HOMEWORK 4: LOGISTIC REGRESSION

10-601 Introduction to Machine Learning (Spring 2018)
Carnegie Mellon University
https://piazza.com/cmu/spring2018/10601
OUT: Feb 16, 2018*
DUE: Feb 25, 2018 11:59 PM
TAs: Sriram, Boyue, Elaine, and Ryan

Summary In this assignment, you will build a "mini Siri", which will be capable of extracting flight information search terms from natural language using a logistic regression model. In Section 1.1 you will warm up by deriving stochastic gradient descent updates for binary logistic regression. Then in Section 2 you will implement a multinomial logistic regression model as the core of your natural language processing system.

START HERE: Instructions

- **Collaboration Policy**: Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 3.4"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the collaboration policy on the website for more information: http://www.cs.cmu.edu/~mgormley/courses/10601-s18/about.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601-s18/about.html
- **Submitting your work:** You will use Gradescope to submit answers to all questions, and Autolab to submit your code. Please follow instructions at the end of this PDF to correctly submit all your code to Autolab.
  - **Gradescope:** For written problems such as derivations, proofs, or plots we will be using Gradescope (https://gradescope.com/). Submissions can be handwritten, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Upon submission, label each question using the template provided. Regrade requests can be made, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted. Each derivation/proof should be completed on a separate page.

*Compiled on Sunday 18th February, 2018 at 20:00

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

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

1 Written Questions [20 pts]

1.1 Binary Logistic Regression [12 pts]

(a) **[2 Points]** In logistic regression, our goal is to learn a set of parameters by maximizing the conditional log likelihood of the data. Assume you are given a dataset with N training examples and M features. Write down a formula for the *negative* conditional log likelihood of the training data in terms of the design matrix $\mathbf{X}$, the labels $\mathbf{y}$, and the parameter vector $\boldsymbol{\theta}$. This will be your objective function $J(\boldsymbol{\theta})$ for gradient descent. (Recall that ith row of the design matrix $\mathbf{X}$ contains the features $\mathbf{x}^{(i)}$ of the ith training example. The ith entry in the vector $\mathbf{y}$ is the label $y^{(i)}$ of the ith training example. Here we assume that each feature vector $\mathbf{x}^{(i)}$ contains a bias *feature*, e.g. $x_1^{(i)} = 1 \ \forall i \in \{1, \ldots, N\}$. As such, the bias parameter is folded into our parameter vector $\boldsymbol{\theta}$.)

(b) **[3 Points]** Derive the partial derivative of the objective function with respect to the mth parameter. That is, derive $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_m}$, where $J(\boldsymbol{\theta})$ is the objective that you provided above. Please show all derivatives can be written in a finite sum form. Show all steps of the derivation.

(c) **[2 Points]** Write gradient descent update rules for logistic regression for arbitrary θ_m.

(d) **[2 Points]** Write down the stochastic gradient descent update for an arbitrary θ_m using the i^{th} training example with features $\mathbf{x^{(i)}}$ and output label $y^{(i)}$.

(e) **[3 Points]** If you train logistic regression for infinite iterations without ℓ_1 or ℓ_2 regularization, the weights can go to infinity. What is an intuitive explanation for this phenomenon? How does regularization help correct the problem?

1.2 Empirical Questions [8 pts]

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

1. **Plots [2 Points]** For *Model 1*, using the data in the largedata folder in the handout, make a plot that shows the average negative log likelihood for the training and validation data sets after each of 100 epochs. The y-axis should show the negative log likelihood and the x-axis should show the number of epochs.
2. **Plots [2 Points]** For *Model 2*, make a plot as in the previous question.
3. **Explanation of Experiment [2 Points]** Write a few sentences explaining the output of the above experiment. In particular do the training and validation log likelihood curves look the same or different? Why?
4. **Results [2 Points]** Make a table with your train and test error for the large data set (found in the largedata folder in the handout) for each of the 2 models after running for 10 epochs.

	Train Error	Test Error
Model 1		
Model 2		

Table 1.1: "Large Data" Results

2 Programming [80 pts]

Your goal in this assignment is to implement a working Natural Language Processing (NLP) system, i.e., a mini Siri, using multinomial logistic regression. You will then use your algorithm to extract flight information from natural text. You will do some very basic feature engineering, though which you will be able to improve the learner's performance on this task. You will write one program: `tagger.{py|java|cpp|m}`. The program you write will be automatically graded using the Autolab system. You may write your programs in **Octave, Python, Java,** or **C++**. However, you should use the same language for all parts below.

2.1 The Tasks and Data Sets

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

The handout contains data from the Airline Travel Information System (ATIS) data set (for more details, see http://deeplearning.net/tutorial/rnnslu.html). Each data set consists of attributes (words) and labels (airline flight information tags). The attributes and tags are separated into sequences (i.e., phrases) with a blank line between each sequence.

The tags are in Begin-Inside-Outside (BIO) format. Tags starting with B indicating the beginning of a piece of information, tags beginning with I indicating a continuation of a previous type of information, and O tags indicating words outside of any information chunk. For example, in the sentence below, Newark is the departure city (`fromloc.city_name`), Los Angeles is the destination (`toloc.city_name`), and the user is requesting flights for Wednesday (`depart_date.day_name`). In this homework, you will treat the BIO tags as arbitrary labels for each word.

```
what    O
flights O
from    O
newark  B-fromloc.city_name
to      O
los     B-toloc.city_name
angeles I-toloc.city_name
on      O
wed     B-depart_date.day_name
```

We have provided you with two subsets of the ATIS data set. Each one is divided into a training, a validation, and a test data set. The toy data set (`toytrain.tsv`, `toyvalidation.tsv`, and `toytest.tsv`) is a small data set that can be used while debugging your code. We have included the reference output files for this toy data set (see directory `toyoutput/`). We have also included a larger data set without reference outputs (`train.csv`, `validation.csv`, `test.csv`). This data set can be used to ensure that your code runs fast enough to pass the autograder tests. Your code should be able to perform one pass 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.

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

2.2 Model Definition

We model the probability that label $y^{(i)} \in \{1, 2, \ldots, K\}$ equals k for the i[th] word in a sequence given a set of features $\mathbf{x}^{(i)} \in \mathbb{R}^M$ as

$$\mathbb{P}(y^{(i)} = k|\mathbf{x}^{(i)}) = \frac{\exp({\boldsymbol{\theta}^{(k)}}^\top \mathbf{x}^{(i)} + \alpha^{(k)})}{\sum_{j=1}^{K} \exp({\boldsymbol{\theta}^{(j)}}^\top \mathbf{x}^{(i)} + \alpha^{(j)})} \tag{2.1}$$

As usual, we can augment each feature vector $\mathbf{x}^{(i)}$ with a bias feature that always takes value 1 (this bias feature will be treated just like any other feature), resulting in the following expression for the likelihood:

$$\mathbb{P}(y^{(i)} = k|\mathbf{x}^{(i)}) = \frac{\exp({\boldsymbol{\theta}^{(k)}}^\top \mathbf{x}^{(i)})}{\sum_{j=1}^{K} \exp({\boldsymbol{\theta}^{(j)}}^\top \mathbf{x}^{(i)})} \tag{2.2}$$

To apply this model, we first need to define the average negative log-likelihood function and derive the corresponding gradient. The average negative log-likelihood function for this model is:

$$J(\boldsymbol{\theta}) = -\frac{1}{N}\ell(\boldsymbol{\theta}) \tag{2.3}$$

$$= -\frac{1}{N} \log \left(\prod_{i=1}^{N} \prod_{k=1}^{K} \mathbb{P}(y^{(i)} = k|\mathbf{x}^{(i)})^{\mathbb{I}(y^{(i)}=k)} \right) \tag{2.4}$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \mathbb{I}(y^{(i)} = k) \log \frac{\exp({\boldsymbol{\theta}^{(k)}}^\top \mathbf{x}^{(i)})}{\sum_{j=1}^{K} \exp({\boldsymbol{\theta}^{(j)}}^\top \mathbf{x}^{(i)})} \tag{2.5}$$

where $\mathbb{I}(\cdot)$ is the indicator function. The gradient of this function is:

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{i=1}^{N} \left(\mathbb{I}(y^{(i)} = k) - \frac{\exp({\boldsymbol{\theta}^{(k)}}^\top \mathbf{x}^{(i)})}{\sum_{j=1}^{K} \exp({\boldsymbol{\theta}^{(j)}}^\top \mathbf{x}^{(i)})} \right) \mathbf{x}^{(i)} \tag{2.6}$$

In this homework, we will use stochastic gradient descent (SGD) to optimize our objective function. In SGD, we are only given one (or a small chunk) of training instances at any given time. In this assignment, we will be using only one training instance in each gradient update. The gradient update will therefore be as follows:

$$\boldsymbol{\theta}^{(k)} \leftarrow \boldsymbol{\theta}^{(k)} - \eta \left[\nabla_{\boldsymbol{\theta}^{(k)}} J^{(i)}(\boldsymbol{\theta}) \right] \tag{2.7}$$

where $\nabla_{\boldsymbol{\theta}^{(k)}} J^{(i)}(\boldsymbol{\theta})$ is defined as follows and is computed for all k before updating any individual $\boldsymbol{\theta}^{(k)}$

$$\nabla_{\boldsymbol{\theta}^{(k)}} J^{(i)}(\boldsymbol{\theta}) = -\left(\mathbb{I}(y^{(i)} = k) - \frac{\exp\left({\boldsymbol{\theta}^{(k)}}^\top \mathbf{x}^{(i)}\right)}{\sum_{j=1}^{K} \exp\left({\boldsymbol{\theta}^{(j)}}^\top \mathbf{x}^{(i)}\right)} \right) \mathbf{x}^{(i)} \tag{2.8}$$

2.3 Implementation

Write a program, `tagger.{py|java|cpp|m}`, that implements a text analyzer using multinomial logistic regression. The file should learn the parameters of a multinomial logistic regression model that predicts a tag (i.e. label) for each word and its corresponding feature vector. The program should output the labels of the training and test examples and calculate training and test error (percentage of incorrectly labeled words).

Your implementation must satisfy the following requirements:

- Initialize all model parameters to 0.
- Use stochastic gradient descent (SGD) to optimize the parameters for a multinomial logistic regression model. The number of times SGD loops through all of the training data (`num_epoch`) will be specified as a command line flag. Set your learning rate as a constant $\eta = 0.5$.
- 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 structures you will use in your logistic regression model using an command line flag (see Section 2.3.4)
- To resolve ties where multiple classes have the same likelihood, choose the label with the smaller ASCII value (e.g., 'Aardvark' is less than 'Apple'; and 'Apple' is less than 'apple']).
- Do not hard-code any aspects of the data sets into your code. We will autograde your programs on multiple (hidden) data sets that include different attributes and output labels.

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

- Write a helper function that produces a sparse representation of the data (see Section 2.3.4 on feature engineering for details on how to do this).
- Write a function that takes a single SGD step on the ith training example. Such a function should take as input the model parameters, the learning rate, and the features and label for the ith training example. It should update the model parameters in place by taking one stochastic gradient step.
- Write a function that takes in a set of features, labels, and model parameters and then outputs the error (percentage of labels incorrectly predicted). You can also write a separate function that takes the same inputs and outputs the negative log-likelihood of the regression model.
- Be sure to handle boundary cases as specified in (see Section 2.3.4)

2.3.1 Command Line Arguments

The autograder runs and evaluates the output from the files generated, using the following command:

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

Where above `[args...]` is a placeholder for eight command-line arguments:`<train_input>` `<validation_input>` `<test_input>` `<train_out>` `<test_out>` `<metrics_out>` `<num_epoch>` `<feature_flag>`. These arguments are described in detail below:

1. `<train_input>`: path to the training input `.tsv` file (see Section 2.1)
2. `<validation_input>`: path to the validation input `.tsv` file (see Section 2.1)
3. `<test_input>`: path to the test input `.tsv` file (see Section 2.1)
4. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 2.3.2)
5. `<test_out>`: path to output `.labels` file to which the prediction on the *test* data should be written (see Section 2.3.2)
6. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and test error should be written (see Section 2.3.3)
7. `<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).
8. `<feature_flag>`: integer taking value 1 or 2 that specifies whether to construct the Model 1 feature set or the Model 2 feature set (see Section 2.3.4)—that is, if `feature_flag`==1 use Model 1 features; if `feature_flag`==2 use Model 2 features

As an example, if you implemented your program in Python, the following command line would run your program on the toy data provided in the handout for 2 epochs using the features from Model 1.

```
$ python tagger.py toytrain.tsv toyvalidation.tsv toytest.tsv \
mode1train_out.labels mode1test_out.labels mode1metrics_out.txt 2 1
```

Important Note: You will not be writing out the predictions on validation data, only on train and test data. The validation data is *only* used to give you an estimate of held-out negative log-likelihood at the end of each epoch during training.[a] See details in Section 2.3.3.

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

2.3.2 Output: Labels Files

Your program should write 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 auto-grader by running your program and evaluating your output file against the reference solution.

Note: You should output your predicted labels using the same string identifiers as the original training data. You should also insert an empty line (again using \n) at the end of each sequence (as is done in the input data files). The first few lines of an example output file are given below.

```
O
O
O
B-fromloc.city_name
O
B-toloc.city_name

O
O
O
O
O
O
O
O
B-city_name
```

2.3.3 Output Metrics

Generate a file where you report the following metrics:

negative log-likelihood After each Stochastic Gradient Descent (SGD) epoch, report training negative log likelihood `likelihood(train)` and validation negative log likelihood `likelihood(validation)`. Theses two likelihoods should be reported at the end of each epoch and prefixed by the epoch number. For example, after the second pass through the training examples, these should be prefixed by `epoch=2`. The total number of train likelihoods you print out should equal `num_epoch`—likewise for the total number of validation likelihoods.

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.

```
epoch=1 likelihood(train): 1.175428
epoch=1 likelihood(validation): 1.665414
epoch=2 likelihood(train): 0.900247
epoch=2 likelihood(validation): 1.317512
error(train): 0.314815
error(test): 0.200000
```

Take care that your output has the exact same format as shown above. There is an equal sign = between the word `epoch` and the epoch number, but no spaces. There should be a single space after the epoch number (e.g. a space after `epoch=1`), and a single space after the colon preceding the metric value (e.g. a space after `epoch=1 likelihood(train):`). Each line should be terminated by a Unix line ending `\n`.

2.3.4 Feature Engineering

Your implementation should have an input argument `<feature_flag>` that specifies one of two types of feature extraction that should be used by the logistic regression model. The two structures are listed below

as probabilities of the labels given the words. In each case w_t specifies the word at position t within a given sequence.

Model 1 $p(y_t \mid w_t, \boldsymbol{\theta})$: This model defines a probability distribution over the current tag y_t using the parameters $\boldsymbol{\theta}$ and a feature vector based on only the current word w_t. This model should be used when `<feature_flag>` is set to 1.

Model 2 $p(y_t \mid w_{t-1}, w_t, w_{t+1}, \boldsymbol{\theta})$: This model defines a probability distribution over the current tag y_t using the parameters $\boldsymbol{\theta}$ and a feature vector based on the previous word w_{t-1}, the current word w_t, and the next word w_{t+1} in the sequence. This model should be used when `<feature_flag>` is set to 2.

It is recommended that you create a separate parsing function for each model that reads in the input .tsv file and outputs a label array and a feature array. Each feature (w_{t-1}, w_t, and w_{t+1}) should use a one-hot encoding scheme. In this scheme, each element of the feature (or label) represents a possible word (or label). All elements of the vector should be zero except for the element corresponding to the word assigned to the feature. For example, if the current word, w_t, is "Boston", and "Boston" corresponds to the i$^{\text{th}}$ element in our list of possible features, w_t will be a vector with the i$^{\text{th}}$ position being 1 and everywhere else being 0. Alternatively, one can sparsely represent the feature value by making w_t a 1-dimensional integer variable and assigning w_t an index corresponding to the i$^{\text{th}}$ word in our list of possible features. This index could them be used to extract parameters from your model weight matrix. The labels can also be encoded using a similar format.

For model 2, be sure that each word has a separate weight for w_{t-1}, w_t, and w_{t+1}. For example, the word "Boston" should have one weight parameter for w_{t-1}, one for w_t and one for w_{t+1}.

For model 2, you will also need to handle boundary cases when you are either looking at the first word or the last word of the sequence. You can do this by adding an extra beginning of sentence ('BOS') feature and end of sentence ('EOS') feature to use in place of a word in the w_{t-1} or w_{t+1} position when t is the first or last word in a sentence, respectively.

2.3.5 Evaluation

Autolab will test your data on a hidden data set with the same format as the two data sets provided in the handout. To ensure that your code can pass the autolab tests in under 5 minutes (the maximum time length) be sure that your function can complete one pass through all of the `largedata` folder training data set (1 epoch) in less than roughly 1 minute for each of the models.

2.4 Autolab Submission

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

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

from the directory containing your code.

Some additional tips: **DO NOT** compress your files; you are just creating a tarball. Do not use tar `-czvf`. **DO NOT** put the above files in a folder and then tar the folder. Autolab is case sensitive, so observe that all your files should be named in **lowercase**. You must submit this file to the corresponding homework link on Autolab. The autograder for Autolab prints out some additional information about the tests that it ran. You

can view this output by selecting "Handin History" from the menu and then clicking one of the scores you received for a submission. For example on this assignment, among other things, the autograder will print out which language it detects (e.g. Python, Octave, C++, Java).

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

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

A Implementation Details for Logistic Regression

A.1 Examples of Features

Here we provide examples of the features constructed by Model 1 and Model 2. Table A.1 shows an example input file, where we have included an additional column t that indexes the position of each word in the sentence. Rather than working directly with this input file, you should transform from the word/tag representation into a label/feature vector representation.

Table A.2 shows the one-hot 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 observed in the training data set. Each row corresponds to a single training example, which we have indexed by i—this was not the case in the input file (Table A.1) which also contained blank lines to distinguish the end of each sentence.

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

Table A.3 shows the sparse representation of the one-hot feature vectors. Each feature vector is now represented by a map from the string name of the feature (e.g. `cur:angeles`) to its value (e.g. 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 *one entry* for each Model 1 feature vector and requires only $O(1)$ space.

Using the same sparse representation of features, we present an example of the features used by Model 2—see Table A.4. Notice that we now have three different feature templates corresponding to the current word w_t (`cur:`), the previous word w_{t-1} (`prev:`), and the next word w_{t+1} (`next:`). The string names of the features use these prefixes (`cur:`, `prev:`,`next:`) to indicate *which* context word we are referring to (w_{t-1}, w_t, w_{t+1}). We need to be careful around the boundaries of the sentence. For example, when $t = 1$ and w_1=`flight`, what is the previous word? Likewise, when $t = 5$ and w_5=`minneapolis`, what is the next word? To address this situation, we have also introduced special symbols that represent the "invisible" beginning-of-sentence (`BOS`) and end-of-sentence (`EOS`) markers. This ensures that each word uniformly has three non-zero entries in its sparse feature vector representation.

A.2 (Even More) Formal Definition of Feature Functions

Next we provide a more formal definition of the feature functions that captures some of nuances of how we handle edge-cases.

> Note that in this case, we've actually taken special care to address the case of out-of-vocabulary words (i.e. words that don't appear in the training data but do appear in the test data). Your TAs went to some lengths when preparing the data to ensure you will *never* encounter an out-of-vocabulary word. However, this presentation makes clear the way you should handle such cases in the real world.

Let $\mathcal{V}$ denote the vocabulary of words found in the training data, and Σ^* denote the set of all possible words. Let $\mathcal{B} \triangleq \{0, 1\}$ denote a set of size two containing the numbers zero and one. Model 1 defines a feature functions $f_1 : \Sigma^* \to \mathcal{B}^{M_1}$ that maps from a word w_t to a feature vector $\mathbf{x}$ of length $M_1 = |\mathcal{V}|$. Why is the

domain of f_1 the set of all possible words and not just those in our vocabulary? Because we must define f_1 so that it handles out-of-vocabulary words correctly since they could appear in the test data.

$$f_1(w_t) \triangleq \begin{cases} \text{one-hot vector } \mathbf{x} \text{ representing the word } w_t & \text{if } w_t \in \mathcal{V} \\ \text{zero vector } \mathbf{0}, & \text{otherwise} \end{cases} \tag{A.1}$$

Model 2 defines a similar feature function $f_2 : (\Sigma^* \times \Sigma^* \times \Sigma^*) \to \mathcal{B}^{M_2}$ that maps from a tuple of three words (w_{t-1}, w_t, w_{t+1}) to a feature vector $\mathbf{x}$ of length $M_2 = 3|\mathcal{V}|$.

$$f_1(w_t) \triangleq \begin{cases} \text{concatentation of three one-hot vectors } \begin{bmatrix} f_1(w_{t-1}) \\ f_1(w_t) \\ f_1(w_{t+1}) \end{bmatrix} & \text{if } w_t \in \mathcal{V} \\ \text{zero vector } \mathbf{0}, & \text{otherwise} \end{cases} \tag{A.2}$$

With these functions defined, we can now provide the following definitions of Model 1 and Model 2.

Model 1 Let the ith training example correspond to a label y_t and its word w_t in the original training data. Then Model 1 defines $p(y_t \mid w_t, \boldsymbol{\theta}) = \mathbb{P}(y^{(i)} = y_t | \mathbf{x}^{(i)} = f_1(w_t), \boldsymbol{\theta})$ where the latter probability is given by the probability in Equation (2.2).

Model 2 Let the ith training example correspond to a label y_t and its word w_t and context w_{t-1}, w_{t+1} in the original training data. Then Model 2 defines $p(y_t \mid w_{t-1}, w_t, w_{t+1}, \boldsymbol{\theta}) = \mathbb{P}(y^{(i)} = y_t | \mathbf{x}^{(i)} = f_2(w_t), \boldsymbol{\theta})$ where the latter probability is given by the probability in Equation (2.2).

In order to correctly provide context words w_{t-1} and w_{t+1} for the first and last words of each sentence respectively, we assume that each sentence is proceeded by a special beginning-of-sentence "BOS" word and followed by an end-of-sentence "EOS" word. Thus, for a sentences of length T, we have $w_0 \triangleq$ "BOS" and $w_{T+1} \triangleq$ "EOS".

A.3 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 $\mathbf{x} \in \mathbb{R}^M$ (in the examples above $\mathbf{x} \in \mathcal{B}^M \equiv \{0, 1\}^M$. When a dense representation of $\mathbf{x}$ (such as that shown in Table A.2) is used, this dot-product requires $O(M)$ computation. Why? Because the dot-product requires a sum over each entry in the vector:

$$\boldsymbol{\theta}^T \mathbf{x} = \sum_{m=1}^{M} \theta_m x_m \tag{A.3}$$

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

$$\boldsymbol{\theta}^T \mathbf{x} = \sum_{m \in \{1,\ldots,M\} \text{ s.t. } x_m \neq 0} \theta_m x_m \tag{A.4}$$

This requires only computation proportional to the number of non-zero entries in $\mathbf{x}$, which is $O(1)$ for Model 1 and Model 2. To ensure that your code runs quickly it is best to write the dot-product in the latter form (Equation (A.4)).

A.4 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.

To represent your feature vectors, you might need multiple data structures. First, you could create a shared mapping from a feature vector name (e.g. `cur:angeles` or `prev:BOS`) to the corresponding index in the dense parameter vector. This shared mapping could be created when the training data is first read-in and then stored for all later computation. Once you know the size of this mapping, you also know the size of the parmeter vector $\boldsymbol{\theta}$.

Another data structure should be used to represent the feature vectors themselves. Here there are several options:

1. For example, one could store exactly the sort of map from string feature names to their values shown in Tables A.3 and A.4. This has the advantage of being quite general and is human interpretable when printed out.
2. Another option would be to directly store a mapping of the integer index in the shared mapping (i.e. the index m) to the value of the feature x_m. This has the advantage of being more space efficient than the first option.
3. Lastly, one can take advantage of the fact that (as in the case of Model 1 and Model 2) we sometimes have only binary features. When this is the case, we can simply store a list of the indices m for which x_m is 1. All other indices are implied to have value x_m of 0. This is the most space efficient of all, but requires a bit more care if you are printing them out to ensure they look correct.

Any of the options above will likely ensure that your code runs fast so long as you are doing the $O(1)$ computation instead of the $O(M)$ version.

Note on out-of-vocabulary features One must be careful to *stop the growth* of the shared mapping after a single pass through the training data. Here's why: when constructing the test data, if a feature did not appear in the mapping, then it must map to some new feature with index $m > M$ where $|\boldsymbol{\theta}| = M$. In the dot-product this corresponds to multiplying x_m by some value θ_m which does not exist explicitly. Accordingly we say that $\theta_m x_m = 0$ whenever $m > M$. So, instead of adding the feature with index $m > M$ in the first place, we just *never add it at all*. This saves us from storing the otherwise useless feature.

A.5 Summary of the Learner

With the tools above, we can now construct a very efficient implementation of multinomial logistic regression. Your code should proceed through roughly the following steps to ensure consistency with our reference implementation:

1. Read in all the training data and use it to construct the shared mapping of feature strings to indices m. Stop the growth of the mapping.
2. Use the shared mapping to construct a label / feature vector representation $(\mathbf{x}^{(i)}, y^{(i)})$ of the training and validation data for either Model 1 or Model 2.
3. Proceed through the first epoch as follows: Iterate through each example in the training data *in or-*

der.[1] For each example, compute the example-specific gradient and update your parameters with one "stochastic gradient" step in $O(1)$ time. Use the fixed learning rate specified above. After all N examples have been visited in this way, you have completed an epoch.

4. Now compute the training and validation likelihood using the current parameters from the end of the previous epoch. Print these out to the metrics file.
5. Repeat the above two steps for each of the remaining epochs, where the number of epochs has been specified on the command line.
6. Use the shared mapping to construct a label / feature vector representation $(\mathbf{x}^{(i)}, y^{(i)})$ of the test data for either Model 1 or Model 2.
7. Use the parameters that were learned to make predictions on the training data set and (separately) the test data set. Write these predictions out to their corresponding files and report the train/test errors.

[1]For this and future assignments, we do not actually want SGD to be stochastic so that we can grade deterministic behavior of your code.

position t	**word** w_t	**tag** y_t
1	flight	O
2	from	O
3	columbus	B-fromloc.city_name
4	to	O
5	minneapolis	B-toloc.city_name
1	los	B-fromloc.city_name
2	angeles	I-fromloc.city_name
3	flights	O

Table A.1: Abstract representation of the input file format. Note that the input file format only *implicitly* specifies the position t within each sentence. Your code could compute this value t explicitly for each sentence. The ith row of this file will be used to construct the ith training example using either Model 1 features (Table A.3) or Model 2 features (Table A.4).

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$											
		cur:aardvark	...	cur:angeles	cur:columbus	cur:flight	cur:flights	cur:from	cur:los	cur:minneapolis	cur:to	...	cur:zebra
1	O	0	...	0	0	1	0	0	0	0	0	...	0
2	O	0	...	0	0	0	0	1	0	0	0	...	0
3	B-fromloc.city_name	0	...	0	1	0	0	0	0	0	0	...	0
4	O	0	...	0	0	0	0	0	0	0	1	...	0
5	B-toloc.city_name	0	...	0	0	0	0	0	0	1	0	...	0
6	B-fromloc.city_name	0	...	0	0	0	0	0	1	0	0	...	0
7	I-fromloc.city_name	0	...	1	0	0	0	0	0	0	0	...	0
8	O	0	...	0	0	0	1	0	0	0	0	...	0

Table A.2: Dense feature representation for Model 1 corresponding to the input file in Table A.1. The ith row corresponds to the ith training example.

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$
1	O	{ "cur:flight"=1 }
2	O	{ "cur:from"=1 }
3	B-fromloc.city_name	{ "cur:columbus"=1 }
4	O	{ "cur:to"=1 }
5	B-toloc.city_name	{ "cur:minneapolis"=1 }
6	B-fromloc.city_name	{ "cur:los"=1 }
7	I-fromloc.city_name	{ "cur:angeles"=1 }
8	O	{ "cur:flights"=1 }

Table A.3: Sparse feature representation for Model 1 corresponding to the input file in Table A.1.

i	**label** $y^{(i)}$	**features** $\mathbf{x}^{(i)}$
1	O	{ "cur:flight"=1, "prev:BOS"=1, "next:from"=1 }
2	O	{ "cur:from"=1, "prev:flight"=1, "next:columbus"=1 }
3	B-fromloc.city_name	{ "cur:columbus"=1, "prev:from"=1, "next:to"=1 }
4	O	{ "cur:to"=1, "prev:columbus"=1, "next:minneapolis"=1 }
5	B-toloc.city_name	{ "cur:minneapolis"=1, "prev:to"=1, "next:EOS"=1 }
6	B-fromloc.city_name	{ "cur:los"=1, "prev:BOS"=1, "next:angeles"=1 }
7	I-fromloc.city_name	{ "cur:angeles"=1, "prev:los"=1, "next:flights"=1 }
8	O	{ "cur:flights"=1, "prev:angeles"=1, "next:EOS"=1 }

Table A.4: Sparse feature representation for Model 2 corresponding to the input file in Table A.1.