STATS 607A (Fall '14): Assignment 1
Due: Sep 30, 2014 23:59:59

Ambuj Tewari

Sep 15, 2014

This version: 4.0 (corrected submission procedure and permission settings)

Ways to earn extra credit

- +1 for all real bugs in the already supplied code that you find and report to the instructor.
- +1 for each problem where your code is in the top 10 percentile (top 5 students) in terms of running time.
- +1 for each python script where Python style guide checker `pep8` doesn't report any issues. You can test for any style guide issues yourself by typing `pep8 <python-script>` on the shell prompt.

How to turn in?

Create a directory called `stats607-fall2014` (case-sensitive) in the `Public` directory already present in your home directory on any of the Bayes machines. Make it readable for the instructor and GSI by typing:
`fs sa ~/Public/stats607-fall2014/ -acl tewaria read kamwong read`
on the Bayes command prompt. Then create a sub-directory called `assignment_one` within the 607 directory. Make that readable too for us by typing:
`fs sa ~/Public/stats607-fall2014/assignment_one/ -acl tewaria read kamwong read`
We will try to find the following 4 files in your home directory after the submission deadline:
`assignment_one_kmeans.py`
`assignment_one_optimization.py`
`assignment_one_nearest_neighbor.py`
`assignment_one_answers.pdf`

The first 3 files should be python scripts that run without any error message. The last file should be a PDF file with answers to all questions below. Make sure the 4 files themselves readable to us. You can check whether permissions are ok by typing:

```
fs listacl ~/Public/stats607-fall2014
fs listacl ~/Public/stats607-fall2014/assignment_one
```

You should see 2 lines in the output (in each case) that say: (under "Normal rights:")
`tewaria rl`
`kamwong rl`

1 K-means (10 points)

In this problem, we will implement the K-means algorithm. Download a test dataset "seeds" from:
http://archive.ics.uci.edu/ml/datasets/seeds

In the file `seeds_dataset.txt`, there are 210 instances of dimension 7 along with a categorical label (1, 2, or 3). Download an (incomplete!) python script by following the following link:
assignment_one_kmeans.py
Click on the "Raw" button on the top right and save the file as `assignment_one_kmeans.py`.

Open the python script in your favorite text editors. You will see a bunch of places where a comment says `TASK x.y(.z)`. These are the places where you have to supply your own code.
Important: Please only change/add code where the tasks require you to do so (sometimes you'll have to replace a `pass` statement with real code, sometimes you'll be changing existing statements). Please *don't* modify the existing code and comments anywhere else! We might use automated scripts to grade your code and not following this suggestion will break those scripts.

We will now briefly describe your tasks. If something is unclear, please don't hesitate to email the instructor and/or GSI.

1.1 Reading in the data (1 point)

We will read in the instances into two lists: `instances` and `labels`. The former will a list of 7-dimensional instance. An instance will itself be represented using a list of length 7. The latter will be a list of the labels (note that the label occurs at the end of each line).

Task 1.1 has 2 subtasks: 1.1.1 and 1.1.2.

1.2 Finding number of unique labels (1 point)

Finish the definition of the function `num_unique_labels` so that it finds out how many unique labels are there in its argument `labels`. Make sure it returns 3 for the labels obtained from the seeds dataset.

1.3 Implementing K-means++ (2 points)

K-means++ is an enhanced version of the classic K-means algorithm. It only differs from the classic algorithm in the way it initializes the K centers. Implement the function `kmeans_plus_plus` so that this enhanced initialization is available by supplying the optional argument `init` to `cluster_using_kmeans`. Its default value is "random" (in this case, the initial centers are simply chosen at uniformly at random without replacement).

Note that this task hasn't been broken down into precisely defined subtasks deliberately. We want to see how well you document your code when given complete freedom. We also want to see how much variation we see in your solutions to this task. Remember those extra credits for running time – this is the task where to get them by beating your fellow students' approaches!

1.4 Cluster assignment step of K-means (2 points)

Implement the function `assign_cluster_ids`.

Task 1.4 has 2 subtasks: 1.4.1 and 1.4.2.

1.5 Center re-computation step of K-means (2 points)

Implement the function `recompute_centers`.

Task 1.5 has 2 subtasks: 1.5.1 and 1.5.2.

1.6 Running the script and discussing the output (2 points)

Once you have supplied all missing pieces of the code, run the script using:
`python assignment_one_kmeans.py`

Answer the following questions:

Q. 1.1 How does the kmeans clustering compare to the provided labels? On how many instances do they disagree?

Q. 1.2 How does the kmeans++ clustering compare to the provided labels? How does it compare to the kmeans clustering? Is there any difference at all?

2 Simple Optimization (12 points)

In the problem, we will implement two simple optimization algorithms: gradient descent and coordinate descent. A gradient descent iteration is simple:

$$x \leftarrow x - \alpha \nabla f(x)$$

where $\alpha > 0$ is a step size parameter.

2.1 Implement gradient descent given gradient evaluator (2 points)

Download the incomplete code from: `assignment_one_optimization.py`
Also download: `losses.py`
and make sure you put the two files in the same directory (the former `import`s the latter).

Implement the function `gradient_descent`. Note that the argument `func_grad` gives you the gradient at an arbitrary point. The gradient as well as the iterates have dimension `dim` and are both represented as lists of floating point numbers (of length `dim`)

Task 2.1 has 2 subtasks: 2.1.1 and 2.1.2.

2.2 Implement coordinate descent given partial derivative evaluator (2 points)

Implement the function `coordinate_descent`. This is the algorithm whose single iteration looks like:

For j **in** $i_1, i_2, \ldots, i_d$
$\quad x_j \leftarrow x_j - \alpha [\nabla f(x)]_j$

where $[\nabla f(x)]_j$ is the partial derivative $\partial f / \partial x_j$ evaluated at x.

There are two version of coordinate descent: cyclic and random. In the cyclic version, the sequence of coordinates we change is deterministic. It is simply $i_1 = 1, i_2 = 2, i_3 = 3, \ldots$ (of course, remember that, in Python, the indices will start from 0 not 1). In the random version, the sequence $i_1, i_2, \ldots, i_d$ is a random draw from the coordinates 1 through d (let us say with replacement). You have to implement both versions. Note that the argument `func_grad_1d` is a function that takes two arguments, x (first) and j (second), and gives you $[\nabla f(x)_j]$.

Task 2.2 has 3 subtasks: 2.2.1, 2.2.2 and 2.2.3 (0.5, 0.5 and 1 point respectively).

2.3 Loss and loss gradient evaluators (4 points)

We will test the two optimization algorithms on functions that arise in maximum-likelihood estimation problems in regression/classification using linear models. In particular, we will consider (unconstrained) minimization of

$$f(\beta) = \frac{1}{n} \sum_{i=1}^{n} \ell(\beta^\top X_i, Y_i) \tag{1}$$

where $X_i \in \mathbb{R}^d, Y_i \in \mathbb{R}$ and $\ell(t, y)$ is a non-negative loss functions such as the squared loss $(t - y)^2$ or the logistic loss $\log(1+\exp(-yt))$ (used in classification settings where $y \in \{-1, +1\}$). These two losses along with the gradients $\ell'(t, y)$ (w.r.t. the first argument t) are already defined in `losses.py` for you. What you have

to do is to define the 3 functions: `loss_calculator`, `loss_grad_calculator`, `loss_grad_1d_calculator` using the equation (1) above and the equations (2) and (3) below respectively:

$$\nabla f(\beta) = \frac{1}{n}\sum_{i=1}^{n} \ell'(\beta^\top X_i, Y_i) X_i \tag{2}$$

$$[\nabla f(\beta)]_j = \frac{1}{n}\sum_{i=1}^{n} \ell'(\beta^\top X_i, Y_i) X_{i,j} \tag{3}$$

where $X_{i,j}$ is the jth feature/covariate of the ith example/input. Note that `loss_calculator` is actually already almost complete except that you will need to implement a helper function `vector_dot` that simply returns $u^\top v = \sum_{j=1}^{d} u_j v_j$ for two vectors u and v (represented as lists of floats).

Task 2.3 has 4 subtasks: 2.3.1 through 2.3.4.

2.4 Testing the optimization routines on regression and logistic regression (2 points)

The `main()` function first creates a regression data set with continuous responses Y_i and then creates a binary classification data set with binary valued labels $Y_i \in \{-1, +1\}$. It runs gradient descent and coordinate descent on both data sets using certain fixed choices for initial points and step sizes (leave them as they are). Your task it to create appropriate functions that can be passed to gradient desct and coordinate descent so that the entire `main()` function executes properly.

Task 2.4 has 2 subtasks: 2.4.1 and 2.4.2

2.5 Running the script and discussing the output (2 points)

After completing all tasks, run the script using:
`python assignment_one_optimization.py`
(making sure that `losses.py` is in the same directory) and answer the following questions.

Q. 2.1 Which optimization method took less time? Does your answer depend on whether you're talking about squared loss or logistic loss case?

Q. 2.2 Do the optimization methods return reasonable solutions? How does the returned solution compare to the true parameter vector? If we ran the optimization to more and more iterations, would you expect the returned solution to converge to the true parameter?

3 Nearest Neighbor Classification (8 points)

In this part, we will implement the k-nearest neighbor classification algorithm for a multiclass classification and compute 10-fold cross-validation errors on a randomly shuffled version of `seeds_dataset.txt` which is available here:
seeds_dataset_shuffled.txt
(Make sure you click on the "Raw"" button to download the raw .txt file). Download the incomplete code from:
assignment_one_nearest_neighbor.py

3.1 Creating folds (2 points)

Your first task is to write the function `get_fold_indices` that'll be used to create the different folds in 10-fold cross validation. It takes three arguments: `n` (sample size), `num_folds` (total number of folds) and `fold_id` (which fold are we creating?). It should return a tuple of lists: `train_indices, test_indices`. It's behavior is best illustrated by an example:

```
>>> get_fold_indices(10, 5, 0)  # 10 samples/5-fold CV/get train-test indices for fold 0
([2, 3, 4, 5, 6, 7, 8, 9], [0, 1])
>>> get_fold_indices(10, 5, 1)
([0, 1, 4, 5, 6, 7, 8, 9], [2, 3])
>>> get_fold_indices(10, 5, 2)
([0, 1, 2, 3, 6, 7, 8, 9], [4, 5])
>>> get_fold_indices(10, 5, 3)
([0, 1, 2, 3, 4, 5, 8, 9], [6, 7])
>>> get_fold_indices(10, 5, 4)
([0, 1, 2, 3, 4, 5, 6, 7], [8, 9])
```

As you can see, when `n` is divisible by `fold_size`, all splits are of equal size. When `n` is not divisible by `fold_size`, the last split will have a slightly larger test indices list and a slightly smaller train indices list. For example, when we have 12 samples and we're doing 5-fold CV, we should get the following behavior:

```
>>> get_fold_indices(12, 5, 0)
([2, 3, 4, 5, 6, 7, 8, 9, 10, 11], [0, 1])
>>> get_fold_indices(12, 5, 1)
([0, 1, 4, 5, 6, 7, 8, 9, 10, 11], [2, 3])
>>> get_fold_indices(12, 5, 2)
([0, 1, 2, 3, 6, 7, 8, 9, 10, 11], [4, 5])
>>> get_fold_indices(12, 5, 3)
([0, 1, 2, 3, 4, 5, 8, 9, 10, 11], [6, 7])
>>> get_fold_indices(12, 5, 4)
([0, 1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11])
```

Task 3.1 has 3 subtasks: 3.1.1, 3.1.2 and 3.1.3 (.5, .5 and 1 points)

3.2 k-nearest neighbor classifier (3 points)

Implement the function `nn_classifier`. Given a `point` to classify, we first sort examples in `train_data` according to their distances from `point`. Then we want to look at the labels of the k-nearest data points. These should get stored in `nearest_k_labels`. Then we find the label that occurs the most in `nearest_k_labels` and return it. If there are ties, we break them at random.

Task 3.2 has 3 subtasks: 3.2.1 through 3.2.3.

3.3 Computing classification error (1 point)

Implement the function `classification_error`. Given a `classifier` and a `data` set with `labels`, it should figure out the number of examples in the data set on which the classifier's label disagrees with the provided label. Finally, it should return the error rate: total number of errors divided by sample size.

3.4 Creating and processing the folds (1 point)

For each fold, use the indices returned to `get_fold_indices` to create a classifier trained on the training and test data sets for that fold. Then evaluate the error rate of that classifier on the test set of that fold.

Task 3.4 has 2 subtasks: 3.4.1 and 3.4.2

3.5 Running the script and discussing the output (1 point)

Once all functions have been implemented run the script (it will read data from `seeds_dataset_shuffled.txt` and will import definitions from `assignment_one_kmeans.py`. So make sure these two files are present in the same directory. Answer the following question.

Q. 3.1 Which k values gave you lowest 10-fold CV errors? If you run the same code on the unshuffled data `seeds_dataset.txt`, would you see higher or lower errors? Why?