Introduction to Machine Learning (Fall 2022)

Homework 1 -- Numpy and ML

Due: Wednesday, September 21, 2022 at 11:00 PM

Welcome to your first homework! Homeworks are designed to be our primary teaching and learning mechanism, with conceptual, math, and coding questions that are designed to highlight the critical ideas in this course. You may choose to tackle the questions in any order, but the homeworks are designed to be followed sequentially. Often, insights from the early problems will help with the later ones.

You have 'free checking'! That means you can check and submit your answer as many times as you want. Your best submission (the one that gives you the most points taking into account correctness and lateness) will be counted---you don't have to worry about it.

After submitting your answers, even if you have gotten a perfect score, we highly encourage you to hit 'View Answer' to look at the staff solution. You may find the staff solutions approached the problems in a different way than you did, which can yield additional insight. Be sure you have gotten your points before hitting 'View Answer', however. You will not be allowed to submit again after viewing the answer.

Each week, we'll provide a Colab notebook for you to use draft and debug your solutions to coding problems (you have better editing and debugging tools there); but you should submit your final solutions here to claim your points.

This week's Colab notebook can be found here: HW01 Colab Notebook (Click Me!)

The homework comes in two parts:

1. Learning to use `numpy`
2. Introduction to linear regression

1) Numpy

Machine learning algorithms almost always boil down to matrix computations, so we'll need a way to efficiently work with matrices.

`numpy` is a package for doing a variety of numerical computations in Python that supports writing very compact and efficient code for handling arrays of data. It is used extensively in many fields requiring numerical analysis, so it is worth getting to know.

We will start every code file that uses `numpy` with `import numpy as np`, so that we can reference `numpy` functions with the `np.` precedent. The fundamental data type in numpy is the multidimensional array, and arrays are usually generated from a nested list of values using the `np.array` command. Every `array` has a `shape` attribute which is a tuple of dimension sizes.

In this class, we will use two-dimensional arrays almost exclusively. That is, **we will use 2D arrays to represent both matrices and vectors!** This is one of several times where we will seem to be unnecessarily fussy about how we construct and manipulate vectors and matrices, but this will make it easier to catch errors in our code. Even if `[[1,2,3]]` and `[1,2,3]` may look the same to us, numpy functions can behave differently depending on which format you use. The first has two dimensions (it's a list of lists), while the second has only one (it's a single list). Using only 2D arrays for both matrices and vectors gives us predictable results from numpy operations.

Using 2D arrays for matrices is clear enough, but what about column and row vectors? We will represent a column vector as a $d \times 1$ array and a row vector as a $1 \times d$ array. So for example, we will represent the three-element column vector,

$$x = \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix},$$

as a 3×1 `numpy` array. This array can be generated with

```
x = np.array([[1],[5],[3]]),
```

or by using the transpose of a 1×3 array (a row vector) as in,

```
x = np.transpose(np.array([[1,5,3]]),
```

where you should take note of the "double" brackets.

It is often more convenient to use the array attribute `.T` , as in

```
x = np.array([[1,5,3]]).T
```

to compute the transpose.

Before you begin, we would like to note that in this assignment we will not accept answers that use `for` or `while` loops. One reason for avoiding loops is efficiency. For many operations, numpy calls a compiled library written in C, and the library is far faster than interpreted Python (in part due to the low-level nature of C, optimizations like vectorization, and in some cases, parallelization). But the more important reason for avoiding loops is that using numpy library calls leads to simpler code that is easier to debug. So, we expect that you should be able to transform loop operations into equivalent operations on numpy arrays, and we will practice this in this assignment.

Of course, there will be more complex algorithms that require loops, but when manipulating matrices you should always look for a solution without loops.

You can find general documentation on `numpy` here.

Numpy functions and features you should be familiar with for this assignment:

- `np.array`
- `np.transpose` (and the equivalent method `a.T`)
- `np.ndarray.shape`
- `np.dot` (and the equivalent method `a.dot(b)`)
- `np.linalg.inv`

- `np.vstack`
- `np.ones`
- `np.sqrt`
- Elementwise operators `+, -, *, /`

Note that in Python, `np.dot(a, b)` is the matrix product `a @ b`, not the dot product $a^T b$.

If you're unfamiliar with numpy and want to see some examples of how to use it, please see this link: Numpy Overview.

1.1) Array Basics

1.1.1) Creating Arrays

Provide an expression that sets `A` to be a 2×3 `numpy` array (2 rows by 3 columns), containing any values you wish.

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.1.2) Transpose

Write a procedure that takes an array and returns the transpose of the array. You can use `np.transpose` or the property `T`, but you may not use loops.

Note: as with other coding problems in 6.036 you do not need to call the procedure; it will be called/tested when submitted.

```python
import numpy as np
def tp(A):
    return np.transpose(A)

```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.2) Shapes

Let `A` be a 4×2 numpy array, `B` be a 4×3 array, and `C` be a 4×1 array. For each of the following expressions, indicate the shape of the result as a tuple of integers (**recall python tuples use parentheses, not square brackets, which are for lists, and a tuple with just one item x in it is written as `(x,)` with a comma**). Write "none" (as a Python string with quotes) if the expression is illegal.

For example,

- If the result array was `[45, 36, 75]`, the shape is `(3,)`
- If the result array was `[[1,2,3],[4,5,6]]`, the shape is `(2,3)`

Hint: If you get stuck, code and run these expressions (with array values of your choosing), then print out the shape using `A.shape`

1.2.1)

`np.array([1,2,3])`

(3,)

Check Formatting Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.2.2)

```
np.array([[1,2,3]])
```

(1,3)

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.2.3)

Reminder: `A` is 4×2, `B` is 4×3, and `C` is 4×1.

```
C * C
```

(4,1)

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.2.4)

Reminder: `A` is 4×2, `B` is 4×3, and `C` is 4×1.

```
np.dot(C, C)
```

None

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.2.5)

Reminder: `A` is 4×2, `B` is 4×3, and `C` is 4×1.

```
np.dot(np.transpose(C), C)
```

(1,1)

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.2.6)

Reminder: `A` is 4×2, `B` is 4×3, and `C` is 4×1.

`np.dot(A, B)`

None

Check Formatting Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.2.7)

Reminder: `A` is 4×2, `B` is 4×3, and `C` is 4×1.

`np.dot(A.T, B)`

(2,3)

Check Formatting Submit View Answer 100.00%

You have infinitely many submissions remaining.

Hint: for more compact and legible code, use `@` for matrix multiplication, instead of `np.dot`. If `A` and `B`, are matrices (2D numpy arrays), then `A @ B = np.dot(A, B)`.

1.3) Indexing vs. Slicing

The shape of the resulting array is different depending on if you use indexing or slicing. **Indexing** refers to selecting particular elements of an array by using a single number (the index) to specify a particular row or column. **Slicing** refers to selecting a subset of the array by specifying a range of indices.

If you're unfamiliar with these terms, and the indexing and slicing rules of arrays, please see the indexing and slicing sections of this link: Numpy Overview (Same as the Numpy Overview link from the introduction). You can also look at the official numpy documentation here.

In the following questions, let `A = np.array([[5,7,10,14],[2,4,8,9]])`. Tell us what the output would be for each of the following expressions. Use brackets `[]` as necessary. If the operation is invalid, write the python string `"none"`.

Note: Remember that Python uses zero-indexing and thus starts counting from 0, not 1. This is different from R and MATLAB.

1.3.1) Indexing

`A[1,2] =` 8

Check Formatting Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.3.2) Indexing, revisited

Reminder: `A = np.array([[5,7,10,14],[2,4,8,9]])`

`A[1,8] =` "none"

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3.3) Slicing

Reminder: `A = np.array([[5,7,10,14],[2,4,8,9]])`

`A[0:1,1:3] =` [[7,10]]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3.4) Slicing, revisited

Reminder: `A = np.array([[5,7,10,14],[2,4,8,9]])`

`A[0:1,1:20] =` [[7,10,14]]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3.5) Lone Colon Slicing

Reminder: `A = np.array([[5,7,10,14],[2,4,8,9]])`

`A[1:,:2] =` [[2,4]]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3.6) Combining Indexing and Slicing

Reminder: `A = np.array([[5,7,10,14],[2,4,8,9]])`

`A[1,1:3]` = [4,8]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3.7) Combining Indexing and Slicing, revisited

Reminder: `A = np.array([[5,7,10,14],[2,4,8,9]])`

`A[:, 1:2]` = [[7],[4]]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3.8) Combining Indexing and Slicing, revisited again

Reminder: `A = np.array([[5,7,10,14],[2,4,8,9]])`

`A[:, 1]` = [7,4]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3.9) Differences

The difference between slicing and indexing is:

- (•) Slicing preserves dimensionality while indexing does not.
- ( ) Indexing preserves dimensionality while slicing does not.
- ( ) Indexing is a dangerous sport for librarians.
- ( ) French chefs prefer slicing.

Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.4) Debugging Advice

Check all the following that are helpful when debugging code:

- [x] Read the problem carefully, and write down any relevant equations or pseudocode
- [x] Google the error and/or check StackOverflow
- [ ] Stare at the screen and hope that the solution dawns on me
- [x] Identify which line of code is causing errors -- `print()` statements are helpful for this
- [x] Check that I'm using the right type of matrix multiplication (* vs @ vs `np.dot()`)
- [x] Check that I'm performing an operation over the correct axis (row or column)
- [x] Read documentation to ensure I am using the function correctly
- [ ] Cry and/or scream into the void
- [x] Work out a small example by hand and ensure my code is giving the correct results at each intermediate step
- [ ] Add a `.T` at the end of a few random variables
- [x] Explain my code to a friend or rubber duck
- [x] Check dimensions of my variables by printing their shapes
- [x] Print out intermediate results in the code, and if using the online code-boxes, find the printed results by clicking the 'Show/Hide Detailed Results'

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

1.5) Coding Practice

Now that we're familiar with numpy arrays, let's practice actually using numpy in our code!

In the following questions, you must get the shapes of the output correct for your answer to be accepted. If your answer contains the right numbers but the grader is still saying your answers are incorrect, check the shapes of your output. The number and placement of brackets need to match!

1.5.1) Row Vector

Write a procedure that takes a list of numbers and returns a 2D `numpy` array representing a **row** vector containing those numbers. Recall that a row vector in our usage will have shape $(1, d)$ where d is the number of elements in the row.

```python
import numpy as np
def rv(value_list):
    return np.array([value_list])

```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.5.2) Column Vector

Write a procedure that takes a list of numbers and returns a 2D numpy array representing a column vector containing those numbers. You can use the `rv` procedure.

```python
import numpy as np
def cv(value_list):
    return np.transpose(np.array([value_list]))

```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.5.3) Length

Write a procedure that takes a column vector and returns the vector's Euclidean length (or equivalently, its magnitude) as a scalar. You may not use `np.linalg.norm`, and you may not use loops.

Remember that the formula for the Euclidean length for a vector $\mathbf{x}$ is:

$$\begin{aligned}\text{length}(\mathbf{x}) &= \sqrt{x_1^2 + x_2^2 + ... + x_n^2} \\ &= \sqrt{\sum_{i=1}^{n} x_i^2}\end{aligned}$$

```python
import numpy as np
def length(col_v):
    elements = np.transpose(col_v)[0]
    print(elements)
    print([x^2 for x in list(elements)])
    print(np.sqrt(sum([x^2 for x in elements])))
    return np.sqrt(sum([x**2 for x in elements]))

```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.5.4) Normalize

Write a procedure that takes a column vector and returns a unit vector (a vector of length 1) in the same direction. You may not use loops. Use your `length` procedure from above (you do not need to define it again).

```
import numpy as np
def normalize(col_v):
    return col_v/length(col_v)
```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.5.5) Last Column

Write a procedure that takes a 2D array and returns the final column as a two dimensional array. You may not use loops. Hint: negative indices are interpreted as counting from the end of the array.

```
import numpy as np
def index_final_col(A):
    return np.array([A[:,-1]]).transpose()
```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.5.6) Matrix inverse

A scalar number x has an inverse x^{-1}, such that $x^{-1}x = 1$, that is, their product is 1. Similarly, a matrix A may have a well-defined inverse A^{-1}, such that $A^{-1}A = I$, where matrix multiplication is used, and I is the identity matrix. Such inverses

generally only exist when A is a square matrix, and just as 0 has no well defined multiplicative inverse, there are also cases when matrices are "singular" and have no well defined inverses.

Write a procedure that takes a matrix A and returns its inverse, A^{-1}. Assume that A is well-formed, such that its inverse exists. Feel free to use routines from `np.linalg`.

```
import numpy as np
def matrix_inverse(A):
    return np.linalg.inv(A)

```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.6) Working with Data in Numpy

1.6.1) Representing data

Mat T. Ricks has collected weight and height data of 3 people and has written it down below:

Weight, Height
150, 5.8
130, 5.5
120, 5.3

He wants to put this into a `numpy` array such that each *column* represents one individual's weight and height (in that order), in the order of individuals as listed. Write code to set `data` equal to the appropriate numpy array:

```python
import numpy as np
data = np.array([[150,130,120], [5.8, 5.5, 5.3]]) #your code here

```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

1.6.2) Matrix Multiplication

Now he wants to compute, for each person, the sum of that person's height and weight, and return the results in a *row* vector with one entry per person. He does this by matrix multiplication using `data` and another `numpy` array. (Remember that column and row vectors are arrays and that, in 6.036, we will *always* represent these as two-dimensional arrays.) He has written the following incorrect code to do so and needs your help to fix it:

```python
import numpy as np
def transform(data):
    return np.dot(np.array([[1,1]]), data) # fix this line

```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

2) Beginning linear regression

We are beginning our study of machine learning with *linear regression* which is a fundamental problem in supervised learning. Please study Sections 2.1 through 2.4 of the Chapter 2 - Regression lecture notes before starting in on these problems.

A hypothesis in linear regression has the form

$$y = \theta^T x + \theta_0$$

where x is a $d \times 1$ input vector, y is a scalar output prediction, θ is a $d \times 1$ parameter vector and θ_0 is a scalar offset parameter.

This week, just to get warmed up, we will consider a simple algorithm for trying to find a hypothesis that fits the data well: we will generate a lot of random hypotheses and see which one has the smallest error on this data, and return that one as our answer. (We don't recommend this method in actual practice, but it gets us started and makes some useful points.)

2.1) Warm-up

Here is a data-set for a regression problem, with $d = 1$ and $n = 5$:

$$\mathcal{D} = ([1], 2), ([2], 1), ([3], 4), ([4], 3), ([5], 5)$$

Recall from the notes that $\mathcal{D}$ is a set of (x, y) (input, output) pairs.

Consider hypothesis $\theta = 1, \theta_0 = 1$. Let our objective $J(\theta, \theta_0; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} (\theta x^{(i)} + \theta_0 - y^{(i)})^2$

What is $J(\theta, \theta_0; \mathcal{D})$? (Note that you can type simple arithmetic expressions into the answer box.)

9/5

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.2) Linear prediction

Assume we are given an input x as a *column* vector and the parameters specifying a linear hypothesis. Let's compute a predicted value.

Write a Python function which is given:

- `x`: input vector $d \times 1$
- `th`: parameter vector $d \times 1$
- `th0`: offset parameter 1×1 or scalar

and returns:

- y value predicted for input `x` by hypothesis `th, th0`

```python
import numpy as np
def lin_reg_predict(x, th, th0):
    return np.dot(th.T, x) + th0
```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.3) Lots of data!

Now assume we are given n points in an array, let's compute predictions for all the points.

Write a Python function which is given:

- `X`: input array $d \times n$
- `th`: parameter vector $d \times 1$
- `th0`: offset parameter 1×1 or scalar

and returns:

- a $1 \times n$ vector y of predicted values, one for each column of `X` for hypothesis `th, th0`

Try to make it so that your answer to this question can be used verbatim as an answer to the previous question.

```python
import numpy as np
def lin_reg_predict(X, th, th0):
    return np.dot(th.T, X) + th0
```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.4) Mean squared error

Given two $1 \times n$ vectors of output values, `Y` and `Y_hat`, compute a 1×1 (or scalar) mean squared error.

- Read about `np.mean`

Write a Python function which is given:

- `Y`: vector of output values $1 \times n$
- `Y_hat`: vector of output values $1 \times n$

and returns:

- a 1×1 array with the mean square error

```python
import numpy as np
def mse(Y, Y_hat):
    print((Y - Y_hat)**2)
    return np.array(np.mean((Y - Y_hat)**2, keepdims=True))

```

Run Code Submit View Answer 100.00%

You have infinitely many submissions remaining.

2.5) More mean squared error

Assume now that you have two $k \times n$ *arrays* of output values, `Y` and `Y_hat`. Each row $(0 \ldots k-1)$ in a matrix represents the results of using a different hypothesis. Compute a $k \times 1$ vector of the mean-squared errors associate with each of the hypotheses (but averaged over all n data points, in each case.)

- Read about the `axis` and `keepdims` arguments to `np.mean`

(Try to make it so that your answer to this question can be used verbatim as an answer to the previous question.)

Write a Python function which is given:

- `Y`: vector of output values $k \times n$
- `Y_hat`: vector of output values $k \times n$

and returns:

- a $k \times 1$ vector of mean squared error values

```python
import numpy as np
def mse(Y, Y_hat):
    return np.array(np.mean((Y - Y_hat)**2, keepdims=True, axis=1))
```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.6) Linear prediction error

Use the `mse` and `lin_reg_predict` procedures to implement a procedure that takes

- `X`: $d \times n$ input array representing n points in d dimensions
- `Y`: $1 \times n$ output vector representing output values for n points
- `th`: parameter vector $d \times 1$
- `th0`: offset 1×1 (or scalar)

and returns

- 1×1 (or scalar) value representing the MSE of hypothesis `th`, `th0` on the data set `X`, `Y`.
- Read about the `axis` argument to `np.mean`

```
import numpy as np
def lin_reg_err(X, Y, th, th0):
    return np.array(np.mean((np.dot(th.T, X) + th0 - Y)**2, axis=1))
```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.7) Our first machine learning algorithm!

The code is below. It takes in

- `X`: $d \times n$ input array representing n points in d dimensions
- `Y`: $1 \times n$ output vector representing output values for n points
- `k`: a number of hypotheses to try

And generates as output

- the tuple `((th, th0), error)` where `th`, `th0` is a hypothesis and `error` is the MSE of that hypothesis on the input data.

```
def random_regress(X, Y, k):
    d, n = X.shape
    thetas = 2 *np.random.rand(d, k) - 1
    th0s = 2* np.random.rand(1, k) - 1
    errors = lin_reg_err(X, Y, thetas, th0s.T)
    i = np.argmin(errors)
    theta, th0 = thetas[:,[i]], th0s[:,[i]]
    return (theta, th0), errors[i]
```

Note that in this code we use `np.random.rand` rather than `np.random.randn` as we saw in the lab. So some of the behavior will be different, and we'll ask some questions about that below.

- Read about `np.random.rand`
- Read about `np.argmin`

Rather than asking you to write the code, we are going to ask you some questions about it.

a. Lines 2 and 3 generate k random hypotheses `(th, th0)`. What problem would we face if we didn't multiply by 2 and subtract 1'?

Pick one.

- ○ The resulting arrays would have the wrong shape.
- ○ The training data would have too much noise.
- ○ The hypotheses we consider would be too similar to each other.
- ◉ The hypotheses we consider would all have positive slope.
- ○ The hypotheses we consider would all always predict positive values.

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

b. What is going on in line 5?

Pick one.

- ○ We are finding the smallest error value over all the hypotheses.
- ◉ We are finding the index of the smallest error value over all the hypotheses.

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

c. When we call `lin_reg_err` in line 4, we have objects with the following dimensions:

- `X`: $d \times n$
- `ths`: $d \times k$
- `th0s`: $1 \times k$

If we want to get a matrix of predictions of all the hypotheses on all the data points, we can write `np.dot(ths.T, X) + th0s.T` But if we do the dimensional analysis here, there's something fishy.

- What is the dimension of `np.dot(ths.T, X)`?

Enter a Python list of two strings (remember quotes). The strings can be one of: `'1'`,`'k'`,`'d'`,`'n'`.

['k','n']

Check Formatting Submit View Answer **100.00%**

You have infinitely many submissions remaining.

- What is the dimension of `th0s.T`?

Enter a Python list (remember brackets) of two strings (remember quotes). The strings can be one of: `'1','k','d','n'`. ['k', '1']

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

- Why does this work?

Pick one.

- magic
- broadcasting
- it's actually wrong

100.00%

You have infinitely many submissions remaining.

Solution: broadcasting

Explanation:

The smaller array th0s.T of shape (k, 1) is "broadcast" across the larger array np.dot(ths.T, X) of shape (k, n) so that a copy of th0s.T is added to each column of np.dot(ths.T, X). See here for the numpy broadcasting rules.

- What would be an explicit numpy call to convert `th0s` into the same shape as `np.dot(ths.T, X)` so the addition is mathematically well defined without any Numpy trickery?

Pick all that are correct.

- [ ] `np.repeat(th0s,n)`
- [ ] `np.repeat(th0s,n,axis=0)`
- [ ] `np.repeat(th0s,n,axis=1)`
- [x] `np.repeat(th0s.T, n, axis=1)`
- [x] `np.repeat(th0s, n, axis=0).T`

Submit | View Answer **100.00%**

You have infinitely many submissions remaining.