HOMEWORK 4: LOGISTIC REGRESSION

10-601 Introduction to Machine Learning (Fall 2018)
Carnegie Mellon University
piazza.com/cmu/fall2018/10601bd
OUT: Sun, Sept 30, 2018*
DUE: Tue, Oct 9, 2018 11:59 PM
TAs: Rongye Shi, Rawal Khirodkar, Jeremy Ong, and Emilio Arroyo-Fang

Summary In this assignment, you will build a sentiment polarity analyzer, which will be capable of analyzing the overall sentiment polarity (positive or negative) . In Section 1 you will warm up by deriving stochastic gradient descent updates for multinomial logistic regression. Then in Section 2 you will implement a binary logistic regression model as the core of your natural language processing system.

START HERE: Instructions

- **Collaboration Policy**: Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 3.4"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the collaboration policy on the website for more information: http://www.cs.cmu.edu/~mgormley/courses/10601bd-f18/about.html#7-academic-integrity-policies
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601bd-f18/about.html#6-general-policies
- **Submitting your work:** You will use Gradescope to submit answers to all questions, and Autolab to submit your code. Please follow instructions at the end of this PDF to correctly submit all your code to Autolab.
  - **Gradescope:** For written problems such as derivations, proofs, or plots we will be using Gradescope (https://gradescope.com/). Submissions can be handwritten, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Upon submission, label each question using the template provided. Regrade requests can be made, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted. Each derivation/proof should be completed on a separate page.
  - **Autolab:** You will submit your code for programming questions on the homework to Autolab (https://autolab.andrew.cmu.edu/). After uploading your code, our grading

*Compiled on Tuesday 9th October, 2018 at 23:09

scripts will autograde your assignment by running your program on a virtual machine (VM). The software installed on the VM is identical to that on `linux.andrew.cmu.edu`, so you should check that your code runs correctly there. If developing locally, check that the version number of the programming language environment (e.g. Python 2.7, Octave 3.8.2, OpenJDK 1.8.0, g++ 4.8.5) and versions of permitted libraries (e.g. `numpy` 1.7.1) match those on `linux.andrew.cmu.edu`. Octave users: Please make sure you do not use any Matlab-specific libraries in your code that might make it fail against our tests. Python3 users: Please include a blank file called python3.txt (case-sensitive) in your tar submission. You have a **total of 10 Autolab submissions**. Use them wisely. In order to not waste Autolab submissions, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly first before any Autolab submission.

- **Materials:** Download from autolab the tar file ("Download handout"). The tar file will contain all the data that you will need in order to complete this assignment.

Linear Algebra Libraries When implementing machine learning algorithms, it is often convenient to have a linear algebra library at your disposal. In this assignment, Java users may use EJML[a] and C++ users Eigen[b]. Details below. (As usual, Python users have numpy; Octave users have built-in matrix support.)

Java EJML is a pure Java linear algebra package with three interfaces. We strongly recommend using the SimpleMatrix interface. Autolab will use EJML version 3.3. The command line arguments above demonstrate how we will call you code. The classpath inclusion `-cp "./lib/ejml-v0.33-libs/*:./"` will ensure that all the EJML jars are on the classpath as well as your code.

C++ Eigen is a header-only library, so there is no linking to worry about—just `#include` whatever components you need. Autolab will use Eigen version 3.3.4. The command line arguments above demonstrate how we will call you code. The argument `-I./lib` will include the `lib/Eigen` subdirectory, which contains all the headers.

We have included the correct versions of EJML/Eigen in the handout.tar for your convenience. Do **not** include EJML or Eigen in your Autolab submission tar; the autograder will ensure that they are in place.

[a] https://ejml.org
[b] http://eigen.tuxfamily.org/

1 Written Questions [20 pts]

1.1 Multinomial Logistic Regression [12 pts]

Multinomial logistic regression, also known as softmax regression and multiclass logistic regression, is a generalization of binary logistic regression. In this problem setting we have a dataset:

$$\mathcal{D} = \left\{ \left(\mathbf{x}^{(1)}, y^{(1)} \right), \ldots, \left(\mathbf{x}^{(N)}, y^{(N)} \right) \right\} \text{ where } \mathbf{x}^{(i)} \in \mathbb{R}^M, y^{(i)} \in \{1, \ldots, K\} \text{ for } i = 1, \ldots, N$$

Here N is the number of training examples, M is the number of features, and K is the number of classes, which is usually greater than two to be interesting and not equivalent to binary logistic regression.

(a) **[2 Points]** To motivate multinomial logistic regression, we will first look at a general way to extend a binary classifier to a multiclass classifier and apply it to logistic regression. One way to extend a binary classifier to a multiclass classifier is to use the One versus Rest (OvR) method. In this method, we will train K binary classifiers, independently. Each classifier $h_i(\mathbf{x})$ will determine if a point $\mathbf{x}$ is in class i or not for $i = 1, \ldots, K$.

Suppose we have successfully trained K *binary logistic regression* classifiers in such a manner. Propose a method for determining the class of a new unlabeled point $\mathbf{x}^{(*)}$ using the set of binary logistic regression classifiers $h_1(\mathbf{x}), \ldots, h_K(\mathbf{x})$. Explain your reasoning.

Solution

Assume any binary logistic regression classifier $h_i(x)$, which learns the weights w_i and outputs a score given by $s_i = w_i^T x^{(i)}$ for the given point $x^{(i)}$.

Now get the scores $s_i, \forall i = 1, 2, ...N$ and compute the label $y^{(i)}$ of the given point $x^{(i)}$ as

$$y^{(i)} = \arg\max s_i$$

This will always asign the correct label to the point because the data point which has the highest score has the highest probability of the belonging to that class, when compared with other classes.

(b) **[1 Points]** Now we would like a method to do multiclass classification without having to train more than one classifier. Multinomial logistic regression is such a method. Remember that in multinomial logistic regression, we have

$$p\left(y \mid \mathbf{x}, \boldsymbol{\Theta}\right) = \frac{\exp\left(\boldsymbol{\theta}_y \mathbf{x}\right)}{\sum_{j=1}^{K} \exp\left(\boldsymbol{\theta}_j \mathbf{x}\right)} = \text{softmax}((\boldsymbol{\Theta}\mathbf{x})_y)$$

where $\boldsymbol{\Theta}$ is the parameter matrix of size $K \times (M+1)$ and $\boldsymbol{\theta}_y$ denotes the yth **row** of $\boldsymbol{\Theta}$, which is the parameter vector for class y. Since we have folded the bias term into $\boldsymbol{\Theta}$ we now have $\mathbf{x} \in \mathbb{R}^{M+1}$. Let us represent class C_k with a *one-hot encoding*, specifically let $C_k \in \mathbb{R}^K$ where the kth entry in C_k is 1, and 0 everywhere else. Let us also define a target matrix $\mathbf{T}$ of size $N \times K$, where the ith **row** of $\mathbf{T}$ is $C_{y^{(i)}}$, where only the $y^{(i)}$th entry is 1, and 0 else where.

Write down the data conditional likelihood $\mathcal{L}(\boldsymbol{\Theta} \mid \mathbf{T}, \mathbf{X})$ in terms of N, K, $\mathbf{T}$ and $p\left(C_j \mid x^{(i)}, \boldsymbol{\Theta}\right)$. Please note that $\mathcal{L}(\boldsymbol{\Theta} \mid \mathbf{T}, \mathbf{X}) = p(\mathbf{T} \mid \boldsymbol{\Theta}, \mathbf{X})$, where likelihood is a function of parameters (not probability), and it is equal in value to the label probability conditioned on data and parameters.

Solution

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\Theta} \mid \mathbf{T}, \mathbf{X}) &= p(\mathbf{T} \mid \boldsymbol{\Theta}, \mathbf{X}) \\
&= \prod_{i=1}^{N} p(\mathbf{T} \mid x^{(i)}, \boldsymbol{\Theta}) \\
&= \prod_{i=i}^{N} \prod_{j=1}^{K} p(C_j \mid x^{(i)}, \boldsymbol{\Theta})^{\mathbf{T}_{i,j}}
\end{aligned}$$

(c) **[1 Points]** Write down the *negative* conditional log-likelihood of the data in terms of N, K, $\mathbf{T}$ and $p\left(C_j \mid x^{(i)}, \mathbf{\Theta}\right)$. This will be your objective function $J(\mathbf{\Theta})$, also known as cross-entropy loss.

Solution

$$
\begin{aligned}
J(\mathbf{\Theta}) &= -\log(\mathcal{L}(\mathbf{\Theta} \mid \mathbf{T}, \mathbf{X})) \\
&= -\log p(\mathbf{T} \mid \mathbf{\Theta}, \mathbf{X}) \\
&= -\log(\prod_{i=i}^{N} \prod_{j=1}^{K} p(C_j \mid x^{(i)}, \mathbf{\Theta})^{\mathbf{T}_{i,j}}) \\
&= \sum_{i=1}^{N} -\log(\prod_{j=1}^{K} p(C_k \mid x^{(i)}, \mathbf{\Theta})^{\mathbf{T}_{i,j}})) \\
&= -\sum_{i=1}^{N} \sum_{j=1}^{K} \mathbf{T}_{i,j} \log(p(C_j \mid x^{(i)}, \mathbf{\Theta}))
\end{aligned}
$$

(d) **[4 Points]** Now let's derive the partial derivative of the objective function with respect to the kth parameter vector $\boldsymbol{\theta}_k$. That is, derive $\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}_k}$, where $J(\boldsymbol{\Theta})$ is the objective function that you provided above. Show that the partial derivative is as follows:

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}_k} = -\sum_{i=1}^{N} \Big(\mathbf{T}_{i,k} - p\left(C_k \mid \mathbf{x}^{(i)}, \boldsymbol{\Theta}\right) \Big) \mathbf{x}^{(i)}$$

Show all steps of the derivation. (Please pay attention that $\mathbf{T}_i$ is $C_{y^{(i)}}$, and you should think about the reason why $y^{(i)}$ disappears in this equation, and what is the relation between $y^{(i)}$ and $\mathbf{T}_{i,k}$).

Solution

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}_k} = -\sum_{i=1}^{N} \frac{\partial}{\partial \boldsymbol{\theta}_k} \Big(\sum_{j=1}^{K} \mathbf{T}_{i,j} \log(p(C_j \mid x^{(i)}, \boldsymbol{\Theta})) \Big)$$

Now consider $\log(p(C_j \mid x^{(i)}, \boldsymbol{\Theta}))$.

$$\begin{aligned} \log(p(C_j \mid x^{(i)}, \boldsymbol{\Theta})) &= \log\left(\frac{exp(\boldsymbol{\theta}_j x^{(i)})}{\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)})} \right) \\ &= \boldsymbol{\theta}_j x^{(i)} - \log\left(\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)}) \right) \end{aligned}$$

Plugging this back into the derivative equation, we get,

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}_k} = -\sum_{i=1}^{N} \frac{\partial}{\partial \boldsymbol{\theta}_k} \Big(\sum_{j=1}^{K} \mathbf{T}_{i,j} \boldsymbol{\theta}_j x^{(i)} - \sum_{j=1}^{K} \mathbf{T}_{i,j} \log\left(\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)}) \right) \Big)$$

Now consider the second term in the above differential equation,

$$\begin{aligned} &= \sum_{j=1}^{K} \mathbf{T}_{i,j} \frac{\partial}{\partial \boldsymbol{\theta}_k} \left(\log\left(\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)}) \right) \right) \\ &= \mathbf{T}_{i,1} x^{(i)} \frac{exp(\boldsymbol{\theta}_k x^{(i)})}{\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)})} + \mathbf{T}_{i,2} x^{(i)} \frac{exp(\boldsymbol{\theta}_k x^{(i)})}{\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)})} + ... + \mathbf{T}_{i,K} x^{(i)} \frac{exp(\boldsymbol{\theta}_k x^{(i)})}{\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)})} \\ &= x^{(i)} \frac{exp(\boldsymbol{\theta}_k x^{(i)})}{\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)})} \Big(\mathbf{T}_{i,1} + \mathbf{T}_{i,2} + ... + \mathbf{T}_{i,K} \Big) \\ &= x^{(i)} p(C_k \mid x^{(i)}, \boldsymbol{\Theta}) * 1 \end{aligned}$$

The sum is 1 because any row of the $\mathbf{T}$ matrix is a one-hot encoding.
Plugging this result back into the original differential, we get

$$\begin{aligned} \frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}_k} &= -\sum_{i=1}^{N} \frac{\partial}{\partial \boldsymbol{\theta}_k} \Big(\sum_{j=1}^{K} \mathbf{T}_{i,j} \boldsymbol{\theta}_j x^{(i)} - \sum_{j=1}^{K} \mathbf{T}_{i,j} \log\left(\sum_{l=1}^{K} exp(\boldsymbol{\theta}_l x^{(i)}) \right) \Big) \\ &= -\sum_{i=1}^{N} \Big(\mathbf{T}_{i,k} x^{(i)} - x^{(i)} p(C_k \mid x^{(i)}, \boldsymbol{\Theta}) \Big) \\ &= -\sum_{i=1}^{N} \Big(\mathbf{T}_{i,k} - p(C_k \mid \mathbf{x}^{(i)}, \boldsymbol{\Theta}) \Big) \mathbf{x}^{(i)} \end{aligned}$$

(e) **[2 Points]** Write down the stochastic gradient descent update steps for an arbitrary θ_k using the i^{th} training example in terms of $\mathbf{x^{(i)}}$, $\mathbf{T}_{i,k}$ and $p\left(C_k \mid \mathbf{x}^{(i)}, \mathbf{\Theta}\right)$.

Solution

$$\begin{aligned}\boldsymbol{\theta}_k &= \boldsymbol{\theta}_k - \eta \frac{\partial J(\mathbf{\Theta})}{\partial \boldsymbol{\theta}_k} \\ &= \boldsymbol{\theta}_k + \eta \sum_{i=1}^{N} \Big(\mathbf{T}_{i,k} - p(C_k \mid \mathbf{x}^{(i)}, \mathbf{\Theta}) \Big) \mathbf{x}^{(i)}\end{aligned}$$

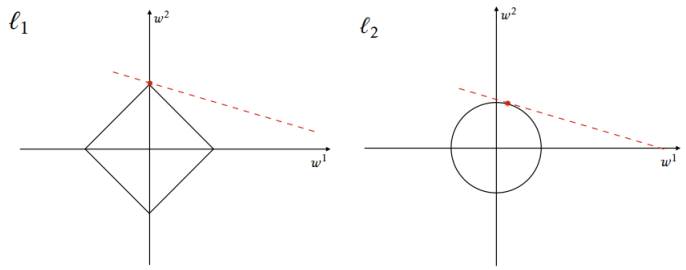
where η is the learning rate of the SGD algorithm.

(f) **[2 Points]** If you train mutinomial logistic regression for infinite iterations without $\ell_1 = |\mathbf{\Theta}|$ (sum of absolute values of all entries in the matrix) or $\ell_2 = ||\mathbf{\Theta}||_2$ (square root of sum of squares of all entries in the matrix) regularization, the weights can go to infinity. What is an intuitive explanation for this phenomenon? How does regularization help correct the problem?

Solution

If we are training with infinite iterations, we are essentially trying to learn all the nuances the training set has, including noise, outliers and subtle patterns that are specific to the training set. If there are noisy samples or samples that are outliers, they can cause the gradient to be very large and this can result in the weight update resulting in very large weights. From here on, the gradients for the new incoming samples can end up being large, resulting in larger weight updates and hence we would keep pushing the weights to infinity.

Regularization helps to solve this problem by allowing for gradient descent optimization while constraining the condition on the weights being learned. For example the l_2 regularization ensures that the sum of squares of all entries in $\mathbf{\Theta}$ is constrained [minimum] (along a sphere equal to $||\ \mathbf{\Theta}\ ||_2$) and when the descent update traverses the contour, it has to keep the weights on the constrained sphere, no matter how large the gradient is.



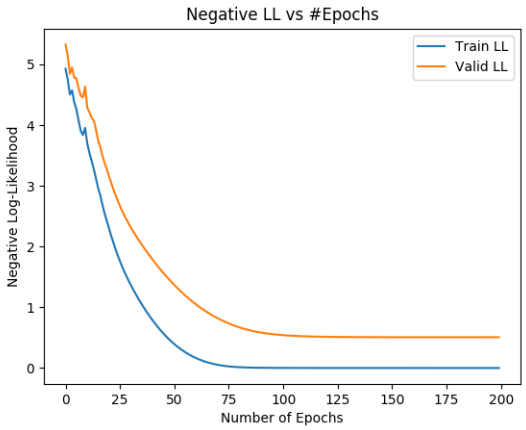
The figure above shows how for $W \in R^2$, we restrict the weights using l_1 and l_2 regularization, respectively.

1.2 Empirical Questions [8 pts]

The following questions should be completed as you work through the programming portion of this assignment (Section 2).

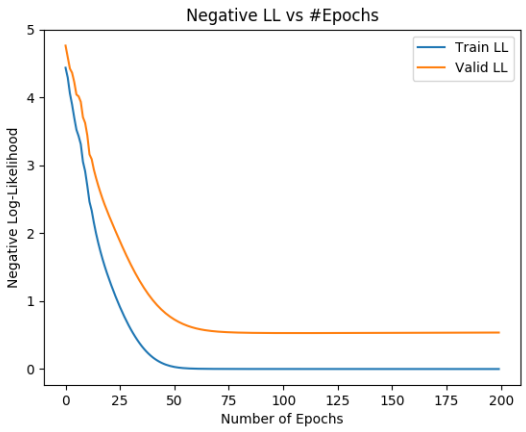
1. **Plots [2 Points]** For *Model 1*, using the data in the largedata folder in the handout, make a plot that shows the average negative log likelihood for the training and validation data sets after each of 200 epochs. The y-axis should show the negative log likelihood and the x-axis should show the number of epochs.

Solution



2. **Plots [2 Points]** For *Model 2*, make a plot as in the previous question.

Solution



3. **Explanation of Experiments [2 Points]** Write a few sentences explaining the output of the above experiments. In particular do the training and validation log likelihood curves look the same or different? Why?

Solution

We can see from the two plots above that Model 2 converges faster than Model 1. This is because we have more relevant features representing the training set (thresholding removes spurious features like 'a', 'the' - which just makes it take longer for the function to learn how to handle them) and hence we learn from them faster and more accurately.

The log likelihood curve of the training and validation sets look same. This is because we are modelling the validation set to not account for unknown words (statistics of the validation and training set are similar) by always referencing a known dictionary, and hence, when the system learns, the validation set behaves very similar to the training set in terms of negative log likelihood. Also, since the negative log likelihood of the training set is decreasing, better weights are being learned, and hence the validation set's negative log likelihood also decreases. The validation set however, does have examples that the model may not have seen before while training and hence we may not be able to get a perfect likelihood (negative log likelihood = 0) between the samples in the validation set and hence, even though the weights have converged and the average negative log likelihood is constant, it is never 0 like the training set.

4. **Results [2 Points]** Make a table with your train and test error for the large data set (found in the largedata folder in the handout) for each of the 2 models after running for 50 epochs.

Solution

	Train Error	Test Error
Model 1	0.15	0.32
Model 2	0.01416	0.1975

Table 1.1: "Large Data" Results

2 Programming [80 pts]

Your goal in this assignment is to implement a working Natural Language Processing (NLP) system, i.e., a sentiment polarity analyzer, using binary logistic regression. You will then use your algorithm to determine whether a review is positive or negative using movie reviews as data. You will do some very basic feature engineering, through which you are able to improve the learner's performance on this task. You will write two programs: `feature.{py|java|cpp|m}` and `lr.{py|java|cpp|m}` to jointly complete the task. The programs you write will be automatically graded using the Autolab system. You may write your programs in **Octave, Python, Java,** or **C++**. However, you should use the same language for all parts below.

2.1 The Tasks and Data Sets

Materials Download the tar file from Autolab ("Download handout"). The tar file will contain all the data that you will need in order to complete this assignment.

The handout contains data from the Movie Review Polarity data set (for more details, see http://www.cs.cornell.edu/people/pabo/movie-review-data/). Currently, the original data is distributed as a collection of separate files (one movie review per file). In the Autolab handout, we have converted this to a one line per example format consisting of the label 0 or 1 in the first column followed by all the words in the movie review (with none of the line breaks) in the second column.

Each data point consists of a label (0 for negatives and 1 for positives) and a attribute (a set of words as a whole). In the attribute, words are separated using white-space (punctuations are also separated with white-space). All characters are lowercased. No fancy pre-processing on the plain text is needed, because we have already done most of the work for you in the handout. We also provide a dictionary file (`dict.txt`) to limit the vocabulary to be considered in this assignments. Actually, this dictionary is constructed from the training data. Examples of the dictionary content are as follows, where the second column is the index of the word. Column one and column two are separated with white-space. Each line in `dict.txt` has the format: `word index\n`.

```
films 0
adapted 1
from 2
comic 3
```

Examples of the data are as follows.

```
1 david spade has a snide , sarcastic sense of humor that works ...
0 " mission to mars " is one of those annoying movies where , in ...
1 anyone who saw alan rickman's finely-realized performances in ...
1 ingredients : man with amnesia who wakes up wanted for murder , ...
1 ingredients : lost parrot trying to get home , friends synopsis : ...
1 note : some may consider portions of the following text to be ...
0 aspiring broadway composer robert ( aaron williams ) secretly ...
0 america's favorite homicidal plaything takes a wicked wife in " ...
```

We have provided you with two subsets of the movie review data set. Each data set is divided into a training, a validation, and a test data set. The small data set (`smalltrain_data.tsv`, `smallvalid_data.tsv`, and `smalltest_data.tsv`) can be used while debugging your code. We have included the reference

output files for this data set after **30 training epochs** (see directory `smalloutput/`). We have also included a larger data set (`train_data.tsv`, `valid_data.tsv`, `test_data.tsv`) with reference outputs for this data set after **60 training epochs** (see directory `largeoutput/`) . This data set can be used to ensure that your code runs fast enough to pass the autograder tests. Your code should be able to perform 60-epoch training and finish predictions through all of the data in around one minute for each of the models: one minute for Model 1 and one minute for Model 2.

The data files are in tab-separated-value (`.tsv`) format. This is identical to a comma-separated-value (`.csv`) format except that instead of separating columns with commas, we separate them with a tab character, `\t`. Each row is ended by a Unix style line ending, `\n`. The first column always contains the label and the second column the set of words: `label\tword1 word2 word3 ... wordN\n`.

2.2 Model Definition

Assume you are given a data set with N training examples and M features. We first write down the *negative* conditional log-likelihood of the training data in terms of the design matrix $\mathbf{X}$, the labels $\mathbf{y}$, and the parameter vector $\boldsymbol{\theta}$. This will be your objective function $J(\boldsymbol{\theta})$ for gradient descent. (Recall that ith row of the design matrix $\mathbf{X}$ contains the features $\mathbf{x}^{(i)}$ of the ith training example. The ith entry in the vector $\mathbf{y}$ is the label $y^{(i)}$ of the ith training example. Here we assume that each feature vector $\mathbf{x}^{(i)}$ contains a bias *feature*, e.g. $x_0^{(i)} = 1 \ \forall i \in \{1, \ldots, N\}$. As such, **the bias parameter is folded into our parameter vector $\boldsymbol{\theta}$.**

Taking $\mathbf{x}^{(i)}$ to be a $(K+1)$-dimensional vector where $x_0^{(i)} = 1$, the likelihood $p\left(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}\right)$ is:

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^{N} p(y^{(i)}|\mathbf{x}^{(i)}, \boldsymbol{\theta}) = \prod_{i=1}^{N} \left(\frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right)^{y^{(i)}} \left(\frac{1}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right)^{\left(1 - y^{(i)}\right)} \tag{2.1}$$

$$= \prod_{i=1}^{N} \frac{\left(e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right)^{y^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \tag{2.2}$$

Hence, the negative conditional log-likelihood is:

$$J(\boldsymbol{\theta}) = -\log p\left(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}\right) = \sum_{i=1}^{N} -y^{(i)} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)}\right) + \log\left(1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right) \tag{2.3}$$

The partial derivative of the negative log-likelihood $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}_j$, $j \in \{0, ..., M\}$ is:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} = -\sum_{i=1}^{N} \mathbf{x}_j^{(i)} \left[y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{2.4}$$

The gradient descent update rule for binary logistic regression for parameter element $\boldsymbol{\theta}_j$ is

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j - \eta \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} \tag{2.5}$$

Then, the stochastic gradient descent update for parameter element $\boldsymbol{\theta}_j$ using the ith datapoint $(\mathbf{x}^{(i)}, y^{(i)})$ is:

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j + \eta \mathbf{x}_j^{(i)} \left[y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{2.6}$$

2.3 Implementation

The implementation consists of two programs, a feature extraction program `feature.{py|java|cpp|m}` and a sentiment analyzer program `lr.{py|java|cpp|m}` using binary logistic regression. The programming pipeline is illustrated as follows.

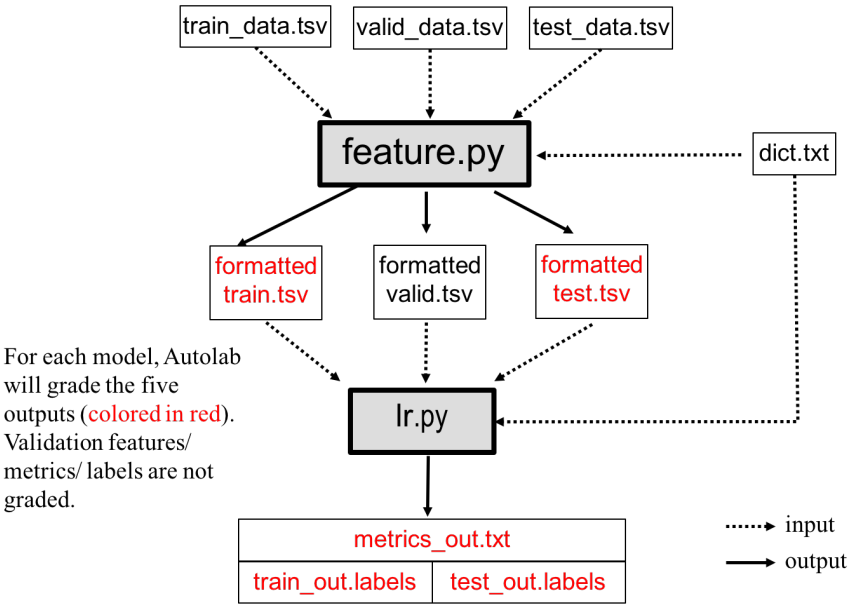


Figure 2.1: Programming pipeline for sentiment analyzer based on binary logistic regression

This first program is `feature.{py|java|cpp|m}`, that converts raw data (e.g., `train_data.tsv`, `valid_data.tsv`, and `test_data.tsv`) into formatted training, validation and test data based on the vocabulary information in the dictionary file `dict.txt`. To be specific, this program is to transfer the whole movie review text into a feature vector using some feature extraction methods. The formatted data sets should be stored in .tsv format. Details of formatted data sets will be introduced in Section 2.3.2 and Section 2.3.5.

The second program is `lr.{py|java|cpp|m}`, that implements a sentiment polarity analyzer using binary logistic regression. The file should learn the parameters of a binary logistic regression model that predicts a sentiment polarity (i.e. label) for the corresponding feature vector of each movie review. The program should output the labels of the training and test examples and calculate training and test error (percentage of incorrectly labeled reviews). As will be discussed later, efficient computation can be obtained with the help of the indexing information in the dictionary file `dict.txt`.

Your implementation must satisfy the following requirements:

- The `feature.{py|java|cpp|m}` must produce a sparse representation of the data using the label-index-value format {`label index[word1]:value1 index[word2]:value2...\n`}. We will use unseen data to test your feature output separately. (see Section 2.3.2 and Section 2.3.5 on feature engineering for details on how to do this).
- Ignore the words not in the vocabulary of `dict.txt` when the analyzer encounters one in the test or validation data.

- Set the trimming threshold to a constant $t = 4$ for Model 2 feature extraction (see Section 2.3.5).
- Initialize all model parameters to 0.
- Use stochastic gradient descent (SGD) to optimize the parameters for a binary logistic regression model. The number of times SGD loops through all of the training data (`num_epoch`) will be specified as a command line flag. Set your learning rate as a constant $\eta = 0.1$.
- Perform stochastic gradient descent updates on the training data **in the order that the data is given in the input file**. Although you would typically shuffle training examples when using stochastic gradient descent, in order to autograde the assignment, we ask that you **DO NOT** shuffle trials in this assignment.
- Be able to select which one of two feature extractions you will use in your logistic regression model using a command line flag (see Section 2.3.5)
- Do not hard-code any aspects of the data sets into your code. We will autograde your programs on multiple (hidden) data sets that include different attributes and output labels.

Careful planning will help you to correctly and concisely implement your program. Here are a few *hints* to get you started.

- Write a function that takes a single SGD step on the ith training example. Such a function should take as input the model parameters, the learning rate, and the features and label for the ith training example. It should update the model parameters in place by taking one stochastic gradient step.
- Write a function that takes in a set of features, labels, and model parameters and then outputs the error (percentage of labels incorrectly predicted). You can also write a separate function that takes the same inputs and outputs the negative log-likelihood of the regression model.

2.3.1 Command Line Arguments

The autograder runs and evaluates the output from the files generated, using the following command (note `feature` will be run before `lr`):

```
For Python: $ python feature.py [args1...]
            $ python lr.py [args2...]
For Java:   $ java feature.java [args1...]
            $ java lr.java [args2...]
For C++:    $ g++ feature.cpp ./a.out [args1...]
            $ g++ lr.cpp ./a.out [args2...]
For Octave: $ octave -qH feature.m [args1...]
            $ octave -qH lr.m [args2...]
```

Where above `[args1...]` is a placeholder for eight command-line arguments:`<train_input>` `<validation_input>` `<test_input>` `<dict_input>` `<formatted_train_out>` `<formatted_validation_out>` `<formatted_test_out>` `<feature_flag>`. These arguments are described in detail below:

1. `<train_input>`: path to the training input `.tsv` file (see Section 2.1)
2. `<validation_input>`: path to the validation input `.tsv` file (see Section 2.1)
3. `<test_input>`: path to the test input `.tsv` file (see Section 2.1)

4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2.1)
5. `<formatted_train_out>`: path to output `.tsv` file to which the feature extractions on the *training* data should be written (see Section 2.3.2)
6. `<formatted_validation_out>`: path to output `.tsv` file to which the feature extractions on the *validation* data should be written (see Section 2.3.2)
7. `<formatted_test_out>`: path to output `.tsv` file to which the feature extractions on the *test* data should be written (see Section 2.3.2)
8. `<feature_flag>`: integer taking value 1 or 2 that specifies whether to construct the Model 1 feature set or the Model 2 feature set (see Section 2.3.5)—that is, if `feature_flag`==1 use Model 1 features; if `feature_flag`==2 use Model 2 features

On the other hand, `[args2...]` is a placeholder for eight command-line arguments:`<formatted_train_input> <formatted_validation_input> <formatted_test_input> <dict_input> <train_out> <test_out> <metrics_out> <num_epoch>`. These arguments are described in detail below:

1. `<formatted_train_input>`: path to the formatted training input `.tsv` file (see Section 2.3.2)
2. `<formatted_validation_input>`: path to the formatted validation input `.tsv` file (see Section 2.3.2)
3. `<formatted_test_input>`: path to the formatted test input `.tsv` file (see Section 2.3.2)
4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2.1)
5. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 2.3.3)
6. `<test_out>`: path to output `.labels` file to which the prediction on the *test* data should be written (see Section 2.3.3)
7. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and test error should be written (see Section 2.3.4)
8. `<num_epoch>`: integer specifying the number of times SGD loops through all of the training data (e.g., if `<num_epoch>` equals 5, then each training example will be used in SGD 5 times).

As an example, if you implemented your program in Python, the following two command lines would run your programs on the data provided in the handout for 60 epochs using the features from Model 1.

```
$ python feature.py train_data.tsv valid_data.tsv test_data.tsv \
dict.txt formatted_train.tsv formatted_valid.tsv formatted_test.tsv 1

$ python lr.py formatted_train.tsv formatted_valid.tsv formatted_test\
.tsv dict.txt train_out.labels test_out.labels metrics_out.txt 60
```

Important Note: You will not be writing out the predictions on validation data, only on train and test data. The validation data is *only* used to give you an estimate of held-out negative log-likelihood at the end of each epoch during training.[a]

[a]For this assignment, we will always specify the number of epochs. However, a more mature implementation would monitor the performance on validation data at the end of each epoch and stop SGD when this validation log-likelihood appears to have converged. You should ***not*** implement such a convergence check for this assignment.

2.3.2 Output: Formatted Data Files

Your `feature` program should write three output `.tsv` files converting original data to formatted data on `<formatted_train_out>`, `<formatted_valid_out>`, and `<formatted_test_out>`. Each should contain the formatted presentation for each example printed on a new line. Use `\n` to create a new line. The format for each line should exactly match

label`\t`index[word1]:value1`\t`index[word2]:value2`\t`...index[wordM]:valueM`\n`

Where above, the first column is label, and the rest are "index[word]:value" feature elements. index[word] is the index of the word in the dictionary, and value is the value of this feature (in this assignment, the value is one or zero). There is a colon, `:`, between index[word] and corresponding value. Columns are separated using a table character, `\t`. The handout contains example `<formatted_train_out>`, `<formatted_valid_out>`, and `<formatted_test_out>` for your reference.

The formatted output will be checked separately by the autograder by running your `feature` program on some unseen data sets and evaluating your output file against the reference formatted files. Examples of content of formatted output file are given below.

```
0       2915:1  21514:1 166:1   32:1    10699:1 305:1   ...
0       7723:1  51:1    8701:1  74:1    370:1   8:1    ...
1       229:1   48:1    326:1   43:1    576:1   55:1    ...
1       8126:1  1349:1  58:1    4709:1  48:1    8319:1  ...
```

2.3.3 Output: Labels Files

Your `lr` program should produce two output `.labels` files containing the predictions of your model on training data (`<train_out>`) and test data (`<test_out>`). Each should contain the predicted labels for each example printed on a new line. Use `\n` to create a new line.

Your labels should exactly match those of a reference implementation – this will be checked by the autograder by running your program and evaluating your output file against the reference solution. Examples of the content of the output file are given below.

```
0
0
1
0
```

2.3.4 Output Metrics

Generate a file where you report the following metrics:

error After the final epoch (i.e. when training has completed fully), report the final training error `error(train)` and test error `error(test)`.

All of your reported numbers should be within 0.01 of the reference solution. The following is the reference solution for large data set with Model 1 feature structure after 60 training epochs. See `model1_metrics_out.txt` in the handout.

```
error(train): 0.074167
error(test): 0.247500
```

Take care that your output has the exact same format as shown above. Each line should be terminated by a Unix line ending `\n`. There is a whitespace character after the colon.

2.3.5 Feature Engineering

Your implementation of `feature.{py|java|cpp|m}` should have an input argument `<feature_flag>` that specifies one of two types of feature extraction structures that should be used by the logistic regression model. The two structures are illustrated below as probabilities of the labels given the inputs.

Model 1 $p(y^{(i)} \mid \mathbf{1_{occur}}(\mathbf{x^{(i)}}, \mathbf{Vocab}), \boldsymbol{\theta})$: This model defines a probability distribution over the current label $y^{(i)}$ using the parameters $\boldsymbol{\theta}$ and a *bag-of-word* feature vector $\mathbf{1_{occur}}(\mathbf{x^{(i)}}, \mathbf{Vocab})$ indicating which word in vocabulary $\mathbf{Vocab}$ of the dictionary occurs at least once in the movie review example $\mathbf{x}^{(i)}$. The entry in the indicator vector associated to the occurring word will set to one (otherwise, it is zero). This bag-of-word model should be used when `<feature_flag>` is set to 1.

Model 2 $p(y^{(i)} \mid \mathbf{1_{trim}}(\mathbf{x^{(i)}}, \mathbf{Vocab}, t), \boldsymbol{\theta})$: This model defines a probability distribution over the current label $y^{(i)}$ using the parameters $\boldsymbol{\theta}$ and a *trimmed* bag-of-word feature vector $\mathbf{1_{trim}}(\mathbf{x^{(i)}}, \mathbf{Vocab}, t)$ indicating (1) which word in vocabulary $\mathbf{Vocab}$ of the dictionary occurs in the movie review example $\mathbf{x}^{(i)}$, AND (2) the *count of the word* is LESS THAN ($<$) threshold t. The entry in the indicator vector associated to the word that satisfies both conditions will set to one (otherwise, it is zero, including no shown and high-frequent words). This trimmed bag-of-word model should be used when `<feature_flag>` is set to 2. In this assignment, use the constant trimming threshold $t = 4$.

The motivation of Model 2 is that keywords that truly represent the sentiment may not occur too frequently, this trimming strategy can make the feature presentation cleaner by removing highly repetitive words that are useless and neutral, such "the", "a", "to", etc. You will observe whether this basic and heuristic strategy based on this intuition will bring in performance improvement.

Note that above $\mathbf{1_{occur}}$ and $\mathbf{1_{trim}}$ are described as a dense feature representation as showed in Tables A.2 for illustration purpose. In your implementation, you should further convert it to the representation in A.3 for Model 1 and the representation in A.5 for Model 2, such that the formatted data outputs match Section 2.3.2.

2.3.6 Evaluation

Autolab will test your implementations on hidden data sets with the same format as the two data sets provided in the handout. `feature` program and `lr` program will be tested separately. To ensure that your code can pass the autolab tests in under 5 minutes (the maximum time length) be sure that your code can complete 60-epoch training and finish predictions through all of the data in the `largedata` folder in around one minute for each of the models.

2.4 Autolab Submission

You must submit a .tar file named `lr.tar` containing `feature.{py|m|java|cpp}` and `lr.{py|m|java|cpp}`. You can create that file by running:

```
tar -cvf lr.tar feature.{py|m|java|cpp} lr.{py|m|java|cpp}
```

from the directory containing your code.

Some additional tips: **DO NOT** compress your files; you are just creating a tarball. Do not use tar `-czvf`. **DO NOT** put the above files in a folder and then tar the folder. Autolab is case sensitive, so observe that all your files should be named in **lowercase**. You must submit this file to the corresponding homework link on Autolab. The autograder for Autolab prints out some additional information about the tests that it ran. You can view this output by selecting "Handin History" from the menu and then clicking one of the scores you received for a submission. For example on this assignment, among other things, the autograder will print out which language it detects (e.g. Python, Octave, C++, Java). **It is recommended that you create a new empty folder somewhere else, copy your implementation files there, and create tarball from there. This can ensure a clean submission without tarring unnecessary files.**

Python3 Users: Please include a blank file called python3.txt (case-sensitive) in your tar submission and we will execute your submitted program using Python 3 instead of Python 2.7.

Note: For this assignment, you may make up to 10 submissions to Autolab before the deadline, but only your last submission will be graded.

3 Collaboration Questions

Please answer the following:

After you have completed all other components of this assignment, report your answers to the collaboration policy questions detailed in the Academic Integrity Policies found here.

1. Did you receive any help whatsoever from anyone in solving this assignment? Is so, include full details.
2. Did you give any help whatsoever to anyone in solving this assignment? Is so, include full details.
3. Did you find or come across code that implements any part of this assignment ? If so, include full details.

Solution

A Implementation Details for Logistic Regression

A.1 Examples of Features

Here we provide examples of the features constructed by Model 1 and Model 2. Table A.1 shows an example input file, where column i indexes the ith movie review example. Rather than working directly with this input file, you should transform from the sentiment/text representation into a label/feature vector representation.

Table A.2 shows the dense occurrence-indicator representation expected for Model 1. The size of each feature vector (i.e. number of feature columns in the table) is equal to the size of the entire vocabulary of words stored in the given `dict.txt` (this dictionary is actually constructed from the same training data in `largeset`). Each row corresponds to a single example, which we have indexed by i.

It would be *highly impractical* to actually store your feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^M$ in the dense representation shown in Table A.2 which takes $O(M)$ space per vector (M is around 40 thousands for the dictionary). This is because the features are extremely sparse: for the second example ($i = 2$), only three of the features is non-zero for Model 1 and only two for Model 2. As such, we now consider a sparse representation of the features that will save both memory and computation.

Table A.3 shows the sparse representation (bag-of-word representation) of the feature vectors. Each feature vector is now represented by a map from the index of the feature (e.g. index`["apple"]`) to its value which is 1. The space savings comes from the fact that we can omit from the map any feature whose value is zero. In this way, the map only contains *non-zero entry* for each Model 1 feature vector.

Using the same sparse representation of features, we present an example of the features used by Model 2. This involves two step: (1) construct the count-of-word representation of the feature vector (see Table A.4); (2) trim/remove the highly repetitive words/features and set the value of all remaining features to one (see Table A.5).

A.2 Efficient Computation of the Dot-Product

In simple linear models like logistic regression, the computation is often dominated by the dot-product $\boldsymbol{\theta}^T \mathbf{x}$ of the parameters $\boldsymbol{\theta} \in \mathbb{R}^M$ with the feature vector $\mathbf{x} \in \mathbb{R}^M$. When a dense representation of $\mathbf{x}$ (such as that shown in Table A.2) is used, this dot-product requires $O(M)$ computation. Why? Because the dot-product requires a sum over each entry in the vector:

$$\boldsymbol{\theta}^T \mathbf{x} = \sum_{m=1}^{M} \theta_m x_m \tag{A.1}$$

However, if our feature vector is represented sparsely, we can observe that the only elements of the feature vector that will contribute a non-zero value to the sum are those where $x_m \neq 0$, since this would allow $\theta_m x_m$ to be nonzero. As such, we can write the dot-product as below:

$$\boldsymbol{\theta}^T \mathbf{x} = \sum_{m \in \{1,\ldots,M\} \text{ s.t. } x_m \neq 0} \theta_m x_m \tag{A.2}$$

This requires only computation proportional to the number of non-zero entries in $\mathbf{x}$, which is generally very small for Model 1 and Model 2 compared to the size of the vocabulary. To ensure that your code runs quickly it is best to write the dot-product in the latter form (Equation (A.2)).

A.3 Data Structures for Fast Dot-Product

Lastly, there is a question of how to implement this dot-product efficiently in practice. The key is choosing appropriate data structures. The most common approach is to choose a dense representation for $\boldsymbol{\theta}$. In C++ or Java, you could choose an array of `float` or `double`. In Python, you could choose a `numpy` array or a list.

To represent your feature vectors, you might need multiple data structures. First, you could create a shared mapping from a feature name (e.g. `apple` or `boy`) to the corresponding index in the dense parameter vector. This shared mapping has already been provided to you in the `dict.txt`, and you can extract the index of the word from the dictionary file for all later computation. In fact, you should be able to construct the dictionary on your own from the training data (we have done this step for you in the handout). Once you know the size of this mapping (which is the size of the dictionary file), you know the size of the parameter vector $\boldsymbol{\theta}$.

Another data structure should be used to represent the feature vectors themselves. This assignment use the option to directly store a mapping from the integer index in the dictionary mapping (i.e. the index m) to the value of the feature x_m. Only the indexs of words satisfying certain conditions will be stored, and all other indexs are implies to have zero value of the feature x_m. This structure option will ensure that your code runs fast so long as you are doing an efficient computation instead of the $O(M)$ version.

Note on out-of-vocabulary features The dictionary in the handout is made from the same training data in the large data set. You may encounter some words in the validation data and the test data that do not appear in the vocabulary mapping. In this assignment, you should ignore those words during prediction and evaluation.

example index i	**sentiment** $y^{(i)}$	**review text** $\mathbf{x}^{(i)}$
1	pos	apple boy , cat dog
2	pos	boy boy : dog dog ; dog dog . dog egg egg
3	neg	apple apple apple apple boy cat cat dog
4	neg	egg fish

Table A.1: Abstract representation of the input file format. The ith row of this file will be used to construct the ith training example using either Model 1 features (Table A.3) or Model 2 features (Table A.5).

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$											
		zoo	...	apple	boy	cat	dog	egg	fish	girl	head	...	zero
1	1	0	...	1	1	1	1	0	0	0	0	...	0
2	1	0	...	0	1	0	1	1	0	0	0	...	0
3	0	0	...	1	1	1	1	0	0	0	0	...	0
4	0	0	...	0	0	0	0	1	1	0	0	...	0

Table A.2: Dense feature representation for Model 1 corresponding to the input file in Table A.1. The ith row corresponds to the ith training example. Each dense feature has the size of the vocabulary in the dictionary. Punctuations are excluded.

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$
1	1	{ index[“apple”]: 1, index[“boy”]: 1, index[“cat”]: 1, index[“dog”]: 1 }
2	1	{ index[“boy”]: 1, index[“dog”]: 1, index[“egg”]: 1 }
3	0	{ index[“apple”]: 1, index[“boy”]: 1, index[“cat”]: 1, index[“dog”]:1 }
4	0	{ index[“egg”]: 1, index[“fish”]: 1 }

Table A.3: Sparse feature representation (bag-of-word representation) for Model 1 corresponding to the input file in Table A.1.

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$
1	1	{ index[“apple”]: 1, index[“boy”]: 1, index[“cat”]: 1, index[“dog”]: 1 }
2	1	{ index[“boy”]: 2, index[“dog”]: 5, index[“egg”]: 2 }
3	0	{ index[“apple”]: 4, index[“boy”]: 1, index[“cat”]: 2, index[“dog”]: 1 }
4	0	{ index[“egg”]: 1, index[“fish”]: 1 }

Table A.4: Count of word representation for Model 2 corresponding to the input file in Table A.1.

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$
1	1	{ index[“apple”]: 1, index[“boy”]: 1, index[“cat”]: 1, index[“dog”]: 1 }
2	1	{ index[“boy”]: 1, index[“egg”]:1 }
3	0	{ index[“boy”]: 1, index[“cat”]: 1, index[“dog”]: 1 }
4	0	{ index[“egg”]: 1, index[“fish”]: 1 }

Table A.5: Sparse feature representation for Model 2 corresponding to the input file in Table A.1. Assume that the trimming threshold is 4. As a result, ”dog” in example 2 and ”apple” in example 3 are removed and the value of all remaining features are reset to value 1.