‘Data Analysis’
Computer Lab Session
Linear Algebra

Master 1 MLDM / CPS^2 / COSI / 3DMT
Saint-Étienne, France

Ievgen Redko

1 Outline

1. Introduction to Python (1/2)
2. Introduction to Python (2/2)
3. Probability, Random Variables and Probability Distributions
4. **Linear Algebra (1/2)**
5. **Linear Algebra (2/2)**
6. Principal Component Analysis
7. Linear Regression (1/2)
8. Linear Regression (2/2)
9. Clustering (1/2)
10. Clustering (2/2)

Outcome

The objective of this lab is to become familiar with Python functions for working with linear algebra, especially with basic functions and instructions for handling matrices in Python, and for being able to compute matrix multiplication and inversion.

2 Quick Recap on Vectors and Matrices in Python

2.1 Constructing Matrix Objects

Matrices can be constructed using the functions `numpy.array()` and `numpy.matrix()`(also denoted by `numpy.mat()`.

For example, Hilbert matrices are often studied in numerical linear algebra because they are easy to construct but have surprising properties.

```
H = np.array([[1, 1/2, 1/3],[1/2, 1/3, 1/4],[1/3, 1/4, 1/5]])
H = np.mat([[1, 1/2, 1/3],[1/2, 1/3, 1/4],[1/3, 1/4, 1/5]])
print(H)
```

Here H is the 3×3 Hilbert matrix, where entry (i, j) is $1/(i + j - 1)$. Note that numpy.matrix() is necessarily a 2D object while numpy.array() can be N-dimensional for any $N > 2$. In general, prefer to use arrays instead of matrices.

The numpy.arange() function will generate a sequence, for example numpy.arange(1,3) will produce the result 1 2. This can be used together with numpy.reshape() command to construct matrices as follows:

```
X = np.array([np.arange(1,13)]).reshape((3,4))

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

will produce a matrix with 3 rows and $12/3 = 4$ columns.

Another useful function for creating matrices in Python is numpy.diag(). It extracts or replaces the diagonal of a matrix, or constructs a diagonal matrix. This function is used for creating identity matrices (a square matrix of size $n \times n$ with ones on the main diagonal and zeros elsewhere), for example:

```
n = 5
print(np.diag(np.arange(n)+1))

[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
```

You can also construct an identity matrix using numpy.eye() function and a matrix of ones or zeros using numpy.zeros() and numpy.ones() commands, respectively.

Remark 1 *Note that numpy.zeros() and numpy.ones() take tuples as arguments that define the shape of the returned object. For instance, to create an array of size* 3×5*, one writes:*

```
np.zeros((3,5))
```

Executing it without the parentheses will throw an error.

```
A = np.zeros(3,5)
TypeError: data type not understood
```

Exercise Use the numpy.array(), numpy.arange() and numpy.tile() functions to construct the following 5×5 Hankel matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

2.2 Matrix Properties

The **dimension** of a matrix is its number of rows and its number of columns. For example,

```
H.shape()
(3, 3)
```

The **determinant** of a 2×2 matrix $\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$ matrix can be calculated as $ad - bc$. For larger square matrices, the calculation becomes more complicated. It can be found in Python using the numpy.linalg.det() function, as in

```
print(np.linalg.det(H))
0.00046296296296296146
```

We can also compute the **trace** (the sum of the diagonal entries) of any matrix using a home-made function such as

```
np.sum(np.diag(data))
```

The trace function uses the numpy.sum() function that returns the sum of all the values present in its arguments. It can be applied to the matrices X and H:

```
np.sum(np.diag(H))
1.5333333333333332
```

The .T function is used to calculate the **matrix transpose** H^T:

```
H.T
[[1.         0.5        0.33333333]
 [0.5        0.33333333 0.25      ]
 [0.33333333 0.25       0.2       ]]
```

2.3 Triangular Matrices

The functions numpy.triu() and numpy.tril() can be used to obtain the lower and upper triangular parts of matrices. The output of the functions is a matrix of with all values below/above the main diagonal set to zero. For example,

```
np.tril(H)
[[1.         0.         0.        ]
 [0.5        0.33333333 0.        ]
 [0.33333333 0.25       0.2       ]]
```

2.4 Matrix arithmetic

Multiplication of a matrix by a scalar constant is the same as multiplication of a vector by a constant. For example, using the H matrix from the previous section, we can multiply each element by 2 as in

```
print(2*H)

[[2.         1.         0.66666667]
 [1.         0.66666667 0.5       ]
 [0.66666667 0.5        0.4       ]]
```

Elementwise addition of matrices also proceeds as for vectors. For example,

```
print(H+H)

[[2.          1.          0.66666667]
 [1.          0.66666667  0.5       ]
 [0.66666667  0.5         0.4       ]]
```

When adding matrices, always ensure that the dimensions match properly. If they do not match correctly, an error message will appear.

The command `X * Y` performs elementwise multiplication. Note that this differs from the usual form of matrix multiplication that we will discuss below. For example,

```
print(H * H)

[[1.          0.25        0.11111111]
 [0.25        0.11111111  0.0625    ]
 [0.11111111  0.0625      0.04      ]]
```

Again, in order for this kind of multiplication to work, the dimensions of the matrices must match. The true matrix multiplication is done via the `numpy.dot()` command as follows:

```
print(np.dot(H,H)) # or print(H.dot(H))

[[1.36111111  0.75        0.525     ]
[0.75        0.42361111  0.3       ]
[0.525       0.3         0.21361111]]
```

2.5 Matrix Inversion

The inverse of a square $n \times n$ matrix A, denoted by A^{-1}, is the solution to the matrix equation $AA^{-1} = I$, where I is the $n \times n$ identity matrix. We can view this as n separate systems of linear equations in n unknowns, whose solutions are the columns of A^{-1}.

For example, with

$$A = \begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix},$$

the matrix equation

$$\begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

is equivalent to the two equations:

$$\begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } \begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} b_{12} \\ b_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The usual computational approach to finding A^{-1} involves solving these two equations.

However, this is not always a good idea! Often the reason we are trying to find A^{-1} is so that we can solve a system $Ax = b$ with the solution $x = A^{-1}b$. It doesn't make sense from a computational point of view to solve n systems of linear equations in order to obtain a result which will be used as the solution to one system. If we know how to solve systems, we should use that knowledge to solve $Ax = b$ directly. Furthermore, using A^{-1} may give a worse approximation to the final result than the direct approach, because there are so many more operations involved, giving opportunities for much more rounding error to creep into our results.

As an example, we compute the inverse of the 3×3 Hilbert matrix introduced previously:

```
Hinv = np.linalg.inv(H)
```

To verify that this is the inverse of H, we can check that the product of $Hinv$ and H is the 3×3 identity:

```
print(H.dot(Hinv))

# Identity matrix
[[ 1.00000000e+00  -1.40628250e-15  -3.33066907e-15]
 [ 0.00000000e+00   1.00000000e+00   0.00000000e+00]
 [-2.22044605e-17  -4.84057239e-15   1.00000000e+00]]
```

Remark 2 *For arbitrary rectangular matrices, use the Moore-Penrose pseudo-inverse function numpy.linalg.pinv().*

2.6 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors can be computed using the function numpy.linalg.eig(). For example, we can find the eigenvalues and eigenvectors of H as follows:

```
w,v = np.linalg.eig(H)
print(w)
print(v)

 # eigenvalues
[1.40831893  0.12232707  0.00268734]

# eigenvectors
[[ 0.82704493   0.54744843   0.12765933]
 [ 0.4598639   -0.52829024  -0.71374689]
 [ 0.32329844  -0.64900666   0.68867153]]
```

To see what this output means, let v_1 denote the first column of the v output, i.e. $[0.827\ \ 0.459\ \ 0.323]^T$. This is the first eigenvector, and it corresponds to the eigenvalue $w_1 = 1.408$.

Thus, $Hv_1 = 1.408v_1$.

Denoting the second and third columns of v by v_2 and v_3, we have $Hv_2 = 0.122v_2$, and $Hv_3 = 0.00268v_3$.

2.7 The Singular Value Decomposition of a Matrix

The singular value decomposition of a square matrix A consists of three square matrices, U, D, and V. The matrix D is a diagonal matrix. The relation among these matrices is $A = UDV^T$.

The matrices U and V are said to be *orthogonal*, which means that $U^{-1} = U^T$ and $V^{-1} = V^T$.

The singular value decomposition of a matrix is often used to obtain accurate solutions to linear systems of equations.

The elements of D are called the *singular values* of A. Note that $A^TA = VD^2V^{-1}$. This is a "similarity transformation" which tells us that the squares of the singular values of A are the eigenvalues of A^TA.

The singular value decomposition can be obtained using the function numpy.linalg.svd(). For example, the singular value decomposition of the 3×3 Hilbert matrix H is

```
U,D,V = np.linalg.svd(H)

# U matrix
[[-0.82704493   0.54744843   0.12765933]
```

```
 [-0.4598639   -0.52829024  -0.71374689]
 [-0.32329844  -0.64900666   0.68867153]]

 # D matrix
[1.40831893  0.12232707  0.00268734] # D matrix of singular values

# V matrix
[[-0.82704493  -0.4598639   -0.32329844]
 [ 0.54744843  -0.52829024  -0.64900666]
 [ 0.12765933  -0.71374689   0.68867153]]
```

We can verify that these components can be multiplied in the appropriate way to reconstruct H:

```
print(U.dot(np.diag(D)).dot(V))  # or np.dot(U*D, V)

[[1.          0.5         0.33333333]
 [0.5         0.33333333  0.25      ]
 [0.33333333  0.25        0.2       ]]
```

Because of the properties of the U, V and D matrices, the singular value decomposition provides a simple way to compute a matrix inverse. For example, $H^{-1} = VD^{-1}U^T$ and can be recalculated as

```
print(V.T.dot(np.diag(1./D)).dot(U.T))

[[    9.   -36.    30.]
 [  -36.   192.  -180.]
 [   30.  -180.   180.]]
```

3 Introduction to Computational Linear Algebra

Exercise 1: Warm-up

Let us consider the following equations:

$$(1) \quad l_1 : 3x_1 - 4x_2 = 6$$

$$(2) \quad l_2 : x_1 + 2x_2 = -3$$

1. Solve this system of equations analytically. What is the solution?
2. Plot the two lines l_1 and l_2. What do you observe?
3. Plot the solution (intersection of the two lines) with two dotted lines.

From this figure it follows that there are 3 possible cases of solutions to the system:

1. Exactly one solution (when the lines intersect in one point)
2. Infinitely many solutions (when the lines coincide).
3. No solutions (when the lines are parallel but not identical); in this case, the system of linear equations is said to be inconsistent.

One of the most important applications of linear algebra is in solving systems of linear equations. For example, we represent the system

$$\left\{\begin{array}{rcr} 3\,x_1 - 4\,x_2 & = & 6 \\ x_1 + 2\,x_2 & = & -3 \end{array}\right\} \text{ as } Ax = b, \text{ where } A = \begin{bmatrix} 3 & -4 \\ 1 & 2 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad b = \begin{bmatrix} 6 \\ -3 \end{bmatrix},$$

and solve it as

$$x = A^{-1}b = \begin{bmatrix} 0.2 & 0.4 \\ -0.1 & 0.3 \end{bmatrix} \begin{bmatrix} 6 \\ -3 \end{bmatrix} = \begin{bmatrix} 0 \\ -1.5 \end{bmatrix}$$

In Python, matrices are inverted and linear systems of equations are solved using the numpy.linalg.solve(). The matrix multiplication can be performed using numpy.dot() function.

Example:

```
a = np.array([[3,-4], [1,2]])
b = np.array([6,-3])
x = np.linalg.solve(a, b)
print(x)
```

The matrix x is the solution:

$$x = \begin{bmatrix} 0 \\ -1.5 \end{bmatrix}.$$

Exercise 2: Another example

With Python, find the solution of the following system of linear equations:

$$(1) \quad 3x_1 + 2x_2 = 7$$

$$(2) \quad x_1 - 3x_2 = -5$$

1. Solve this system using numpy.linalg.solve() function.

2. Solve this system using the direct inversion of matrix A (numpy.linalg.inv) and its further right-multiplication by b. Do the two solutions coincide?

3. Replace the second equation with

$$-\frac{9}{2}x_1 - 3x_2 = -5$$

and run your code once again. Did it work this time? Why? Calculate the determinant of the coefficient matrix to justify your response.

4. Modify your code using the numpy.linalg.lstlq command and a try-catch conditional structure with LinAlgError.

Exercise 3: Solving a real-world problem

Need to check

An insurance company has four types of policies, which we will label A, B, C, and D.

- They have a total of 245 921 policies.
- The annual income from each policy is 10 € for type A, 30 € for type B, 50 € for type C, and 100 € for type D.
- The total annual income for all policies is 7 304 620 €.
- The claims on these policies arise at different rates. The expected number of type A claims is 0.1 claims per year, type B 0.15 claims per year, type C 0.03 claims per year, and type D 0.5 claims per year.
- The total expected number of claims for the company is 34 390.48 per year.
- The expected size of the claims is different for each policy type. For type A, it is 50 €, for type B it is 180 €, for type C it is 1500 €, and for type D it is 250 €.
- The expected total claim amount is 6 864 693 €. This is the sum over all policies of the expected size of claim times the expected number of claims in a year.

Use Python to answer the following questions:

1. Find the total number of each type of policy.
2. Find the total income and total expected claim size for each type of policy.

✓

Exercise 4: Polynomials

Let $[x_1, x_2, x_3, x_4, x_5, x_6]^T = [10, 11, 12, 13, 14, 15]^T$. Find the coefficients of the quintic polynomial $f(x)$ for which

$$[f(x_1), f(x_2), f(x_3), f(x_4), f(x_5), f x_6)]^T = [25, 16, 26, 19, 21, 20]^T.$$

The quintic polynomial $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5$ can be viewed as the matrix product of the row vector $[1, x, x^2, x^3, x^4, x^5]$ with the column vector $[a_0, a_1, a_2, a_3, a_4, a_5]^T$.

Work out the matrix version of this to give $[f(x_1), f(x_2), f(x_3), f(x_4), f(x_5), f(x_6)]^T$.

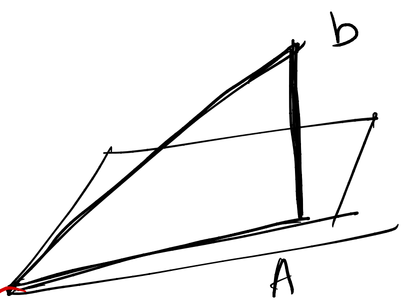
✓

Exercise 5: Spectrum of a matrix

Let X be a matrix defined as

$$X = \begin{bmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 9 \end{bmatrix}$$

1. Calculate the matrix $H = X(X^TX)^{-1}X^T$
2. Calculate the eigenvalues and eigenvectors of H.
3. Calculate the trace of the matrix H, and compare with the sum of the eigenvalues.
4. Calculate the determinant of the matrix H, and compare with the product of the eigenvalues.
5. Verify that the columns of X and $I - H$ are eigenvectors of H, here $HX = X$ and $H(I - H) = 0$.



from $A^T(A\hat{x}-\hat{b}) = 0$

$\Sigma^2 x = V \Sigma U x$

I. Redko

Exercise 6: Singular value decomposition

1. Generate a random two-dimensional matrix X of arbitrary size n.
2. Set the second row of X to be a linear combination of the two rows with arbitrary coefficients such that:
$$X(1,:) = \alpha_1 X(0,:) + \alpha_2 X(1,:)$$
where $\alpha_1 > \alpha_2$.
3. Calculate the correlation between the two rows and make sure that they are correlated. Confirm it by plotting X in two dimensions.
4. Decompose matrix X using the SVD method. Use Eckart-Young theorem to find the best rank-one approximations of X. Plot it together with X. What do you observe?
5. Propose a solution to a system of linear equations based on SVD for an arbitrary matrix A and a vector b having the lowest l_2 norm, ie,
$$\min_x \|Ax - b\|_2^2.$$
6. Verify that the obtained analytical solution coincides with the Moore-Penrose pseudo-inverse.
7. Test you solution on the following example
$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 0 & 0 \end{bmatrix}, b = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}.$$

Exercise 7: Finding the largest eigenvalue with the power method

1. Implement power method that finds the largest eigenvalue and its associated eigenvector as follows:
```
# A is an NxN matrix, num_simulations is the number of iterations
def power_iteration(A, num_simulations):
    # Choose a random eigenvector x having the size N to start with
        # do until the required number of iterations is reached
    for _ in range(num_simulations):
        # calculate the matrix-by-vector product Ax
        # calculate the norm of the Ax product
        # re normalize the vector using the norm Ax/norm(Ax)
        # calculate the eigenvalue as x^T A x/x^T x

    return x, eigenvalue
```
2. Generate a 3×3 symmetric matrix A as follows:
$$A = \frac{B + B^T}{2},$$
where B is any 3×3 matrix with strictly positive values.
3. Run your algorithm and check if the obtained result coincides with that of numpy.linalg.eig() command.
4. Modify your function to store all intermediate value of the eigenvalue and plot its convergence to the true one.