase λ.

Now answer the following questions:

(f) [1 point] Can the decision boundary in unregularized logistic regression change if we add an additional variable that is a duplicate of one of the variables already in the model? Give an intuitive answer (Hint: think about the model assumptions).

(g) Let us say our $\mathcal{D}_{\text{train}}$ has n examples, and only one feature x_1. Now we create a dataset $\mathcal{D}_{\text{mod}}$, with n examples and two features x_1 and x_2 where each example in $\mathcal{D}_{\text{mod}}$ contains the feature on $\mathcal{D}_{\text{train}}$ twice, and the same label. We learn a logistic regression from $\mathcal{D}_{\text{train}}$, which will give us two parameters: w_0 and w_1. We also learn a logistic regression from $\mathcal{D}_{\text{mod}}$, which will give us three paramters: w'_0, w'_1, w'_2.

i. [2 points] Write down the **unregularized** log-likelihood we want to maximize for each of the two logistic regressions.

ii. [3.5 points] Given the log-likelihood functions, what is the relationship between (w_0, w_1) and (w'_0, w'_1, w'_2)? Using this relationship, answer question (e) again here, more formally.

iii. [1 bonus point] Would your answer for the previous question change if we were using L2 regularization?. Argue why or why not (Remember we don't regularize w_0).

Problem 1.2

On Slide 7 of the first lecture, we presented multi-class logistic regression, where $Y \in \{y_1, ..., y_R\}$. Here, we have a simplified version, with no w_0. When $k < R$, the posterior probability is given by:

$$P(Y = y_k|X) = \frac{exp(\langle w_k, X\rangle)}{1 + \sum_{j=1}^{R-1} exp(\langle w_j, X\rangle)}$$

For $k = R$, the posterior is:

$$P(Y = y_k|X) = \frac{1}{1 + \sum_{j=1}^{R-1} exp(\langle w_j, X\rangle)}$$

Where $\langle w_j, X\rangle = \sum_{i=1}^{n} w_{ji} X_i$ (i.e. the dot product). We can replace the two equations above by a single equation, to simplify notation. For such, we introduce a fixed, pseudo parameter vector $w_R = [0, 0, 0, ..., 0]$. Now, for any label y_k, we write:

$$P(Y = y_k|X) = \frac{exp(\langle w_k, X\rangle)}{1 + \sum_{j=1}^{R-1} exp(\langle w_j, X\rangle)}$$

(a) [4 points] How many parameters do we need to estimate? What are these parameters?

(b) [4 points] Given N training samples $\{(x^1, y^1), (x^2, y^2), ..., (x^N, y^N)\}$, write down explicitly the log-likelihood function and simplify it as much as you can:

$$L(w_1, ..., w_{R-1}) = \sum_{j=1}^{N} ln(P(y^j|x^j, w))$$

(c) [4 points] Compute the gradient of L with respect to each w_k and simplify it.

(d) [4 points] Now add the regularization term $\frac{\lambda}{2}$ and define a new objective function:

$$L(w_1, ..., w_{R-1}) = \sum_{j=1}^{N} ln(P(y^j|x^j, w)) - \frac{\lambda}{2} \sum_{l=1}^{R-1} \|w_l\|_2^2$$

Compute the gradient of this new L with respect to each w_k

Problem 1.3

The Count-Min sketch of Cormode and Muthukrishnan is biased. That is, the estimated count $\hat{a}_i$ for element $i \in \{1, \ldots, N\}$ is always higher than (or equal to) the true count a_i. Reminder: The count a_i is the number of times we see element i in the sequence. In this question, you will develop a simple unbiased sketch, *Simple-Count*, (with weaker convergence rates than the Count-Min sketch).

First, we will start with the simplest version of Simple-Count: Let g be a hash function chosen from a family G of independent hashes, such that g maps each i to either $+1$ or -1 with equal probability: [1]:

$$P(g(i) = +1) = P(g(i) = -1) = 1/2.$$

We now define h, the accumulator of our sketch. When we observe element i in the sequence, we simply update:

$$h = h + g(i).$$

Now, if we would like to predict the count for element i, we simply return:

$$\hat{a}_i = h\ g(i).$$

Given this sketch, please answer the following questions:

[1] The randomness arises from the fact that the hash function g is drawn randomly from the family G. Given a hash function g, the mapping $g : \{1, \ldots, N\}$ is deterministic. All expectations, etc. are taken with respect to the distribution of g.

(a) [2 points] Let a_i be the true counts for each element i. Express h in terms of the a_i and $g(i)$ only.

(b) [2 points] What is the expected value of $g(i)$, denoted by $E[g(i)]$?

(c) [4 points] Prove that $\hat{a}_i = h\ g(i)$ is an unbiased estimate of a_i, i.e., $E[\hat{a}_i] = a_i$. Hint: use linearity of expectations, $E[u + v] = E[u] + E[v]$, and the fact that $g(i)$ and $g(j)$ are independent.

(d) [4 points] Prove that the variance of our estimate $Var(\hat{a}_i)$ is given by:

$$Var(\hat{a}_i) = \sum_{j \in \{1,\ldots,N\}: j \neq i} a_j^2.$$

Hint: recall that $Var(X) = E[X^2] - (E[X])^2$.

(e) [4 points] We will now bound the probability of getting a bad estimate. In particular, after n steps, we will say our estimate $\hat{a}_i$ is ϵ-bad if, for $\epsilon > 0$:

$$|\hat{a}_i - a_i| \geq \epsilon n.$$

To prove our bound, we will use Chebyshev's inequality: If X is a random variable, and $\alpha > 0$, then:

$$P(|X - E[X]| \geq \alpha) \leq \frac{Var(X)}{\alpha^2}.$$

Use Chebyshev's inequality to prove that the probability δ of getting a bad estimate for $\hat{a}_i$ is bounded by:

$$\delta \leq \frac{Var(\hat{a}_i)}{\epsilon^2 n^2} \leq \frac{1}{\epsilon^2}.$$

(f) The bound in the previous question is going to be vacuous for sufficiently small ϵ. To address this issue, we will expand the number of hash functions in our sketch. Let's introduce a set of k independent hash functions g_j with the same properties as g above. Now, we will create h_j, in analogy to the h function above, for each g_j. When we see element i in the sequence, we will update each h_j by:

$$h_j = h_j + g_j(i).$$

Now, if we would like to predict the count for element i, we simply return the average:

$$\hat{a}_i = \frac{1}{k} \sum_{j=1}^{k} h_j\ g_j(i).$$

For this sketch, prove that:

i. [2 points] The variance of $\hat{a}_i$ is now bounded by:

$$Var(\hat{a}_i) \leq \frac{n^2}{k}.$$

Hint: The estimates obtained by each hash function are independent.

ii. [2 points] Use this result and the Chebyshev's inequality as above to prove that for any $\epsilon > 0, \delta > 0$, the probability of getting an ϵ-bad estimate of $\hat{a}_i$ will be lower than δ if we use $k \geq \frac{1}{\delta\epsilon^2}$ hash functions.

Problem 1.4

Logistic Regression for Ads Click Prediction[40 + 5 bonus points]

In this problem, you will train a logistic regression model to predict the Click Through Rate (CTR) on a dataset with ∼1 million examples. The CTR provides a measure of the popularity of an advertisement, and the features we will use for prediction include attributes of the ad and the user. You will also implement the hashing kernel, where the features are hashed into a smaller space. At the end, there is an extra credit component for implementing multitask logistic regression for personalized CTR prediction (see the "Weinberger, Kilian, et al." paper from the reading list).

Dataset

The dataset we will consider comes from the 2012 KDD Cup Track 2. Here, a user types a query and a set of ads are displayed and we observe which ad was clicked. For example:

1. Alice went to the famous search engine Elgoog, and typed the query "big data".
2. Besides the search result, Elgoog displayed 3 ads each with some short text including its title, description, etc.
3. Alice then clicked on the first advertisement.

This completes a **SESSION**. At the end of this session Elgoog logged 3 records:

Clicked = 1	Depth = 3	Position = 1	Alice	Text of Ad1
Clicked = 0	Depth = 3	Position = 2	Alice	Text of Ad2
Clicked = 0	Depth = 3	Position = 3	Alice	Text of Ad3

In addition, the log contains information about Alice's age and gender. Here is the format of a complete row of our training data:

Clicked	Depth	Position	Userid	Gender	Age	Text Tokens of Ad

Let's go through each field in detail:

- "Clicked" is either 0 or 1, indicating whether the ad is clicked.
- "Depth" takes a value in $\{1, 2, \ldots, \}$ specifying the number of ads displayed in the session.
- "Position" takes a value in $\{1, 2, \ldots, Depth\}$ specifying the rank of the ad among all the ads displayed in the session.
- "Userid" is an integer id of the user.
- "Age" takes a value in $\{0, 1, 2, 3, 4, 5, 6\}$, indicating different ranges of a user's age: '0' if the age is unkown, '1' for (0, 12], '2' for (12, 18], '3' for (19, 24], '4' for (24, 30], '5' for (30, 40], and '6' for greater than 40.
- "Gender" takes a value in $\{-1, 0, 1\}$, where -1 stands for male, 1 stands for female, and 0 means that the gender is unknown.
- "Text Tokens" is a comma separated list of token ids. For example: "15,251,599" means "token_15", "token_251", and "token_599". (Note that due to privacy issues, the mapping from token ids to words is not revealed to us in this dataset, e.g., "token_32" to "big".)

Here is an example that illustrates the concept of features "Depth" and "Position". Suppose the list below was returned by Elgoog as a response to Alice's query. The list has $depth = 3$. "Big Data" has $position = 1$, "Machine Learning" has $position = 2$ and so forth.

Big Data
Machine Learning
Cloud Computing

Here is a sample from the training data:

0 | 2 | 2 | 280151 | 1 | 2 | 0,1,154,173,183,188,214,234,26,3,32,36,37,4503,51,679,7740,8,94

The test data are in the same format except that they do not have the first label field, which is stored in a separate file named "test_label.txt". Some data points do not have user information. In these cases, the userid, age, and gender are set to zero.

Feature Representation

In class, we simply denote

$$x^t = [x_1^t, \ldots, x_d^t] \tag{1}$$

as an abstract feature vector. In the real world, however, constructing the feature vector requires some thought.

- First of all, not everything in the data should be treated as a feature. In this dataset, "Userid" should not be treated as feature.
- Similarly, we cannot directly use the list of token ids as features in Eq. 1 since the numbers are arbitrarily assigned and thus meaningless for the purposes of regression. Instead, we should think of the list of token ids $L \equiv [l_1, l_2, l_3, \ldots]$ as a compact representation of a sparse binary vector $\mathbf{b}$ where $\mathbf{b}[i] = 1 \quad \forall i \in L$. It is important to think in terms of the binary representation but implement the code using a compact representation.
- As for the rest of the features: "Depth", "Position", "Age", and "Gender", they are scalar variables, so please use their original value as the feature.

Accessing and Processing the Data

(a) Download "clickprediction_data.zip" from the course website.

(b) After unzipping the folder, there should be three files: train.txt, test.txt and test_label.txt.

(c) For instructions on setting up Java/Eclipse and using the starter code, please read section at the end of the file.

1. Warm up

We begin by simply assessing various attributes of the dataset, primarily to ensure that it is correctly accessed and parsed.

If you are using the starter code, please complete the functions in "analysis/BasicAnalysis.{java,py}". In the starter code, you will find "analysis/DummyLoader.{java,py}" as sample code for initializing the dataset, iterating over each row, parsing the text, and printing out results. **NOTE**: You will have to change the location of the dataset in `main` to reflect where you put the data on your system.

(a) [1 point] Report the average CTR for the training data (Number of clicks / Number of examples).

(b) [2 points] How many unique tokens are there in the training data? What about the test data? How many tokens appear in both datasets?

(c) [2 points] How many unique users are there in each age group in the training data? What about the test data?

2. Stochastic Gradient Descent

Recall that stochastic gradient descent (SGD) performs a gradient descent using a noisy estimate of the full gradient based on just the current example.

(a) [3 points] Write down the equation for the weight update step. That is, how to update weights w^t using the data point (x^t, y^t), where $x^t \equiv [x_1^t, x_2^t, \ldots, x_d^t]$ is the feature vector for example t, and $y^t \in \{0, 1\}$ is the label.

(b) For stepsizes $\eta = \{0.001, 0.01, 0.05\}$ and without regularization, implement SGD and train the weights by making one pass over the dataset. **Use only one pass over the data on all subsequent questions as well.** For each step size:

- [3 points] Plot the average loss $\overline{L}$ as a function of the number of steps T, where

$$\overline{L}(T) = \frac{1}{T} \sum_{t=1}^{T} (\hat{y}^t - y^t)^2$$

  where $\hat{y^t}$ is the predicted label of example x^t using the weights w^{t-1}. Record the average loss every 100 steps, e.g. $[100, 200, 300, \ldots]$.
- [3 points] Report the l_2 norm of the weights at the end of the pass.
- [3 points] Use the weights to predict the CTRs for the test data. Recall that "test_label.txt" contains the labels for the test data. Report the RMSE (root mean square error) of your predicted CTR. Also report the RMSE of the baseline prediction you got from the Warm Up. (Do not expect a huge improvement since the label distribution is biased. Elgoog still makes a huge profit even with a 0.1% improvement in accuracy.)
  Hint: you can use the given Util/EvalUtil.{java,py} to compute RMSE.

(c) [3 points] For $\eta = 0.01$, report the weights for the following features: intercept, "Position", "Depth", "Gender", and "Age". Provide an interpretation of the effect of each feature on the probability of a click based on these inferred weights.

Hint
Java (and Python) users: You need to complete the "LogisticRegression.{java,py}". Ignore the lambda and "performDelayedRegularization()" for now, which will be useful in the next question "Regularization".

Big data is often sparse. In this problem, the feature space is huge (the order is on the size of the entire token vocabulary). Fortunately, you do not need to update every feature for every data point. Why? Because a data point only has a few tokens, and the gradient of w_i will be non-zero if and only if feature i is non-zero. In other words, you just need to update the weights corresponding to the tokens that appear in the current example. Other weights will stay the same. Taking advantage of the data sparsity is one of the key weapons for attacking big data problems.

3. Regularization

Notice that the l_2 norm of the weights in the previous part is not small and keeps growing as we get more and more data. It is necessary to add l_2 regularization to each update step. However, the regularization is not a sparse update. At every step, the regularization affects the weights for all of the features, not just the ones that appeared in the current example. To deal with this issue, we will try to be as lazy as possible. What if at each iteration we just regularize the weights that affect the current example and hope for the best? Unfortunately, this is too lazy because it will be unfair to features that appear frequently.

The trick is to delay the regularization for w_i until we encounter a data point that affects it. Suppose feature i appeared for the first time at time t_1. No regularization of w_i is needed because its value is 0. Then at time t_2, feature i shows up again. You know that the regularization for w_i was delayed for $t_2 - t_1 - 1$ steps, so its time to let it pay. How much? Each step of the regularization downweights w_i by a factor of $(1 - \eta * \lambda)$, so the total is $(1 - \eta * \lambda)^{t_2 - t_1 - 1}$. To implement the lazy regularization, you need to keep track of the update timestamp for the weights on sparse tokens.

NOTE: Only regularize the feature weights, not the intercept.

(a) [5 points] Implement the regularization, and train the weights again using stepsize $\eta = 0.05$ with λ ranging from 0 to 0.014 spaced by 0.002, e.g. $[0, 0.002, 0.004, \ldots, 0.014]$.

   i. Plot the l_2 norm of the weights as a function of λ.

   ii. Is there a consistent trend in the l_2 norm as λ increases? Why does this make sense?

   iii. As we increase $\lambda \to \infty$, what will the l_2 norm converge to?

(b) [5 points] Predict the CTR for the test data and evaluate the RMSE. Plot the RMSE as a function as λ.

Hint

Java (and Python) users: You need to complete the function "performDelayedReg-

ularization()" in "LogisticRegression.{java,py}". "Weights.accessTime" is a map for keeping track of the access time of token weights. For example, "w.accessTime.get(256)" should return the most recent time when the weight for token_256 was updated, or `null` if it's never been updated before.

4. Hashing Kernel [10 points]

The "Weinberger, Kilian, et al." paper introduces an unbiased hash kernel $\phi : \mathcal{X} \to \mathcal{F}$. The original feature space $\mathcal{X}$ is transformed into a space $\mathcal{F}$ with lower dimension through two hash functions: $h : \mathcal{I} \to \{0, \ldots, m-1\}$, and $\xi : \mathcal{I} \to \{+1, -1\}$, where $\mathcal{I}$ indexes the original feature space $\mathcal{X}$. In this problem, we only ask you to hash the text features, keeping the rest of the features as before. Therefore, $\mathcal{I}$ will be the space of all token ids.

The new feature vector (for the text features) $\phi(x)$ will be an m-dimensional array, where the $\phi(x)_i = \sum_{j:h(j)=i} \xi(j) X_j$. Now, we can run the same SGD algorithm in the hashed feature space. The sparse updating and lazy regularization tricks still apply.

Train the weights in the hashed feature space with $m = \{101, 12277, 1573549\}$, $\lambda = 0.001$ and stepsize $\eta = 0.01$. Report the RMSE of the predicted CTRs for all 3 cases.

Hint
Java (and Python) users: Complete the "HashDataInstance.{java,py}" and "LogisticRegressionWithHashing.{java,py}". The starter code has two hash functions in "util/HashUtil.class" where you can use as h and ξ. Ignore the personalized flag. Make sure the runtime does not depend on the size of the hash space m.

5. Extra Credit: Personalization [5 bonus points]

If you have read and understood the "Weinberger, Kilian, et al." paper in its entirety, you can implement a personalized version of CTR prediction. It's just a few lines of code to change: Instead of hashing each feature once, you hash it again with the userid. The rest remains the same.

Implement the personalized logistic regression with hashing. Train the weights using $\eta = 0.01$, $m = 12277$, and $\lambda = 0.001$.

(a) Report the RMSE on the test data (including all users).

(b) Report the RMSE just based on the subset of users who appear both in the test and training data.

Instructions for starter code and setup If you use the starter code, here are files you need to print out and attach to the end of your writeup:

- BasicAnalysis.{java,py}
- HashedDataInstance.{java,py}
- LogisticRegression.{java,py}
- LogisticRegressionWithHashing.{java,py}

Java

(a) Eclipse is a good editor for programming Java. Download Eclipse Classic 4.2.1 at:

http://www.eclipse.org/downloads/

To install, just unzip the downloaded file. Double click the eclipse executable to launch.

(b) To import the starter code, go to the menu File → Import. Under general, select Existing Projects into Workspace, and click Next. Choose "Select archive file", and find "ClickPrediction.zip" that you downloaded from the course website. Click Finish.

Python

(a) Eclipse can also be used for editing Python (with the PyDev plugin), although you may already have your favorite editor. If you choose to use Eclipse, follow the instructions above for downloading Eclipse and see the section entitled "Installing with the update site" at http://pydev.org/manual_101_install.html for installing PyDev.

(b) To run the code after unzipping, you will need to set your `PYTHONPATH` to the root of the project. For example, when running from the command line, from `/path/to/ClickPrediction/`, you need to run

```
PYTHONPATH=. python analysis/DummyLoader.py
```

General advice for Java/Python users

(a) If you get a `Nan`, either you divided something by zero or the $exp(w^T x)$ overflowed.

(b) Be careful when dividing an integer. Java and Python perform rounding for integer division. For example: x = 1; x/2; gives you zero. Use x/2.0; instead. When you have two integer variables, cast one into double (Java) or float (Python).

(c) (i) Java: Use `map.containsKey(key)` before asking a value from a map. Or use "Integer" and "Double" object to store the return from `map.get(key)`, and check `null`. `int x = map.getKey(y)` will throw an exception if the key `y` does not exist.

(ii) Python: Use `map.has_key(key)` before asking a value from a map. Or use `map.get(key)` and check against `None`. `x = map[y]` will throw an exception if the key `y` does not exist.

(d) If you are using the starter code, remember to call Dataset.reset() after every pass of the data.

(e) (Java) If you get "out of heap space error", you probably need a larger heap space for jvm. In Eclipse, go to the menu run → run configuration. On the left panel, select the application you just ran, on the right panel, select the Arguments next to Main. Type `-Xmx1g` on the second input box (VM arguments). This will ensure 1G heap space, which should be enough for this homework.