Assignment 2 Report

```
--./
|
|-- lsq.py: Plot for LSQ
|-- perceptron.py: Plot for Perceptron
|-- logistic.py: Experiment code for part 2
```

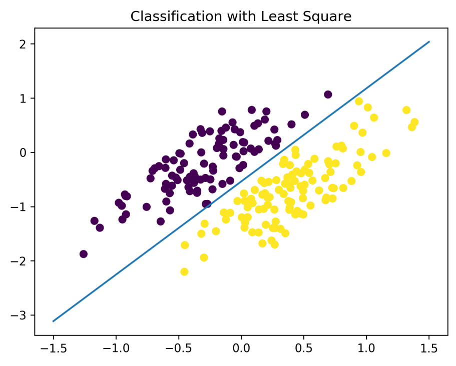
Here is the links of my answers.

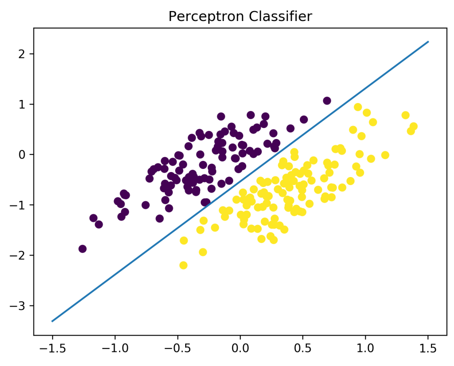
- Part 1
- Part 2

Least Square Classifier and Perceptron Algorithm

Accuracy:

- LSQ: 1.0
- Perceptron: 0.9 ~ 1.0 (Due to SGD)





Multiclass Logistic Regression

Notations:

- N is the number of the training set and K is the number of class.
- Upper $^{(n)}$ denotes the training data, e.g. $y_k^{(n)}$ represents value y_k in the n-th training data in the training set.
- Upper case character denotes a matrix, e.g. $\mathbf{W}$ is a weight matrix, which we want to learn by training.
- Lower case character in bold denotes a vector, e.g. $\mathbf{t}$ is a one-hot vector whose non-zero value denotes the corresponding type of an input $\mathbf{x}$.

To start with, given an input feature vector $\phi = \phi(\mathbf{x})$ and a weight matrix $\mathbf{W}$, our output $\mathbf{y}$ is a K-dimensional vector such that

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_k \\ \vdots \\ y_K \end{bmatrix}$$

where

$$y_k = p(\mathcal{C}_k|\mathbf{W}, \phi) = \frac{\exp(a_k)}{\sum_{j=1}^{K} \exp(a_j)}$$
$$a_k = \mathbf{w_k^T}\phi$$

Notice that $\sum_k y_k = 1$ since $\mathbf{y}$ is a softmax function with respect to $\mathbf{a}$. And the prediction t is given by $t = \arg\max_k y_k$.

The likelihood function, which we want to maximize, is given by

$$p(\mathbf{T}|\mathbf{W}, \Phi) = \prod_{n=1}^{N} p(\mathbf{t}^{(n)}|\mathbf{W}, \phi^{(n)}) = \prod_{n=1}^{N}\prod_{k=1}^{K} {y_k^{(n)}}^{t_k^{(n)}}$$

where $\mathbf{t}$ is the one-hot target vector, for a feature vector ϕ belongs to class $\mathcal{C}_k$, in which only t_k is 1. Taking the negative logarithm of it then gives the log likelihood function

$$E(\mathbf{W}) = -\ln p(\mathbf{T}|\mathbf{W}, \Phi) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \ln y_k^{(n)}$$

which is also known as **cross-entropy** error function for the multi-class classification problem. Now we need to find the minimum of it using the method of Gradient Descent. Before that we must get to know the gradient of the error function, which is solved as follows.

Computation of derivatives

We first take the derivatives of y_k with respect to a_j, which is given by

$$\frac{\partial y_k}{\partial a_j} = \begin{cases} \frac{\exp(a_k)(\Sigma - \exp(a_j))}{\Sigma^2} & k = j \\ \frac{-\exp(a_k)\exp(a_j)}{\Sigma^2} & k \neq j \end{cases}$$

where $\Sigma = \sum_{j=1}^{K} \exp(a_j)$.

Notice that the result above can be generalized into such form

$$\frac{\partial y_k}{\partial a_j} = y_k \cdot (I_{kj} - y_j)$$

where I_{kj} are the elements of the identity matrix. Using this, we can then obtain

$$\frac{\partial E}{\partial w_j} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \frac{\partial \ln y_k^{(n)}}{\partial w_j} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \frac{\partial \ln y_k^{(n)}}{\partial y_k^{(n)}} \frac{\partial y_k^{(n)}}{\partial a_j^{(n)}} \frac{\partial a_j^{(n)}}{\partial w_j} = -\sum_{n=1}^{N}\sum_{k=1}^{K} \frac{t_k^{(n)}}{y_k^{(n)}} \cdot y_k^{(n)} \cdot (I_{kj} - y_j^{(n)}) \cdot \phi^{(n)} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \cdot (I_{kj} - y_j^{(n)}) \cdot \phi^{(n)}$$

Further more, in a more precise form we get

$$\frac{\partial E}{\partial w_j} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \cdot (I_{kj} - y_j^{(n)}) \cdot \phi^{(n)} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} I_{kj} \cdot \phi^{(n)} - \sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} y_j^{(n)} \cdot \phi^{(n)} = -\sum_{n=1}^{N} t_j^{(n)} \cdot \phi - \sum_{n=1}^{N} y_j^{(n)} \cdot \phi^{(n)} = -\sum_{n=1}^{N} (t_j^{(n)} - y_j^{(n)}) \cdot \phi^{(n)}$$

where we use $\sum_j t_j^{(n)} = 1$.

Finally, we find that the derivative is just the error times the input, as follows

$$\frac{\partial E}{\partial w_j} = \sum_{n=1}^{N} (y_j^{(n)} - t_j^{(n)}) \cdot \phi^{(n)}$$

If we add a regularization term, this will become

$$\frac{\partial E}{\partial w_j} = \sum_{n=1}^{N} (y_j^{(n)} - t_j^{(n)}) \cdot \phi^{(n)} + \lambda w_j$$

Note that the bias w_0 should not be regularized, since it just simply governs the bias and has nothing to do with overfitting.

Gradient Descent

- Good old Batch Gradient Descent works as follows

1. For each $(\mathbf{y}^{(i)}, \mathbf{t}^{(i)})$ in the training set, repeats the following update until converge, e.g. $||\nabla_w E|| < \epsilon$.

$$\begin{aligned}
\mathbf{W} := \mathbf{W} - \alpha \nabla_w E = \mathbf{W} - \alpha \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_K} \end{bmatrix} = \mathbf{W} - \alpha \begin{bmatrix} \frac{\partial E}{\partial w_{ij}} \end{bmatrix} \\
= \mathbf{W} - \alpha \begin{bmatrix} \mathbf{y}^{(1)} - \mathbf{t}^{(1)} & \dots & \mathbf{y}^{(N)} - \mathbf{t}^{(N)} \end{bmatrix} \begin{bmatrix} \phi^{(1)} \\ \vdots \\ \phi^{(N)} \end{bmatrix} \\
= \mathbf{W} - \alpha(\begin{bmatrix} \mathbf{y}^{(1)} & \dots & \mathbf{y}^{(N)} \end{bmatrix} - \begin{bmatrix} \mathbf{t}^{(1)} & \dots & \mathbf{t}^{(N)} \end{bmatrix}) \begin{bmatrix} \phi^{(1)} \\ \vdots \\ \phi^{(N)} \end{bmatrix}
\end{aligned}$$

- Stochastic Gradient Descent, which is more efficient and practical method, belows are some of its variants. The main idea is to pick only one sample from the trainning set to update $\mathbf{W}$, which is illustrated as follows.

  1. Shuffle the training set $\mathcal{T}$.
  2. For each $(\mathbf{y}^{(i)}, \mathbf{t}^{(i)})$ in $\mathcal{T}$, update $\mathbf{W}$ as follows

$$\mathbf{W} := \mathbf{W} - \begin{bmatrix} (y_1^{(i)} - t_1^{(i)})\phi^{(i)} \\ \vdots \\ (y_K^{(i)} - t_K^{(i)})\phi^{(i)} \end{bmatrix}$$

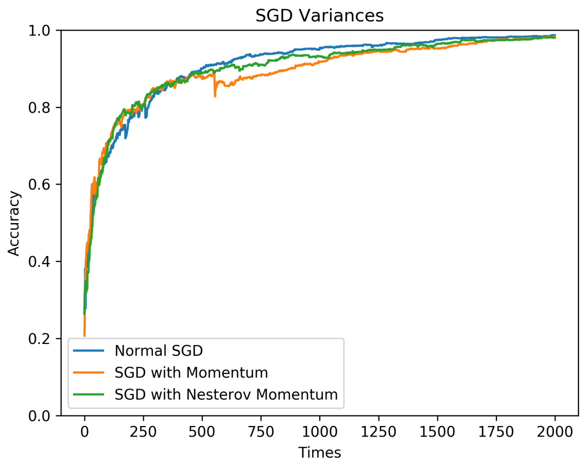
- Above is the most trivial verision of SGD. It's faster than BGD, and we can even do better in both efficiency and accuracy in practice, however. Here are two variances of SGD.

1. SGD with **Momentum**

   Intuition behind this variance can be derived from

2. SGD with **Nesterov Momentum**

> Nesterov Momentum is a slightly different version of the momentum update that has recently been gaining popularity.



- Mini-batch Gradient Descent takes some samples each time, not the whole in BGD or just single sample in SGD, to calculate the gradient, and then updates $\mathbf{W}$ as follows.
  1. Shuffle the training set $\mathcal{T}$.
  2. Divide $\mathcal{T}$ into `num_batch` groups, denoted by $\mathcal{T}_j$, which contains N_j samples.
  3. For each $\mathcal{T}_n$ in $\mathcal{T}$, update $\mathbf{W}$ as BGD, simply replacing N by N_i

$$
\begin{aligned}
\mathbf{W} :&= \mathbf{W} - \alpha \nabla_w E \\
&= \mathbf{W} - \alpha \begin{bmatrix} \mathbf{y}^{(1)} - \mathbf{t}^{(1)} & \dots & \mathbf{y}^{(N_i)} - \mathbf{t}^{(N_i)} \end{bmatrix} \begin{bmatrix} \phi^{(1)} \\ \vdots \\ \phi^{(N_i)} \end{bmatrix} \\
&= \mathbf{W} - \alpha(\begin{bmatrix} \mathbf{y}^{(1)} & \dots & \mathbf{y}^{(N_i)} \end{bmatrix} - \begin{bmatrix} \mathbf{t}^{(1)} & \dots & \mathbf{t}^{(N_i)} \end{bmatrix}) \begin{bmatrix} \phi^{(1)} \\ \vdots \\ \phi^{(N_i)} \end{bmatrix}
\end{aligned}
$$

For the requirements

1. Preprocessing
   - Process the text. I use `str.replace()` method to replace all useless character for a whitespace. And simply split them by space `str.split()` .
   - Get a dictionary for words **whose occurrence is greater than `min_count` and smaller than `max_count`** since words with extremely high frequency are also useless such as "the", "a". This can be done by run through the dataset, keeping trace of every word's occurrence and select only those legal words. A `dict` in python, with word as key and index as value, is suitable for this work to guarantee the efficiency since it's a hash table. Also store the index of words using another dictionary. `list()`, which converts `dict` into `list`, and `index()` are useful.
   - Run through the data again, output their multi-hot feature vector according to the index.

- Processing target vector is rather easy, just simply convert it into an one-hot vector.

2. The computation of $\frac{\partial E}{\partial w_j}$ has been shown above.

   (1) When adding L2 regularization, we should not regularize the bias term since it does't control the shape for the curve and has nothing to do with overfitting.

   (2) To check the correction of the gradient calculation, I have tried three different ways, whose mechanism behind is almost the same.

   One way is to focus on the value of the loglikelihood function $E(\mathbf{W})$, which is expected to descent if our gradient is correct.

   The second way is to trace the difference of $\mathbf{W}$ made by the gradient descent, i.e. the norm of the gradient itself, which can be measured by **Matrix Norms**, e.g. The Frobenius Matrix Norm $||A||_F = (\sum_i \sum_j |a_{ij}|^2)^{\frac{1}{2}}$. Intuitively, this value will descent and will also converge to 0 when $\mathbf{W}$ converges.

   Another more strict way is to use the centered formula:

$$\frac{\partial E}{\partial \mathbf{W}_i} \approx \frac{E(\mathbf{W}+\Delta\mathbf{W}_i) - E(\mathbf{W}-\Delta\mathbf{W}_i)}{2\Delta\mathbf{W}_i}$$

   Using this to test only on several dimensions.

   But in practice, I find it much easier to work with the matrix norms and accuracy.

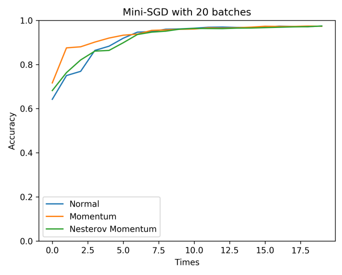
3. (1) If the learning rate is too small, gradient descent will be performed slowly. If the learning rate is too large, gradient descent may fail to converge, or even diverge.

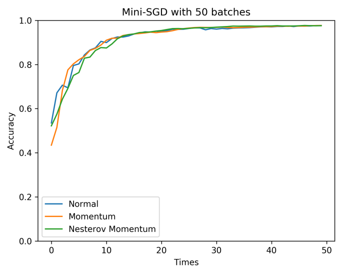
   (2) To determine when to terminate the training, I have tried two methods.

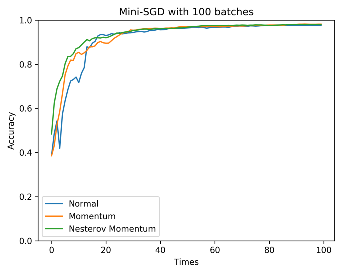
   - The first way is to simply to fix the number of epochs `num_epoch`. After repeating `num_epoch` epochs, GD stops.
   - Another is to set a threshold for $||\Delta\mathbf{W}||$, which measures the difference of $\mathbf{W}$ between updates. For simplicity, I use the Frobenius Matrix Norm. When $||\Delta\mathbf{W}|| < \epsilon$, it seems that $\mathbf{W}$ won't change much in the future. Thus the algorithm should be terminated.

4. (1) It is easy to verified that SGD and Mini-batch GD is much more faster than GD in the most cases. Among the three GD algorithms, mini-batch GD seems to have the best balance between efficiency and accuracy. Here is the plot with different optimizations and batches, the number of parts which the whole dataset are split into.

   We can see that the curves of momentum versions are **smoother** than normal mini-batch. But it seems that when batches is not large enough, i.e., like a full-batch, Nesteriv Momentum optimizations failed. On the contrary, if the number of batches is large enough, both variances of Mini-batch GD works better, since it's closer to a SDG, in which momentum can make a difference. Thus momentum is good at randomly gradient descent.







   (2) Pros and cons:

    - GD: It guarantees to decrease the loss in every update, however slow.
    - SGD: Faster than GD and slower than Mini-batch and, yet not stable.
    - Mini-batch: Has the best balance between efficiency and stability with proper `num_batch` and is most commonly used in practice. But adding another hyper-parameter make the model a little more complicate.

5. With learning rate $a = 0.1$ and $\lambda = 0.1$

   BGD with 100 epochs: 85%

   SGD with 2000 updates (about 1 epoch): 78% (Maybe slow to converge.)

   Mini-batch GD with 10 batches and 2000 updates: 86%

Reference

Multiclass classifier:

https://www.cs.toronto.edu/~urtasun/courses/CSC411_Fall16/07_multiclass.pdf

Gradient Descent:

https://towardsdatascience.com/difference-between-batch-gradient-descent-and-stochastic-gradient-descent-1187f1291aa1

SGD:

http://cs231n.github.io/neural-networks-3/#sgd

Momentum SDG:

https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d