CS542 (Fall 2023) Programming Assignment 1
Logistic Regression Classifier

Due September 22, 2023

You are given `logistic_regression.py`, and `movie_reviews.zip`, the NLTK movie review corpus. Reviews are separated into a training set (80% of the data) and a development set (10% of the data). A testing set (10% of the data) has been held out and is not given to you. Within each set, reviews are sorted by sentiment (positive/negative). The files are already tokenized. Each review is in its own file. You are also given `movie_reviews_small.zip`, the toy corpus from HW1, in the same format.

You will need to use Numpy for this assignment. A description of useful Numpy functions is in the Appendix.

Assignment

Your task is to implement a logistic regression classifier. Specifically, in `logistic_regression.py`, you should fill in the following functions:

- `load_data(self, data_set)`: This function should, given a folder of documents (training, development, or testing), return a list of `filenames`, and dictionaries of `classes` and `documents` such that:
  - `classes[filename]` = class of the document
  - `documents[filename]` = feature vector for the document

  You may find it helpful to store the classes in terms of their indices, rather than their names; you can use `self.class_dict` to translate between class names and indices. To get the feature vector for a document, you can use the `featurize` function, described below.

- `featurize(self, document)`: This function should, given a document (as a list of words), return a feature vector. Note that, letting $|F|$ be the number of features, this function returns a vector of length $|F| + 1$. Furthermore, the last element of the vector should always be 1. If we consider our parameter vector `self.theta` to have form $\begin{bmatrix} w_1 & \dots & w_n & b \end{bmatrix}$ and our feature vector to have form $\begin{bmatrix} x_1 & \dots & x_n & 1 \end{bmatrix}$, then we can see that:

  $$\begin{bmatrix} x_1 & \dots & x_n & 1 \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \\ b \end{bmatrix} = \sum_{j=1}^{n} x_j w_j + 1 \times b = \mathbf{x} \cdot \mathbf{w} + b$$

  In this way, the last element of the vector, corresponding to the bias, is a "dummy feature" with value 1.

  What features should you use? You can start with word count features, as in the Naive Bayes example and HW1. You may find it helpful, especially if you are using a lot of word count features, to use `self.feature_dict` to translate between feature names and indices, but it is not required. Furthermore, you are not limited to word count features either. Examples of other feature types you may use include:

  - Lexicon count features, e.g. counts of words in positive or negative sentiment lexicons. The only thing you cannot do is to use an outside sentiment lexicon; we want you to do your own feature selection .
  - Binary word features, e.g. for some word, 1 if the word is in the document, 0 otherwise.
  - Document statistics, e.g. the total number of words in the document.
  - Any other features you can think of. Be creative!

- `train(self, train_set, batch_size=3, n_epochs=1, eta=0.1)`: Given a folder of training documents, this function loads the dataset (using the `load_data` function), splits the data into mini-batches (`minibatch` being a list of filenames in the current mini-batch), and shuffles the data after every epoch; these tasks have already been done for you. Your task is to implement the following for each mini-batch:

  1. Create and fill in a matrix $\mathbf{x}$ and a vector $\mathbf{y}$. Letting m be the number of documents in the mini-batch[1], $\mathbf{x}$ should have shape $(m, |F|+1)$ and $\mathbf{y}$ should have shape $(m,)$. Intuitively, $\mathbf{x}$ consists of the feature vectors for each document in the mini-batch, stacked on top of each other, and $\mathbf{y}$ contains the classes for those documents, in the same order. In other words, $\mathbf{x}$ and $\mathbf{y}$ should have the following forms:
  $$\mathbf{x} = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} & 1 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

  2. Compute $\hat{\mathbf{y}} = \sigma(\mathbf{x} \cdot \theta)$. The sigmoid function σ has already been implemented for you. Note the order of the operands $\mathbf{x} \cdot \theta$ (rather than $\theta \cdot \mathbf{x}$). To see why this should be the case, consider the shapes of $\mathbf{x}$ and θ: $(m, |F|+1)$ and $(|F|+1,)$, respectively. $\sigma(\mathbf{x} \cdot \theta)$ results in an array of shape $(m,)$, corresponding to $\hat{y}^{(i)}$ for each document i in the mini-batch. $\theta \cdot \mathbf{x}$, meanwhile, raises an error because their shapes are not aligned.

  3. Update the cross-entropy loss. Recall that the loss (for a single example) is $L_{CE}(\hat{y}, y) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$. We want to calculate the average train loss over the whole training set, but since we are dividing by the number of training documents at the end, it is okay to just keep a running sum.

  4. Compute the average gradient $\nabla L = \dfrac{1}{m}\Big(\mathbf{x}^T \cdot (\hat{\mathbf{y}} - \mathbf{y})\Big)$. This equation may look different from other gradient equations you have seen, so let us take a closer look. First, what is $\mathbf{x}^T \cdot (\hat{\mathbf{y}} - \mathbf{y})$?

[1] Note that this is not necessarily equal to the `batch_size`; if the `batch_size` does not evenly divide the number of training documents, one of the mini-batches will be smaller than the others.

$$
\begin{bmatrix} x_1^{(1)} & \dots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_n^{(1)} & \dots & x_n^{(m)} \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \hat{y}^{(1)} - y^{(1)} \\ \vdots \\ \hat{y}^{(m)} - y^{(m)} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) x_1^{(i)} \\ \vdots \\ \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) x_n^{(i)} \\ \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) \end{bmatrix}
$$

$$
= \begin{bmatrix} \displaystyle\sum_{i=1}^m \left(\frac{\partial L}{\partial w_1}\right)^{(i)} \\ \vdots \\ \displaystyle\sum_{i=1}^m \left(\frac{\partial L}{\partial w_n}\right)^{(i)} \\ \displaystyle\sum_{i=1}^m \left(\frac{\partial L}{\partial b}\right)^{(i)} \end{bmatrix}
$$

$$
= \sum_{i=1}^m (\nabla L)^{(i)}
$$

It computes the sum of the gradients for each document i in the mini-batch! Then to get the average gradient, we just divide by m, the number of documents in the mini-batch.

5. Update the weights (and bias). We can use the equation $\theta_{t+1} = \theta_t - \eta \nabla L$.

As a general note, you should avoid unnecessary for loops, by taking advantage of Numpy *universal* functions, that automatically operate element-wise over arrays. These include arithmetic operations such as `-`, `*`, and `numpy.log`, as well as functions that use them, like `sigma`. The last four steps can and should be done in one line of code each.

- `test(self, dev_set)`: This function should, given a folder of development (or testing) documents, return a dictionary of `results` such that:

  - `results[filename]['correct']` = correct class
  - `results[filename]['predicted']` = predicted class

  Again, you are free to store the classes in terms of their indices, rather than their names. You are also free to process the development/ testing documents one at a time, rather than in mini-batches. Recall that $P(y = 1|\mathbf{x}) = \hat{y}$. We will use 0.5 as the decision boundary; in other words:

  $$\text{predicted class} = \begin{cases} 1, & \text{if } P(y = 1|\mathbf{x}) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

- `evaluate(self, results)`: This function should, given the results of `test`, compute precision, recall, and F1 score for each class, as well as the overall accuracy, and print them in a readable format. Note that although you should fill in this function in order to have some idea of how well your model is performing, this function will *not* count toward your grade.

Hyperparameter tuning

In addition to changing your set of features, another way to improve the performance of your model is to tune your set of *hyperparameters*. Unlike parameters such as θ, which are optimized during training, the values of the hyperparameters are set beforehand. In this assignment, the hyperparameters are the `batch_size`, `n_epochs`, and learning rate `eta`.

You should experiment with different values for your hyperparameters. One possibility is to perform *grid search*: for each hyperparameter, choose a set of possible values. For example, you might test values of `batch_size` in $\{1, 2, 4, 8, 16, \ldots\}$, `n_epochs` in $\{1, 2, 4, 8, 16, \ldots\}$, and `eta` in $\{\ldots, 0.025, 0.05, 0.1, 0.2, 0.4, \ldots\}$. Then train and test your model with each combination of hyperparameter values. That being said, you can use any method you want to tune your hyperparameters. Turn in your model that performs best on the development set.

Write-up

You should also prepare a short write-up that includes at least the following:

- What additional features you included/tried in your classifier
- How you tuned the hyperparameters, and their final values
- Your evaluation results on the development set (you do not need to include any results on the toy data)

Submission Instructions

Please submit two files: your write-up (in PDF format), and `logistic_regression.py`. Do not include any data.

Appendix: Useful Numpy functions

You may find the following Numpy functions useful:

- `numpy.zeros(shape)`: Returns an array of given `shape`, filled with zeros.
- `numpy.log(x)`: If `x` is a number, returns the (natural) log of `x`. If `x` is an array, returns an array containing the logs of each element of `x`.
- `numpy.dot(a, b)`: Returns the product of `a` and `b`, where the type of product depends on the shapes of `a` and `b`.
- `numpy.argmax(a, axis=None)`: Given an array `a`, returns the argmax(es) along an axis. If no axis is given, returns the argmax over the entire array (flattening it into a vector if necessary).
- `numpy.sum(a, axis=None)`: Given an array `a`, returns the sum(s) over an axis. If no axis is given, returns the sum over the entire array.
- `numpy.trace(a)`: Given a matrix `a`, returns the sum along the diagonal.