CS 4824 / ECE 4424 HW 2 Programming Portion Due: March 10, 2020

General Instructions: You only need to upload the Python file: `linearclassifier.py`, that you are required to modify for this assignment. For this assignment, you will run an experiment comparing two different types of linear classifiers for multi-class classification. The two models you will implement are perceptron and logistic regression. We have covered both these models in the class in the context of binary classification. We will be using their multi-class variants in this assignment. We first introduce some notations and then discuss the two models:

Notations We are given a training set comprising of N labeled records, $\mathcal{D} = \{(\boldsymbol{x}_j, y_j)\}_{j=1}^{N}$, where $\boldsymbol{x}_j$ denotes the input vector on the j^{th} record and $y_j \in \{1, 2, \ldots, C\}$ denotes its class label that can take any one of the C values. The size of $\boldsymbol{x}_j$ is equal to $(d+1) \times 1$, where d is the number of input attributes such that the first d elements of $\boldsymbol{x}_j$ correspond to the attributes while the last element is always equal to 1 (to account for the bias term of linear classifiers as we will see later). In matrix notations, we can write the set of input vector as $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N]$.

For both the linear classifiers, our goal is to learn weight parameters $\boldsymbol{w}_c$ for every class label c, where $\boldsymbol{w}_c$ is a vector of size $(d+1) \times 1$, where the first d elements correspond to the weights for input attributes while the last element coresonds to the bias term. The weight vectors across all class labels can be represented in a matrix forrm as $W = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_C]$. We can also conveniently flatten the weight matrix W in column-major order (using the NumPy function `W.ravel(order='F')`) to obtain:

$$\boldsymbol{w}_{\text{flat}} = \text{flat}(W) = \begin{bmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ \vdots \\ \boldsymbol{w}_C \end{bmatrix}, \tag{1}$$

where $\boldsymbol{w}_{\text{flat}}$ is a vector of length $C(d+1) \times 1$. Given the weight matrix W, we can make predictions on a data record, $\boldsymbol{x}_j$, using the following prediction function:

$$\hat{y}_j = \underset{c \in \{1,\ldots,C\}}{\arg\max}\ \boldsymbol{w}_c^\top \boldsymbol{x}_j, \tag{2}$$

where $a^\top$ represents the transpose of a. While both the linear classifiers, perceptron and logistic regression, use the same prediction function for making multi-class classification decisions, they differ in their approach for learning the model parameters W given the training set, as described in the following.

Multi-class Perceptron In the multi-class variant of the perceptron classifier, we consider an iterative algorithm for learning the weight matrix W based on the following update equation on a certain epoch at a training record, $\boldsymbol{x}_j$:

$$\boldsymbol{w}_{y_j} \leftarrow \boldsymbol{w}_{y_j} + \lambda \times \boldsymbol{x}_j \tag{3}$$

$$\boldsymbol{w}_{\hat{y}_j} \leftarrow \boldsymbol{w}_{\hat{y}_j} - \lambda \times \boldsymbol{x}_j \tag{4}$$

where λ is a hyper-parameter correspond to the learning rate of the algorithm.

Multi-class Logistic Regression In muli-class logistic regression, the probability that a data record $\boldsymbol{x}_j$ belongs to class label i is given by:

$$p_{ij} = Pr(y_j = i | \boldsymbol{x}_j, W) = \frac{\exp(\boldsymbol{w}_i^\top \boldsymbol{x}_j)}{\sum_{c=1}^{C} \exp(\boldsymbol{w}_c^\top \boldsymbol{x}_j)}, \tag{5}$$

where the matrix $P = [p_{ij}]$ stores the probabilities across all C classes and for all N training records. Note that $\sum_i p_{ij} = 1$. The negative log-likelihood of the training set given weight matrix W is then given by:

$$E(W) = -\sum_{i=1}^{C}\sum_{j=1}^{N} \mathbb{I}(y_j = i) \times \log(p_{ij}), \tag{6}$$

$$= \sum_{j=1}^{N} \log\left(\sum_{c=1}^{C} \exp(\boldsymbol{w}_c^\top \boldsymbol{x}_j)\right) - \sum_{j=1}^{N} \boldsymbol{w}_{y_j}^\top \boldsymbol{x}_j, \tag{7}$$

where $\mathbb{I}(a)$ represents the indicator function which is equal to 1 only if the condition a is true. The gradient of the negative log likelihood is then given by:

$$\nabla_{\boldsymbol{w}_i} E(W) = \sum_{j=1}^{N} \boldsymbol{x}_j \left(\frac{\exp(\boldsymbol{w}_i^\top \boldsymbol{x}_j)}{\sum_{c=1}^{C} \exp(\boldsymbol{w}_c^\top \boldsymbol{x}_j)} - \mathbb{I}(y_j = i)\right) \tag{8}$$

$$= \sum_{j=1}^{N} \boldsymbol{x}_j \left(p_{ij} - \mathbb{I}(y_j = i)\right) \tag{9}$$

While the Newton-Raphson optimization algorithm discussed in the class for learning W for logistic regression requires the first-order derivatives $\nabla_{\boldsymbol{w}_i} E(W)$ as well as the second-order Hessian matrix, here we consider a simplified "quasi-Newton" optimization algorithm that only requires the first-order derivatives and approximates the Hessian without explicitly computing it. This algorithm is referred to as the BFGS algorithm, after the names of its inventors, Broyden-Fletcher-Goldfarb-Shanno[1]. You do not need to know the details of how BFGS works for the purpose of this assignment. You can the BFGS optimizer using the SciPy package `minimize(method='BFGS')`, to learn W given a formula for the gradient of the negative log likelihood.

Experiments

You'll build the learning algorithms and prediction functions for perceptron and logistic regression models. You should be able to run experiments on synthetic data and on real medical data. These experiments are already set up for you in the two iPython notebooks: `synthetic_experiments.ipynb` and `cardio_experiments.ipynb`. The cardiotocography data includes measurements from fetal cardiotocograms that medical experts diagnosed into 10 possible diagnosis classes.[2] Both experiments will measure the results for the perceptron and logistic regression models on the two datasets and see how the results of the perceptron model change as we run multiple epochs of learning.

You are required to use Python 3.6 or higher for this assignment. We will grade your code in a Python 3.7 environment (most likely 3.7.3).

Tasks

1. (4 points) There are three unfinished functions in `linearclassifier.py` that you need to complete. The first function you should finish is `linear_predict`, which takes a set of data and a model object as input and outputs the predicted labels for each example. This linear prediction should be done with matrix operations and should take approximately two to five lines of code. This function will be used as the predictor function for both learning algorithms.

   The `model` is a `dict` that contains only one entry. The key is `'weights'`, and the value is the weight matrix for the multi-class linear classifier.

2. (4 points) Implement `perceptron_update`, which should run one perceptron learning step. It receives as input **one** example and its label and modifies the `model` object in place. In other words, the function should change the weights of the `model` dictionary to the new weights after applying the perceptron update equation. Finally, the function should return whether the perceptron was correct on

[1] see: `https://en.wikipedia.org/wiki/Broyden-Fletcher-Goldfarb-Shanno_algorithm`

[2] `https://archive.ics.uci.edu/ml/datasets/Cardiotocography`

the given example. Your `perceptron_update` function should update the numpy matrix stored in `model['weights']` *in-place*. Once you complete this task, you should be able to run the perceptron experiments. The notebooks should generate plots of the training and testing accuracy as you run multiple epochs of learning, as well as display the training and test accuracy values at the final epoch.

3. (7 points) Implement `log_reg_nll`, which is a function defined inside `log_reg_train` that returns the negative log likelihood as well as its gradient with respect to model weights. The function `log_reg_nll` returns a length-two tuple containing (1) the value of the negative log-likelihood—i.e., Equation (7)—and (2) the gradient of the negative log-likelihood with respect to the model weights—i.e., Equation (9).

   The first cell of the logistic regression portion in each notebook tests whether your objective function and gradient agree with each other by comparing your provided gradient with a numerically estimated gradient of your objective function. Use that to validate that you implemented the function correctly.

   Make sure to read and understand the other code in `log_reg_train`, which uses the function and gradient you compute and passes it into a general purpose numerical optimization tool to perform weight updates using the BFGS algorithm. You technically won't be graded on your understanding of this part for this homework, but it's an important concept in machine learning that will come up again.

   Once you complete this task, you should be able to run the logistic regression experiment. The notebooks should display the training and test accuracy values of the logistic regression model.

Note on preventing numerical overflow. In some cases, when you try to calculate $\frac{\exp(s_i)}{\sum_j \exp(s_j)}$, one or more of the $\exp$ operations may overflow. Here are two ways to handle this:

(a) You can subtract a constant from each of the scores. Let a be some constant. Then mathematically, $\frac{\exp(s_i)}{\sum_j \exp(s_j)} \equiv \frac{\exp(s_i - a)}{\sum_j \exp(s_j - a)}$. If you use this method, a good choice for a is $a = \max_i s_i$ because then the maximum value of all the $\exp(s_i - a) = \exp(0) = 1$, and you can be sure that none of these overflow. (Some may underflow, but that's okay.)

(b) Another approach is to take advantage of the fact that I gave you the logsumexp function (which basically does something very similar to option (a). You can calculate the whole fraction in log space. Taking the log of the expression, you get

$$\log\left(\frac{\exp(s_i)}{\sum_j \exp(s_j)}\right) = \log\exp(s_i) - \log\left(\sum_j \exp(s_j)\right) = s_i - \text{logsumexp}(\boldsymbol{s}),$$

where the last expression is the logsumexp operation run on the full vector $\boldsymbol{s} = [s_1, \ldots, s_C]$.

Overflow should only really happen if the weights or features are too large in magnitude. Sometimes the optimization may explore weight values that are too large, so having these tricks to prevent overflow can be helpful, but other times the optimization never tries these large values, so adding them just overcomplicates your code.