COS302/SML305 – Princeton University, Spring 2022
Assignment #2

Due: February 7, 2022 at 11:59pm

Upload at: https://www.gradescope.com/courses/214271/assignments/1003646

Remember to append your Colab PDF as explained in the first homework, with all outputs visible.
When you print to PDF it may be helpful to scale at 95% or so to get everything on the page.

Problem 1 (23pts)

Imagine that you have an old-fashioned balance scale as shown below.

You have five objects of unknown mass m_1, m_2, m_3, m_4, m_5. You would like to find the mass of each of these objects using the scale. If you put one or more of the objects on each side of the scale, you will be able to measure the difference in weight. This scale does not work, however, if one side is left empty.

(A) You make a measurement of objects $\{1, 3\}$ against objects $\{4, 5\}$ and get a difference of 3kg. Write this as a linear equation for the masses.

(B) What is the minimum number of such measurements necessary to recover all five masses?

(C) If the answer to the above question is N, come up with $N - 1$ measurements that, when combined with the answer to (A) above would make it possible to recover the masses.

(D) What would $\boldsymbol{A}$ and $\boldsymbol{b}$ be in the matrix form of the linear system corresponding to the N measurements? That is, if you were to set this up as $\boldsymbol{Am} = \boldsymbol{b}$ where $\boldsymbol{m}$ is the vector of unknown masses. Denote the result of the ith measurement as b_i. (Typeset these using the `bmatrix` environment.)

Problem 2 (25pts)

The core library for numerical computation in Python is called NumPy. NumPy provides useful abstractions to create and manipulate multidimensional arrays (e.g., vectors, matrices, tensors), and functions that are thin layers over high-performance C/C++/Fortran BLAS and LAPACK libraries. Conventionally you would start your numeric Python code with an `import numpy as np`. The most basic object is the NumPy array, which is a multi-dimensional array usually made up of type `double`, i.e., 64-bit floating point numbers. A vector is usually a one-dimensional array and a matrix is a two-dimensional array:

```
example_vector = np.array([50.0, 1.4, 0.4, 0.23, 1.04])
example_matrix = np.array([[ 3.4, 1.10,   0.8 ],
                           [ 4.2, 100.0, -1.4],
                           [ 1.1, 0.44, 9.4]])
```

Higher-dimensional arrays (tensors) are possible, but come up less often than vectors and matrices. The tuple made up of the size each dimension is called the *shape* and can be accessed as an attribute, so:

```
print(example_vector.shape, example_matrix.shape)
# Output: (5,) (3,3)
```

Create a Colab notebook following the same pattern as the previous assignment, and do the following:

(A) Use `arange` to create a variable named `foo` that stores an array of numbers from 0 to 29, inclusive. Print `foo` and its shape.

(B) Use the `reshape` function to change `foo` to a validly-shaped two-dimensional matrix and store it in a new variable called `bar`. Print `bar` and its shape.

(C) Create a third variable, `baz` that reshapes it into a valid three-dimensional shape. Print `baz` and its shape.

(D) There are several different ways to index into NumPy arrays. Use two-dimensional array indexing to set the first value in the second row of `bar` to -1. Now look at `foo` and `baz`. Did they change? Explain what's going on. (Hint: does `reshape` return a view or a copy?)

(E) Another thing that comes up a lot with array shapes is thinking about how to aggregate over specific dimensions. Figure out how the NumPy `sum` function works (and the `axis` argument in particular) and do the following:

- (i) Sum `baz` over its second dimension and print the result.
- (ii) Sum `baz` over its third dimension and print the result.
- (iii) Sum `baz` over **both** its first and third dimensions and print the result.

(F) Along with shaping and indexing, we also do a lot of slicing which is where you index with ranges to get subvectors and sometimes submatrices. Write code to do the following:

- (i) Slice out the second row of `bar` and print it.
- (ii) Slice out the last column of `bar` using the -1 notation and print it.
- (iii) Slice out the top right 2×2 submatrix of `bar` and print it.

Problem 3 (25pts)

One feature of NumPy that is powerful but tricky is the ability to perform broadcasting, which really just refers to repeatedly performing an operation over one or more dimensions. Start a new problem in your Colab notebook and when you're done with the entire assignment, follow the same procedure for appending a PDF and inserting the URL as you did in the previous assignment. The notebook URL is near the end of the assignment.

(A) The most basic kind of broadcast is with a scalar, in which you can perform a binary operation (e.g., add, multiply, ...) on an array and a scalar, the effect is to perform that operation with the scalar for every element of the array. To try this out, create a vector $1, 2, \ldots, 10$ by adding 1 to the result of the `arange` function.

(B) Now, create a 10×10 matrix $\boldsymbol{A}$ in which $A_{ij} = i + j$. You'll be able to do this using the vector you just created, and adding it to a reshaped version of itself.

(C) A very common use of broadcasting is to standardize data, i.e., to make it have zero mean and unit variance. First, create a fake "data set" with 50 examples, each with five dimensions.

```
import numpy.random as npr
data = np.exp(npr.randn(50,5))
```

You don't worry too much about what this code is doing at this stage of the course, but for completeness: it imports the NumPy random number generation library, then generates a 50×5 matrix of standard normal random variates and exponentiates them. The effect of this is to have a pretend data set of 50 independent and identically-distributed vectors from a log-normal distribution.

(D) Now, compute the mean and standard deviation of each column. This should result in two vectors of length 5. You'll need to think a little bit about how to use the `axis` argument to `mean` and `std`. Store these vectors into variables and print both of them.

(E) Now standardize the `data` matrix by 1) subtracting the mean off of each column, and 2) dividing each column by its standard deviation. Do this via broadcasting, and store the result in a matrix called `normalized`. To verify that you successfully did it, compute the mean and standard deviation of the columns of `normalized` and print them out.

Problem 4 (25pts)

Fairly often in machine learning and other disciplines that use linear algebra to formalize computational problems, you see explicit matrix inverses like $\boldsymbol{A}^{-1}\boldsymbol{b}$ in papers and books. Of course, you learn in linear algebra classes how to perform such an inverse, and in the NumPy linear algebra library you'll see a function that will compute the inverse. It seems sensible right? And it's right there on the page: *Invert the matrix* $\boldsymbol{A}$ *and then multiply by the vector* $\boldsymbol{b}$. How hard can it be?

Well... don't do that. When you see $\boldsymbol{A}^{-1}\boldsymbol{b}$ you should read that as "the $\boldsymbol{x}$ that is the result of solving the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$". You should use something like the `solve` function in the NumPy linear algebra package. If you have to solve multiple such systems with the same matrix, you might think about performing an LU decomposition, a topic we'll return to in a couple of weeks.

The issue is one of numerical stability, which boils down to the fact that 1) floating point numbers on computers don't have infinite precision, and 2) some quantities (e.g., infinite sums or numerical integrals) require us to truncate and approximate if we want the computation to complete. These issues are deep and this course won't delve into them significantly. However, the "explicit inverse" issue is worth looking at just to see an example.

(A) A Vandermonde matrix is a matrix generated from a vector in which each column of the matrix is an integer power starting from zero. So, if I have a column vector $[x_1, x_2, \ldots, x_N]^T$, then the associated (square) Vandermonde matrix would be

$$\boldsymbol{V} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & x_N^3 & \cdots & x_N^{N-1} \end{bmatrix}$$

Use what you learned about broadcasting in the previous problem to write a function that will produce a Vandermonde matrix for a vector $[1, 2, \ldots, N]^T$ for any N. Do it without using a loop. Here's a stub to get you started:

```
def vandermonde(N):
  vec = np.arange(N)+1
  # Finish me.
```

Use your function for $N = 12$, store it in variable named `vander`, and print the result.

(B) Now, let's make a pretend linear system problem with this matrix. Create a vector of all ones, of length 12 and call it `x`. Perform a matrix-vector multiplication of `vander` with the vector you just created and store that in a new vector and call it `b`. Print the vector `b`.

(C) First, solve the linear system the naïve way, pretending like you don't know `x`. Import `numpy.linalg`, invert `V` and multiply it by `b`. Print out your result. What should you get for your answer? If the answer is different than what you expected, write a sentence about that difference.

(D) Now, solve the same linear system using `solve`. Print out the result. Does it seem more or less in line with what you'd expect?

Problem 5 (2pts)
Approximately how many hours did this assignment take you to complete?

My notebook URL: https://colab.research.google.com/XXXXXXXXXXXXXXXXXXXXXX