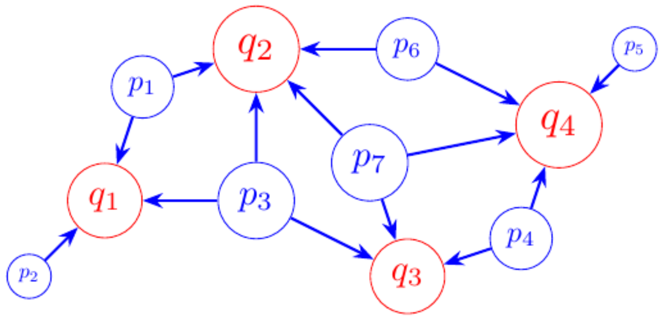
Семинар 6

1. Лекция 6: Транспортная задача

Транспортная задача является важной как с практической точки зрения, так и с исторической. В своей базовой постановке задача очень просто описывается, однако ее решение имеет некоторые особенности.

Предположим, что наша экономика включает в себя m заводов и n потребителей. Каждый завод i физически может поставить не более p_i единиц продукции. Каждый потребитель j предъявляет спрос не более чем на q_j единиц продукции. Стоимость доставки единицы продукции с завода i потребителю j равна c_{ij}, а соответствующий объем равен x_{ij}.



T.J. Sargent, J. Stachurski, CC BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0/

Основной задачей является нахождение такого набора значений $\{x_{ij}\}$, который будет минимизировать совокупные издержки на транспортировку

$$\min_x \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$\sum_{i=1}^{m} x_{ij} = q_j$$

$$\sum_{j=1}^{n} x_{ij} = p_i$$

$$x_{ij} \geqslant 0$$

Также можно показать, что

$$\sum_{j=1}^{n} q_j = \sum_{j=1}^{n} \sum_{i=1}^{m} x_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} = \sum_{i=1}^{m} p_i$$

1.1. Векторизация задачи

Описанная выше задача сформулирована в терминах матриц:

- матрица затрат $C = (c_{ij})$;
- матрица целевых переменных $X = (x_{ij})$;
- векторы производственных возможностей p и спроса q.

Но максимум, на что согласны описанные выше функции `linprog` и `linprog_simplex`, — это принимать матрицы в качестве элементов ограничений.

Надо как-то перейти к векторному представлению задачи.

Первый, достаточно прямолнейный способ, следующий

$$
\begin{aligned}
\min_X \ & \operatorname{tr} C'X \\
X\mathbf{1}_n &= p \\
X'\mathbf{1}_m &= q \\
X &\geqslant 0
\end{aligned}
$$

Действительно, матрица $C'X$ имеет размер $n \times n$, причем на главной диагонали находятся члены вида

$$\sum_{i=1}^{m} c_{ij} x_{ij}$$

которые при взятии следа матрицы дают нам целевую функцию.

Этот способ не совсем нам подходит! Почему?

Достичь желаемого результата нам поможет операция `vec`. Данная операция превращает матрицу в вектор-столбец, размещая ее столбцы друг на друга:

$$\operatorname{vec}(X) = \operatorname{vec}\left([X_1 \mid X_2 \mid \cdots \mid X_n]\right) = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{pmatrix}$$

Очевидно, что целевая функция может быть записана как $\operatorname{vec}(C)' \operatorname{vec}(X)$.

Для преобразования ограничений нам понадобится **произведение Кронеккера** $\otimes$

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{pmatrix}$$

Данное произведение и операция `vec` связаны следующим соотношением

$$\operatorname{vec}(AXB) = (B' \otimes A) \operatorname{vec}(X)$$

Имеем

$$p = X\mathbf{1}_n = \operatorname{vec}(X\mathbf{1}_n) = \operatorname{vec}(I_m X \mathbf{1}_n) = (\mathbf{1}_n' \otimes I_m) \operatorname{vec}(X)$$

Для преобразования второго ограничения заметим, что $\mathrm{vec}(x') = \mathrm{vec}(x) = x$ для любого **вектора** x. Тогда

$$q = X'\mathbf{1}_m = I_n X'\mathbf{1}_m = (\mathbf{1}_m' X I_n)' = \mathrm{vec}[(\mathbf{1}_m' X I_n)'] = \mathrm{vec}(\mathbf{1}_m' X I_n) = (I_n \otimes \mathbf{1}_m') \, \mathrm{vec}(X)$$

В итоге получим, что ограничения модели преобразуются в следующее ограничение

$$\begin{pmatrix} \mathbf{1}_n' \otimes I_m \\ I_n \otimes \mathbf{1}_m' \end{pmatrix} \mathrm{vec}(X) = \begin{pmatrix} p \\ q \end{pmatrix}$$

Итого, задача примет вид

$$\begin{gathered} \min_X \ \mathrm{vec}(C)' \, \mathrm{vec}(X) \\ \begin{pmatrix} \mathbf{1}_n' \otimes I_m \\ I_n \otimes \mathbf{1}_m' \end{pmatrix} \mathrm{vec}(X) = \begin{pmatrix} p \\ q \end{pmatrix} \\ X \geqslant 0 \end{gathered}$$

Эта задача выражена через векторы, а «эту задачу мы решать умеем»!

1.2. Продолжение экспериментов

[1]:
```
# %%capture
# %reset -f
# %pip install -U POT
# %pip install -U networkx
```

[2]:
```
import numpy as np

from scipy.optimize import linprog
from quantecon.optimize.linprog_simplex import␣
↪linprog_simplex

from ortools.linear_solver import pywraplp

import sympy as sp

import ot  # POT
```

[3]:
```
m = 3
n = 5

p = np.array([50, 100, 150], dtype=np.float64)
q = np.array([25, 115, 60, 30, 70], dtype=np.float64)
C = np.array(
    [
        [10, 15, 20, 20, 40],
        [20, 40, 15, 30, 30],
        [30, 35, 40, 55, 25],
    ],
    dtype=np.float64,
```

```
)
```

Подготовим все для функций `linprog` и `linprog_simplex`.

[4]:
```
A1 = np.kron(np.ones((1, n)), np.identity(m))
A2 = np.kron(np.identity(n), np.ones((1, m)))

A = np.vstack([A1, A2])
b = np.hstack([p, q])

C_vec = C.flatten("F")
```

[5]:
```
res_linprog = linprog(C_vec, A_eq=A, b_eq=b)
res_linprog.fun, res_linprog.x.reshape((m, n), order="F")
```

[5]:
```
(7225.0,
 array([[ 0., 50.,  0.,  0.,  0.],
        [10.,  0., 60., 30.,  0.],
        [15., 65.,  0.,  0., 70.]]))
```

Матрица вырождена. Действительно, сумма первых трех строк равна сумме последних пяти. Это означает, что **любая** строка матрицы A может быть выражена через линейную комбинацию оставшихся.

[6]:
```
A, np.linalg.det(A.T @ A)
```

[6]:
```
(array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1.,␣
↪0., 0.],
        [0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0.,␣
↪1., 0.],
        [0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0.,␣
↪0., 1.],
        [1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,␣
↪0., 0.],
        [0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0.,␣
↪0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0.,␣
↪0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0.,␣
↪0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,␣
↪1., 1.]]),
 np.float64(0.0))
```

Другими словами, мы можем безболезненно избавиться от одной строки.

[7]:
```
%time res = linprog(C_vec, A_eq=A[:-1, :], b_eq=b[:-1])
res.fun, res.x.reshape((m, n), order="F")
```

```
CPU times: total: 0 ns
Wall time: 4 ms
```

```
[7]: (7225.0,
      array([[ 0., 50.,  0.,  0.,  0.],
             [10.,  0., 60., 30.,  0.],
             [15., 65.,  0.,  0., 70.]]))
```

Решим ту же задачу при помощи `linprog_simplex`.

```
[8]: %time res_simplex = linprog_simplex(-C_vec, A_eq=A, b_eq=b)
     res_simplex.fun, res_simplex.x.reshape((m, n), order="F")
```

```
CPU times: total: 1.44 s
Wall time: 1.49 s
```

```
[8]: (-7225.0,
      array([[15., 35.,  0.,  0.,  0.],
             [10.,  0., 60., 30.,  0.],
             [ 0., 80.,  0.,  0., 70.]]))
```

Обратите внимание на «магическое» слово `%time`, которое позволяет определить скорость исполнения выражения. Ранее мы уже использовали другое магическое слово `%reset`, сбрасывающее все окружение: модули, переменные…

Мы можем видеть, что версия из пакета `quantecon` в 20 раз быстрее!

Воспользуемся специализированным пакетом `POT`, содержащим методы для решения транспортных задач.

```
[9]: %time res_pot = ot.emd(p, q, C)
     res_pot, np.sum(C * res_pot)
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

```
[9]: (array([[15., 35.,  0.,  0.,  0.],
             [10.,  0., 60., 30.,  0.],
             [ 0., 80.,  0.,  0., 70.]]),
      np.float64(7225.0))
```

Данный метод принимает на вход **матрицу** C и векторы предложения p и спроса q.

1.2.1. Ortools

Время для надуманных сложностей: попробуем экзотическое программирование с `ortools`.

```
[10]: solver = pywraplp.Solver.CreateSolver("GLOP")
```

Создадим массив с иксами, массив размером, совпадающим с `C`.

```
[11]: xs = [
          [solver.NumVar(0.0, solver.infinity(), f"x_{i}{j}") for␣
       ↪j in range(n)]
          for i in range(m)
      ]
```

Зададим ограничения нашей задачи. Сначала мы создаем объект `Constraint` внутри нашего решателя, передавая ему нижнюю и верхнюю границы. Так как у нас равенство, то передадим одинаковые границы.

Затем во вложенном цикле добавляем в ограничения переменные с соответствующими коэффициентами.

[12]:
```
constraints_p = []

for i in range(m):
    constraints_p.append(solver.Constraint(p[i], p[i]))

    for j in range(n):
        constraints_p[i].SetCoefficient(xs[i][j], 1)
```

[13]:
```
constraints_q = []

for j in range(n):
    constraints_q.append(solver.Constraint(q[j], q[j]))

    for i in range(m):
        constraints_q[j].SetCoefficient(xs[i][j], 1)
```

Создадим цель для решателя. Во вложенных циклах добавляем в цель переменные с коэффициентами, взятыми из матрицы `C`. В конце объявляем цель задачей минимизации.

[14]:
```
objective = solver.Objective()
for i in range(m):
    for j in range(n):
        objective.SetCoefficient(xs[i][j], C[i][j])
objective.SetMinimization()
```

На прошлом семинаре мы решали задачу прямым вызовом `Maximize`, что «под капотом» делает то же самое. Обратите внимание, что в прошлый раз мы получали значение целевой функции при помощи

```
solver.Objective().Value()
```

Сейчас же мы явно создали цель и обращаемся к ней напрямую.

[15]:
```
status = solver.Solve()
if status == solver.OPTIMAL:
    print(f"Решение: {objective.Value()}")

    for i in range(m):
        for j in range(n):
            print(f"x_{i}{j} = {xs[i][j].solution_value():2.
↪1f}")
```

```
Решение: 7225.0
x_00 = 0.0
x_01 = 50.0
```

```
x_02 = 0.0
x_03 = 0.0
x_04 = 0.0
x_10 = 10.0
x_11 = 0.0
x_12 = 60.0
x_13 = 30.0
x_14 = 0.0
x_20 = 15.0
x_21 = 65.0
x_22 = 0.0
x_23 = 0.0
x_24 = 70.0
```

[16]:
```
res_ortools = np.array(
    [[xs[i][j].solution_value() for j in range(n)] for i in range(m)]
)
```

1.2.2. SymPy

Аналогично `ortools` создадим массив с символами, которые поймет `sympy`.

[17]:
```
xs = np.array([[sp.Symbol(f"x_{{{i},{j}}}") for j in range(n)] for i in range(m)])
```

Создадим списки с ограничениями. В цикле мы сначала присоединяем к списку 0, к которому во вложенном цикле добавляем иксы с соответствующими коэффициентами.

[18]:
```
restrictions_p = []

for i in range(m):
    restrictions_p.append(0)

    for j in range(n):
        restrictions_p[i] += xs[i][j]

    restrictions_p[i] = sp.Eq(restrictions_p[i], p[i])
```

[19]:
```
restrictions_q = []

for j in range(n):
    restrictions_q.append(0)

    for i in range(m):
        restrictions_q[j] += xs[i][j]

    restrictions_q[j] = sp.Eq(restrictions_q[j], q[j])
```

Так как функция `lpmin` не ограничивает значения переменных, мы должны сделать это вручную.

[20]:
```
restrictions_x = []

for i in range(m):
    for j in range(n):
        restrictions_x.append(xs[i][j] >= 0)
```

Зададим целевую функцию.

[21]:
```
Q = 0
for j in range(n):
    for i in range(m):
        Q += C[i][j] * xs[i][j]
```

[22]:
```
res = sp.solvers.lpmin(Q, restrictions_p + restrictions_q␣
 ↪+ restrictions_x)
res
```

[22]:
```
(7225.00000000000,
 {x_{0,0}: 0,
  x_{0,1}: 35.0000000000000,
  x_{0,2}: 0,
  x_{0,3}: 15.0000000000000,
  x_{0,4}: 0,
  x_{1,0}: 25.0000000000000,
  x_{1,1}: 0,
  x_{1,2}: 60.0000000000000,
  x_{1,3}: 15.0000000000000,
  x_{1,4}: 0,
  x_{2,0}: 0,
  x_{2,1}: 80.0000000000000,
  x_{2,2}: 0,
  x_{2,3}: 0,
  x_{2,4}: 70.0000000000000})
```

Для того, чтобы получить матрицу значений иксов воспользуемся методом `subs`.

[23]:
```
res_sympy = np.array(
    [[float(xs[i][j].subs(res[1])) for j in range(n)] for i␣
 ↪in range(m)]
)
```

1.2.3. Сравним решения

Мы уже убедились, что все методы дают одинаковое значение целевой функции. Сравним иксы.

[24]:
```
(
    res_linprog.x.reshape((m, n), order="F"),
```

```python
    res_simplex.x.reshape((m, n), order="F"),
    res_pot,
    res_ortools,
    res_sympy,
)
```

[24]:
```
(array([[ 0., 50.,  0.,  0.,  0.],
        [10.,  0., 60., 30.,  0.],
        [15., 65.,  0.,  0., 70.]]),
 array([[15., 35.,  0.,  0.,  0.],
        [10.,  0., 60., 30.,  0.],
        [ 0., 80.,  0.,  0., 70.]]),
 array([[15., 35.,  0.,  0.,  0.],
        [10.,  0., 60., 30.,  0.],
        [ 0., 80.,  0.,  0., 70.]]),
 array([[ 0., 50.,  0.,  0.,  0.],
        [10.,  0., 60., 30.,  0.],
        [15., 65.,  0.,  0., 70.]]),
 array([[ 0., 35.,  0., 15.,  0.],
        [25.,  0., 60., 15.,  0.],
        [ 0., 80.,  0.,  0., 70.]]))
```