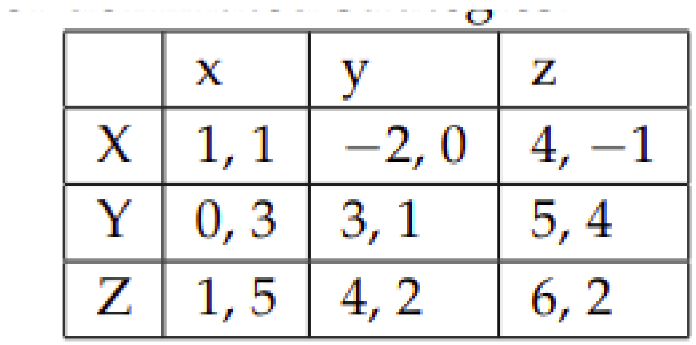
AppliedW12

May 23, 2024

[1]:
```
from IPython.display import Image, display
from matplotlib import pyplot as plt
from itertools import combinations
import numpy as np
import nashpy
```

1 Question 2

[2]:
```
display(Image(filename='game_dominance2.png'))
```



	x	y	z
X	1, 1	−2, 0	4, −1
Y	0, 3	3, 1	5, 4
Z	1, 5	4, 2	6, 2

We'll use the same pprocess as in question 1.

Start with the row player: - Strategy X is not dominated. When the column player plays column x, the row player can play row X - Strategy Y is dominated. When the column player plays column x, the row player can play row X. When the column player plays column y, the row player can play row Z. When the column player plays column z, the row player can play row Z - Strategy Z is also not dominated.

Now move on to the column player, noting that we will ignore row Y. - column x is not dominated. When the row player plays row X, the column player should play column x. - column y is dominated. When the row player plays row X, the column player should play column x. When the row player plays row Z, the column player should play column x. - column z is also dominated.

Now move back to the row player, noting that we will ignore row Y and columns y and z. -

The column player has one strategy: x - The row player has two strategies X and Z. - Neither row strategy is better than the other row the row player as both have the same payoff. Note that the players do not care about minimising the opposition's payoff, just maximising their own. - No strategy is dominated in this round for the row player and there are no more stragegies to eliminate for the column player.

We are stuck. We have reached a situation where iterative removal of dominated strategies fails to find a profile which is never dominated.

2 Question 3

[3]:
```
class bimatrix_game:
    def __init__(self, A, B):
        self.A = A
        self.B = B

    def pure_nash(self, verbose=False):
        row_br = []
        for col in range(self.B.shape[1]):
            for row in range(self.A.shape[0]):
                if self.A[row, col] == max(self.A[:, col]):
                    row_br += [[row, col]]
        row_br = np.array(row_br)
        if verbose: print(row_br)

        col_br = []
        for row in range(self.A.shape[0]):
            for col in range(self.B.shape[1]):
                if self.B[row, col] == max(self.B[row, :]):
                    col_br += [[row,col]]
        col_br = np.array(col_br)
        if verbose: print(col_br)

        nashes = []
        for br in row_br:
            single_strategy_is_best_response = br == col_br
            profile_pair_is_best_response = np.
↪all(single_strategy_is_best_response, axis=1)
            best_response_pair_exists = np.any(profile_pair_is_best_response)
            if best_response_pair_exists:
                nashes += [br]

        return nashes
```

[4]:
```
A2 = np.array([
    [5, 7, 2],
    [8, 6, 5],
    [1, 8, 4]
])

B2 = A2.T

game = bimatrix_game(A=A2, B=B2)

game.pure_nash(verbose=False)
```

[4]:
```
[array([2, 1]), array([1, 2])]
```

3 Question 4

[5]:
```
A3 = np.array([
    [0, 50, 40],
    [40, 0, 50],
    [50, 40, 0]
])

B3 = A3.T

A3, B3
```

[5]:
```
(array([[ 0, 50, 40],
        [40,  0, 50],
        [50, 40,  0]]),
 array([[ 0, 40, 50],
        [50,  0, 40],
        [40, 50,  0]]))
```

[6]:
```
game = nashpy.Game(A3, B3)
for nash in game.support_enumeration():
    print(nash)
```

```
(array([0.33333333, 0.33333333, 0.33333333]), array([0.33333333, 0.33333333,
0.33333333]))
```

First, let us assume that the column player B will play strategy 1 with probability α, strategy 2 with probability β and strategy 3 with probability γ and that any response that the row player can play will have idential payoff; v. Thus, the row player's payoffs can be described as:

$$0\alpha + 50\beta + 40\gamma = v \tag{1}$$

$$40\alpha + 0\beta + 50\gamma = v \tag{2}$$

$$50\alpha + 40\beta + 0\gamma = v \tag{3}$$

and we also know that the $\alpha + \beta + \gamma = 1$

re-arranging these four equations into a matrix equation gives;

[7]:
```
A_prime = np.array([
    [0, 50, 40, -1],
    [40, 0, 50, -1],
    [50, 40, 0, -1],
    [1, 1, 1, 0]
])
Right_Hand_Side_A = np.array([0, 0, 0, 1])

A_prime, Right_Hand_Side_A
```

[7]:
```
(array([[ 0, 50, 40, -1],
        [40,  0, 50, -1],
        [50, 40,  0, -1],
        [ 1,  1,  1,  0]]),
 array([0, 0, 0, 1]))
```

By solving this system of equations we can find the best payoff that the row player can get (v) in response to the column player's strategy (α, β, γ)

[8]:
```
sol = np.linalg.solve(A_prime, Right_Hand_Side_A)
stratA = sol[:-1]
payoffB = sol[-1]

stratA, payoffB
```

[8]:
```
(array([0.33333333, 0.33333333, 0.33333333]), 30.0)
```

We should now check if there is any pure strategy outside the support which has a better payoff that what we just got. However, since this is a full support strategy, there exists no such strategy to check.

We must now perform a similar check for the row player A. Since A=B.T, and the support is identical for both players, the we know that the results will be identical. i.e;

[9]:
```
B_prime = np.array([
    [0, 50, 40, -1],
    [40, 0, 50, -1],
    [50, 40, 0, -1],
    [1, 1, 1, 0]
])
Right_Hand_Side_B = np.array([0, 0, 0, 1])

B_prime, Right_Hand_Side_B
```

```
[9]: (array([[ 0, 50, 40, -1],
             [40,  0, 50, -1],
             [50, 40,  0, -1],
             [ 1,  1,  1,  0]]),
      array([0, 0, 0, 1]))
```

```
[10]: sol = np.linalg.solve(B_prime, Right_Hand_Side_B)
      stratB = sol[:-1]
      payoffA = sol[-1]

      stratB, payoffA
```

```
[10]: (array([0.33333333, 0.33333333, 0.33333333]), 30.0)
```

Once again, we should now check if there is any pure strategy outside the support which has a better payoff that what we just got. However, since this is a full support strategy, there exists no such strategy to check.

We also need to check if this is a valid solution. All of the values in the profile are valid, as they are all between 0 and 1, and they sum to exactly 1.

Thus, it is reasonable to say that there exists a nash equilibrium with full support with the mixture $([1/3, 1/3, 1/3], [1/3, 1/3, 1/3])$ and payoffs $(30, 30)$.

If we were to enumerate all supports of equal size, and repeat this procedure for each of them, we will have implemented the Support Enumeration algorithm.

4 Question 5

```
[11]: class SEM:
          def __init__(self, A, B, tol):
              self.A = A
              self.B = B
              self.tol = tol

          def all_profiles(self, verbose=False):
              An = self.A.shape[0]
              Bn = self.B.shape[1]
              n = min(An, Bn)
              profiles = []
              for k in range(2, n+1):
                  combsA = combinations(range(n), k)
                  for stratA in combsA:
                      combsB = combinations(range(n), k)
                      for stratB in combsB:
                          profile = (stratA, stratB)
                          if verbose: print(profile)
                          profiles += [profile]
```

```
        return profiles

    def augmented_matrix(self, M, profile, row = True, verbose=False):
        M_prime = M[profile[0], :]
        M_prime = M_prime[:, profile[1]]

        M_prime = M_prime if row else M_prime.T

        M_aug = np.hstack([M_prime, -np.ones(shape=((M_prime.shape[1]), 1))])
        M_aug = np.vstack([M_aug, np.ones(shape=((1, M_aug.shape[1])))])
        M_aug[-1, -1] = 0

        if verbose: print(M_aug)

        return M_aug

    def best_response(self, profile, verbose=False):

        A_aug = self.augmented_matrix(self.A, profile, verbose=verbose, row=True)
        B_aug = self.augmented_matrix(self.B, profile, verbose=verbose,␣
↪row=False)
        rhs = np.zeros(A_aug.shape[0])
        rhs[-1] = 1

        if (np.abs(np.linalg.det(A_aug)) <= self.tol) or (np.abs(np.linalg.
↪det(B_aug)) <= self.tol): # singular matrices can't be nash as no solution␣
↪exists
            if verbose: print('At least one system is singular')
            return False, None

        A_sol = np.linalg.solve(A_aug, rhs)
        B_sol = np.linalg.solve(B_aug, rhs)

        solution_profile = (A_sol[:-1], B_sol[:-1])
        payoffs = (A_sol[-1], B_sol[-1])

        if verbose: print(solution_profile, payoffs)

        return solution_profile, payoffs

    def is_nash(self, profile, solution, payoffs, tol=1e-16, verbose=False):
        nA = self.A.shape[0]
        nB = self.B.shape[1]

        if np.abs(np.sum(solution[0])-1) > tol or np.abs(np.sum(solution[1])-1)␣
↪> tol:
```

```
            if verbose: print('Sum of probabilities != 1') # not nash if not␣
↪probability density (sum=1)
            return False
        if (min(solution[0]) < 0) or (min(solution[1]) < 0): # not nash if not␣
↪probability density (non-negative)
            if verbose: print('Negative Probabilities')
            return False
        if (min(solution[0]) > 1) or (min(solution[1]) > 1): # not nash if not␣
↪probability density (max=1)
            if verbose: print('Probabilities >1 exist')
            return False

        x = np.zeros(nA) # row solution
        for i, profile_idx in enumerate(profile[0]):
            x[profile_idx] = solution[0][i]

        y = np.zeros(nB) # colum solution
        for i, profile_idx in enumerate(profile[1]):
            y[profile_idx] = solution[1][i]

        pure_payoffs_A = np.dot(self.A, y) # checking for pure strategies in A␣
↪outside support that beat x
        for i in range(len(pure_payoffs_A)):
            if (x[i] == 0) and (pure_payoffs_A[i] - payoffs[0] >= self.tol):
                if verbose: print("Pure strategy outside A's profile exists with␣
↪better payoff")
                return False

        pure_payoffs_B = np.dot(x, self.B)  # checking for pure strategies in B␣
↪outside support that beat y
        for i in range(len(pure_payoffs_B)):
            if (y[i] == 0) and (pure_payoffs_B[i] - payoffs[1]>= self.tol):
                if verbose: print("Pure strategy outside B's profile exists with␣
↪better payoff")
                return False

        if verbose: print("We have a nash!")
        return True # unless one of the matrices is singular, this is nash

    def solve(self, verbose=False):
        if verbose: print('Generating supports')
        profiles = self.all_profiles(verbose=verbose)
        nashes = []
        for profile in profiles:
            if verbose: print('Testing {0}'.format(profile))
            solution, payoffs = self.best_response(profile, verbose=verbose)
            if solution and self.is_nash(profile, solution, payoffs,␣
↪verbose=verbose):
                A_strat = np.zeros(self.A.shape[0])
                A_strat[list(profile[0])] = solution[0]

                B_strat = np.zeros(self.B.shape[1])
                B_strat[list(profile[1])] = solution[1]

                nashes += [(A_strat, B_strat, payoffs)]
        return nashes
```

```
[12]: game = SEM(A=A2, B=B2, tol=1e-10)
      game.solve(verbose=False)
```

```
[12]: [(array([0.25, 0.75, 0.  ]), array([0.25, 0.75, 0.  ]), (6.5, 6.5)),
       (array([0.75, 0.25, 0.  ]),
        array([0.        , 0.22222222, 0.77777778]),
        (5.75, 6.444444444444445)),
       (array([0.        , 0.22222222, 0.77777778]),
        array([0.75, 0.25, 0.  ]),
        (6.444444444444445, 5.75)),
       (array([0.        , 0.33333333, 0.66666667]),
        array([0.        , 0.33333333, 0.66666667]),
        (5.333333333333333, 5.333333333333333)),
       (array([0.20833333, 0.75      , 0.04166667]),
        array([0.20833333, 0.75      , 0.04166667]),
        (6.375, 6.375))]
```

```
[13]: game = SEM(A=A3, B=B3, tol=1e-10)
      game.solve(verbose=False)
```

```
[13]: [(array([0.33333333, 0.33333333, 0.33333333]),
        array([0.33333333, 0.33333333, 0.33333333]),
        (30.0, 30.0))]
```

```
[14]: import nashpy
```

```
[15]: game = nashpy.Game(A2, B2)
      for nash in game.support_enumeration():
          print(nash)
```

```
(array([0., 1., 0.]), array([0., 0., 1.]))
(array([0., 0., 1.]), array([0., 1., 0.]))
(array([0.25, 0.75, 0.  ]), array([0.25, 0.75, 0.  ]))
(array([1.58603289e-17, 3.33333333e-01, 6.66666667e-01]), array([1.58603289e-17,
3.33333333e-01, 6.66666667e-01]))
(array([0.20833333, 0.75      , 0.04166667]), array([0.20833333, 0.75      ,
0.04166667]))
```

```
[17]: game = nashpy.Game(A2, B2)
      for nash in game.vertex_enumeration():
          print(nash)
```

```
(array([0., 0., 1.]), array([0., 1., 0.]))
(array([0.        , 0.33333333, 0.66666667]), array([0.        , 0.33333333,
0.66666667]))
(array([0., 1., 0.]), array([0., 0., 1.]))
(array([ 2.50000000e-01,  7.50000000e-01, -4.51028104e-17]), array([
2.50000000e-01,  7.50000000e-01, -4.51028104e-17]))
(array([0.20833333, 0.75      , 0.04166667]), array([0.20833333, 0.75      ,
0.04166667]))
```

```
[ ]:
```