CS2204

Assignment No. 3

Purpose:
The goal of this assignment is to help familiarize you with using NumPy in an actual science domain. It will introduce you to manipulating NumPy arrays and using pre-defined mathematical subroutine libraries. You will also be using the *timeit* library to compare the efficiency of routines you write with those routines provided through NumPy. This is meant to illustrate that the benefits of using an external library extend beyond convenience.

Background:
There are a number of standard problem domains within scientific literature which computers are very good at solving and apply across a variety of domains. One such domain is classification. Given a set of parameters or measurements on an object, can the computer identify which class that object belongs to.

There are a large number of ways to solve classification problems. [For a short list see http://en.wikipedia.org/wiki/Classification_%28machine_learning%29#Algorithms.] The method you will be using is a linear classifier implemented with least-squares linear regression.

The problem space for this assignment is data collected from breast cancer biopsies. Each biopsy has 10 measurements associated with the observed cell nuclei. In order they are:

a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness (perimeter^2 / area - 1.0)
g) concavity (severity of concave portions of the contour)
h) concave points (number of concave portions of the contour)
i) symmetry
j) fractal dimension ("coastline approximation" - 1)

In addition, the correct diagnosis of the reading is included in a separate file.

Simply put, your task is to write a program that reads in a training dataset, computes a prediction vector based on the data and then tests the prediction vector against a second dataset.

Functional Specifications:

1. Reading in the Data

Read the data into a NumPy array. For proper execution, the index semantics of this array should be arrayname[row,column]. The data is split into two separate files, the first file (patients*.txt) has all the measurements. The second file (results*.txt) has only the diagnoses given.

Write a function called `get_patient_data` that takes a single parameter, the name of the file as a string for the measurements and returns a NumPy array with all the measurements. The array should be a two-dimensional array of shape (N,10) where N is the number of rows in the file [the array will have 10 columns since each row has 10 data values].

Then, write a second function called `get_diagnoses` to read in the diagnoses related to the measurements. This function will also only have a string filename as a parameter and return a NumPy array. The array should be a two-dimensional array of shape (N,1) where N is the number of rows in the file [the array will have 1 column since each row has 1 data value].

You can open a file for reading in Python with the statement: `fin = open(filename,'r')`
After opening fin, you can use a for-loop to access each line of data from the file as a string:

```
for row in fin:
   # process the data in the string object 'row'
```

When done processing the file, you can close it with: `fin.close()`

Each row of data will be a string. You will need to convert that string into a list containing all the data values as floating point numbers, and in the end create a list of those lists (i.e., the result will be a list of lists, where each sublist represents one row of data from the file). To convert a single row string into a single list, you can use a combination of string operations (to convert the string into a list of substrings) and list comprehensions (to convert the list of substrings into a list of floats). You then use a cumulative sum strategy to append the newly generated list to the end of your list of lists.

Finally, convert the list of lists into a 2-dimensional NumPy array and return it. Or, you can just use the `NumPy.readtxt` method. Either way should work.

2. The easy way

If you have read in the data correctly (each patient matched to a single row) then you should be able to use NumPy's `linalg.lstsq()` method to simply solve for the approximate solution to the system of equations – this code has been provided for you. The first element of the tuple that is returned is the solution. In the next section you will be trying to get the same solution value.

Yes, it really is that easy. This is part of the power of pre-built libraries. You don't have to rewrite code that someone else wrote.

…except this time you do. To prove a point of course!

3. The hard way

Now it's time to buckle down and take a quick look at some linear algebra. You don't need to know a whole lot about linear algebra for this assignment, but the explanation below describes the basics you need to know.
We are going to write a method called my_solve() that will take in our measurement matrix (*A*), and our diagnosis matrix (b) and then return the solution matrix. The data as it is currently read in forms a system of equations, which looks something like the following.

$$\begin{bmatrix} texture_0 & \dots & fractal_0 \\ \vdots & \ddots & \vdots \\ texture_n & \dots & fractal_n \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_9 \end{bmatrix} = \begin{bmatrix} diagnosis_0 \\ \vdots \\ diagnosis_n \end{bmatrix}$$

From here on we will call the measurement matrix/array *A* and the diagnosis matrix/array *b*.
Since this equation has more rows than columns and real world data doesn't play nice, we can be highly certain that there is no single solution to this equation. So, instead we find the closest approximation we can. The definition of "closest approximation" varies, but in this case it means the line in 10-space that has the smallest Euclidean distance from all the data points.

It's okay if not all of this makes sense, because luckily Karl Gauss has done most of the heavy lifting for us and proved the least square solution is also the solution to the following equation:
$A^TAx = A^Tb$
Where A^T is the transpose of *A* (by definition A[row, col] = A^T[col, row] for all elements in *A,* or the reflection of a across the NW-SE diagonal). The matrix A^TA will be a 10x10 matrix and A^Tb a 10x1 matrix.
So, the first step is to get the transpose of the matrix. NumPy matrices allow you to take the transpose, but it's a view/alias. Views still point to the original data, which is bad because we may want to change values without modifying the originals. So, write your own transpose function which returns a new array using NumPy's transpose and copy functions.

Next comes the multiplication. Again, this is the hard way, but NumPy helps you out. Matrices have a dot() function that implements dot multiplication (there is more than one way to multiply two matrices, but that's another topic). So, once we've called transpose on our *A* matrix, we can get A^TA and A^Tb. Not too tough yet, but the fun part is next.

We need to solve the linear equation for our *x* values. Let's write another method called gauss() to do this. The gauss() method should have two parameters (the first to receive the result of A^TA and the second for the result of A^Tb in that order) and the method will return the solution matrix. The easiest way to do this is to move the results of our multiplications into an augmented matrix, which a fancy way of saying smash them together. So A^Tb becomes another column on the right side of A^TA. We'll call this new matrix *aug* for short. Aug will have the same number of rows as A^TA but will have one extra column. You can use the shape attribute on A^TA to find it dimensions then create a zero matrix of the desired size. Finally copy the values of A^TA and A^Tb into aug.

When the data is collected into our augmented matrix we put it into row-reduced form. The steps are pretty simple:

Over all the rows in the augmented matrix in order:

a) Select the given row, this is the pivot row. The pivot value is at *aug[pivot, pivot]*
b) Divide all elements in the pivot row by the value *aug[pivot, pivot]*. This normalizes the row (makes the element on the diagonal a one).
c) For every row that is not the pivot row, do:
   1) Find the scaling value for the row. It's the value in the pivot column of the current row divided by the pivot value (*aug[row, pivot]/aug[pivot, pivot]*)
   2) Subtract the scaled values of the pivot row from the corresponding values in the current row and store them back in the current row. So, subtract (*scaling value)*aug[pivot, col]* from *aug[row, col]* and store in *aug[row, col]*. This zeros out all the elements of the pivot column except for the pivot value.

The end result is that we have an identity matrix in the first part of the matrix and the last column has the least-square results for the data. We want to return this column as the solution matrix from gauss(). This same column is then returned from my_solve().

4. Looking at differences

Ok, so we've done it the easy way and then the hard way. You can check yourself at this point by running both versions of the solution using one of the given data sets. If you've everything is working correctly you will get the same solution matrix from both methods. You have been provided with code that will compare the results of your my_solve() against those produced by linalg.lstsq().

Now let's see how good our results are. You are given code that will use your solution matrix on a second dataset ('patients1.txt') and comparing your predictions with the actual diagnoses' ('results1.txt'). The code reports:

a) Percent correct
b) Percent false positives

c) Percent false negatives

…and two efficiency values:

a) Runtime of the NumPy implementation (over 300 calls)
b) Runtime of your custom implementation (over 300 calls)

A false positive is any situation where the prediction says positive and the actual results say negative. Likewise, a false negative is any situation where the prediction says negative and the actual results say positive.

Submission for grading:

When you have completed your work on this assignment, please submit the **breast_cancer.py** source file for grading. Submit ONLY the source file – please do not submit a zip file containing your entire project. You submit the files by visiting the assignment page in Oak and attaching the files (one at a time) by clicking on the "Browse my computer" button and finding the file to attach.

Grading:

This project is worth 50 points. Your grade on this project will be based on the following:

1. The correctness of your methods implemented in the breast_cancer.py file.
2. The use of built-in Python capabilities (list comprehensions, array sections, etc.).
3. The use of good programming style.

You should also review the syllabus regarding the penalties for late programming assignments.