LECTURE 3 - MARCH 21, 2024

PLOTTING MULTIPLE GRAPHS

WE CONSIDER AGAIN THE EXPECTED UTILITY EXERCISE SEEN IN LECTURE 2. GET THE CODE FROM UniStudium, COPY AND PASTE IT IN A NEW NOTEBOOK.

```
[ ]: # List of tuples of normal parameters
     normals = [(10, 0.8), (9, 0.3), (9, 0.4), (10, 0.5), (11, 0.6)]
     # Exponential utility parameter
     a = 0.1

     # List for EU values
     eus = []   ← EMPTY LIST

     # Compute the expected utility
     for (mu, sigma) in normals:
         eu = expected_utility(mu, sigma, a)
         eus.append(eu)

     # Maximum expected utility
     m = max(eus)

     # Index of the maximum EU
     i = eus.index(m)
                                        PARAMETERS OF THE
                                        NORMAL MAXIMIZING THE EU
     print('Choose:', normals[i])
```

```
[ ]: # Plot the utility function
     import matplotlib.pyplot as plt

     x = np.linspace(-10, 10, 100)
     y = u(x, a)

     plt.title('Utility function')
     plt.xlabel('$ x $')
     plt.ylabel('$ u(x) $')
     plt.plot(x,y)
```

FOR MATH MODE IN LABELS

```
[ ]: # Clear the plot
     plt.clf()

     # Plot the normal densities
     x = np.linspace(5, 15, 200)
     plt.title('Normal densities')
     plt.xlabel('$ x $')

     for (mu, sigma) in normals:
         label = '$ \mu = $' + str(mu) +
                 ', $ \sigma = $' + str(sigma)
         y = f(x, mu, sigma)
         plt.plot(x,y, label = label)

     plt.legend()
     plt.show()
```

MORE POINTS: BETTER GRAPH

STRING CONCATENATION

CONVERT TO A STRING

HOMEWORK: COMPUTATION OF THE QUANTILES

LET X BE A RANDOM VARIABLE WITH CUMULATIVE DISTRIBUTION FUNCTION $F(x) = \mathbb{P}(X \leq x)$.

FOR $\alpha \in (0,1)$, DEFINE THE <u>α-QUANTILE</u> AS

$$q_\alpha = \inf \{ x \in \mathbb{R} \mid F(x) \geq \alpha \}.$$

IF $F(x)$ IS CONTINUOUS WE NEED TO FIND q_α SOLVING

$$F(q_\alpha) = \alpha$$

USE THE FUNCTION norm.ppf() TO COMPUTE THE α-QUANTILE OF PREVIOUS NORMAL DISTRIBUTIONS FOR $\alpha = 0.05$ AND FIND THE MINIMUM AND THE MAXIMUM OF QUANTILES.

REPEAT THE ANALYSIS FOR $\alpha = 0.95$.

ARRAYS USING NUMPY

LISTS ARE NOT EFFICIENT: NUMPY'S ARRAYS ARE EFFICIENT AND SUPPORT ARRAY-ORIENTED PROGRAMMING.

TO USE THEM WE NEED TO IMPORT NUMPY:

```
import numpy as np
```

ONE-DIMENSIONAL ARRAYS (VECTORS)

```
[]: import numpy as np

     # Create an array from a list
     a = np.array([5, 6, 1])
```

INDICES: 0, 1, 2

```
[]: type(a)
```

```
[]: a
```

IMPORTANT: TO VISIT THE ELEMENTS OF AN ARRAY WE CAN USE A for CYCLE WORKING EXACTLY AS WE DID IN LECTURE 2 FOR A LIST. WE CAN ITERATE THE ELEMENTS OR USE AN INDEX IN range(len(a)).

ARITHMETIC OPERATIONS AND COMPARISONS ARE EXECUTED ELEMENT-WISE:

```
[]: a * 2
```

```
[]: a ** 3
```

```
[]: a
```

THE OPERATIONS DON'T CHANGE a

```
[]: a == (a ** 3)
```

```
[]: b = np.array([-1, 5, 7])
```

```
[]: # Linear combination (arrays of same size)
     c = 2 * a - 3 * b
     c
```

EVERY ARRAY HAS METHODS sum, min, max, mean, std, var:

```
[]: a.sum()
```

```
[]: a.mean()
```

```
[]: a.std()
```

WE CAN COMPUTE THE SCALAR PRODUCT THROUGH THE FUNCTION np.dot()

$$a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}, \qquad a \cdot b = (a, b) = a_1 b_1 + a_2 b_2 + a_3 b_3$$

<u>EXAMPLE:</u> COMPUTE THE WEIGHTED MEAN OF

MARK	27	28	30	29	25
CFU	9	6	6	9	6

```
[]: marks = np.array([27, 28, 30, 29, 25])
     cfus  = np.array([9, 6, 6, 9, 6])

     wmean = np.dot(marks, cfus) / cfus.sum()

     print('Weighted mean:', wmean)
```

WE CAN CREATE SPECIAL ARRAYS

INCLUDED
EXCLUDED

```
[]: np.arange(1, 4)
```

WE GET array([1, 2, 3]) ← INDICES 0 1 2

INCLUDED EXCLUDED
BACKWARD

```
[]: np.arange(4, 1, -1)
```

WE GET array([4, 3, 2]) ← INDICES 0 1 2

```
[]: np.zeros(5)
```

WE GET array([0., 0., 0., 0., 0.]) ← INDICES 0 1 2 3 4

```
[]: np.ones(5)
```

WE GET array([1., 1., 1., 1., 1.]) ← INDICES 0 1 2 3 4

```
[]: np.full(5, 13)
```

WE GET array([13, 13, 13, 13, 13]) ← INDICES 0 1 2 3 4

TO PLOT THE GRAPH OF A FUNCTION WE HAVE ALREADY USED THE FOLLOWING

INCLUDED
NUMBER OF POINTS

```
[]: np.linspace(0, 1, 5)
```

WE GET array([0., 0.25, 0.5, 0.75, 1.])

- ## TWO-DIMENSIONAL ARRAYS (MATRICES)

WE CAN BUILD A TWO-DIMENSIONAL ARRAY USING A LIST OF LISTS (ROWS OF THE MATRIX)

ROW INDICES → ← COLUMN INDICES

$$A = \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 10 & -1 & 5 \\ 1 & 0 & 12 & 7 \\ 2 & 5 & -3 & 8 \end{array}$$

[ ]:
```
A = np.array([[10, -1, 5], [0, 12, 7], [5, -3, 8]])
A
```

TO SEE THE SHAPE (NUMBER OF ROWS AND COLUMNS)

[ ]:
```
A.shape
```

TO ACCESS THE ELEMENT IN ROW i AND COL. j:

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

IMPORTANT: A ONE-DIMENSIONAL ARRAY IS NEITHER A $1 \times n$ NOR A $n \times 1$ MATRIX, WE NEED TO RESHAPE IT TO USE IT IN LINEAR ALGEBRA.

[ ]:
```
b
b.shape
```

[ ]:
```
b = b.reshape(3, 1)
b
b.shape
```

[ ]:
```
B = np.array([[1, 2, 6], [-1, 0, 7]])
B
```

```
[]: # Transpose of a matrix
    C = B.T
    C
```

TO COMPUTE THE MATRIX MULTIPLICATION (ROW BY COLUMN) THE SHAPES MUST BE

$$n \times m \qquad m \times \ell$$

EQUAL

WE CAN COMPUTE IT WITH np.matmul() OR THE OPERATOR @

```
[]: np.matmul(A, B)
[]: A @ B
```

THE SAME ⇒ ERROR!

```
[]: np.matmul(A, C)
[]: A @ C
```

THE SAME

$$\begin{bmatrix} 10 & -1 & 5 \\ 0 & 12 & 7 \\ 5 & -3 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 & -1 \\ 2 & 0 \\ 6 & 7 \end{bmatrix}$$

WE CAN CREATE SPECIAL MATRICES

```
[]: np.zeros((2,3))
[]: np.ones((2,3))
[]: np.identity(3)
```

LINEAR ALGEBRA WITH NumPy

GIVEN A MATRIX A WE CAN CREATE A SUBMATRIX SELECTING SOME ROWS AND/OR COLUMNS.

[]:
```python
import numpy as np
A = np.array([[87, 96, 70], [100, 87, 90],
              [94, 77, 90], [100, 81, 82]])
```

COLUMN INDICES

ROW INDICES

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{bmatrix} 87 & 96 & 70 \\ 100 & 87 & 90 \\ 94 & 77 & 90 \\ 100 & 81 & 82 \end{bmatrix} \end{array}$$

SUBSET OF ROWS

[]:
```python
# Only row 1
A[1]
```

[]:
```python
# Rows from 0 to 1
A[0:2]
```

EXCLUDED

[]:
```python
# Rows 1 and 3
A[[1,3]]
```

LIST OF ROW INDICES: IT WORKS ALSO WITH A TUPLE OF ROW INDICES (1, 3)

SUBSET OF COLUMNS

[]:
```python
# Only column 0
A[:, 0]
```

ALL ROWS IN THE COLUMN

[]:
```
# Columns from 1 to 2
A[:, 1:3]
```
EXCLUDED

[]:
```
# Columns 0 and 2
A[:, [0,2]]
```
LIST OF COLUMN INDICES: IT WORKS ALSO WITH A TUPLE OF COLUMN INDICES (0,2)

TO ADD A COLUMN TO A MATRIX

[]:
```
# Create a column vector
b = np.array([10, 2, 50, 3])
b = b.reshape(4,1)

# Concatenate the column to A
B = np.concatenate((A,b), axis=1)
```
TO ADD A COLUMN

TO COMPUTE THE <u>DETERMINANT</u> OF A $\boxed{\text{SQUARE}}$ MATRIX WE USE THE FUNCTION np.linalg.det()

[]:
```
np.linalg.det(A)
```
← ERROR! A IS NOT SQUARE

[]:
```
np.linalg.det(B)
```

IF B IS SQUARE AND $\det(B) \neq 0$, THEN ITS <u>INVERSE</u> MATRIX B^{-1} IS A SQUARE MATRIX SUCH THAT $B \cdot B^{-1} = B^{-1} \cdot B = I$ ← IDENTITY MATRIX

$$\begin{bmatrix} 1 & \cdots & 0 \\ 0 & 1 & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$$

[]:
```
# Inverse of matrix B
Binv = np.linalg.inv(B)
B @ Binv
Binv @ B
```
MATRIX PRODUCT (ROW BY COLUMN)

TESTING LINEAR DEPENDENCE

GIVEN n VECTORS $\underline{a}_1, \underline{a}_2, \dots, \underline{a}_n \in \mathbb{R}^m$, THEY ARE LINEAR DEPENDENT IF WE CAN FIND n NUMBERS $x_1, x_2, \dots, x_n \in \mathbb{R}$ NOT ALL ZERO SUCH THAT

NULL VECTOR

$$\boxed{x_1 \underline{a}_1 + x_2 \underline{a}_2 + \dots + x_n \underline{a}_n = \underline{0}}$$

EQUIVALENTLY, WE CAN FIND A VECTOR, LET'S SAY $\underline{a}_n$, AND $n-1$ NUMBERS $y_1, y_2, \dots, y_{n-1} \in \mathbb{R}$ SUCH THAT

$$\boxed{\underline{a}_n = y_1 \underline{a}_1 + y_2 \underline{a}_2 + \dots + y_{n-1} \underline{a}_{n-1}}$$

$\Rightarrow$ $\underline{a}_n$ IS REDUNDANT

THE RANK OF $A = [\underline{a}_1 \ \underline{a}_2 \ \cdots \ \underline{a}_n]$ IS THE MAXIMUM NUMBER OF LINEAR INDEPENDENT COLUMN VECTORS. WE COMPUTE IT WITH THE FUNCTION `np.linalg.matrix_rank()`

```
[]: A = np.array([[1,2,3],[1,2,5],[1,2,1]])
     A
```

```
[]: # Compute the rank of the matrix
     np.linalg.matrix_rank(A)
```

HOMEWORK

GIVEN A MATRIX A AND A NUMBER r, DEFINE A FUNCTION THAT EXTRACTS FROM A A SUBMATRIX WITH r LINEARLY INDEPENDENT COLUMNS (IF THEY EXIST).

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

LIBRARY TO GENERATE SUBSETS OF INDICES

```
[]: A = np.array([[1, 2, 3], [1,2,5], [1,2,1]])
```

```
[]: # Find a subset of r linearly independent
     columns of A (if it exists)
    def non_redundant (A, r):
        rows, cols = A.shape
        indices = range(cols)  #[0, 1, ..., cols-1]

        # Iterate among all subsets of r columns
        for subcols in itertools.combinations(indices, r):
            subA = A[:, subcols]
            r_subA = np.linalg.matrix_rank (subA)
            if r_subA == r:
                return subA
```

SUBSETS OF INDICES WITH r ELEMENTS

SUBMATRIX

RANK

FOUND A SUBMATRIX WITH r LIN. INDEP. COL.

```
[]: non_redundant(A, 1)
```

```
[]: non_redundant (A,2)
```

```
[]: non_redundant (A,3)
```

IT DOES NOT RETURN ANYTHING SINCE THERE ARE NO 3 LIN. INDEP. COLUMNS IN A !!