Programación Avanzada
Daniel Gregorio Longino
Tarea 16

1. Funciones

A continuación la primera función que calcula el frente de Paretto.

```
def is_pareto_efficient_dumb(costs):
    """
    Find the pareto-efficient points
    :param costs: An (n_points, n_costs) array
    :return: A (n_points, ) boolean array, indicating whether each point is Pareto efficient
    """
    is_efficient = np.ones(costs.shape[0], dtype = bool)
    for i, c in enumerate(costs):
        is_efficient[i] = np.all(np.any(costs[:i]<c, axis=1)) and np.all(np.any(costs[i+1:]<c, axis=1))
    return is_efficient
```

Dado un subconjunto S finito de R^n lo que hacemos primero es colocar estos puntos en una matriz `costs`, donde las filas de esta matriz son los vectores de S. Luego la función `is_pareto_efficient_dumb` trabaja de la siguiente manera: tomamos una fila `c` de `costs`, y comparamos si `c` $>$ `c'` para cada `c'`$\neq$ `c`, la comparacion se realiza entrada a entrada. Esta comparación se realiza en dos pasos, primero con todos las filas que están por arriba de `c` (`costs[:i]`$<$`c`) y luego con todas las filas que están por debajo de `c` (`costs[i+1:]`$<$`c`).

Si existe una fila `c'`$\neq$ `c` tal que `c'`$\geq$`c` , entonces al realizar la comparación (`c`$>$`c'`) obtendremos un arreglo con entradas booleanas de la forma

`[False,False,...,False]`,

en consecuencia

`np.any(costs[:i]<c, axis=1)` o
`np.any(costs[i+1:]<c, axis=1)`

tendrá en una de sus entradas `False`, y por lo tanto

`np.all(np.any(costs[:i]<c, axis=1))` o
`np.all(np.any(costs[i+1:]<c, axis=1))`

será `False`. Esto nos indicará que `c` no puede estár en la frente de Paretto pues existe una `fila c'`$\neq$ `c` con `c'`$\geq$ `c`.

En caso contrario, es decir, si nunca se obtiene una lista de la forma

`[False,False,...,False]`,

eso indica que

`np.all(np.any(costs[:i]<c, axis=1))` y
`np.all(np.any(costs[i+1:]<c, axis=1))`

son ambos iguales a `True` y por lo tanto `c` está en el frente de Paretto.

```
def is_pareto_efficient_simple(costs):
    """
    Find the pareto-efficient points
    :param costs: An (n_points, n_costs) array
    :return: A (n_points, ) boolean array, indicating whether each point is Pareto efficient
    """
    is_efficient = np.ones(costs.shape[0], dtype = bool)
    for i, c in enumerate(costs):

        if is_efficient[i]:
            is_efficient[is_efficient] = np.any(costs[is_efficient]>c, axis=1)  # Keep any po
            is_efficient[i] = True  # And keep self
    return is_efficient
```

Básicamente, lo que hace la función anterior es que para cada iteración se fija en un elemento `c` y realiza la comparación `c'>c` para cada vector `c'` (`costs[is_efficient]>c`), nuevamente la comparación se realiza entrada a entrada. Claro, si encuentra en un vector `c'`$\neq$ `c` tal que `c'`$\leq$ `c`, entonces al realizar la comparacion se obtendrá un arreglo de la forma `[False,False,...,False]` y por lo tanto

`np.any(costs[is_efficient]>c, axis=1)`

tendrá un `False` en una de sus entradas. Lo anterior implica que `c'` no puede estar en el frente de Paretto (porque `c'`$\leq$ `c`). Entonces la correspondiente entrada para `c'` en `is_efficient` se vuelve False, y este elemento ya no se considera para la iteración siguiente.

Por otro lado, si al finalizar todas las iteraciones la entrada correspondiente a `c` en `is_efficient` es `True`, eso implica que para cada vector `c'` distinto de `c`, el arreglo que se obtiene al realizar la comparación `c>c'`, tiene al menos un `True` en una de sus entradas, y por lo tanto no puede pasar que exista un véctor `c''` tal que `c''`$\geq$ `c`. Entonces, efectivamente, `is_efficient` es un arreglo booleano que nos dice si el vector correspondiente a cada una de sus entradas está en el frente de Paretto o no.

```python
def is_pareto_efficient(costs, return_mask = True):
    """
    Find the pareto-efficient points
    :param costs: An (n_points, n_costs) array
    :param return_mask: True to return a mask
    :return: An array of indices of pareto-efficient points.
        If return_mask is True, this will be an (n_points, ) boolean array
        Otherwise it will be a (n_efficient_points, ) integer array of indices.
    """
    is_efficient = np.arange(costs.shape[0])
    n_points = costs.shape[0]
    next_point_index = 0  # Next index in the is_efficient array to search for
    while next_point_index<len(costs):
        nondominated_point_mask = np.any(costs>costs[next_point_index], axis=1)
        nondominated_point_mask[next_point_index] = True
        is_efficient = is_efficient[nondominated_point_mask]  # Remove dominated points
        costs = costs[nondominated_point_mask]
        next_point_index = np.sum(nondominated_point_mask[:next_point_index])+1

    if return_mask:
        is_efficient_mask = np.zeros(n_points, dtype = bool)
        is_efficient_mask[is_efficient] = True
        return is_efficient_mask
    else:
        return is_efficient
```

La función anterior trabaja de la siguiente manera. Crea un arreglo con los número enteros del 1 a $n-1$, donde n `=len(costs)` (`np.arange(costs.shape[0])`). Luego empieza a iterar de la siguiente forma: el contador inicia en 0 (`next_point_index = 0`). Se toma el vector `c` correspondiente al indice `next_point_index` (`costs[next_point_index]`) y se compara con todos los otros vectores. La instrucción `costs>costs[next_point_index]` nos regresa un arreglo X de las mismas dimensiones que `costs` pero con entradas boolenas, donde X_{ij} es igual a `True` si la `j`-ésima entrada del `i`-ésimo elemento de `costs` es mayor que la `j`-ésima entrada de `c`, en caso contrario se coloca `False`. Observe que si existe un vector `c'`$\neq$ `c` tal que `c'`$\leq$ `c`, entonces al realizar la comparación `c'`$>$ `c`, obtendremos un arreglo con entradas unicamente iguales a `False`, en consecuencia la instrucción `np.any(costs>costs[next_point_index], axis=1)` nos dará un arreglo en donde una de sus entradas es `False`. Note que en este caso `c'` no puede estar en el frente de Paretto porque `c'`$\leq$`c` (entrada a entrada). Debido a que la instrucción `c>c` siempre regresa un arreglo donde todas sus entradas son iguales a `False`, es importante asignar `True` a la correspondiente entrada de `c` en el arreglo `nondominated_point_mask` y esto se logra mediante la instrucción

`nondominated_point_mask[next_point_index] = True`.

Los índices correspondientes a estos elementos `c'` en `is_efficient` deben ser removidos y esto se hace mediante la instrucción

`is_efficient = is_efficient[nondominated_point_mask]` .

Como hemos eliminado estos vectores dominados, entonces para la siguiente iteración hay que considerar el siguiente vector no dominado más inmediato a `c`. Este vector tendrá asociado el índice

```
np.sum(nondominated_point_mask[:next_point_index])+1
```

en el nuevo arreglo `is_efficient`.

Finalmente hay que ir guardando los elementos que no están en la cara de Paretto, simplemente asociando False a los elementos que son dominados por alguien más. Esto mediante las intrucciones

```
is_efficient_mask = np.zeros(n_points, dtype = bool)
        is_efficient_mask[is_efficient] = True
```

2. Tiempo de ejecución

A continuación el tiempo promedio de ejecución de cada algoritmo para encontrar el frente de Paretto para los datos del archivo `statistics.csv`.

Cuando se usó la función `pareto_mine` tomó un tiempo de 109.96121177299938 para encontrar el frente de Paretto.

```
58  vectors = []
59  vectorsp = []
60  elements = []
61  df = pd.read_csv("statistics.csv")
62  for i in range(5050):
63       vectors.append(list(df.loc[i])[2:])
64       vectorsp.append(np.array(list(df.loc[i])[2:]))
65       elements.append(list(df.loc[i]))
66
67
68  print("paretto_mine: "
69      f"{(timeit.timeit('paretto_mine(vectorsp)', number=1, globals=globals()))}")
70
```

```
paretto_mine: 109.96121177299938
```

Cuando se usó la función `is_pareto_efficient_dumb` y se ejecutó 5 veces tomó un tiempo promedio de 1.6277886680007214 para encontrar el frente de Paretto.

```
19  vectors = []
20  elements = []
21  df = pd.read_csv("statistics.csv")
22  for i in range(5050):
23       vectors.append(list(df.loc[i])[2:])
24
25       elements.append(list(df.loc[i]))
26
27  costs = np.matrix(vectors)
28
29  print("is_pareto_efficient_dumb: "
30      f"{(timeit.timeit('is_pareto_efficient_dumb(costs)', number=5, globals=globals()))/5}")
31
```

```
is_pareto_efficient_dumb: 1.6277886680007214
```

Cuando se usó la función `is_pareto_efficient_simple` y se ejecutó 5 veces tomó un tiempo promedio de 1.0391682974004652 para encontrar el frente de Paretto.

```
34  vectors = []
35  elements = []
36  df = pd.read_csv("statistics.csv")
37  for i in range(5050):
38      vectors.append(list(df.loc[i])[2:])
39
40      elements.append(list(df.loc[i]))
41
42  costs = np.array(vectors)
43
44  print("is_pareto_efficient: "
45      f"{(timeit.timeit('is_pareto_efficient(costs)', number=5, globals=globals()))/5}")
46
```

```
is_pareto_efficient_simple: 1.0391682974004652
[Finished in 8.0s]
```

Cuando se usó la función `is_pareto_efficient` y se ejecutó 5 veces tomó un tiempo promedio de 1.0745536136004374 para encontrar el frente de Paretto.

```
34  vectors = []
35  elements = []
36  df = pd.read_csv("statistics.csv")
37  for i in range(5050):
38      vectors.append(list(df.loc[i])[2:])
39
40      elements.append(list(df.loc[i]))
41
42  costs = np.array(vectors)
43
44  print("is_pareto_efficient: "
45      f"{(timeit.timeit('is_pareto_efficient(costs)', number=5, globals=globals()))/5}")
46
47
```

```
is_pareto_efficient: 1.0571832538000308
[Finished in 8.0s]
```