03-HOMEWORK-ProjectionsAndDeterminants

September 30, 2021

1 Projections and Determinants

In this assignment, you will write code to implement some of the formulas that we derived in class.

[1]:
```
import numpy as np
```

1.0.1 Orthogonal Projections

Problem 1 Consider the vector space $\mathbb{R}^5$ with the standard dot product. Let $U \subset \mathbb{R}^5$ be the linear subspace

$$U = \text{span}[\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 0 \\ -7 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}]$$

Let

$$\vec{x} = \begin{bmatrix} -10 \\ -9 \\ 0 \\ 0 \\ 5 \end{bmatrix}.$$

Using the formula that we derived in class, determine $\pi_U(\vec{v})$, the orthogonal projection of $\vec{v}$ onto U. Do this in steps, as indicated in the comments of the following code blocks.

[5]:
```
# Define a numpy array B whose columns form a basis for U. Print your matrix B.
# Columns of U are lin. ind. so can write them into matrix B
B = np.array([[0,-1,1],
              [1,0,1],
              [2,0,1],
              [3,-7,1],
              [4,2,1]])
print(B)
```

```
[[ 0 -1  1]
 [ 1  0  1]
 [ 2  0  1]
 [ 3 -7  1]
 [ 4  2  1]]
```

[21]:
```python
# Compute the matrix B*(B^T*B)^{-1}*B^T from the projection formula. Store this␣
↪matrix as P (for 'projection').
# Print your matrix P.
P = B@np.linalg.inv(np.transpose(B)@B)@np.transpose(B)
print(P)
```

```
[[ 6.00000000e-01  4.00000000e-01  2.00000000e-01  1.11022302e-16
  -2.00000000e-01]
 [ 4.00000000e-01  3.25910064e-01  2.28265525e-01 -3.42612420e-02
   8.00856531e-02]
 [ 2.00000000e-01  2.28265525e-01  2.30835118e-01  5.35331906e-02
   2.87366167e-01]
 [ 8.32667268e-17 -3.42612420e-02  5.35331906e-02  9.95717345e-01
  -1.49892934e-02]
 [-2.00000000e-01  8.00856531e-02  2.87366167e-01 -1.49892934e-02
   8.47537473e-01]]
```

[11]:
```python
# Compute the projection pi_U(x) by matrix multiplying P with the column vector␣
↪x defined above. Print your answer.
x = np.array([[-10],
              [-9],
              [0],
              [0],
              [5]])
pi_U = P@x
print(pi_U)
```

```
[[-10.6       ]
 [ -6.53276231]
 [ -2.61755889]
 [  0.23340471]
 [  5.51691649]]
```

We saw in class that the formula for the projection matrix greatly simplifies if we choose an *orthogonal* basis for our subspace. In this case, the projection matrix is given by BB^T.

The function `orth` from the `scipy` package automatically performs Gram-Schmidt orthogonalization. The function is imported below.

[13]:
```python
from scipy.linalg import orth
```

Problem 2 Run your matrix `B` from above through the `orth` function and call the output `B_orth`. Use this orthogonalized basis matrix to recompute the projection matrix, and calle the result `P_orth`. Print your new projection matrix. Check that `P` (from above) and `P_orth` are really the same — a handy function for doing this is `np.allclose(P,P_orth)` which returns `True` if all corresponding entries of the two matrices are approximately equal.

[22]:
```python
## Problem 2 code goes here.
B_orth = orth(B)
```

```
P_orth = B_orth@np.transpose(B_orth)
print(P_orth)
np.allclose(P,P_orth)
```

```
[[ 6.00000000e-01  4.00000000e-01  2.00000000e-01 -1.31327982e-16
  -2.00000000e-01]
 [ 4.00000000e-01  3.25910064e-01  2.28265525e-01 -3.42612420e-02
   8.00856531e-02]
 [ 2.00000000e-01  2.28265525e-01  2.30835118e-01  5.35331906e-02
   2.87366167e-01]
 [-1.31327982e-16 -3.42612420e-02  5.35331906e-02  9.95717345e-01
  -1.49892934e-02]
 [-2.00000000e-01  8.00856531e-02  2.87366167e-01 -1.49892934e-02
   8.47537473e-01]]
```

[22]:
```
True
```

1.0.2 Determinants

The following function computes the determinant of a 2x2 matrix (entered as a numpy array).

[23]:
```
def two_by_two_determinant(A):

    # Input: 2x2 numpy array
    # Output: determinant of the matrix, which is a number

    det = A[0,0]*A[1,1] - A[0,1]*A[1,0]

    return det
```

It's always good to test your code! I typically test on a couple of simple examples where I know the answer, and then something more random.

[24]:
```
I = np.array([[1,0],
              [0,1]])

print(f'The determinant of \n {I} \n is {two_by_two_determinant(I)}')

J = np.array([[0,1],
              [1,0]])

print(f'The determinant of \n {J} \n is {two_by_two_determinant(J)}')

B = np.random.rand(2,2)

print(f'The determinant of \n {B} \n is {two_by_two_determinant(B)}')
```

```
The determinant of
 [[1 0]
```

```
 [0 1]]
 is 1
The determinant of
 [[0 1]
 [1 0]]
 is -1
The determinant of
 [[0.73527305 0.92600654]
 [0.72891443 0.44649898]]
 is -0.34668086011351545
```

Seems like it's working!

Problem 3 Write a function called `three_by_three_determinant` that computes the determinant of a 3x3 matrix. There are several ways to do this; feel free to use the `two_by_two_determinant` function within your code (if you want to).

[43]:
```
## Problem 2 code goes here
def three_by_three_determinant(A):

    # Input: 3x3 numpy array
    # Output: determinant of the matrix, which is a number


    det = A[0,0] * (A[1,1] * A[2,2] - A[2,1] * A[1,2])\
        - A[0,1] * (A[1,0] * A[2,2] - A[1,2] * A[2,0])\
        + A[0,2] * (A[1,0] * A[2,1] - A[1,1] * A[2,0])

    return det
```

Problem 4 Test your function by computing the determinant of the 3x3 identity matrix and compute the determinant of a random 3x3 matrix.

[45]:
```
## Problem 3 code goes here
I_3 = np.identity(3, dtype=int)

print(f'The determinant of \n {I_3} \n is {three_by_three_determinant(I_3)}')

R = np.random.rand(3,3)

print(f'The determinant of \n {R} \n is {three_by_three_determinant(R)}')
```

```
The determinant of
 [[1 0 0]
 [0 1 0]
 [0 0 1]]
 is 1
The determinant of
 [[0.51153866 0.75641833 0.88984108]
 [0.21842741 0.78926988 0.53848848]
```

```
[0.23047642 0.70789736 0.45743445]]
is -0.016288795998822526
```

Of course, determinant functions are built into `numpy`. The point of the problems above was to practice a bit of coding. We may as well test our functions against the built-in numpy function.

Here is a test of the `2_by_2_determinant` function. I will generate a collection of random matrices, compute their determinants using my function and the `numpy` function and find the percentage of instances where the outputs agree. To account for tiny numerical errors, I'll use a function called `isclose` which determines equality up to a small tolerance.

[31]:
```
from math import isclose
```

[32]:
```
num_trials = 20 #Number of random matrices to generate.

successful_trials = 0 # initialize a counter for number of successful trials

for j in range(num_trials):

    A = np.random.rand(2,2)
    det1 = two_by_two_determinant(A) # Determinant computed via my function
    det2 = np.linalg.det(A) # Determinant as computed by numpy

    if isclose(det1,det2):
        successful_trials += 1 # If the determinants are approximately equal,␣
↪add one to the count of good trials

print(f'The success rate is {successful_trials/num_trials*100} %')
```

```
The success rate is 100.0 %
```

Looks good!

Problem 5 Run a similar experiment to test whether your `3_by_3_determinant` function agrees with the `numpy` determinant function.

[46]:
```
## Problem 4 code goes here
num_trials = 20 #Number of random matrices to generate.

successful_trials = 0 # initialize a counter for number of successful trials

for j in range(num_trials):

    A = np.random.rand(3,3)
    det1 = three_by_three_determinant(A) # Determinant computed via my function
    det2 = np.linalg.det(A) # Determinant as computed by numpy

    if isclose(det1,det2):
        successful_trials += 1 # If the determinants are approximately equal,␣
↪add one to the count of good trials
```

```python
print(f'The success rate is {successful_trials/num_trials*100} %')
```

```
The success rate is 100.0 %
```