

## Tarea 16

Materia: Programación Avanzada.

Nombre: Germán Jordi Arreortúa Reyes

Fecha: martes, 5 de julio de 2022

Indicaciones:

Dados dos puntos  $x$  y  $y$  en  $R^n$ ,  $x$  domina a  $y$  si y solo si  $x$  es mayor o igual que  $y$  coordenada a coordenada. Construya un programa en Python que encuentre el frente de Pareto de un conjunto finito  $A$  de puntos en  $R^n$ . Es decir, el máximo subconjunto de puntos en  $A$  que no son dominados por algún elemento de  $A$ .

1. Pruebe su programa con los datos en el archivo `statistics.csv`. La respuesta está en archivo `pareto_frontier.csv`.
2. Utilice la biblioteca `timeit` (<https://docs.python.org/es/3/library/timeit.html>) para determinar el tiempo de ejecución de su programa.
3. Repita el punto anterior utilizando los métodos `is_pareto_efficient_dumb`, `is_pareto_efficient_simple` e `is_pareto_efficient` que aparecen en <https://stackoverflow.com/questions/32791911/fast-calculation-of-pareto-front-in-python>.
4. Reporte la comparación de tiempos en los cuatro métodos (el suyo y los otros tres) y explique con el mayor detalle posible por qué funciona cada uno de estos métodos.

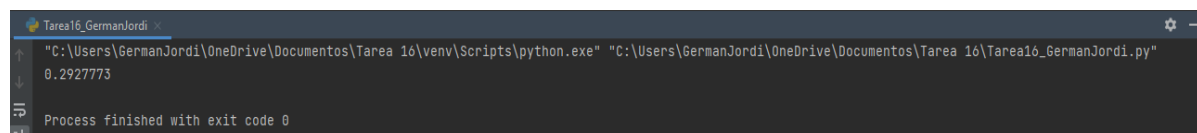
4.

### Tiempo de ejecución de los programas.

- Programa creado, `frente_de_pareto()`

```
print(timeit.timeit('frente_de_pareto(costos)', number=100,
globals=globals()))
```

Se obtuvo un tiempo de 0.2927773

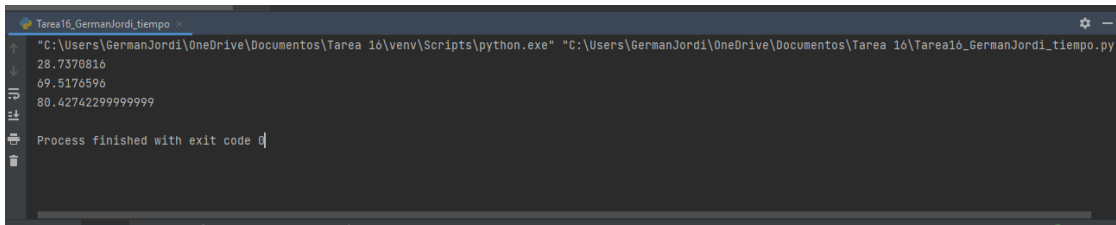


- Para las otras funciones

```
if __name__ == '__main__':
    df = pd.read_csv('statistics.csv')
    costos1 = -np.array(df[['APR', 'SHARPE', 'PRICE']])
    print(timeit.timeit('is_pareto_efficient_dumb(costos1)', number=100,
globals=globals()))
    print(timeit.timeit('is_pareto_efficient_simple(costos1)',
number=100, globals=globals()))
    print(timeit.timeit('is_pareto_efficient(costos1)', number=100,
globals=globals()))
```

Se obtuvieron los siguientes tiempos respectivamente.

- 28.7370816
- 69.5176596
- 80.42742299999999



```
Tarea16_GermanJordi_tiempo
"C:\Users\GermanJordi\OneDrive\Documentos\Tarea 16\venv\Scripts\python.exe" "C:\Users\GermanJordi\OneDrive\Documentos\Tarea 16\Tarea16_GermanJordi_tiempo.py"
28.7370816
69.5176596
80.42742299999999
Process finished with exit code 0
```

A continuación se explica con detalle las funciones, incluido la que realicé.

- **Función que realicé frente\_de\_pareto()**

```
def frente_de_pareto(datos):  
    """  
    Encuentra el frente de Pareto de una lista de  
    puntos en  $R^n$   
    :param datos: arreglo  
    :return: Un arreglo con los elementos  
    del frente de Pareto  
    """  
    indices_a_comparar = []  
    for ind in range(len(datos[0])):  
        if not isinstance(datos[0][ind], str):  
            indices_a_comparar.append(ind)  
  
    i = 0  
    while i < len(datos):  
        j = i + 1  
        while j < len(datos):  
            if all(datos[i][k] <= datos[j][k] for k in  
indices_a_comparar):  
                datos.pop(i)  
                j = i + 1  
            else:  
                if all(datos[j][k] <= datos[i][k] for k in  
indices_a_comparar):  
                    datos.pop(j)  
                else:  
                    j += 1  
            i += 1  
    return datos
```

Esta función recibe una lista con los elementos que se quiere encontrar el frente de Pareto, esta función maximiza es decir encuentra los elementos que no son dominados por ningún otro elemento y los devuelve en una lista.

Así pues la función recibe una lista (datos), luego indices\_a\_comparar son los índices de datos[0] tal que los elementos de datos[0] correspondiente a dichos índices son no son cadenas, suponiendo que los elementos de datos[0] son cadenas o números. Esto se hace ya que en el caso de encontrar el frente de Pareto de archivo 'statistics.csv' encontramos que las primeras dos columnas son de cadenas y no queremos incluirlas en las comparaciones.

El primer while lo que hace es que va ir recorriendo datos y el segundo while va ir recorriendo los elementos de datos que estén después del datos[i] y lo va a comparar con datos[i] mientras que j sea menor que la longitud de datos.

Antes de comenzar el while,  $i = 0$  luego entramos al while y comprueba si  $i < \text{len}(\text{datos})$ , luego  $j = i + 1$  y comprueba si  $j$  es menor que  $\text{len}(\text{datos})$ . Luego

```
if all(datos[i][k] <= datos[j][k] for k in indices_a_comparar):
```

Comprueba si cada entrada con índices en índices\_a\_comparar de datos[j] es mayor o igual que las respectivas de datos[i], es decir comprueba si datos[j] domina a datos[i], si ese es el caso entonces se elimina datos[i] pues no está en el frente de Pareto, y luego  $j = i + 1$ , esto se hace debido a que por ejemplo si tenemos lo siguiente

$$\begin{array}{ccccccc} [[...], [...], [...], \dots, [...], \dots, [...]] \\ i & i+1 & & & j & & \end{array}$$

Luego al eliminar datos[i] se tiene que datos[i] es ahora lo que había en datos[i+1], por lo cual se tiene que comparar datos[i] con datos[i+1]. Ahora si datos[j] no domina a datos[i] entonces se comprueba si datos[i] domina a datos[j], si ese es el caso entonces se elimina datos[j] y en este caso no es necesario redefinir j, porque j no cambia solo se eliminó el contenido en datos[j] y el elemento que ocupa la posición j se tiene que comparar con datos[i]. Ahora si ninguno domino al otro entonces quiere decir que puede estar en el frente de Pareto y por lo tanto aumentamos en 1 la j, y cuando j se mayor igual con  $\text{len}(\text{datos})$  entonces quiere decir que ya comparamos a datos[j] con todos los elementos posteriores a datos[i] y por lo cual datos[i] está en el frente de Pareto pues si k es menor que i entonces datos[k] no domina a datos[i] ni viceversa pues si no, se hubiera eliminado alguno cuando se hacía el while para k, y si k es mayor que i entonces ninguno domina al otro. Posteriormente aumentamos la i en 1.

Así pues al finalizar datos será una lista con los elementos que no son dominados por ningún otro es decir los elementos que están en el frente de Pareto.

- **def is\_pareto\_efficient\_dumb()**

```
# Very slow for many datapoints. Fastest for many costs, most readable
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
```

Esta y las próximas funciones minimizan por lo cual al comprobar el tiempo de ejecución se pasaron los datos a negativos. La función recibe un numpy.ndarray (costs) y devuelve un numpy.ndarray

(is\_efficient) de booleanos indicando los elementos que son de Pareto, es decir, si un elemento en la posición i de costs es de Pareto entonces en la posición i de is\_efficient será True.

Primeramente analizaremos cada parte de la función.

is\_efficient = np.ones(costs.shape[0], dtype = bool), se inicializa como un numpy.ndarray del mismo tamaño que costs de puros True's.

En el ciclo for iteramos sobre enumerate(cost). La función enumerate devuelve un iterador con la información del índice, y el elemento que corresponde a ese índice por ejemplo.

```
prueba = enumerate(np.array([[1, 2], [2, 5], [3, 6]]))
for i, c in prueba:
    print(i, c)
```

Obtenemos

```
0 [1 2]
1 [2 5]
2 [3 6]
```

costs[:i] > c, compara entrada por entrada los elementos de costs de índices menores que i con las de c y devuelve un arreglo donde cada entrada de los elementos del arreglo tiene un valor booleano si la entrada es mayor que la respectiva entrada de c. Veamos un ejemplo de cómo funciona esto.

```
valores = np.array([[1, 2], [7, 3], [8, 9], [3, 6]])
print(valores[:3] > [3, 6])
```

Obtenemos:

```
[[False False]
 [ True False]
 [ True  True]]
```

Puesto que cada entrada de [1, 2] es menor que la respectiva entrada de [3, 6] entonces tenemos [False False], en cambio puesto  $7 > 3$  y  $3 < 6$  entonces devuelve un [True False].

np.any(costs[:i] > c, axis=1) devuelve una lista con los valores booleanos obtenidos a partir de cost[:i] > c. Tenemos lo siguiente al aplicar np.any al ejemplo anterior.

```
[False True True]
```

Así pues si una entrada de costs[:i] > c tiene al menos un True entonces aparecerá un True en np.any(costs[:i] > c, axis=1), en la misma posición que la de costs[:i] > c.

np.all(np.any(costs[:i]>c, axis=1)), evalúa un arreglo de y devuelve True si todos sus valores son True. Por ejemplo al evaluar el ejemplo anterior obtendremos False.

Así pues is\_efficient[i] será False si existe un elemento de costs de índice menor o mayor que i tal que sus entradas sean menores o iguales que las correspondientes entradas de c, es decir si existe un elemento en cost que sea dominado por c, por lo cual c no está en el frente de Pareto, en caso contrario será True es decir cuando c no domina a ningún elemento y es minimal en el sentido de Pareto.

Notemos que si cost tiene dos elementos iguales entonces al hacer las comparaciones tenderemos en costs[:i] > c o en costs[i+1:]>c aparecerá un elemento con puros False, siendo c algún elemento que se repite en cost y por lo cual is\_efficient[i] será False, lo cual nos genera un problema ya que cost[i] puede ser un elemento que este en el frente de Pareto.

Finalmente `is_efficient` será una arreglo de elementos booleanos donde `True` indica que es esa posición el elemento de `cost` es de Pareto.

- **`def is_pareto_efficient_simple()`**

```
# Fairly fast for many datapoints, less fast for many costs, somewhat
readable
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)
            is_efficient[i] = True # And keep self
    return is_efficient
```

Al igual que la función anterior `is_efficient` se inicializa como un `numpy.ndarray` de puros `True`'s del mismo tamaño que `cost`.

Si `A` y `B` son `numpy.ndarray` del mismo tamaño, con `B` solo de elementos booleanos entonces podemos operar los valores de `A` para los cuales en la misma posición en `B` se tiene `True`. Por ejemplo

```
>>> import numpy as np
>>> booleanos = [True, False, False, True, True]
>>> valores = np.array([1, 2, 3, 4, 5])
>>> booleanos = [True, False, False, True, True]
>>> valores[booleanos]
array([1, 4, 5])
>>> valores1
array([1, 2, 3, 4, 5])
>>> valores[booleanos] <= 4
array([ True,  True, False])
```

Así pues en

```
for i, c in enumerate(costs):
    if is_efficient[i]:
        is_efficient[is_efficient] = np.any(costs[is_efficient] < c, axis=1)
        is_efficient[i] = True # And keep self
```

Tenemos lo siguiente, el ciclo `for` corremos `i, c` sobre `enumerate(costs)`, luego si `is_efficient[i]` es `True` entonces para las entradas de `costs` que tengan el valor de `True` en la respectiva entrada de `is_efficient`, se comprueba si alguna de sus entradas es menor que la respectiva entrada de `c`, si se cumple esto entonces se le asigna el valor de `True` en caso contrario será `False`. Puesto que en la

comparación también está incluido c entonces tenemos un valor de False por lo cual es la penúltima línea tenemos `is_efficient[i] = True`.

Así pues en resumen para cada elemento de costs que tenga el valor de True en `is_efficient`, se comprueba si al menos una de sus entradas es menor que la respectiva entrada de c, es decir si dicho elemento no domina a c, pues es caso contrario como como estamos minimizando quiere decir que dicho elemento no puede estar en el frente de Pareto y por lo cual se le asignara el valor de False. Así pues en la siguiente iteración no se verifica si este elemento puede estar en el frente de Pareto pues domina a otro elemento de costs. Así pues finalizar el proceso solo tendrán True aquellos que estén en el frente de Pareto

- **def is\_pareto\_efficient()**

```
# Faster than is_pareto_efficient_simple, but less readable.
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
        # Remove dominated points
        is_efficient = is_efficient[nondominated_point_mask]
        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
```

Tenemos que `is_efficient` se inicializa como un `numpy.ndarray` con los índices de costs es decir es un `numpy.ndarray` con los valores de 0 hasta la longitud de costs menos 1, y lo que hace el programa es eliminar los índices de los elementos de costs que dominan a algún otro elemento de costs y al finalizar `is_efficient` tendrá los índices de los elementos de costs que están en el frente de Pareto. Se tiene que `n_points` es el número de elementos de cost y `next_point_index = 0`.

En el ciclo `while` tenemos que `nondominated_point_mask = np.any( costs < costs [next_point_index] , axis=1)` es un `numpy.ndarray` de valores booleanos, `nondominated_point_mask[k]` es False

si cada entrada de `cost[k]` es mayor o igual que `costs[next_point_index]`, es decir si `cost[k]` domina a `costs[next_point_index]`, caso contrario será `True`. Así pues `nondominated_point_mask` indica que valores de `costs` no son dominados por `costs[next_point_index]`. Puesto que en `np.any(costs < costs[next_point_index], axis=1)` se incluye la comparación de `costs[next_point_index]` consigo misma la cual dará `False` entonces es necesario cambiarla por `True`.

Luego con `is_efficient = is_efficient[nondominated_point_mask]` y `costs = costs[nondominated_point_mask]` se eliminan los índices y los elementos que dominan a `costs[next_point_index]` en `is_efficient` como en `costs` respectivamente. Posteriormente `next_point_index = np.sum(nondominated_point_mask[:next_point_index]+1)`, es la suma de los elementos que son `True` en `nondominated_point_mask` con índices menores que `next_point_index` y a esto se le suma 1, esto se hace porque `costs` va cambiando de tamaño pues se eliminaron los elementos que dominan a `costs[next_point_index]`. Así pues para pasarnos el siguiente de `costs` tenemos que contar cuantos elementos de `nondominated_point_mask` con índices menores que `next_point_index` fueron `True`, pues estos son lo que no se eliminaron, así pues el índice de `costs[next_point_index]` después de la eliminación es `np.sum(nondominated_point_mask[:next_point_index])` y al sumarle 1 nos pasamos al siguiente elemento. Así pues al finalizar el ciclo `while` tendremos que `costs` tiene los elementos que no son dominados por ningún otro elemento, `is_efficient` tendrá los índices del `costs` original de los elementos que están en el frente de Pareto. Posteriormente se hace la comparación `if return_mask`. Si `return_mask` es `True` entonces se crea un `np.ndarray` de `n_points` de puros `False`'s y con `is_efficient_mask[is_efficient] = True`, se asigna el valor de `True` a los índices de `is_efficient_mask` que aparecen en `is_efficient` y finalmente la función devolverá `is_efficient_mask`. Si `return_mask = False` entonces la función devuelve `is_efficient`.