Solutions Lecture 3 (Chapter 5)

Make sure to import Numpy, and the linear optimization packages `linprog` and `pulp`, to be able to do all exercises.

```
import numpy as np
from scipy.optimize import linprog
import pulp
```

Question 1

Suppose that $m \geq n$ and that $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. If the columns of A are independent, i.e., if the rank of A is n, then the unique least squares solution to

$$\min_x ||Ax - b||_2.$$

is given by $x^* = (A^T A)^{-1} A^T b$. (We need the assumption that $m \geq n$ to guarantee that $(A^T A)^{-1}$ exists.)

Write a function that takes as input a matrix A and vector b. It should return the least squares solution x^* if A has full rank, and a message saying that A is not of full rank if not. You may use an if/else-statement for this case distinction.

It should give the following output on the inputs below.

```
A = np.array([
 [0.61776137, 0.9880466,  0.93526716],
 [0.25839379, 0.66983101, 0.55140378],
 [0.42598981, 0.47034738, 0.67713771],
 [0.05808045, 0.60282648, 0.968094  ],
 [0.6152295,  0.18224588, 0.23185116]])

b = np.array([
 [0.68873052],
 [0.01618962],
 [0.67498566],
 [0.73718501],
 [0.0448842 ]])

print(least_squares(A,b))
```

```
[[ 0.05234646]
```

```
[-0.76893707]
[ 1.3285266 ]]
```

```
A = np.array([
 [0.61776137, 0.9880466,  0.9880466],
 [0.25839379, 0.66983101, 0.66983101],
 [0.42598981, 0.47034738, 0.47034738],
 [0.05808045, 0.60282648, 0.60282648],
 [0.6152295,  0.60282648, 0.60282648]])

b = np.array([0.68873052,0.01618962, 0.67498566, 0.73718501, 0.0448842])

# Columns are not linearly independent
print(least_squares(A,b))
```

```
Matrix A is not of full rank
```

```
def least_squares(A,b):
    m, n = np.shape(A)
    rank_A = np.linalg.matrix_rank(A)

    if rank_A == n:
        return (np.linalg.inv(A.T @ A) @ A.T) @ b
    else:
        return "Matrix A is not of full rank"
```

Question 2

In this exercise we are going to implement Cramer's rule for solving the system $Ax = b$ in case $A \in \mathbb{R}^{n \times n}$ is an invertible matrix. Cramer's rule determines the solution $x = [x_0, \dots, x_{n-1}]$ by

$$x_i = \frac{\det(A_i)}{\det(A)} \quad \text{for } i = 0, \dots, n-1,$$

where A_i is the matrix A in which column i is replaced by b.

Write a function `cramer(A,b)` that computes the solution x according to Cramer's rule (you may use one for-loop in your implementation).

Your function should give the following output on the given input.

```
A = np.array([
 [0.61776137, 0.9880466,  0.93526716],
 [0.25839379, 0.66983101, 0.55140378],
 [0.42598981, 0.47034738, 0.67713771]
 ])

b = np.array([0.68873052, 0.01618962, 0.67498566])
```

```
# Solution with Cramer's rule
print(cramer(A,b))

# Solution with np.linalg.solve() for comparison
print(np.linalg.solve(A,b))
```

```
[ 2.88381098 -0.97013863 -0.14352512]
[ 2.88381098 -0.97013863 -0.14352512]
```

```
def cramer(A, b):
    m, n = np.shape(A)

    x = np.zeros(m)
    # Compute determinant after replacing i-th column by `b`.
    for i in range(m):
        A_mod = A.copy()
        A_mod[:, i] = b
        x[i] = np.linalg.det(A_mod)
    return x / np.linalg.det(A)
```

Question 3

Assume we are given a matrix $A \in \mathbb{R}^{m \times n}$ with the property that every subset of m columns of A is linearly independent. This is, e.g., typically the case if the numbers in A would be randomly generated (we will see this later in the course).

One way to obtain a solution to an underdetermined system ($n > m$) system $Ax = b$ is to pick a list of indices $R \subseteq \{0, \dots, n-1\}$ of size $|R| = m$. Set $x_i = 0$ for all $i \notin R$ (there are $n - m$ such variables), and determine the values in $x_R = (x_i)_{i \in R}$ by solving the system $Bx_R = b$, where B is the matrix containing the columns of A corresponding to the indices in R.

Write a function `underdetermined(A,b,R)` that takes as input arrays A, b and a list R as above, and returns a solution $x \in \mathbb{R}^n$ to $Ax = b$ satisfying the above description.

Your function should give the following output on the given input.

```
A = np.array([
 [0.83, 0.43, 0.16, 0.01, 0.86, 0.87, 0.36, 0.68],
 [0.29, 0.81, 0.78, 0.97, 0.2,  0.28, 0.44, 0.07],
 [0.18, 0.81, 0.91, 0.88, 0.54, 0.89, 0.53, 0.88]])

b = np.array([0.68873052, 0.01618962, 0.67498566])

R = [0,4,7]

# Determine solution
x = underdetermined(A,b,R)
print(x)
```

```
# Verify correctness
print(A @ x - b) # All numbers almost zero (e-16 is numerical inaccuracies)
```

```
[-3.16315793  0.          0.          0.          5.31389279  0.
  0.         -1.84675911]
[-2.22044605e-16  1.28369537e-16  1.11022302e-16]
```

```
def underdetermined(A,b,R):
    m, n = np.shape(A)

    x = np.zeros(n)
    x[R] = np.linalg.solve(A[:,R],b)
    return x
```

Question 4

In this exercise, we will look into the basics of principle component analysis (PCA); we emphasize this is not a complete explanation, but hopefully helps develop some intuition.

We are given an $m \times n$ matrix A containing n data points, that all have m features. That is, every column of A corresponds to a data point. It could, e.g., be that we obtained the weight, height, IQ, and age of a group of 100 people. In this case we would have $m = 4$ features and $n = 100$ data points.

For a given feature, the sample mean $\bar{\mu}_i$ of row i in A is defined as the average of all the values in that row. The (symmetric) $m \times m$ covariance matrix S is defined by

$$S = \frac{1}{n-1} BB^T$$

where B is the matrix that is obtained from A by substracting from every row its sample mean (so that every row has mean zero).

a) Write a function `covariance()` that takes as input the matrix A and outputs the matrix S.

```
# Covariance matrix function
def covariance(A):
    m,n = np.shape(A)

    # Compute matrix S
    B = A - np.mean(A, axis=1)[:,None]
    S = 1/(n-1)*(B @ B.T)
    return S
```

The covariance matrix contains on its diagonals the sample variance

$$\frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{\mu}_i)^2$$

and on its off-diagonal elements the covariance of features i and j at position (i, j). The trace of S, the total variance in the data, is equal to the sum of the eigenvalues of the matrix S, because of the spectral

decomposition (recall from your Linear Algebra course)

$$S = \sum_{i=0}^{m-1} \lambda_i v_i v_i^T$$

where λ_i are the eigenvalues of S and v_i the eigenvector of λ_i for $i = 1, \dots, m$.

b) Write a function `ordered_components()` that takes as input a symmetric matrix S and outputs a vector containing the eigenvalues in ascending order (smallest to largest), and a matrix containing the corresponding eigenvectors ordered according to the order of the eigenvalues. As an example, if we have three eigenvalues $\{\lambda_0, \lambda_1, \lambda_2\}$ with $\lambda_2 \leq \lambda_0 \leq \lambda_1$, then the output should be $[\lambda_2, \lambda_0, \lambda_1]$ and the matrix should be $[v_2\ v_0\ v_1]$ where the v_i are the eigenvectors corresponding to eigenvalues λ_i. Hint: The function `np.argsort()` from Chapter 4 might be useful here.

```
def ordered_components(S):
    eig_values, eig_vectors = np.linalg.eig(S)

    order = np.argsort(eig_values)
    return eig_values[order], eig_vectors[:,order]
```

We can view the covariance matrix as a linear sum of "components" being the eigenvectors of S, and the corresponding eigenvalues indicates how important a component is for the data, i.e., how much it contributes to the total variance of the data.

c) Write a function `contributions` that takes a vector $x = [x_0, \dots, x_{n-1}]$ as input and output the numbers

$$y_i = \frac{\sum_{j=i}^{n-1} x_j}{\sum_{j=0}^{n-1} x_j}$$

i.e., y_i is the fraction that the last i numbers contribute to the sum of all numbers in x.

```
def contributions(x):
    x = x[-1::-1] # Reverse the array
    y = np.cumsum(x)/np.sum(x)
    return y[-1::-1]
```

Your functions in $a) - c)$ should give the following output on the indicated input.

```
# Stylized data in which there is clear correlation
# between feature 0 and 1 (but not with the others)
n = 10
weight = np.linspace(60,90,n)
height = np.linspace(150,190,n)
IQ = 110*np.ones(n)
age = np.repeat([23,24],np.round(n/2))

A = np.vstack((weight,height,IQ,age)) #Can also stack > 2 arrays with vstack()
```

```
A = np.around(A,decimals=1) # Round values in A to one decimal
print('A = \n', A)

# Compute covariance matrix
S = covariance(A)
S = np.around(S, decimals = 1) # Round values in S to one decimal
print('S = \n',S)

# Compute eigenvalues and vectors
eig_S, eig_vector_S = ordered_components(S)
print('Eigenvalues are \n',eig_S)
print('Eigenvectors are \n',eig_vector_S)

# Compute eigenvalue contributions of k largest eigenvalues
eig_contributions = contributions(eig_S)
print('Contributions of k largest eigenvalues \n', eig_contributions)
```

```
A =
 [[ 60.   63.3  66.7  70.   73.3  76.7  80.   83.3  86.7  90. ]
 [150.  154.4 158.9 163.3 167.8 172.2 176.7 181.1 185.6 190. ]
 [110.  110.  110.  110.  110.  110.  110.  110.  110.  110. ]
 [ 23.   23.   23.   23.   23.   24.   24.   24.   24.   24. ]]
S =
 [[101.9 136.    0.    4.6]
 [136.  181.4   0.    6.2]
 [  0.    0.    0.    0. ]
 [  4.6   6.2   0.    0.3]]
Eigenvalues are
 [-5.02441935e-02  0.00000000e+00  9.98121483e-02  2.83550432e+02]
Eigenvectors are
 [[-0.76817281  0.         -0.22461531 -0.59954858]
 [ 0.58467307  0.          0.13549239 -0.7998745 ]
 [ 0.          1.          0.          0.        ]
 [-0.26089832  0.          0.96498175 -0.02724496]]
Contributions of k largest eigenvalues
 [1.         1.00017717 1.00017717 0.99982522]
```

From the output we can see that the largest eigenvalue is much larger than the others, and accounts for almost all of the total variance (sum of the eigenvalues). This implies that the data in A, which are data points in $\mathbb{R}^4$, are essentially contained in a 1-dimensionsal subspace of $\mathbb{R}^4$ formed by the "principle component" being the eigenvector of the largest eigenvalue, and so the "dimension of the data" is in this sense smaller than that of the data itself, i.e.,

$$S \approx \lambda_{\max} v_{\max} v_{\max}^T.$$

Such insights can help identify which features are stastistically important in the data, and which are not.

Question 5

Consider a knapsack problem in which we have $k = n^2$ items $I = \{1, \dots, k\}$ that all have a weight w_i and value v_i for $i \in I$. We want to select a subset of items whose total weight is at most W, whilst their total value is maximized at the same time. Furthermore, we can select at most r from the first n items $\{1, \dots, n\}$, at most r from the second n items $\{n+1, \dots, 2n\}$ etc…

This problem can be modelled with the following binary linear optimization problem, where the modeling means $x_i = 1$ if item i is included, and $x_i = 0$ otherwise.

$$
\begin{array}{lllll}
\max & \displaystyle\sum_{i=1}^{n^2} v_i x_i & & & \\
\text{s.t.} & \displaystyle\sum_{i=a\cdot n+1}^{(a+1)\cdot n} x_i & \leq & r & \text{for } a = 0, \dots, n-1 \\
& \displaystyle\sum_{i=1}^{n^2} w_i x_i & \leq & W & \\
& x_i & \in & [0,1] & \text{for all } i = 1, \dots, n^2 \\
& x_i & \in & \{0,1\} & \text{for all } i = 1, \dots, n^2
\end{array}
$$

Define a function `knapsack(n,v,w,W,r)` that takes as input n, the weight and values vectors $v, w \in \mathbb{R}^{n^2}$, and the numbers W and r, and returns the optimal solution to the problem above as a binary vector using `linprog`. Generate the constraint matrix using NumPy functions; it might help to start with `np.eye(n)` which is the $n \times n$ identity matrix. You may use list comprehension once in your function when defining the bounds on the variables. As an example, note that for $k = 9$, i.e., $n = 3$, the constraint matrix of the inequalities should look like this:

$$
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 & w_9
\end{bmatrix}.
$$

```
n = 3
w = np.arange(1,n**2+1)
v = np.arange(n**2+1,1,-1)
W = 3*n**2
r = 2

solution = knapsack(n,v,w,W,r)
print('Solution is \n',solution)
```

```
Solution is
 [ 1.  1. -0.  1.  1. -0.  1.  1.  0.]
```

```
# from scipy.optimize import linprog (already did this at top of exercise file)

def knapsack(n,v,w,W,r):
    w = w[None,:] # Turn into vector of shape (1,n**2);
```

```
                    # this is important for np.vstack() later on

    # Coefficients of the objective function
    c = -v # Maximization to minimization

    # Matrix of the inequality constraints (Ux <= z)
    x = np.eye(n)
    x = np.repeat(x,n).reshape(n,n**2)
    U = np.vstack((x,w))

    # RHS of the inequality constraints (Ux <= z)
    z = np.hstack((r*np.ones(n),W))

    # Bounds for the variables x1 and x2 (l = 0, u = 1)
    x_bounds = [(0,1) for i in range(n**2)]

    # Optimize model
    result = linprog(c, A_ub=U, b_ub=z, \
                     bounds=x_bounds, \
                     integrality=1)

    return result.x
```

Question 6

Implement and solve the problem from Question 5 with the PuLP package, i.e., write the function `knapsack_pulp(n,v,w,W,r)` that yields the same output.

```
n = 3
w = np.arange(1,n**2+1)
v = np.arange(n**2+1,1,-1)
W = 3*n**2
r = 2

solution = knapsack_pulp(n,v,w,W,r)
print('Solution is \n',solution)
```

```
Solution is
 [1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0]
```

```
# import pulp (already did this at top of exercise file)

def knapsack_pulp(n,v,w,W,r):
    # n does not necessarily have to be an input argument
    # We could have obtained it from v with np.sqrt(np.size(v)).astype(int)

    k = n**2 # Total number of items
```

```
    # Instantiate problem
    prob = pulp.LpProblem("Knapsack", pulp.LpMaximize)

    # With list comprehension
    x = np.array([pulp.LpVariable(f"x_{i}", cat='Binary') for i in range(k)])

    # Set objective function
    prob += pulp.lpSum(v*x)

    # Set constraints
    for a in range(n):
        prob += pulp.lpSum([x[i] for i in range(a*n,(a+1)*n)]) <= r

    prob += pulp.lpSum(w*x) <= W

    # Solve the problem
    prob.solve()

    return [pulp.value(x[i]) for i in range(k)]
```