CSCI 567, Spring'18
Michael Shindler

Programming Set #3

Due: 11:59 pm, March 18, 2018

General instructions

Due date and time March. 12th, 11:59pm

Starting point Your repository will have now a directory 'homework3/'. Please do not change the name of this repository or the names of any files we have added to it. Please perform a `git pull` to retrieve these files. You will find within it:

Download the data Please download **mnist_subset.json** from Piazza/Resource/Homework. Please **DO NOT** push it into your repository when you submit your results. Otherwise, you will get 20% deduction of your score of this assignment.

High-level descriptions

Dataset We will use **mnist_subset** (images of handwritten digits from 0 to 9). This is the same subset of the full MNIST that we used for Homework 1 and Homework 2. As before, the dataset is stored in a JSON-formated file **mnist_subset.json**. You can access its training, validation, and test splits using the keys 'train', 'valid', and 'test', respectively. For example, suppose we load **mnist_subset.json** to the variable x. Then, $x['train']$ refers to the training set of **mnist_subset**. This set is a list with two elements: $x['train'][0]$ containing the features of size N (samples) $\times D$ (dimension of features), and $x['train'][1]$ containing the corresponding labels of size N.

Tasks You will be asked to implement the linear support vector machine (SVM) for binary classification (Sect. 1). Specifically, you will

- finish implementing the following three functions—`objective_function`, `pegasos_train`, and `pegasos_test`—in `pegasos.py`. Refer to `pegasos.py` and Sect. 1 for more information.
- Run the script `pegasos.sh` after you finish your implementation. This will output `pegasos.json`.
- add, commit, and push (1) the `pegasos.py`, and (2) the `pegasos.json` file that you have created.

You will be asked to implement the boosting algorithms with decision stump (Sect. 2). Specifically, you will

- finish implementing the following classes — `boosting`, `decision_stump`, `AdaBoost` and `LogitBoost`. Refer to `boosting.py`, `decision_stump.py` and Sect. 2 for more information.
- Run the script `boosting.sh` after you finish your implementation. This will output `boosting.json`.
- Add, commit, and push (1) the `boosting.py`, (2) the `decision_stump.py`, and (3) the `boosting.json`.

You will be asked to implement the decision tree classifier (Sect. 3). Specifically, you will

- finish implementing the following classes — `DecisionTree`, `TreeNode`. Refer to `decision_tree.py` and Sect. 3 for more information.
- Run the script `decision_tree.sh` after you finish your implementation. This will output `decision_tree.json`.
- Add, commit, and push (1) the `decision_tree.py`, (2) the `decision_tree.json`

As in the previous homework, you are not responsible for loading/pre-processing data; we have done that for you (e.g., we have pre-processed the data to have to class: digits 0–4 and digits 5–9.). For specific instructions, please refer to text in Sect. 1 and 2 and the instructions in `pegasos.py` and `boosting.py`.

Cautions Please do not import packages that are not listed in the provided code. Follow the instructions in each section strictly to code up your solutions. **Do not change the output format**. **Do not modify the code unless we instruct you to do so**. A homework solution that does not match the provided setup, such as format, name, initializations, etc., **will not** be graded. It is your responsibility to **make sure that your code runs with the provided commands and scripts on the VM**. Finally, make sure that you **git add, commit, and push all the required files**, including your code and generated output files.

Problem 1 Pegasos: a stochastic gradient based solver for linear SVM

In this question, you will build a linear SVM classifier using the Pegasos algorithm [1]. Given a training set $\{(\mathbf{x}_n \in \mathbb{R}^D, y_n \in \{1, -1\})\}_{n=1}^N$, the primal formulation of linear SVM is as follows

$$\min_{\mathbf{w}} \frac{\lambda}{2}||\mathbf{w}||_2^2 + \frac{1}{N}\sum_n \max\{0, 1 - y_n \mathbf{w}^T \mathbf{x}_n\}. \tag{1}$$

Instead of turning it into dual formulation, we are going to solve the primal formulation directly with a gradient-base algorithm. *Note that here we include the bias term b into parameter* $\mathbf{w}$ *by appending* $\mathbf{x}$ *with* 1.

In (batch) gradient descent, at each iteration of parameter update, we compute the gradients for all data points and take the average (or sum). When the training set is large (i.e., N is a large number), this is often too computationally expensive (for example, a too large chunk of data cannot be held in memory). Stochastic gradient descent with mini-batch alleviates this issue by computing the gradient on a *subset* of the data at each iteration.

One key issue of using (stochastic) gradient descent to solve eq. (1) is that $\max\{0, z\}$ is not differentiable at $z = 0$. You have seen this issue in Homework 2, where we use the Heaviside function to deal with it. In this question, you are going to learn and implement Pegasos, a representative solver of eq. (1) that applies stochastic gradient descent with mini-batch and explicitly takes care of the non-differentiable issue.

The pseudocode of Pegasos is given in Algorithm 1. At the t-th iteration of parameter update, we first take a mini-batch of data A_t of size K `[step (3)]`, and define $A_t^+ \subset A_t$ to be the subset of samples for which $\mathbf{w}_t$ suffers a non-zero loss `[step (4)]`. Next we set the learning rate $\eta_t = 1/(\lambda t)$ `[step (5)]`. We then perform a **two-step parameter update** as follows. We first compute $(1 - \eta_t \lambda)\mathbf{w}_t$ and for all samples $(\mathbf{x}, y) \in A_t^+$ we add the vector $\frac{y\eta_t}{K}\mathbf{x}$ to $(1 - \eta_t\lambda)\mathbf{w}_t$. We denote the resulting vector by $\mathbf{w}_{t+\frac{1}{2}}$ `[step (6)]`. This step can also be written as $\mathbf{w}_{t+\frac{1}{2}} = \mathbf{w}_t - \eta_t \nabla_t$ where

$$\nabla_t = \lambda \mathbf{w}_t - \frac{1}{|A_t|}\sum_{(\mathbf{x},y)\in A_t^+} y\mathbf{x}$$

The definition of the hinge loss, together with the Heaviside function, implies that ∇_t is the gradient of the objective function on the mini-batch A_t at $\mathbf{w}_t$. Last, we set $\mathbf{w}_{t+1}$ to be the projection of $\mathbf{w}_{t+\frac{1}{2}}$ onto the set

$$B = \{\mathbf{w} : ||\mathbf{w}|| \leq 1/\sqrt{\lambda}\}$$

This is obtained by scaling $\mathbf{w}_{t+1}$ by $\min\{1, \frac{1/\sqrt{\lambda}}{||\mathbf{w}_{t+\frac{1}{2}}||}\}$ `[step (e)]`. For details of Pegasos algorithm you may refer to the original paper [1].

Now you are going to implement Pegasos and train a binary linear SVM classifier on the dataset **mnist_subset.json**. You are going to implement three functions—`objective_function`, `pegasos_train`, and `pegasos_test`—in a script named `pegasos.py`. You will find detailed programming instructions in the script.

1.1 Finish the implementation of the function `objective_function`, which corresponds to the objective function in the primal formulation of SVM.

1.2 Finish the implementation of the function `pegasos_train`, which corresponds to Algorithm 1.

Algorithm 1 The Pegasos algorithm

```
Require: A training set $S = \{(\mathbf{x}_n \in \mathbb{R}^D, y_n \in \{1, -1\})\}_{n=1}^N$, the total number of iterations $T$, the batch size $K$, and the regularization parameter $\lambda$.
Ensure: The last weight $\mathbf{w}_{T+1}$.
1: Initialization Choose $\mathbf{w}_1$ s.t. $||\mathbf{w}_1|| \leq \frac{1}{\sqrt{\lambda}}$.
2: for $t = 1, 2, \cdots, T$ do
3:     Randomly choose a subset of data $A_t \in S$, where $|A_t| = K$
4:     Set $A_t^+ = \{(\mathbf{x}, y) \in A_t : y\mathbf{w}_t^T\mathbf{x}_t < 1\}$
5:     Set $\eta_t = \dfrac{1}{\lambda t}$
6:     Set $\mathbf{w}_{t+\frac{1}{2}} = (1 - \eta_t\lambda)\mathbf{w}_t + \dfrac{\eta_t}{K}\sum_{(\mathbf{x},y)\in A_t^+} y\mathbf{x}$
7:     Set $\mathbf{w}_{t+1} = \min\{1, \dfrac{1/\sqrt{\lambda}}{||\mathbf{w}_{t+\frac{1}{2}}||}\}\mathbf{w}_{t+\frac{1}{2}}$
8: end for
```

1.3 After you train your model, run your classifier on the test set and report the accuracy, which is defined as:

$$\frac{\text{\# of correctly classified test samples}}{\text{\# of test samples}}$$

Finish the implementation of the function `pegasos_test`.

1.4 After you complete above steps, run `pegasos.sh`, which will run the Pegasos algorithm for 500 iterations with 6 settings (mini-batch size $K = 100$ with different $\lambda \in \{0.01, 0.1, 1\}$ and $\lambda = 0.1$ with different $K \in \{1, 10, 1000\}$), and output a `pegasos.json` that records the test accuracy and the value of objective function at each iteration during the training process.

What to do and submit: run script `pegasos.sh`. It will generate `pegasos.json`. Add, commit, and push both `pegasos.py` and `pegasos.json` before the due date.

1.5 Based on `pegasos.json`, you are encouraged to make plots of the objective function value versus the number of iterations (i.e., a convergence curve) in different settings of λ and k, but you are not required to submit these plots.

References

[1] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, Andrew Cotter *Mathematical Programming, 2011, Volume 127, Number 1, Page 3*. Pegasos: primal estimated sub-gradient solver for SVM.

Problem 2 Boosting

In the lectures, you learned an algorithm, boosting, which constructs a strong (binary) classifier based on iteratively adding one weak (binary) classifier into it. A weak classifier is learned to maximize the weighted training accuracy at the corresponding iteration, and the weight of each training example is updated after every iteration.

In this question, you will implement a weak classifier, the decision stump. You will further implement two different yet related boosting algorithms, AdaBoost and LogitBoost.

2.1 Forward stagewise additive modeling Given N samples $\{\mathbf{x}_n, y_n\}$, where $y_n \in \{+1, -1\}$, the goal of boosting is to solve the following optimization problem:

$$\min_h \sum_n L(y_n, h(\mathbf{x}_n)), \tag{2}$$

where $L(y, \hat{y})$ is some loss function.

Given a set of weak classifiers $\mathcal{H}$, we would like to construct a strong classifier as

$$h(\mathbf{x}) = \text{sign}\left[\sum_{t=0}^{T-1} \beta_t h_t(\mathbf{x})\right]. \tag{3}$$

Since directly finding the optimal h is hard, we can solve it sequentially. Suppose we have built a classifier $f_{t-1}(\mathbf{x})$ at step $t-1$. At step t, we would like to improve f_{t-1} by adding a new classifier $h_t(\mathbf{x}) \in \mathcal{H}$ to construct the new classifier $f_t(\mathbf{x})$ as

$$f_t(\mathbf{x}) = f_{t-1}(\mathbf{x}) + \beta_t h_t(\mathbf{x}). \tag{4}$$

Therefore at step t, we would like to find the optimal h_t^* and β_t^* by solving the following opimization problem:

$$(h_t^*(\mathbf{x}), \beta_t^*) = \underset{h_t(\mathbf{x}), \beta_t}{\text{argmin}} \sum_n L(y_n, f_{t-1}(\mathbf{x}_n) + \beta_t h_t(\mathbf{x}_n)) \tag{5}$$

Please read the class "Boosting" defined in `boosting.py`, and then complete the class by implementing the "TODO" part(s) as indicated in `boosting.py`.

2.2 Decision stump A decision stump $h \in \mathcal{H}$ is a classifier characterized by a triplet $(s \in \{+1, -1\}, b \in \mathbb{R}, d \in \{0, 1, \cdots, D-1\})$ such that

$$h_{(s,b,d)}(\mathbf{x}) = \begin{cases} s, & \text{if } x_d > b, \\ -s, & \text{otherwise.} \end{cases} \tag{6}$$

That is, each decision stump function only looks at a single dimension x_d of the input vector $\mathbf{x}$, and check whether x_d is larger than b. Then s decides which label to give if $x_d > b$.

Please first read `classifier.py` and `decision_stump.py`, and then complete the class "DecisionStump" by implementing the "TODO" part(s) as indicated in `decision_stump.py`.

2.3 AdaBoost AdaBoost is a powerful and popular boosting method, for which we utilize the exponential loss

$$L_{exp} = e^{-y_n f(\mathbf{x}_n)} \tag{7}$$

as the loss function. Algorithm 2 summarizes the algorithm of AdaBoost.

Please read the class "AdaBoost" defined in `boosting.py`, and then complete the class by implementing the "TODO" part(s) as indicated in `boosting.py`.

Algorithm 2 The AdaBoost algorithm

```
Require: $\mathcal{H}$: A set of classifiers, where $h \in \mathcal{H}$ takes a $D$-dimensional vector as input and outputs either $+1$
    or $-1$, a training set $\{(\mathbf{x}_n \in \mathbb{R}^D, y_n \in \{+1, -1\})\}_{n=0}^{N-1}$, the total number of iterations T.
Ensure: Learn $h(\mathbf{x}) = \text{sign}\left[\sum_{t=0}^{T-1} \beta_t h_t(\mathbf{x})\right]$, where $h_t \in \mathcal{H}$ and $\beta_t \in \mathbb{R}$.
1: Initialization $w_0(n) = \frac{1}{N}, \forall n \in \{0, 1, \cdots, N-1\}$.
2: for $t = 0, 1, \cdots, T-1$ do
3:     find $h_t = \text{argmin}_{h \in \mathcal{H}} \sum_n w_t(n) \mathbb{I}[y_n \neq h(\mathbf{x}_n)]$
4:     compute $\epsilon_t = \sum_n w_t(n) \mathbb{I}[y_n \neq h_t(\mathbf{x}_n)]$
5:     compute $\beta_t = \frac{1}{2} \log \frac{1 - \epsilon_t}{\epsilon_t}$
6:     compute $w_{t+1}(n) = \begin{cases} w_t(n) \exp(-\beta_t), & \text{if } y_n = h_t(\mathbf{x}_n) \\ w_t(n) \exp(\beta_t), & \text{otherwise} \end{cases}, \forall n \in \{0, 1, \cdots, N-1\}$
7:     normalize $w_{t+1}(n) \leftarrow \frac{w_{t+1}(n)}{\sum_{n'} w_{t+1}(n')}, \forall n \in \{0, 1, \cdots, N-1\}$
8: end for
```

2.4 LogitBoost LogitBoost is another popular boosting algorithm which utilizes the logloss

$$L_{log} = \log(1 + e^{-2y_n f(\mathbf{x}_n)}). \tag{8}$$

As proved in problem set 3, the exponential loss does not correspond to the log likelihood of any well-behaved probabilistic model. Therefore we cannot recover probability estimates from the boosted classifier $h(\mathbf{x})$. Algorithm 3 summarizes the algorithm of LogitBoost.

Algorithm 3 The LogitBoost algorithm

```
Require: $\mathcal{H}$: A set of classifiers, where $h \in \mathcal{H}$ takes a $D$-dimensional vector as input and outputs either $+1$
    or $-1$, a training set $\{(\mathbf{x}_n \in \mathbb{R}^D, y_n \in \{+1, -1\})\}_{n=0}^{N-1}$, the total number of iterations T.
Ensure: Learn $h(\mathbf{x}) = \text{sign}\left[\sum_{t=0}^{T-1} \beta_t h_t(\mathbf{x})\right]$, where $h_t \in \mathcal{H}$ and $\beta_t \in \mathbb{R}$.
1: Initialization $\pi_0(n) = \frac{1}{2}, \forall n \in \{0, 1, \cdots, N-1\}$.
2: for $t = 0, 1, \cdots, T-1$ do
3:     compute the working response $z_t(n) = \frac{(y_n + 1)/2 - \pi_t(n)}{\pi_t(n)(1 - \pi_t(n))}$
4:     compute the weights $w_t(n) = \pi_t(n)(1 - \pi_t(n))$
5:     find $h_t = \text{argmin}_{h \in \mathcal{H}} \sum_n w_t(n) \left[z_t(n) - h(\mathbf{x}_n)\right]^2$
6:     update $f_{t+1}(\mathbf{x}) = f_t(\mathbf{x}) + \frac{1}{2} h_t(\mathbf{x})$
7:     compute $\pi_{t+1}(n) = \frac{1}{1 + \exp(-2 f_{t+1}(\mathbf{x}_n))}$
8: end for
```

Please read the class "LogitBoost" defined in `boosting.py`, and then complete the class by implementing the "TODO" part(s) as indicated in `boosting.py`.

2.5 Final submission Please run `boosting.sh`, which will generate `boosting.json`. Add, commit, and push `boosting.py`, `decision_stump.py` and `boosting.json` before the due date.

Problem 3 Decision Tree

In the lecture, you learned the basic knowledge of decision tree classifier. In this question, you will implement a simple decision tree classifier. For simplicity, only discrete features are considered here.

3.1 Conditional Entropy In the lecture, you learned how to pick the attribute to split. For each attribute, you can calculate the weighted average entropy of each branch, which is also called conditional entropy. In this question, you will implement the function to calculate conditional entropy for given branches.

Please read the notes of "conditional_entropy" defined in `decision_tree.py`, and then complete the function by implementing the "TODO" part as indicated in `decision_tree.py`.

3.2 Tree construction The building of a decision tree involves growing and pruning. For simplicity, in this programming set, you only need to consider how the grow a tree and it is not necessarily a binary tree. You may want to think about what drawbacks it could have.

The tree can be constructed by recursively splitting the nodes if needed. Algorithm 4 summarizes the algorithm of decision tree.

Algorithm 4 The recursive procedure of decision tree algorithm

```
1:  function TreeNode.split(self)
2:      if self.splittable then
3:          find the best attribute to split the node
4:          split the node into child_nodes
5:          for child_node in child_nodes do
6:              child_node.split()
7:          end for
8:      end if
9:  end function
10: Initialization Construct the root_node
11: ROOT_NODE.SPLIT(())
```

For each given node, each possible splitting will be examined and the best one will be chosen using conditional entropy. Then the corresponding child nodes will be examined recursively.

Please read the notes of "split" defined in `decision_tree.py` carefully, and then complete the function by implementing the "TODO" parts as indicated in `decision_tree.py`.

3.3 Final submission Please run `decision_tree.sh`, which will generate `decision_tree.json`. Add, commit, and push `decision_tree.py`, `decision_tree.json` before the due date.