STATS 607A (Fall '15): Assignment 2
Due: Oct 10, 2015 11:55 PM

Ambuj Tewari

Sep 30, 2015

This version: 1.0

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?

Zip the following 4 files:
`assignment_two_pagerank.py`
`assignment_two_svm.py`
`assignment_two_adaboost.py`
`assignment_two_answers.pdf`
into a single `.zip` file and name it `assignment_two_uniqname.zip` where `uniqname` is your U of M uniqname. Upload your zip file to ctools using the "Assignments" link on the left. Make sure you are submitting "Assignment 2".

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.

1 Pagerank (8 points)

In this assignment, we will implement 3 more algorithms from the list of "top 10 data mining algorithms" mentioned in the previous assignment. In the first problem, we will implement the Pagerank algorithm. Download the test files `example_index.txt` and `example_arcs.txt` from:
https://github.com/ambujtewari/stats607a-fall2015/tree/master/homeworks/datasets
In the file `example_index.txt`, there are $n = 106$ websites along with index numbers starting from 0 (and ending with 105). In the file `example_arcs.txt`, there are 141 edges with one edge per line (source first, destination second). For example, the first line is:

```
7 5
```

which means that the website with index 7 links to the website with index 5.

We will implement Pagerank using the power method as explained in the document `Pagerank excerpt.pdf` on ctools (look under "Resources"). We will pretty much base our Python implementation of Pagerank (so called because of its co-inventor Larry Page, one of the founders of Google and a very well-known University

of Michigan alumnus!) on the description provided in `Pagerank excerpt.pdf`. In case, you're hungry for more, here the book the excerpt is from:
http://press.princeton.edu/titles/8216.html

Download an (incomplete!) python script by following the following link:
assignment_two_pagerank.py
Click on the "Raw" button on the top right and save the file as `assignment_two_pagerank.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 Creating the adjacency matrix (0.5 points)

Important: In Problem 1, you are NOT supposed to use any explicit `for` or `while` loops, nor can you use any list comprehensions. You have to use `numpy` functions and `ndarray` methods to do everything. The only `while` loop we will need is in the power iteration implemention. That loop is already present – don't add any new loops in your solutions or your submission file may be completely rejected.

We want to create an adjacency matrix `adj_mat` (containing only 0s and 1s) such that `adj_mat[i, j]` is 1 iff there is an edge from i to j. Supply the code for this in TASK 1.1

1.2 Finding dangling nodes (0.5 points)

Dangling nodes are those from which there are no out-links (in terms of the adjacency matrix, the entire row corresponding to a dangling node is zero). In TASK 1.2, create a 1-d array `dangling` such that `dangling[i] == 1` if `i` is a dangling node and is 0 otherwise.

1.3 Normalize non-zero rows of adjacency matrix (0.5 points)

In TASK 1.3, normalize each nonzero row of `adj_mat` to have row-sum 1 (leave the zero rows untouched) to get `adj_mat_norm`.

1.4 Make a rank one change to get a stochastic matrix (0.5 points)

In TASK 1.4, create a rank one change matrix $\frac{1}{n}\mathbf{a}\mathbf{e}^T$ where $\mathbf{a}$ is the dangling node indicator vector and $\mathbf{e}$ is the vector of 1's. We will add this rank one matrix to `adj_mat_norm` to get a stochastic matrix `adj_mat_stoch`.

Recall that a stochastic matrix is a non-negative square matrix each of whose rows sums to exactly 1.0. There are a couple of `assert` statements in the code asserting that a matrix is supposed to be stochastic. Use of `assert` statements is common in situations like this: you expect something to be true and if it is not true, something probably went wrong and so an exception should be raised. More about Python `assert` statement can be found here:
http://www.tutorialspoint.com/python/assertions_in_python.htm
`assert` statements have been added for illustration only. None of your Tasks involve changing them.

1.5 Implement the function `compute_pageranks` (2 points)

You will implement the power iteration according to eq. (4.5.1) in the Pagerank extract uploaded on ctools. Note that `compute_pageranks` gets called on the `google_mat` which is a mixture (or convex combination) of the two stochastic matrices: `adj_mat_stoch` and the $n \times n$ matrix with $\frac{1}{n}$'s in it (this latter matrix

corresponds to a completely random walk where we just randomly jump from one node to another without even looking at the hyperlink structure). The weight on the former matrix is α and on the latter is $1-\alpha$. We have chosen $\alpha = 0.85$ (don't change it).

TASK 1.5 has 2 subtasks: 1.5.1 and 1.5.2.

1.6 Top 10 page ranks (1 point)

In TASK 1.6, extract the indices of pages with top 10 pageranks (largest pagerank on top) and then print their names using the extracted indices.

1.7 Extracting indices of eigenvalues close to 1 (1 point)

In TASK 1.7, we want to find out indices of eigenvalues (returned by `numpy.eig()`) that are (numerically close to) 1.

1.8 Normalize corresponding eigenvector to be a probability distribution (1 point)

This is your TASK 1.8. The eigenvectors returned by `eig()` are normalized differently: you want to normalize it so that it is a probability distribution (i.e., a non-negative vector with entries summing to 1.0).

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

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

Answer the following questions:

Q. 1.1 What are the top 10 websites by pagerank? Is there any relationship between the top 10 nodes by pagerank and the top 10 nodes according to in-degree?

Q. 1.2 Did your power iteration computation result `pageranks` agree with result of `eig` (which has been named `pageranks_eig` in the code)?

2 Support Vector Machine (9 points)

In this problem, we will implement the Support Vector Machine. Download the "ionosphere" data set `ionosphere.data` from:
https://github.com/ambujtewari/stats607a-fall2015/tree/master/homeworks/datasets
This is a binary classification data set with $n = 351$ instances/examples/feature vectors along with labels ("b" or "g" for "bad" and "good"). we will code "g" as label 1 and "b" as label 0. The dimension of each example is $d = 34$. Each line of the above file has an example and its labels at the end (with values separated by commas).

We will implement SVM by using a minimization function in `scipy.optimize` that can handle constraints. Our minimization problem will be

$$\begin{aligned}\text{minimize} \quad f(\alpha) &:= -\sum_{i=1}^{n}\alpha_i + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j K(x_i, x_j) \\ &= -\alpha^\top \mathbf{1} + \frac{1}{2}\alpha^T(\mathbf{K}\odot YY^\top)\alpha\end{aligned}$$

subject to

$$0 \le \alpha_i \le C,\ 1 \le i \le n$$

where $\mathbf{1}$ is a vector of ones, $\mathbf{K}$ is the kernel matrix with entries $K(x_i, x_j)$ and $YY^\top$ is a rank-one matrix with i, j entry $y_i y_j$. Also note that $\odot$ denotes Hadamard (i.e., entry-wise) product. We will use the default value of $C = 1.0$. Here x_i's are n d-dimensional instances and y_i's are their *signed labels* (i.e. think of label 1 as $+1$ and 0 as -1). Once the minimization is done, a new point x_{new} is given label 1 if

$$\sum_{i=1}^{n} \alpha_i y_i K(x_{\text{new}}, x_i) > 0 \tag{1}$$

and 0 otherwise. Note that any x_i (and the corresponding signed label y_i) with a zero α_i need not be kept around once training is done since it plays no role in the classifier. Other instances, viz. those with non-zero α_i's, are called "support vectors".

Here K is a "kernel function". Its job is to compute inner products between instances after they've been mapped into some high dimensional space. The details of these high dimensional mappings need not concern us here. We will simply use 4 pre-defined kernels in the file `kernels.py`. We will train an SVM using each of the 4 kernels on the first 200 instances and labels. Then we will use the learned classifier to get a confusion matrix by testing it on the remaining 151 instances. Finally, we will report a bunch of evaluation measures for each of the 4 choices of the kernel function.

Download an (incomplete!) python script by following the following link:
assignment_two_svm.py
Click on the "Raw" button on the top right and save the file as `assignment_two_svm.py`. Do the same for:
kernels.py
Complete the following tasks.

Important: In this problem, do NOT add any `for` or `while` loops of your own. You may use list comprehensions if you need them. Try and use `numpy` functions and `ndarray` methods as much as possible.

2.1 Creating the kernel matrix (0.5 points)

Create an $n \times n$ symmetric matrix $\mathbf{K}$ (`kernel_mat` in the code) such that `kernel_mat[i, j]` is $K(x_i, x_j)$.

(I couldn't figure out an elegant way to compute the kernel matrix given instances and a kernel function without using at least 1 `for` loop. The existing code uses 2 nested loops. Keep it that way in your solution and but anyone who emails me a valid solution without loops which is more time-efficient than the 2 loop version we have will get a $+1$ extra credit.)

2.2 Creating the function evaluator (0.5 points)

Supply code for the function `func` so that it evaluates the function $f(\alpha)$ above given α as a 1-d array `alpha`.

2.3 Creating the gradient evaluator (1 points)

Supply code for the gradient evaluator `func_deriv` so that it evaluates the function $\nabla f(\alpha)$ above given α as a 1-d array `alpha`. Remember than `func_deriv` will return a length n 1-d array, not a scalar. Also note the formula for the gradient:

$$\nabla f(\alpha) = -\mathbf{1} + (\mathbf{K} \odot YY^\top)\alpha \ .$$

2.4 Constraints (0.5 point)

The constraint $0 \leq \alpha_i \leq C$ will be represented by a tuple `(0.0, C)`. The list of constraints `box_constraints` should be a list containing n such tuples.

2.5 Retaining non-zero $\alpha_i y_i$ (1 point)

The signed labels y_i are represented in the code as `pm_labels`. Once the optimization terminates, the vector `alpha_y` is formed to store the n values $\alpha_i y_i$. Set `alpha_y_nz` as just the non-zero $\alpha_i y_i$s.

2.6 Retaining the support vectors (0.5 points)

Store the support vectors in `support_vectors`.

2.7 Creating a classifier (1 point)

Write the function `classifier` so that it returns the right label computed according to (1). Use the variables `kernel_func, support_vectors, alpha_y_nz` to do this.

2.8 Evaluating the classifier (2 points)

Make sure you know what a confusion matrix is: http://en.wikipedia.org/wiki/Confusion_matrix

Implement the function `evaluate_classifier` that should return the confusion matrix for `classifier` on `instances, labels`.

TASK 8 has 4 subtasks each worth 0.5 points.

2.9 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_two_svm.py`

Answer the following questions:

Q. 2.1 Which kernel gave you best accuracy?

Q. 2.2 Which class is more difficult to classify? Positive (label 1 = "g")? Negative (label 0 = "b")? Is there any noticeable difference?

3 Adaboost (8 points)

In this problem, we will implement the Adaboost algorithm. We will continue to use the "ionosphere" data set that we used in the SVM problem.

We will implement the algorithm as described in Fig. 5 of the `top10algorithms.pdf` document uploaded in the Resources sections of the ctools website.

Download an (incomplete!) python script by following the following link:
assignment_two_adaboost.py
Click on the "Raw" button on the top right and save the file as `assignment_two_adaboost.py`. Complete the following tasks.

Important: In this problem, it is OKAY to use `for` and `while` loops of your own. You may use list comprehensions if you need them. However, do try and use `numpy` functions and `ndarray` methods as much as possible. Any efficient solutions without using any loops will be awarded +2 extra credits.

3.1 Train weak learner (2 points)

The first step in the Adaboost loop is to train a weak learner. Remember Adaboost is a "boosting" algorithm: it combines weak classifiers to create a stronger classifier. In this step:

$$h_t = \mathcal{L}(\mathcal{D}, D_t)$$

we run a weak learning algorithm $\mathcal{L}$ on the our dataset $\mathcal{D}$ (`instances, labels` in the code) using distribution D_t (`diet` in the code) over the n examples (Fig. 5 uses m instead of n).

Implement the function `weak_learner` so that it learns a particular type of weak classifier called a decision stump (or 1-dimensional threshold function):
http://en.wikipedia.org/wiki/Decision_stump

This is a classifier of the type $1[x_j < \theta]$ or $1[x_j > \theta]$ where 1[condition] is 1 is condition is true and 0 otherwise. Note that a decision stump looks at only a single feature x_j in an instance $x \in \mathbb{R}^d$. That's why they're pretty weak classifiers on their own. In the function, you have to find the j and threshold $\theta \in [-1, 1]$ (since features in the ionosphere datasets are in $[-1, 1]$) that give the least D weighted error on the dataset. Then you have to return a *classifier* that uses that feature, threshold combination. Note that classifiers in your code should output unsigned binary labels 0, 1 (and not signed label -1, +1).

What's D weighted error? That's the next task.

3.2 Compute weighted error (1 point)

Implement the function `compute_error` that computes the weighted error of h (`h` in code) on a data set given weight distribution D (`dist` in code):

$$\sum_{i=1}^{n} D(i) 1[h(X_i) \neq Y_i]$$

where $X_i \in \mathbb{R}^d$ is the ith instance and Y_i is the ith (unsigned) label.

3.3 Update distribution (2 points)

Once a weak classifier has been learned, Adaboost "focuses" its attention on examples where the weak classifier doesn't do a good job: it increases the weight (in the distribution D_t) on examples that are misclassified and decreases the weight on those correctly classified. That is, if $h_t(X_i) \neq Y_i$ then $D_{t+1}(i) = D_t(i) \exp(\alpha_t)/Z_t$ otherwise $D_{t+1}(i) = D_t(i) \exp(-\alpha_t)/Z_t$ where α_t is a positive number (already computed in the code as `alpha`). Note that the normalization constant Z_t is necessary to make sure the new D_{t+1} is still a distribution over n examples. You'll have to make sure you normalize the distribution after doing the exponential updates.

3.4 Return the Adaboost classifier after all iterations are done (1 point)

The current code just runs Adaboost for $T = 20$ iterations (which, in my experience, is enough to get around 90% test set accuracy on ionosphere).

Once all iterations are done, the α_ts and h_ts and available as (α_t, h_t) tuples in a list of tuples called `alpha_h` in the code. Your task is to write the (nested) function `classifier` that, for input point x, returns 1 if $\sum_{t=1}^{T} \alpha_t (2h_t(x) - 1) > 0$ and 0 otherwise.

Note that Fig. 5 has $h_t(x)$ not $2h_t(x) - 1$ since it assumes that the weak classifiers outputs signed labels while our weak classifiers will return unsigned labels.

3.5 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_two_adaboost.py`

Answer the following questions:

Q. 3.1 What was the test accuracy of Adaboost?

Q. 3.2 (A bit open-ended) What do you think will happen to the test accuracy as T (`num_iters` in Adaboost code) increases beyond 20? Will it continue to increase to 1? Increase but converge to a value less than 1? Oscillate? First increase up to a point and then decrease? Provide the reasoning behind your answer.