HOMEWORK 4: LOGISTIC REGRESSION

10-301/10-601 Introduction to Machine Learning (Fall 2020)
https://www.cs.cmu.edu/~10601/
DUE: Wednesday, October 14, 2020 11:59 PM

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 the Written component, you will warm up by deriving stochastic gradient descent updates for binary and multinomial logistic regression. Then in the Programming component, you will implement a binary logistic regression model as the core of your natural language processing system.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: https://www.cs.cmu.edu/~10601
- **Late Submission Policy:** See the late submission policy here: https://www.cs.cmu.edu/~10601
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to Gradescope.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, we will be using Gradescope (https://gradescope.com/). Please use the provided template. Submissions must be written in LaTeX. Regrade requests can be made, however this gives the staff the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted. Each derivation/proof should be completed in the boxes provided. For short answer questions you **should not** include your work in your solution. If you include your work in your solutions, your assignment may not be graded correctly by our AI assisted grader.
  - **Programming:** You will submit your code for programming questions on the homework to Gradescope (https://gradescope.com). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). When you are developing, check that the version number of the programming language environment (e.g. Python 3.6.9, OpenJDK 11.0.5, g++ 7.4.0) and versions of permitted libraries (e.g. `numpy` 1.17.0 and `scipy` 1.4.1) match those used on Gradescope. You have unlimited Gradescope programming submissions. However, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly first before submitting you code to Gradescope.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the writeup and template on Piazza.

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] or ND4J[b] and C++ users Eigen[c]. Details below. (As usual, Python users have NumPy.)

EJML for Java EJML is a pure Java linear algebra package with three interfaces. We strongly recommend using the SimpleMatrix interface. The autograder will use EJML version 0.38. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.38-libs/*:linalg_lib/nd4j-v1.0.0-beta7-libs/*:./"` to ensure that all the EJML jars are on the classpath as well as your code.

ND4J for Java ND4J is a library for multidimensional tensors with an interface akin to Python's NumPy. The autograder will use ND4J version 1.0.0-beta7. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.38-libs/*:linalg_lib/nd4j-v1.0.0-beta7-libs/*:./"` to ensure that all the ND4J jars are on the classpath as well as your code.

Eigen for C++ Eigen is a header-only library, so there is no linking to worry about—just `#include` whatever components you need. The autograder will use Eigen version 3.3.7. The command line arguments above demonstrate how we will call you code. When compiling your code we will include, the argument `-I./linalg_lib` in order to include the `linalg_lib/Eigen` subdirectory, which contains all the headers.

We have included the correct versions of EJML/ND4J/Eigen in the `linalg_lib.zip` posted on the Piazza Resources page for your convenience. It contains the same `linalg_lib/` directory that we will include in the current working directory when running your tests. Do **not** include EJML, ND4J, or Eigen in your homework submission; the autograder will ensure that they are in place.

[a] https://ejml.org
[b] https://deeplearning4j.org/docs/latest/nd4j-overview
[c] http://eigen.tuxfamily.org/

Written Questions (30 points)

1. Logistic Regression and Stochastic Gradient Descent

(a) (1 point) Which of the following are true about logistic regression?

Select all that apply:

☐ Our formulation of binary logistic regression will work with both continuous and binary features.

☐ Binary Logistic Regression will form a linear decision boundary in our feature space.

☐ The function $\sigma(x) = \frac{1}{1+e^{-x}}$ is convex.

☐ The negative log likelihood function for logistic regression. $-\frac{1}{N}\sum_{i=1}^{N}\log(\sigma(\mathbf{x}^{(i)}))$ is non-convex so gradient descent may get stuck in a sub-optimal local minimum.

☐ None of above.

(b) (1 point) The negative log-likelihood $J(\boldsymbol{\theta})$ for binary logistic regression can be expressed as

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} -y^{(i)}\left(\boldsymbol{\theta}^T\mathbf{x}^{(i)}\right) + \log\left(1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})\right)$$

where $\mathbf{x}^{(i)} \in \mathbb{R}^{M+1}$ is the column vector of the feature values of ith data point, $y^{(i)} \in \{0,1\}$ is the i-th class label, $\boldsymbol{\theta} \in \mathbb{R}^{M+1}$ is the weight vector. When we want to perform logistic ridge regression (i.e. with ℓ_2 regularization), we modify our objective function to be

$$f(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda\frac{1}{2}\sum_{j=0}^{M}\theta_j^2$$

where λ is the regularization weight, θ_j is the jth element in the weight vector $\boldsymbol{\theta}$. Suppose we are updating θ_k with learning rate α, which of the following is the correct expression for the update?

Select one:

◯ $\theta_k \leftarrow \theta_k + \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(y^{(i)} - \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) + \lambda\theta_k$

◯ $\theta_k \leftarrow \theta_k + \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(-y^{(i)} + \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) - \lambda\theta_k$

◯ $\theta_k \leftarrow \theta_k - \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(-y^{(i)} + \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) + \lambda\theta_k$

◯ $\theta_k \leftarrow \theta_k - \alpha\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k}$ where $\frac{\partial f(\boldsymbol{\theta})}{\partial\theta_k} = \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}\left(-y^{(i)} - \frac{\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x}^{(i)})}\right) + \lambda\theta_k$

2. Binary Logistic Regression on a Small Dataset

The following questions should be completed before you start the programming component of this assignment.

The following dataset consists of 4 training examples, where $x_k^{(i)}$ denotes the k-th dimension of the i-th training example, and the corresponding label $y^{(i)}$. $k \in \{1, 2, 3, 4, 5\}$ and $i \in \{1, 2, 3, 4\}$

i	x_1	x_2	x_3	x_4	x_5	y
1	0	0	1	0	1	0
2	0	1	0	0	0	1
3	0	1	1	0	0	1
4	1	0	0	1	0	0

A binary logistic regression model is trained on this data. After n iterations, the parameter vector $\theta = [1.5, 2, 1, 2, 3]^T$. *Note*: There is no bias term used in this problem.

Use the data above to answer the following questions. For all numerical answers, please use one number rounded to the fourth decimal place, e.g., 0.1234.

Showing your work in these questions is optional, but it is recommended to help us understand where any misconceptions may occur.

(a) (1 point) Calculate $J(\boldsymbol{\theta})$, $\frac{1}{N}$ times the negative log-likelihood over the given data after iteration n (Note here we are using natural log, i.e., the base is e).

$J(\boldsymbol{\theta})$

Work

(b) (2 points) Calculate the gradients $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j}$ with respect to θ_j, for all $j \in \{1, 2, 3, 4, 5\}$

$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_1$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_2$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_3$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_4$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_5$

Work

(c) (1 point) Update the parameters following the parameter update step $\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j - \alpha \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j}$ and give the final (numerical) value of the vector θ. use learning rate $\alpha = 1$.

θ_1	θ_2	θ_3	θ_4	θ_5

Work

(d) (2 points) The following table shows the sparse feature representation for the given data

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$
1	0	$\{x_3 : 1, x_5 : 1\}$
2	1	$\{x_2 : 1\}$
3	1	$\{x_2 : 1, x_3 : 1\}$
4	0	$\{x_1 : 1, x_4 : 1\}$

Calculate the probability $p\left(\mathbf{y}|\mathbf{X} = \mathbf{x}^{(\mathbf{3})}, \boldsymbol{\theta}\right)$ after the update step in (c), using **only k unique multiplication operations** where k is the number of non-zero features in $\mathbf{x}^{(\mathbf{3})}$. Explicitly show these multiplication operations.

Answer including work showing multiplication operations

3. Multinomial Logistic Regression

Multinomial logistic regression, also known as softmax regression or 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 possible classes, which is usually greater than two to be interesting.

(a) (1 point) 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. Which of the following are the possible ways to determine the class for each unlabelled data point $\mathbf{x}^{(i)}$? No need to worry about tie breaking.

Select all that apply:

☐ Train K classifiers. Each trained classifier $h_k(\mathbf{x}^{(i)})$ will determine if a point $\mathbf{x}^{(i)}$ is in class k or not for $k = 1, \ldots, K$. The class that has the highest probability from the K classifiers will be the predicted label of point $\mathbf{x}^{(i)}$

☐ Train K classifiers. Each trained classifier $h_k(\mathbf{x}^{(i)})$ will determine if a point $\mathbf{x}^{(i)}$ is in class k or not for $k = 1, \ldots, K$. The class that has the highest 'confidence score' from the K classifiers will be the predicted label of point $\mathbf{x}^{(i)}$, where a 'confidence score' of classifier k is defined as $\mathbf{w}_k^T \mathbf{x}^{(i)} + b_k$

☐ Train $\frac{K(K-1)}{2}$ classifiers. Each trained classifier $h_{k,j}(\mathbf{x}^{(i)})$ will determine if a point $\mathbf{x}^{(i)}$ is in class k or class j for all unique combinations of k and j, where $k \neq j$, i.e. for $k \in \{1, \ldots, K\}$ and $j \in \{k+1, \ldots, K\}$. The class label that is predicted the most number of times by all classifiers is assigned to point $\mathbf{x}^{(i)}$

☐ None of the above

(b) (2 points) Now we would like a method to do multiclass classification by training a single classifier that considers all classes simultaneously. Multinomial logistic regression is such a method. Remember that in multinomial logistic regression, the probability of random variable Y taking on class value k given the input data point $\mathbf{x}$ is:

$$P(Y = k \mid \mathbf{x}, \boldsymbol{\Theta}) = \frac{\exp(\boldsymbol{\Theta}_k \mathbf{x})}{\sum_{q=1}^{K} \exp(\boldsymbol{\Theta}_q \mathbf{x})} = \text{softmax}(\boldsymbol{\Theta}\mathbf{x})_k \tag{1}$$

where $\boldsymbol{\Theta}$ is the parameter matrix of size $K \times (M+1)$, and $\boldsymbol{\Theta}_k$ denotes the k-th **row** of $\boldsymbol{\Theta}$, which is the parameter vector for the k-th class. Assume that we have folded the bias term into $\boldsymbol{\Theta}$, so we have $\mathbf{x}^{(i)} \in \mathbb{R}^{M+1}$.

The likelihood that the i-th training label takes on the observed label $y^{(i)}$ given the i-th input data point is:

$$P\left(Y^{(i)} = y^{(i)} \mid \mathbf{x}^{(i)}, \boldsymbol{\Theta}\right) = \frac{\exp\left(\boldsymbol{\Theta}_{y^{(i)}} \mathbf{x}^{(i)}\right)}{\sum_{j=1}^{K} \exp\left(\boldsymbol{\Theta}_j \mathbf{x}^{(i)}\right)} = \text{softmax}(\boldsymbol{\Theta}\mathbf{x}^{(i)})_{y^{(i)}} \tag{2}$$

Additionally, we assume that the $Y^{(i)}$ are independent and identically distributed, so the conditional likelihood may be written as a product over all data points:

$$\mathcal{L}(\mathbf{\Theta}; \mathbf{y}, \mathbf{X}) = p(y^{(1)}, \cdots, y^{(N)} \mid \mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(N)}, \mathbf{\Theta}) = \prod_{i=1}^{N} P\left(Y^{(i)} = y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{\Theta}\right) \quad (3)$$

Recall from binary logistic regression that we considered both $P(Y = 1 \mid \mathbf{x})$ and $P(Y = 0 \mid \mathbf{x})$ cases in the conditional likelihood formulation by raising each factor to the appropriate power of $y^{(i)}$:

$$\mathcal{L}_{\text{binary}}(\boldsymbol{\theta}; \mathbf{y}, \mathbf{X}) = \prod_{i=1}^{N} p\left(1 \mid \mathbf{x}^{(i)}, \boldsymbol{\theta}\right)^{y^{(i)}} P\left(0 \mid \mathbf{x}^{(i)}, \boldsymbol{\theta}\right)^{\left(1-y^{(i)}\right)} \quad (4)$$

This won't quite work with multinomial logistic regression because $y^{(i)} \in \{1 \cdots K\}$ rather than $\{0, 1\}$. So, we will make use of the indicator function $\mathbb{I}(y^{(i)} = k)$ to generate a 1 when the label of the i-th point is class k and a 0 otherwise. For convenience (and to avoid indicator function notation everywhere), we can convert all of the label data $y^{(i)}$ into a binary matrix $\mathbf{T}$ of size $N \times K$, where the i, j entry of $\mathbf{T}$ is $T_{ij} = \mathbb{I}(y^{(i)} = j)$. Note that each row of $\mathbf{T}$ will have exactly one 1.

Now, write the conditional likelihood $\mathcal{L}(\mathbf{\Theta}; \mathbf{T}, \mathbf{X})$ from Equation 3 in terms of N, K, T_{ij} and $p\left(j \mid \mathbf{x}^{(i)}, \mathbf{\Theta}\right)$.

Your Answer

(c) (2 points) Write the objective function, $J(\mathbf{\Theta})$, as $\frac{1}{N}$ times the *negative* conditional log-likelihood in terms of N, K, T_{ij} and $p\left(j \mid x^{(i)}, \mathbf{\Theta}\right)$. This objective function is also known as cross-entropy loss. To help you with the next part, write down the objective function after replacing $p\left(j \mid x^{(i)}, \mathbf{\Theta}\right)$ using equation 2 given in the previous part. Do not include the literal term "softmax" in your answer.

Your Answer

(d) (5 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:

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\Theta}_k} = -\frac{1}{N} \sum_{i=1}^{N} \left(T_{ik} - p\left(k \mid \mathbf{x}^{(i)}, \boldsymbol{\Theta} \right) \right) \mathbf{x}^{(i)}$$

Show all steps of the derivation. (**Hint:** A good first step would be to simplify your answer from part c as much as you can, if you haven't already done so in the previous part)

Your Answer

(e) (2 points) Write pseudocode corresponding to the stochastic gradient descent **update steps** for an arbitrary $\mathbf{\Theta}_k$ for $k = 1, \ldots, K$, using the i-th training example in terms of $\mathbf{x}^{(i)}$, T_{ik}, $p\left(k \mid \mathbf{x}^{(i)}, \mathbf{\Theta}\right)$, and learning rate α.

Hint: Recall the buggy SGD program from lecture.

Your Answer

(f) (1 point) If you train multinomial logistic regression for infinite iterations without ℓ_1 regularization (sum of absolute values of all entries in the matrix) or ℓ_2 (square root of sum of squares of all entries in the matrix) regularization, the weights can go to infinity in magnitude. What is an explanation for this phenomenon? (**Hint**: Think about what happens to the probabilities if we train an unregularized logistic regression, and the role of the weights when calculating such probabilities)

Your Answer

(g) (1 point) How does regularization such as ℓ_1 and ℓ_2 help correct the problem?

Select all that apply:

☐ ℓ_1 regularization prevents weights from going to infinity by penalizing the count of non-zero weights.

☐ ℓ_1 regularization prevents weights from going to infinity by reducing some of the weights to 0, effectively removing some of the features.

☐ ℓ_2 regularization prevents weights from going to infinity by reducing the value of some of the weights to *close* to 0 (reducing the effect of a feature but not necessarily removing it).

☐ None of the above.

4. Programming Empirical Questions

The following questions should be completed as you work through the programming component of this assignment.

(a) (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.

Your Answer

(b) (2 points) For *Model 2*, make a plot as in the previous question.

Your Answer

(c) (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?

Your Answer

(d) (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. Please use one number rounded to the fourth decimal place, e.g., 0.1234.

Your Answer

	Train Error	Test Error
Model 1	?	?
Model 2	?	?

Table 1: "Large Data" Results

Collaboration Questions

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.

Your Answer

Programming (70 points)

1 The Task

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}` and `lr.{py|java|cpp}` to jointly complete the task. The programs you write will be automatically graded using Gradescope. You may write your programs in **Python, Java,** or **C++**. However, you should use the same language for all parts below.

Note: Before starting the programming, you should work through the written component to get a good understanding of important concepts that are useful for this programming component.

2 The Datasets

Datasets Download the zip file from Piazza, which contains all the data that you will need in order to complete this assignment. The handout contains data from the Movie Review Polarity dataset. [1] In the data files, each line is a data point that consists of a label (0 for negatives and 1 for positives) and a attribute (a set of words as a whole). The label and attribute are separated by a tab.[2] In the attribute, words are separated using white-space (punctuations are also separated with white-space). All characters are lowercased. The format of each data point (each line) is `label\tword1 word2 word3 ... wordN\n`.

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 dataset. Each dataset is divided into a training, a validation, and a test dataset.

The small dataset (`smalldata/train_data.tsv`, `valid_data.tsv`, and `test_data.tsv`) can be used while debugging your code. We have included the reference output files for this dataset after **30 training epochs** (see directory `smalloutput/`). We have also included a larger dataset (`largedata/train_data.tsv`, `valid_data.tsv`, `test_data.tsv`) with reference outputs for this dataset after **60 training epochs** (see directory `largeoutput/`) . This dataset 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 less than one minute for each of the models: one minute for Model 1 and one minute for Model 2.

[1] for more details, see http://www.cs.cornell.edu/people/pabo/movie-review-data/

[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`

Dictionary We also provide a dictionary file (`dict.txt`) to limit the vocabulary to be considered in this assignment (this dictionary is constructed from the training data, so it includes all the words from the training data, but some words in validation and test data may not be present in the dictionary). Each line in the dictionary file is in the following format: `word\tindex\n`. Words (column 1) and indexes (column 2) are separated with whitespace. Examples of the dictionary content are as follows:

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

3 Model Definition

Assume you are given a dataset with N training examples and M features. We first write down the $\frac{1}{N}$ times 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 i-th row of the design matrix $\mathbf{X}$ contains the features $\mathbf{x}^{(i)}$ of the i-th training example. The i-th entry in the vector $\mathbf{y}$ is the label $y^{(i)}$ of the i-th 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 $(M+1)$-dimensional vector where $x_0^{(i)} = 1$, the likelihood $p\left(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}\right)$ is:

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^{N} p(y^{(i)} \mid \mathbf{x}^{(i)}, \boldsymbol{\theta}) = \prod_{i=1}^{N} \left(\frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right)^{y^{(i)}} \left(\frac{1}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right)^{\left(1 - y^{(i)}\right)} \tag{5}$$

$$= \prod_{i=1}^{N} \frac{\left(e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right)^{y^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \tag{6}$$

Hence, $J(\boldsymbol{\theta})$, that is $\frac{1}{N}$ times the negative conditional log-likelihood, is:

$$J(\boldsymbol{\theta}) = -\frac{1}{N} \log p\left(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}\right) = \frac{1}{N} \sum_{i=1}^{N} -y^{(i)} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)}\right) + \log\left(1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right) \tag{7}$$

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

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} = -\frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} \left[y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{8}$$

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

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j - \alpha \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} \tag{9}$$

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

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j + \alpha \frac{x_j^{(i)}}{N} \left[y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{10}$$

4 Implementation

4.1 Overview

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

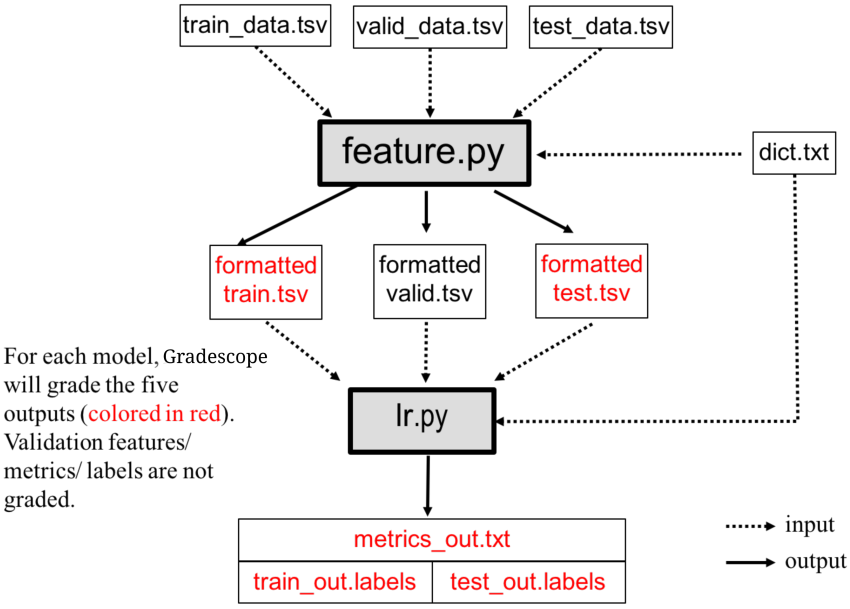

Figure 1: Programming pipeline for sentiment analyzer based on binary logistic regression

This first program is `feature.{py|java|cpp}`, 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 datasets should be stored in .tsv format. Details of formatted datasets will be introduced in Section 4.2 and Section 5.1.

The second program is `lr.{py|java|cpp}`, 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 discussed in Appendix A.2 and A.3, efficient computation can be obtained with the help of the indexing information in the dictionary file `dict.txt`.

4.2 Feature Engineering

Your implementation of `feature.{py|java|cpp}` 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 $p\left(y^{(i)} \mid \phi_1\left(\mathbf{x}^{(i)}\right), \boldsymbol{\theta}\right)$: This model uses a *bag-of-words* feature vector $\phi_1\left(\mathbf{x}^{(i)}\right) = \mathbf{1}_{\mathbf{occur}}(\mathbf{x^{(i)}}, \mathbf{Vocab})$ indicating which words in vocabulary **Vocab** of the dictionary occur at least once in the movie review example $\mathbf{x}^{(i)}$. Specifically, there are $V = |\mathbf{Vocab}|$ entries in this indicator vector, and the j-th entry will be set to 1 if the j-th word in **Vocab** occurs at least once in the movie review. The j-th entry will be set to 0 otherwise. This bag-of-words model should be used when `<feature_flag>` is set to 1.

Model 2 $p\left(y^{(i)} \mid \phi_2\left(\mathbf{x}^{(i)}\right), \boldsymbol{\theta}\right)$: This model uses a *trimmed* bag-of-words feature vector $\phi_2\left(\mathbf{x}^{(i)}\right) = \mathbf{1}_{\mathbf{trim}}\left(\mathbf{x}^{(i)}, \mathbf{Vocab}, \mathbf{t}\right)$ indicating: (1) which word in vocabulary **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 with a word that satisfies both conditions will be set to 1 (otherwise, it will be 0). This trimmed bag-of-words 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 as "the", "a", "to", etc. You will observe whether this basic and intuitive strategy will improve performance.

Note that above $\mathbf{1}_{\mathbf{occur}}$ and $\mathbf{1}_{\mathbf{trim}}$ are described as a dense feature representation as shown in Table 3 for illustration purpose. In your implementation, you should further convert it to the representation in Table 4 for Model 1 and the representation in Table 6 for Model 2, such that the formatted data outputs match Section 5.1.

4.3 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...]
```

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)
2. `<validation_input>`: path to the validation input `.tsv` file (see Section 2)
3. `<test_input>`: path to the test input `.tsv` file (see Section 2)
4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2)
5. `<formatted_train_out>`: path to output `.tsv` file to which the feature extractions on the *training* data should be written (see Section 5.1)

6. `<formatted_validation_out>`: path to output `.tsv` file to which the feature extractions on the *validation* data should be written (see Section 5.1)
7. `<formatted_test_out>`: path to output `.tsv` file to which the feature extractions on the *test* data should be written (see Section 5.1)
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 4.2)—that is, if `feature_flag`==1 use Model 1 features; if `feature_flag`==2 use Model 2 features

Likewise, `[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 5.1)
2. `<formatted_validation_input>`: path to the formatted validation input `.tsv` file (see Section 5.1)
3. `<formatted_test_input>`: path to the formatted test input `.tsv` file (see Section 5.1)
4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2)
5. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 5.2)
6. `<test_out>`: path to output `.labels` file to which the prediction on the *test* data should be written (see Section 5.2)
7. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and test error should be written (see Section 5.3)
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. You are asked to graph the negative log-likelihood vs. epoch of the validation and training data in Programming Empirical Questions section. [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.

5 Program Outputs

5.1 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 datasets 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  ...
```

5.2 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
```

5.3 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 dataset with Model 1 feature structure after 60 training epochs. See `model1_metrics_out.txt` in the handout.

```
error(train): 0.092500
error(test): 0.222500
```

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.

6 Evaluation and Submission

6.1 Evaluation

Gradescope will test your implementations on hidden datasets with the same format as the two datasets provided in the handout. `feature` program and `lr` program will be tested separately. To ensure that your code can pass the gradescope 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.

6.2 Requirements

Your implementation must satisfy the following requirements:

- The `feature.{py|java|cpp}` 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 5.1 and Section 4.2 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 4.2).
- 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 $\alpha = 0.1$.
- 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 4.2)
- Do not hard-code any aspects of the datasets into your code. We will autograde your programs on multiple (hidden) datasets that include different attributes and output labels.

6.3 Hints

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

- Work through the written component.
- (**Python only**) We strongly recommend you to write your own text parser rather than using parser provided in various packages.
- Be sure to check that the output from your `feature.{py|java|cpp}` matches the reference output given exactly before proceeding to `lr.{py|java|cpp}`.
- Write a function that takes a single SGD step on the i-th training example. Such a function should take as input the model parameters, the learning rate, and the features and label for the i-th 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.
- You can either treat the bias term as separate variable, or fold it into the parameter vector. In either case, make sure you update the bias term correctly.

6.4 Gradescope Submission

You should submit your `feature.{py|java|cpp}` and `lr.{py|java|cpp}` to Gradescope. Note: please do not use other file names. This will cause problems for the autograder to correctly detect and run your code.

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 2 shows an example input file, where column i indexes the i-th movie review example. Rather than working directly with this input file, you should transform the sentiment/text representation into a label/feature vector representation.

Table 3 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 3 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 4 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 5); (2) trim/remove the highly repetitive words/features and set the value of all remaining features to one (see Table 6).

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 3) is used, this dot-product requires $O(M)$ computation. Why? Because the dot-product requires a sum over each entry in the vector:

$$\boldsymbol{\theta}^T \mathbf{x} = \sum_{m=1}^{M} \theta_m x_m \tag{11}$$

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:

$$\boldsymbol{\theta}^T \mathbf{x} = \sum_{m \in \{1,\ldots,M\} \text{ s.t. } x_m \neq 0} \theta_m x_m \tag{12}$$

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 (12)).

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 uses 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 indices of words satisfying certain conditions will be stored, and all other indices implicitly 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 for 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 2: 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 4) or Model 2 features (Table 6).

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 3: Dense feature representation for Model 1 corresponding to the input file in Table 2. 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 4: Sparse feature representation (bag-of-word representation) for Model 1 corresponding to the input file in Table 2.

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

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 6: Sparse feature representation for Model 2 corresponding to the input file in Table 2. 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.