HOMEWORK 4: LOGISTIC REGRESSION

10-301/10-601 Introduction to Machine Learning (Fall 2021)
http://mlcourse.org
OUT: Saturday, October 2, 2021
DUE: Monday, October 11, 2021 11:59 PM
TAs: Gopi Krishna, Roshan, Justin, Youngjoo, Jingyun

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 logistic regression. Then in the Programming component, you will implement a logistic regression model as the core of your natural language processing system.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **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, please use the provided template. Submissions can be handwritten onto the template, 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. Each derivation/proof should be completed in the boxes provided. If you do not follow the template, 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 3.9.6, OpenJDK 11.0.11, g++ 7.5.0) and versions of permitted libraries (e.g. `numpy` 1.21.2 and `scipy` 1.7.1) match those used on Gradescope. You have 10 Gradescope programming submissions. 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 may use 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 0.41. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.41-libs/*:linalg_lib/nd4j-v1.0.0-M1.1-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-M1.1. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.41-libs/*:linalg_lib/nd4j-v1.0.0-M1.1-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 3.4.0. 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 Coursework page of the course website 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://javadoc.io/doc/org.nd4j/nd4j-api/latest/index.html
[c] http://eigen.tuxfamily.org/

Instructions for Specific Problem Types

For "Select One" questions, please fill in the appropriate bubble completely:

Select One: Who taught this course?

● Matt Gormley / Henry Chai

○ Marie Curie

○ Noam Chomsky

If you need to change your answer, you may cross out the previous answer and bubble in the new answer:

Select One: Who taught this course?

● Matt Gormley / Henry Chai

○ Marie Curie

● Noam Chomsky

For "Select all that apply" questions, please fill in all appropriate squares completely:

Select all that apply: Which are scientists?

□ Stephen Hawking

■ **Albert Einstein**

□ Isaac Newton

□ None of the above

Again, if you need to change your answer, you may cross out the previous answer(s) and bubble in the new answer(s):

Select all that apply: Which are scientists?

■ Stephen Hawking

■ Albert Einstein

■ Isaac Newton

■ I don't know

For questions where you must fill in a blank, please make sure your final answer is fully included in the given space. You may cross out answers or parts of answers, but the final answer must still be within the given space.

Fill in the blank: What is the course number?

10-601

10-7601

1 Written Questions (20 points)

1.1 Logistic Regression and Stochastic Gradient Descent

1. (2 points) 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.

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

3. (2 points) Data is separable in one dimension if there exists a threshold t such that all values less than t have one class label and all values $\geq t$ have the other class label. If you train logistic regression for infinite iterations without regularization on training data that is separable in at least one dimension, the corresponding weight(s) 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

It is possible that the the linear decision boundary is vertical to one dimension, e.g x1=t, since the training data is separable in at least one dimension. In this situation, the iterations may be infinite without regularization and the resulting weights can go to infinity in magnitude.

4. (2 points) How does regularization such as ℓ_1 and ℓ_2 help correct the problem in part (c)?

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.

1.2 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.

1. (2 points) 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).

$J(\boldsymbol{\theta})$

1.9309

Work

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

$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_1$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_2$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_3$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_4$	$\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta}_5$
0.2427	-0.0417	0.2336	0.2427	0.2455

Work

3. (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 updated (numerical) value of the vector θ. use learning rate $\alpha = 1$.

θ_1	θ_2	θ_3	θ_4	θ_5
1.2573	2.0417	0.7664	1.7573	2.7545

Work

1.3 Logistic Regression on an Image Dataset

Images can be represented numerically as a vector of values for each pixel. Image classification tasks use this vector of pixel values as features to predict an image label. An automobile company is trying to gather data by asking participants to submit grayscale images of cars. Each pixel has an intensity value in the continuous range $[0, 1]$. The company is attempting to run a logistic regression model to predict whether the photo actually contains a car. After training the model initially on a training dataset, the company achieves a mediocre test error. The company wants to improve the model and offers monetary compensation to people who can submit photos that contain a car and make the model predict "false" (i.e., a false negative), as well as photos that do not contain a car and make the model predict "true" (i.e., a false positive). Furthermore, the company releases the parameters of their learned logistic regression model. Let's investigate how to use these parameters to understand the model's weaknesses.

1. (2 points) Given the company's model parameters $\boldsymbol{\theta}$ (i.e., the logistic regression coefficients), gradient ascent can be used to find the vector of pixel values that maximizes the "car" prediction. What is the gradient update rule to do so? Write (1) the objective function, (2) the gradient of the objective function with respect to the feature *values* and (3) the update rule. Use $\mathbf{x}$ as the input vector.

Your Answer

The objective function $J(x|\theta)$ is the log likelihood of predicting a "car" given the company's model parameters θ and input x: $J(x|\theta) = log(\frac{1}{1+exp(-\theta^T x)})$.
The gradient of the objective function with respect to the feature values is $\Delta J(x|\theta) = \frac{1}{1+exp(\theta^T x)}\theta$.
The update rule of the gradient ascent is $x = x + \eta \Delta J(x|\theta)$, where η is the learning rate.

2. (1 point) Modify the procedure from part (a) to find the image that minimizes the "car" prediction.

Your Answer

Minimizing the "car" prediction is equivalent to maximizing the "not car" prediction. So the objective function is $\tilde{J}(x|\theta) = log(1 - \frac{1}{1+exp(-\theta^T x)})$, which is still a concave function.
Again we use gradient ascent to the new objective function $\tilde{J}(x|\theta)$. Thus, the update rule changes to $x = x + \eta \Delta \tilde{J}(x|\theta)$, where η is the learning rate and $\Delta \tilde{J}(x|\theta) = -\frac{1}{1+exp(-\theta^T x)}\theta$
.

3. (2 points) To generate an image, we require the feature values to be in the range $[0, 1]$. Propose a different procedure that optimizes the "car" prediction subject to this constraint and does not require a gradient calculation. What's the runtime of this procedure?

Your Answer

It is obvious the log likelihood is monotone increasing with respect to $\theta^T x$, so the maximizer under the interval $[0, 1]$ is $x = (x_1, \ldots, x_m)$, where $x_j = \mathbf{1}\{\theta_j \geq 0\}$, m is the dimension of pixel vector.

4. (2 points) Now let's consider whether logistic regression is well-suited for this task. Suppose the exact same white car in a dark background is used to generate train and test data. The training photos were captured with the side view of the car centered in the photo at a distance of between 3-50 meters from the camera. **Select which (if any) of the below descriptions of a test image you expect the model to correctly predict as "car."**

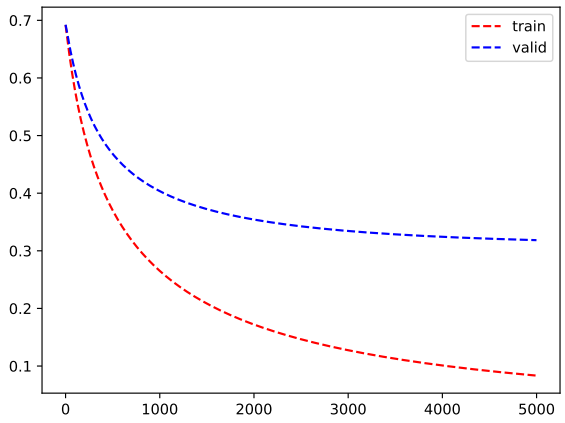
- ☐ The car is centered and 60 meters from the camera.
- ☐ The car is located in the upper-right hand corner of the frame.
- ☒ The car in a training image is replaced with an equal size white cardboard cutout of the car.
- ☐ The background of one of the training images is changed to white.
- ☐ None of the above.

1.4 Programming Empirical Questions

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

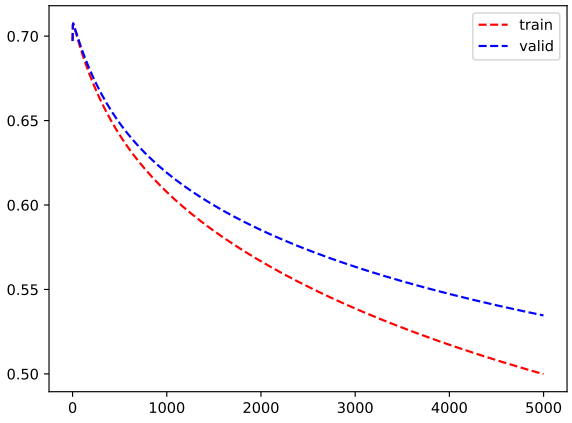
1. (2 points) For *Model 1*, using the data in the `largedata` folder in the handout, make a plot that shows the *average* negative log likelihood for the training and validation data sets after each of 5,000 epochs. The y-axis should show the negative log likelihood and the x-axis should show the number of epochs. (Note that running the code for 5,000 epochs might take longer than one minute. This is okay since we won't run your code for more than 500 epochs during auto-grading.)

Your Answer



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

Your Answer



3. (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? Without looking at the plots, which model would you have predicted to perform better (i.e. get lower error rates faster) and why? Which model actually performed better in practice and why might this be the case?

Your Answer

For both Model1 and Model2, the log likelihood of training is always higher than the log likelihood of validation and the gap is increasing with respect to the number of epoches. In addition, the gap between training and validation is larger for Model1 than Model2. This is because that the trained parameters are aimed to maximize the log likelihood of the training data, so applying the trained parameters to validation data will results in smaller log likelihood compared to that to training data. In addition, as the number of epoches increases, it is more likely to be overfitting, which is why the gap increases for large epoches.
Without looking at the plots, I predicted Model 2 performs better since it seems that the feature engineering method of Model 2 captures more information. However, in fact Model 1 performed better and converged very quickly. This is possible because that Model 1 embeds much more features so it can extract more information from the data.

4. (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 5,000 epochs. Please use one number rounded to the fourth decimal place, e.g., 0.1234.

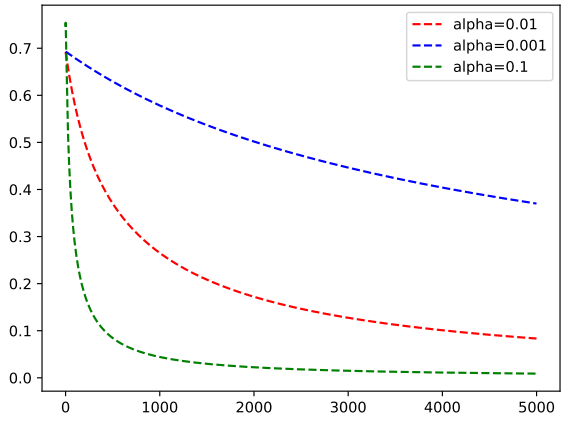
Your Answer

	Train Error	Test Error
Model 1	0.0000	0.1400
Model 2	0.2142	0.2550

Table 1: "Large Data" Results

5. (2 points) For *Model 1*, using the data in the largedata folder of the handout, make a plot comparing the training average negative log-likelihood over epochs for three different values for the learning rates, $\alpha \in \{0.001, 0.01, 0.1\}$. The y-axis should show the negative log likelihood, the x-axis should show the number of epochs (from 0 to 5,000 epochs), and the plot should contain three curves corresponding to the three values of α. Provide a legend that indicates α for each curve.

Your Answer



6. (2 points) Compare how quickly each curve in the previous question converges.

Your Answer

Based on the comparison in the above plot, it is obvious that larger learning rate corresponds to faster convergence in this case.

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

None

2 Programming (70 points)

2.1 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.

2.2 The Datasets

Datasets Download the zip file from the course website, 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 **500 training epochs** for both Model 1 and Model 2 (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 **500 training epochs** for both Model 1 and Model 2 (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 500-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.

Dictionary We also provide a dictionary file (`dict.txt`) to limit the vocabulary to be considered in

[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`

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 index\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
```

Word2vec Feature Dictionary For Model 2, we provide an additional feature dictionary used for feature extraction (`word2vec.txt`). To build this feature dictionary, we used pre-trained word2vec models trained from text in Wikipedia to generate dense vector representations of words in `dict.txt`. The file contains the same words as those in `dict.txt` in the same order. In each line, the feature dictionary contains a word and then 300 floating point numbers separated by *tabs*.

2.3 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:

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

$$= \prod_{i=1}^{N} \frac{\left(e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right)^{y^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \tag{2}$$

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

$$J(\boldsymbol{\theta}) = -\frac{1}{N} \log p\left(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}\right) = \frac{1}{N} \sum_{i=1}^{N} -y^{(i)} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)}\right) + \log \left(1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right) \tag{3}$$

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

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} = -\frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} \left[y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{4}$$

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

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j - \alpha \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} \tag{5}$$

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

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j + \alpha \frac{x_j^{(i)}}{N} \left[y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{6}$$

2.4 Implementation

2.4.1 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.

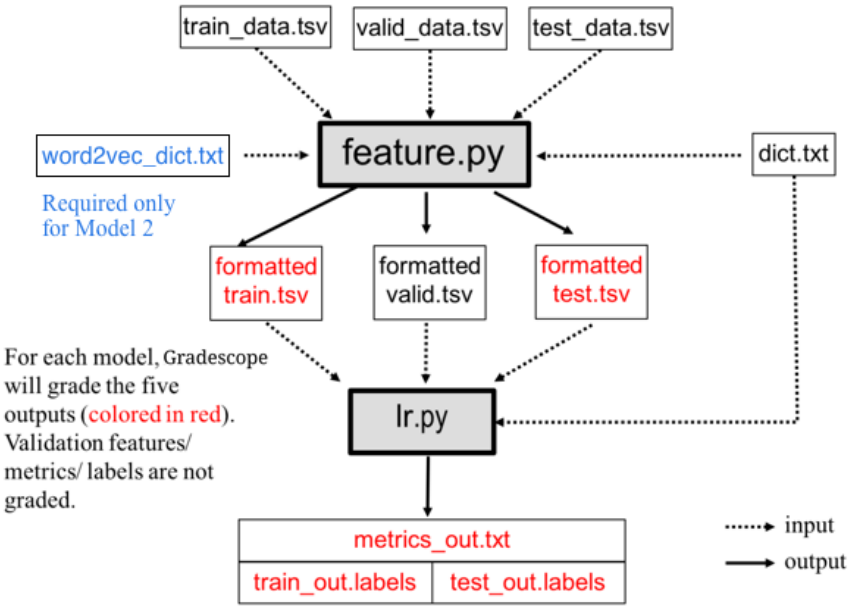


Figure 1: Programming pipeline for sentiment analyzer based on binary logistic regression

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`. For Model 2, the program also takes in an additional feature dictionary file `word2vec.txt`. To be specific, this program is to transfer the whole movie review text into a feature vector using some feature extraction methods. You should *not* be folding in a bias term into the formatted output. The formatted datasets should be stored in .tsv format. Details of formatted datasets will be introduced in Section 2.4.2 and Section 2.5.1.

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. This program can either be implemented by folding in the bias term and weight or not. The program should output the labels of the training and test examples and calculate training and test error (percentage of incorrectly labeled reviews).

2.4.2 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.

2.4.3 Model 1

Model 1 is written as $p\left(y^{(i)} \mid \boldsymbol{\phi}_1\left(\mathbf{x}^{(i)}\right), \boldsymbol{\theta}\right)$. This model uses a *bag-of-words* feature vector $\boldsymbol{\phi}_1\left(\mathbf{x}^{(i)}\right) = \mathbf{1}_{\mathbf{occur}}(\mathbf{x}^{(\mathbf{i})}, \mathbf{Vocab})$ indicating which words in vocabulary $\mathbf{Vocab}$ 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 $\mathbf{Vocab}$ 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.

2.4.4 Model 2

Model 2 is written as $p\left(y^{(i)} \mid \boldsymbol{\phi}_2\left(\mathbf{x}^{(i)}\right), \boldsymbol{\theta}\right)$. This model makes use of *word2vec* embeddings, which are reduced dimension vector representations (features) of words. These feature vectors will be provided in `word2vec.txt`, which contains the *word2vec* embeddings of `15k` words. Note that not every word in the movie review examples will be included in the provided `word2vec.txt` file.

For this model, there will be two steps in the feature engineering process:

1. First, we would like to exclude words from the review that are not included in the word2vec dictionary. Let I_i be the set of indices of words in $\mathbf{x}^{(i)}$ that are included in the *word2vec* dictionary. Then, we can define $\mathbf{x}\_\text{trim}^{(i)} = \mathbf{x}^{(i)}[I_i]$, where $\mathbf{x}^{(i)}[I_i]$ trims vector $\mathbf{x}^{(i)}$ by only including elements of $\mathbf{x}^{(i)}$ with indices in I_i.

2. Second, we want to take the trimmed vector $\mathbf{x}\_\text{trim}^{(i)}$ and convert it to the final feature vector given by the following equation:

$$\boldsymbol{\phi}_2\left(\mathbf{x}^{(i)}\right) = \frac{1}{J}\sum_{j=1}^{J} word2vec(\mathbf{x}\_\text{trim}_j^{(i)})$$

   where J denotes the number of words in $\mathbf{x}\_\text{trim}^{(i)}$ and $\mathbf{x}\_\text{trim}_j^{(i)}$ is the j-th word in $\mathbf{x}\_\text{trim}^{(i)}$.

   In the given equation, $word2vec(\mathbf{x}\_\text{trim}_j^{(i)}) \in \mathbb{R}^{300}$ is the *word2vec* feature vector for the word $\mathbf{x}\_\text{trim}_j^{(i)}$.

The following **example** provides a reference for Model 2:

- Let $\mathbf{x}^{(i)}$ denote the sentence "`a hot dog is not a sandwich because it is not square`".
- A toy *word2vec* dictionary is given as follows (note that unlike the following text, contents in the actual input file given will be tab-separated):

```
hot          0.1    0.2    0.3
not         -0.1    0.2   -0.3
sandwich     0.0   -0.2    0.4
square       0.2   -0.1    0.5
```

- Then, $\mathbf{x}\_\text{trim}^{(i)}$ denotes the trimmed review "`hot not sandwich not square`". Note that in this trimmed text, the words that are not in the *word2vec* dictionary are excluded. Also note that we keep the order of words and do not de-duplicate words in the trimmed text.
- The feature for $\mathbf{x}^{(i)}$ can be calculated as

$$\begin{aligned}\phi_2(\mathbf{x}^{(i)}) &= \frac{1}{5}\big(word2vec(\text{hot}) + 2 \cdot word2vec(\text{not}) + word2vec(\text{sandwich}) + word2vec(\text{square})\big) \\ &= \begin{bmatrix}0.02 & 0.06 & 0.12\end{bmatrix}^T.\end{aligned}$$

The motivation of Model 2 is that pre-trained feature representations such as word2vec embeddings may provide richer information about sentiment of a sentence than the simpler bag-of-words features. You will observe whether using pre-trained word embeddings to build feature vectors will improve performance and accuracy.

2.4.5 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 nine command-line arguments: `<train_input>` `<validation_input>` `<test_input>` `<dict_input>` `<formatted_train_out>` `<formatted_validation_out>` `<formatted_test_out>` `<feature_flag>` `<feature_dictionary_input>`. These arguments are described in detail below:

1. `<train_input>`: path to the training input `.tsv` file (see Section 2.2)
2. `<validation_input>`: path to the validation input `.tsv` file (see Section 2.2)
3. `<test_input>`: path to the test input `.tsv` file (see Section 2.2)
4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2.2)
5. `<formatted_train_out>`: path to output `.tsv` file to which the feature extractions on the *training* data should be written (see Section 2.5.1)
6. `<formatted_validation_out>`: path to output `.tsv` file to which the feature extractions on the *validation* data should be written (see Section 2.5.1)
7. `<formatted_test_out>`: path to output `.tsv` file to which the feature extractions on the *test* data should be written (see Section 2.5.1)
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 2.4.2)—that is, if `feature_flag`==1 use Model 1 features; if `feature_flag`==2 use Model 2 features
9. `<feature_dictionary_input>`: path to the word2vec feature dictionary `.tsv` file (see Section 2.2).

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 2.5.1)
2. `<formatted_validation_input>`: path to the formatted validation input `.tsv` file (see Section 2.5.1)
3. `<formatted_test_input>`: path to the formatted test input `.tsv` file (see Section 2.5.1)
4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2.2)
5. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 2.5.2)
6. `<test_out>`: path to output `.labels` file to which the prediction on the *test* data should be written (see Section 2.5.2)
7. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and test error should be written (see Section 2.5.3)
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 500 epochs using the features from Model 1.

```
$ python feature.py largedata/train_data.tsv largedata/valid_data.tsv
largedata/test_data.tsv dict.txt largeoutput/formatted_train.tsv
largeoutput/formatted_valid.tsv largeoutput/formatted_test.tsv
1 word2vec.txt

$ python lr.py largeoutput/formatted_train.tsv
largeoutput/formatted_valid.tsv largeoutput/formatted_test.tsv
dict.txt largeoutput/train_out.labels largeoutput/test_out.labels
largeoutput/metrics_out.txt 500
```

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.

2.5 Program Outputs

2.5.1 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`value1`\t`value2`\t`value3`\t`...valueM`\n`

Where above, the first column is label, and the rest are "value" feature elements. Value is the value of the feature at index `i` of the formatted output row. In this assignment, for Model 1 the value is one or zero depending on whether dictionary word `i` is present in the sentence. For Model 2, the rows are the summed up word2vec vectors for all the words present in the dictionary. 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. For Model 2, please round to 6 decimal places (e.g. 0.123456). Examples of content of formatted output file are given below.

For Model 1:

```
0       1       1       0       1       0       1       1       ...
0       1       0       0       1       0       1       1       ...
1       1       1       1       0       0       1       0       ...
1       0       1       1       0       0       0       1       ...
```

For Model 2:

```
1.000000        -0.217109       -0.157321       -0.193579       ...
0.098980        -1.059955       0.309624        -0.397892       ...
0.560687        1.878663        -1.354379       0.148609        ...
1.731375        -0.843260       1.445480        -2.125001       ...
```

2.5.2 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
```

2.5.3 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 and round the error values to 6 decimal places. The following is the reference solution for large dataset with Model 1 feature structure after

500 training epochs. See
`model1_metrics_out.txt` in the handout.

```
error(train): 0.049167
error(test): 0.150000
```

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.

2.6 Evaluation and Submission

2.6.1 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 500-epoch (for model 1 and 2, respectively) training and finish predictions through all of the data in the `largedata` folder in around one minute.

2.6.2 Requirements

Your implementation must satisfy the following requirements:

- The `feature.{py|java|cpp}` must produce a dense representation of the data according to the `feature_flag`. We will use unseen data to test your feature output separately. (see Section 2.5.1 and Section 2.4.2 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.
- 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 $\alpha = 0.01$ for both Model 1 and Model 2 when comparing your outputs with the reference output provided in the handout (you will need to change the learning rate for some of the empirical questions, however).
- 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 2.4.2)
- 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.

2.6.3 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.
- For reading in the word2vec embeddings, and to read the formatted inputs, we strongly recommend you utilize a linear algebra library to speed up vector computations.
- 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.

2.6.4 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.

A.2 Efficient Computation of the Dot-Product

Using a for loop to perform a dot product is very slow and inefficient, especially in python. There are linear algebra libraries optimized to compute the dot product for you. With that said, you may find the following linear algebra libraries for efficient dot product computation useful:

NumPy (Python)

```
np.dot(A, B)
```

EJML (Java)

```
dot(A, B)
```

ND4J (Java)

```
dot(A, B)
```

Eigen (C++)

```
A.dot(B)
```

where A and B are vectors or matrices

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 or Model 2 features.

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.