Homework 4
Naive Bayes and Logistic Regression

Machine Learning (Spring 2020)
OUT: Mar. 21, 2020
DUE: Apr. 4, 2020, 11:55 PM

START HERE: Instructions

- **Late days:** The homework is due Wednesday April 4st, at 11:55PM. Late submissions will lead to -10 points punishment **per late day**, and **no submissions** are allowed for this homework after the "Accept Until" due day, which is April 7rd, at 11:55PM.
- **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 inspiration (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 and resources fully and completely (e.g., "Jane explained to me what is asked in Question 3.4" or "I found an explanation of conditional independence on page 17 of Mitchell's textbook"). Second, write up your solution independently: close the book and all of your notes, and send collaborators out of the room, so that the solution comes directly from you and you alone.
- **Programming:**
  - **Python:** You must write your code in Python.
  - **Autograding:** Problem 3 is partially autograded. Code file "evaluation.py" will tell you how well you are doing in each sub-problem. The teaching assistant will run your code. To make sure your code executes correctly on our machines, you should avoid using libraries which are not present in the *basic* python libraries and numpy.
- **Submitting your work:** All answers will be submitted electronically through the NYU Classes website.
  - **Writeup:** The write-up file should be in PDF format. We recommend that you use LaTeX. PDF files exported from doc/docx files are also accepted. Please do not submit hand-written solutions. The write-up file should be named as "solution.pdf"
  - **Code:** You should only make changes to "solution.py".
  - **Putting it all together:** Compress your submission *as a zip file.* and name it with your NetID. The teaching assistant will only look into "solution.pdf" and "solution.py".

Problem 1: Naive Bayes vs. Logistic Regression [15 pts]

Assume that we are performing a binary classification task with n samples. Each sample has p binary features $X_1, \ldots, X_p \in \{0, 1\}$, and a corresponding label $Y \in \{0, 1\}$.

1. For each of Naive Bayes and Logistic Regression briefly state:
   (a) The formula for the conditional likelihood, $P(Y = 1 \mid X_1, \ldots, X_p)$, assumed by each model. **[4 pts]**

   (b) The classification rule. **[2 pts]**

   (c) The parameters we have to estimate. **[2 pts]**

   (d) The method we use to do Maximum Likelihood Estimation (MLE) of the parameters. **[2 pts]**

2. Prove that we can write the Naive Bayes class distribution $P(Y = 1 \mid X_1, \ldots, X_p)$ in a form that matches the Logistic class distribution. To start, it will help to write $P(X_j = 1 \mid Y = 1) = \theta_j$. Hint: this makes $P(X_j = x \mid Y = 1) = \theta_j^x (1 - \theta_j)^{1-x}$. **[5 pts]**

Problem 2: Multi-class Logistic Regression [35 pts]

In class we learned about binary logistic regression. In this problem we consider the more general case where we have more than two classes. Let us denote the number of classes by C. Then, the conditional probability of the output class being c, given the input data point $\boldsymbol{x}$ is given by the following expression:

$$P(y = c \mid \boldsymbol{x}, \mathrm{W}) = \frac{e^{\boldsymbol{w}_c^\top \boldsymbol{x}}}{\sum_{c'=1}^{C} e^{\boldsymbol{w}_{c'}^\top \boldsymbol{x}}}, \tag{1}$$

where $c \in \{1, \dots, C\}$ and where W is a matrix whose rows are the weight vectors of each class, $\boldsymbol{w}_c$ for $c \in \{1, \dots, C\}$. During the training phase of the algorithm we are given a set of n input-output pairs, $\{\langle \boldsymbol{x}_i, y_i \rangle\}_{i=1}^n$ and we want to *learn* the values of the weight vectors for each class that maximize the conditional likelihood of the output labels, $\{y_i\}_{i=1}^n$, given the input data $\{\boldsymbol{x}_i\}_{i=1}^n$ and those weights, W. That is, we want to solve the following optimization problem:

$$\mathrm{W}^* = \underset{\mathrm{W}}{\operatorname{argmax}} \prod_{i=1}^{n} P(y_i \mid \boldsymbol{x}_i, \mathrm{W}).$$

Note here that we use a product over all training data points and not a joint probability because we assume that those data points are *independent and identically distributed (i.i.d.)*.

1. In class we discussed about a different form of multi-class logistic regression:

$$\begin{aligned} P(y = c \mid \boldsymbol{x}, \mathrm{V}) &= \frac{e^{\boldsymbol{v}_c^\top \boldsymbol{x}}}{1 + \sum_{c'=1}^{C-1} e^{\boldsymbol{v}_{c'}^\top \boldsymbol{x}}}, \\ P(y = C \mid \boldsymbol{x}, \mathrm{V}) &= \frac{1}{1 + \sum_{c'=1}^{C-1} e^{\boldsymbol{v}_{c'}^\top \boldsymbol{x}}}, \end{aligned} \tag{2}$$

   where $c \in \{1, \dots, C-1\}$ and where V is a matrix whose rows are the weight vectors of each class (except the last one), $\boldsymbol{w}_c$ for $c \in \{1, \dots, C-1\}$.

   (a) Show that **[1 pt]**

   $$\frac{e^x}{1 + e^x} = \frac{1}{e^{-x} + 1}$$

   (b) Show that **[1 pt]**

   $$\frac{e^{x_c}}{\sum_{c'=1}^{C} e^{x_{c'}}} = \frac{e^{x_c + \delta}}{\sum_{c'=1}^{C} e^{x_{c'} + \delta}}$$

   for any real number δ.

   (c) Show that equation 2 is equivalent to equation 1. Hint: Given $\boldsymbol{w}_c$, $c \in \{1, \dots, C\}$, what value should $\boldsymbol{v}_c$, $c \in \{1, \dots, C-1\}$ take to make them equivalent? **[2 pts]**

   (d) Explain why equation 2 is useful in programming. Hint: e^x may cause overflow if x is too large. **[1 pt]**

2. When we actually implement this algorithm, instead of maximizing the conditional likelihood of the training data, we choose to maximize the log of that quantity, namely the conditional log-likelihood.

   (a) Provide two (2) potential reasons why we might want to do that. **[5 pts]**

   (b) Explain why the solution we obtain by maximizing the log of a function is the same as the solution we obtain by maximizing the function itself. **[5 pts]**

3. In order to solve the optimization problem of the training phase we need to use some numerical solver. Most solvers require that we provide a function that computes the objective function value given some weights (i.e. the quantity within the "arg max" operator) and the gradient of that objective function (i.e. its first derivatives) and they take care of solving the problem for us. So, in order to implement the algorithm we need to derive those functions.

(a) Derive the conditional log-likelihood function for the multi-class logistic regression model. You may call that function $l(\mathrm{W})$. **[10 pts]**

(b) Derive the gradient of that function with respect to the weight vector of class c. That is, derive the value of $\nabla_{\boldsymbol{w}_c} l(\mathrm{W})$. You may call the gradient $\boldsymbol{g}_c(\mathrm{W})$. **[10 pts]**
Note: The gradient of a function $f(\boldsymbol{x})$ with respect to vector $\boldsymbol{x}$ is itself a vector, whose i^{th} entry is defined as $\frac{\partial f(\boldsymbol{x})}{\partial x_i}$, where x_i is the i^{th} element of vector $\boldsymbol{x}$.

Problem 3: Implementing Binary Logistic Regression [50 pts]

In this question you will code *Binary Logistic Regression*.

Binary Logistic Regression

The binary logistic regression is modeled as

$$\mathrm{P}(\mathrm{Y} = 1 \mid \mathbf{X} = \mathbf{x}; \mathbf{w}) \;=\; \sigma(\mathbf{w}^\top \mathbf{x})$$

where $\mathbf{x}$ is the feature vector, $\mathbf{w}$ is the weight vector, and σ is the logistic sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Both $\mathbf{x}$ and $\mathbf{w}$ are vectors in $\mathbb{R}^f$, where f is the number of features.

In the lecture, we learned that our goal is to **maximize** the Conditional Log Likelihood. This is equivalent to **minimize** the **Negative** Conditional Log Likelihood, and we can regard the Negative Conditional Log Likelihood a loss function:

$$\begin{aligned}\mathcal{L}(\mathbf{x}, y; \mathbf{w}) &= \;-\sum_{y \in \{0,1\}} y \cdot \log \mathrm{P}(\mathrm{Y} = y \mid \mathbf{X} = \mathbf{x}) \\ &= \;-y \cdot \log\left[\sigma(l)\right] - (1 - y) \cdot \log\left[1 - \sigma(l)\right]\end{aligned}$$

where $\mathcal{L}$ denotes this loss, and $l = \log\left[\frac{\mathrm{P}(Y=1|\mathbf{X}=\mathbf{x})}{\mathrm{P}(Y=0|\mathbf{X}=\mathbf{x})}\right] = \mathbf{w}^\top \mathbf{x}$ is often called the *logit*.

In a few weeks, you will learn that this $\mathcal{L}$ is a special case of Cross Entropy, and cross entropy will be the common loss function we use for deep learning. Therefore, we can again define:

$$\sigma\text{-xent}(l) = -y \cdot \log\left[\sigma(l)\right] - (1 - y) \cdot \log\left[1 - \sigma(l)\right]$$

as the *Sigmoid Cross Entropy* function. You will learn about the derivation soon (see Keith's note on NYU Classes if you are interested), but for this homework you can take this for granted.

Stochastic Gradient Descent

We will use Stochastic Gradient Descent (SGD) algorithm to optimize $\mathcal{L}(\mathbf{x}, y; \mathbf{w})$.

The basic idea of SGD is that, we randomly take a data point $(\mathbf{x}, y)$ from the dataset, calculate its loss function, get the derivative of the weight vector $\mathbf{w}_t$:

$$\mathbf{g}_{\mathbf{w}_t} = \nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}, y; \mathbf{w}_t)$$

and update $\mathbf{w}$ using the derivative:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \mathbf{g}_{\mathbf{w}_t}$$

where hyper-parameter λ is called the *learning rate*. (Note that in the lecture, the derivative is computed with respect to individual dimensions of $\mathbf{w}$, and here we are just using the vector notation, which is more compact. You should be able to understand both.)

Batching

In practice, we usually take a batch of data points instead of only one. Suppose that we take s (named as *batch size*) data points $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_s, y_s)$. We will calculate the average loss

$$\mathcal{L}(\mathbf{x}_{1:s}, y_{1:s}; \mathbf{w}_t) = \frac{1}{s} \sum_{i=1}^{s} \mathcal{L}(\mathbf{x}_i, y_i; \mathbf{w}_t)$$

and its derivative with respect to $\mathbf{w}_t$:

$$\mathbf{g}_{\mathbf{w}_t} = \nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}_{1:s}, y_{1:s}; \mathbf{w}_t) = \frac{1}{s} \sum_{i=1}^{s} \nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}_i, y_i; \mathbf{w}_t)$$

The updating rule stays the same.

Back-propagation

Calculating the derivatives is in general a difficult task. In the lecture, we can compute the derivative all at once by hands because the logistic form is not very complicated. For more complicated forms, the common solution is *backpropagation* algorithm. Backpropogation makes the computation easier by taking advantage of the chain rule and breaking the derivative into multiple, simpler parts. When we apply backpropagation to $\mathcal{L}(\mathbf{x}_{1:s}, y_{1:s}; \mathbf{w}_t)$, we have:

$$\nabla_{\mathbf{w}_t} \mathcal{L}(\mathbf{x}_{1:s}, y_{1:s}; \mathbf{w}_t) = \sum_{i=1}^{s} \frac{\partial \sigma\text{-xent}(l_i)}{\partial l_i} \cdot \nabla_{\mathbf{w}_t} l_i$$

Here, we will calculate $\frac{\partial \sigma\text{-xent}(l_i)}{\partial l_i}$ and $\nabla_{\mathbf{w}_t} l_i$ separately, and product-sum them up. You will learn about the general form of backpropagation in the second half of this semester, and now you can take them for granted.

Epochs

The process that we go over the entire dataset batch by batch and update the weight vector is called an *epoch* (it was called an "iteration" in class, however, I will use "epoch" to be consistent with the deep learning community).

As epochs grow, the loss decreases. The number of epochs for training is a hyper-parameter and you may change it to find the best configuration for the task you are working on.

Submission

Remember to follow the detailed coding and submission instructions at the top of this file!

- **SUBMISSION CHECKLIST**
  - Submission executes on our machines in less than 1 minutes. Each second exceeded will lead to 2 points for punishment.
  - Submission for this problem should only contain "solution.py'.
  - Your code file should be smaller than 1MB.

Sub-problems

1. **Linear Layer [6 pts]**
   Implement a batched version of linear transformation:

   $$l_i = \mathbf{w}^\top \mathbf{x}_i$$

   in class LinearLayerForward, method __call__(self, weights, xs, ctx=None).

   Parameters:

   - weights: $\mathbf{w}$. It is a numpy array of shape $(s,)$
   - xs: batched version of feature vector $\mathbf{x}$. It is a numpy array with shape (s, f). xs[i] represents $\mathbf{x}_i$.
   - ctx: an optional parameter used to store values necessary for calculating the gradient. It is an empty python dict. It is up to you what should be put in it. The modified version of ctx will be passed to LinearLayerBackward.
     Note: When the forward pass is called without further need to calculate the gradient (such as in prediction), ctx is None. Your implementation should be able to handle this case.

   Returned value:

   - logits: batched version of logits. It should be a numpy array with shape $(s,)$. logits[i] represents l_i.

2. **Sigmoid Cross Entropy [9 pts]**
Implement a batched version of σ-xent$(\cdot)$ function

$$\mathcal{L}(l_{1:s}) = \frac{1}{s}\sum_{i=1}^{s}\mathcal{L}(l_i) = \frac{1}{s}\sum_{i=1}^{s}\sigma\text{-xent}(l_i)$$

in class SigmoidCrossEntropyForward, method __call__(self, logits, ys, ctx=None).

Parameters:

- logits: batched version of logits. It is exactly what LinearLayerForward returns.
- ys: batched version of ground truth y. It is a numpy array with shape $(s,)$. ys[i] represents y_i.
- ctx: an optional parameter used to store values necessary for calculating the gradient. It is an empty python dict. It is up to you what should be put in it. The modified version of ctx will be passed to SigmoidCrossEntropyBackward.

Returned value:

- average_loss: $\mathcal{L}(l_{1:s})$. It should be a scalar.

Note: Be careful that e^x may cause overflow problem. You may solve the overflow problem by using both the left hand side and right hand side of Problem 2-1-(a).

3. **Gradient of Sigmoid Cross Entropy [9 pts]**
Get the derivative of logits

$$g_{l_i} = \frac{\partial \sigma\text{-xent}(l_i)}{\partial l_i}$$

in class SigmoidCrossEntropyBackward, method __call__(self, ctx, dloss).

Parameters:

- ctx: a python dict produced in SigmoidCrossEntropyForward.
- dloss: a scalar. It is always 1.0. Do not ask why here, you will know that next month.

Returned value:

- dlogits: batched version of logits' derivatives. It should be a numpy array with shape $(s,)$. dlogits[i] represents g_{l_i}.

Note: Be careful again that e^x may cause overflow problem.

4. **Gradient of Linear Layer [6 pts]**
Get the derivative of the weight vector

$$\mathbf{g_w} = \sum_{i=1}^{s} g_{l_i} \cdot \nabla_{\mathbf{w}_t} l_i$$

in class LinearLayerBackward, method __call__(self, ctx, dlogits).

Parameters:

- ctx: a python dict produced in LinearLayerForward.
- dlogits: batched version of logits' derivatives, g_{l_i}. It is exactly what SigmoidCrossEntropyBackward returns.

Returned value:

- dw: the derivative of the weight vector, $\mathbf{g_w}$. It should be a numpy array with shape $(f,)$.

5. **Update Function [5 pts]**
   Update the weight vector
   $$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \mathbf{g}_{\mathbf{w}_t}$$
   in class LinearLayerUpdate, method __call__(self, weights, dw, learning_rate=1.0).

   Parameters:

   - weights: the weight vector before being updated, $\mathbf{w}_t$. It is a numpy array with shape $(f,)$.
   - dw: the derivative of the weight vector, $\mathbf{g}_{\mathbf{w}_t}$. It is a numpy array with shape $(f,)$.
   - learning_rate: the learning rate, λ.

   Returned value:

   - new_weights: the updated weight vector, $\mathbf{w}_{t+1}$. It should be a numpy array with shape $(s,)$.

   Hint: This function is extremely simple. This problem is a point-giving problem.

6. **Logistic Regression Classifier with Logits [5 pts]**
   Make email classification in class Prediction, method __call__(self, logits).

   Parameters:

   - logits: batched version of logits. It is a numpy array with shape $(s,)$. logits[i] represents the logit value of the i^{th} email in the batch.

   Returned value:

   - predictions: batched version of classification results. It should be a **boolean** numpy array with shape $(s,)$. predictions[i] represents the classification result of the i^{th} email in the batch.

   Hint: This function is extremely simple. This problem is a point-giving problem.

7. **Tuning Hyper-parameters and Report Your Test Error Rate [10 pts]**
   Tuning hyper-parameters in python dict, opts. These hyper-parameters are

   - threshold: the threshold for filtering word list.
   - num_epochs: the number of epochs to train the model.
   - batch_size: the batch size s.
   - init_weight_scale: the standard deviation $\sigma_{\mathbf{w}}$ for the Gaussian distribution $\mathcal{N}(0, \sigma_{\mathbf{w}})$ from which each dimension of the initial weight vector is sampled.
   - learning_rate: the learning rate λ.

   By running "validation.py", you will train the model on the training set, and for each epoch, you will get the average training loss, the error rate on the training set, and the error rate on the validation set. **You should tune the hyper-parameters until the error rate (for the last epoch) on the validation set goes down to zero.** Please report all combinations of hyper-parameters you tried and their corresponding error rate on the validation set.

   After hyper-parameter tuning, by using the latest combination of hyper-parameters and running "test.py", you will train the model on the combination of the training set and the validation set, and you will get the error rate on the test set. Please report the final test error rate.

Evaluation

You may use “evaluation.py” to evaluate your solutions. Grading would also based on the result produced by “evaluation.py”. For each sub-problem, if your program pass the test and the grader (which is a human) find your code is legal (by looking into it), you will get full points for that sub-problem.

For each sub-problem,

- **Linear Layer**: If you see “LinearLayerForward test succeeds.”, your code pass the test.
- **Gradient of Linear Layer**: If you see “LinearLayerBackward test succeeds.”, your code pass the test.
- **Update Function**: If you see “LinearLayerUpdate test succeeds.”, your code pass the test.
- **Sigmoid Cross Entropy**: If you see “SigmoidCrossEntropyForward test succeeds.”, your code pass the test.
- **Gradient of Sigmoid Cross Entropy**: If you see “SigmoidCrossEntropyBackward test succeeds.”, your code pass the test.
- **Logistic Regression Classifier with Logits**: If you see “Prediction test succeeds.”, your code pass the test.
- **Tuning Hyper-parameters and Report Your Test Error Rate**: If you see “Model test succeeds.”, your code pass the test.