Exploring Logistic Regression and Linear Classification

1 Introduction

In this assignment, you will implement logistic regression model on a toy dataset for binary classification and linear classification model on a toy dataset for multiclass classification.Your key goal in this assignment is to correctly implement these models and analyse the results you obtain.

Starter code and dataset are available here: https://github.com/ashutoshbsathe/cs725-hw

NOTE: You are not allowed to use any python package other than python standard packages and the others mentioned in requirements.txt.

2 Data

The `data/` directory from our repository contains our toy datasets. The `data/binary` directory contains dataset for binary classification. The `data/iris` directory contains dataset for multi-class classification. Each of these directories contains the training and validation data in the form of NumPy arrays.

The train-set source (input) is named as `train_x.npy` and the train-set target is named as `train_y.npy`. Similarly the validation-set source (input) is named as `val_x.npy` and the validation-set target is named as `val_y.npy`. For binary classification the number of features for each sample are 2 and the number of classes are also 2 while for multi-class classification the number of features for each sample are 4 and the number of classes are 3.

3 Starter Code

The starter code contains following files:

1. *args.py*: This file specifies the python classes for training arguments and visualization arguments. You can check all the arguemnts and their default values in this file. You should not modify this file.
2. *model.py*: Contains boilerplate implementation of logistic regression and linear classification. You must go through comments of each function to understand their expected behavior. For both the models, you need to implement `calculate_loss()`, `calculate_gradient()`, `update_weights()` and `get_prediction()` functions.
3. *train.py*: A minimal training loop that makes use of above functions. Also neatly saves training history and the weights that lead to best validation accuracy to a folder. You should not modify this file.
4. *train_with_visualization.py*: Similar to `train.py` above but also visualizes decision boundary of logistic regression after every gradient update.
5. *utils.py*: Various utlity functions used by other scripts. Do not use this file for defining your own utilities, make sure your implementation is contained entirely in `model.py` itself.
6. *evaluate_submission.py*: Use this after completing the assignment to get an idea of what accuracy and loss values the autograder will see with your models. Do note that the actual autograder will use the test split of both of these models which is not available to you during the assignment.

Specifically, your task is to complete the logistic regression and linear classifier implementation in *model.py*.

4 Report

After the implementation is complete, study the effects of various design choices of hyper-parameters like learning rate, number of training epochs, momentum etc. The values of these hyper-parameters can be set accordingly through argument passing while executing *train.py* on CLI. The exact command can be found on the github repository. In your report you must provide analysis of the following:

1. **Learning rate** **[4 marks (logistic regression) + 6 marks (linear classifier)]**
   – *Effect of the learning rate on the loss and accuracy.* – run the experiment with different values of the hyper-parameter *learning_rate* and analyse how does it affect the training and validation performances as measured by accuracy and loss. Every time you run an experiment with certain hyper-parameters the boiler-plate code will automatically save the plots of corresponding curves in the directory *log_dir*. You can monitor it there. The values of learning rate you need to try and test are 10^{-1}, 10^{-2}, 10^{-4} and 10^{-6}. These can be specified at the *train.py* CLI as `1e-1`, `1e-2`, `1e-4` and `1e-6` respectively.
2. **Number of epochs** **[4 marks (logistic regression) + 6 marks (linear classifier)]**
   – *Effect of the number of epochs on the loss and accuracy.* – with the best value you get for *learning_rate* above, run the experiment with different values of the hyper-parameter *num_epochs* and analyse how does it affect the training and validation performance as measured by accuracy and loss. The plots of corresponding curves for loss and accuracy can be found in the directory *log_dir* same as previous. The values of number of epochs you need to try and test are 100, 250, 500 and 1000.
3. **Momentum** **[2 marks (logistic regression) + 3 marks (linear classifier)]**
   – *Effect of the momentum on the loss and accuracy.* – with the best values you get for *learning_rate* and *num_epochs* above, run the experiment with different values of the hyper-parameter *momentum* and analyse how does it affect the training and validation performance as measured by accuracy and loss. The plots of corresponding curves for loss and accuracy can be found in the directory *log_dir* same as previous. Run the experiment with $(\mu - 0.9)$ and without $(\mu = 0)$ momentum and perform analysis.

The main focus of this investigation is to analyze the effect of hyper-parameter values on loss and accuracy. For all the questions, you must perform analysis for both datasets and provide comments about differences and similarities between the results you observe.

Apart from the above analysis you are also required to include answers to the following in the report.

1. Mathematical derivation of the gradient term for logistic regression which you must have used while implementation of logistic regression. **[2 marks]**
2. Study on iris. How to adopt logistic regression to multi-class setting. **[3 marks]**

5 Submission Structure

Once you are done with the assignment, submit (1) your best models for both the datasets (2) your code for reproducing the best results i.e. your *model.py* (3) the report with the results you obtain. Since the assignment will be autograded, it is important to maintain the submission structure as mentioned below:

```
submission/
    model.py
    best_binary.weights.npy
    best_iris.weights.npy
    report.pdf
```

Pack everything under `submission/` directory under a `.tar.gz` archive named as `rollno1_rollno2.tar.gz` and upload that to Moodle. Refer to the *README.md* on the repo for more concrete instructions.

6 Recall

For logistic regression with 2 classes, given a training pair (x_i, y_i), you can compute the loss as $L(w) = y_i \log p + (1 - y_i) \log(1 - p)$ where $p = \sigma(z) = \frac{1}{1+\exp(-z)}$ and $z = w^T x_i$. Here w are the trainable model parameters that must be updated according to the gradient $\partial L/\partial W$. Derive the equation of the gradient i.e. $\nabla L(w) = \partial L/\partial W$ and carefully implement it in NumPy. We can consider a multiclass linear classifier with m classes as m logistic regression classifiers with 1-vs-all strategy. Let's say $m = 3$ like we have for the *iris* dataset in the assignment, you will need to create 3 sets of weights each corresponding to one logistic regression. When computing gradient and loss for class 1, you would consider examples corresponding to classes 2 and 3 as negative examples. How can you compute the gradient and loss for every class? What will be the final label assigning strategy?

For updating your parameters, you can perform gradient descent as $w = w - \lambda \nabla L(w)$ where λ is the chosen learning rate. A modification to this update rule called "momentum update" enjoys better convergence rate. This requires maintaining an extra set of parameters called v. v is initialized to zero at the same time w is initialized. Then the actual update is written as follows:

$$v = \mu v - \lambda \nabla L(w)$$
$$w = w + v$$

This introduces μ as the additional "momentum" hyperparameter whose value is typically 0.9. Notice how at $\mu = 0$, the update rule reduces to $w = w - \lambda \nabla L(w)$. In question 3 as described above, you must experiment with $\mu = 0.9$ (with "momentum") and $\mu = 0$ (without "momentum" / normal gradient descent). Note that implementing "momentum" will also require slight modification to the `__init__()` method of both classes to include the additional v parameter.

More info about momentum update: https://cs231n.github.io/neural-networks-3/.