Tarea 16
Materia: Programación Avanzada.
Nombre: Germán Jordi Arreortúa Reyes
Fecha: martes, 5 de julio de 2022
Indicaciones:
Dados dos puntos x e 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.

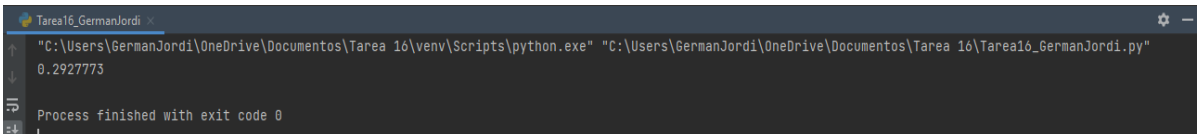
4.

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

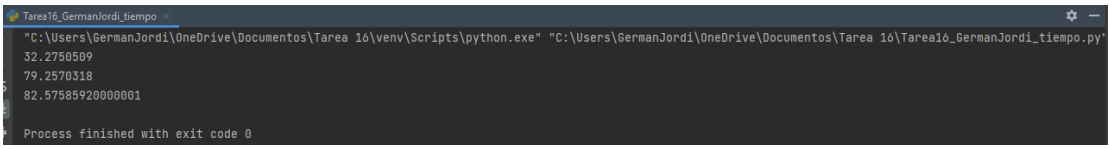


- Paras las otras funciones

```
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.

- 32.2750509
- 79.2570318
- 82.57585920000001



A continuación se explica con detalle las funciones, incluido la 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: lista con los elementos de
    los cuales se quiere obtener el frende de pareto
    :return: Una lista con los elementos
    del frente de pareto
    """
    indices_a_comparar = []
    for r in range(len(datos[0])):
        if isinstance(datos[0][r], (int, float)):
            indices_a_comparar.append(r)

    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 máxima 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 es una lista con índices de datos[0] que son números reales y son los que se van a utilizar en las comparaciones para encontrar el frente de Pareto. Esto se hace ya que por ejemplo en el caso de encontrar el frente de Pareto de archivo ‘statistics.csv’ encontramos columnas que son cadenas y no queremos incluir 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] hasta que sean menor que la longitud de datos.
Entonces antes de comenzar el while, i = 0 luego entramos al while y comprueba si i < len(datos), luego j= i+1 y comprueba si j es menor que len(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

[[…], […], […], … , […], … […]
i i +1 j

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 nuevo 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 len(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
```

La función recibe un numpy.ndarray (costs) y devuelve un numpy.ndarray (is_efficinet) de booleanos indicando los elementos que no dominan a ningún otro elemento, es decir, si un

elemento en la posición i de costs no domina a ningún otro entonces en la posición i de is_efficiente será True.

Primeramente analizaremos hace 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 entradas de c, es decir si existe un elemento en cost que sea dominado por c, en caso contrario será True.
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 no es dominado por ningún otro elemento es decir los elementos no dominados.

- **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
```

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_effcient se comprobara si al menos una entrada alguna de sus entradas es menor que la respectiva entrada de c, si se cumple esto entonces se le asignara 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 es que para cada elemento de costs que tenga el valor de True en is_efficient, se comprueba si es dominado por c, si este es el caso entonces se le asignara el valor False y como estamos minimizando quiere decir que c no está en el frente de Pareto. Notemos que la condición if is_efficient[i], se pone porque si is_efficient[i] es False entonces quiere decir que el elemento c de costs asociado a i domino a un elemento anterior a él y por lo cual c no está en el frente de Pareto, y si existe un elemento el cual domina a c entonces también domina a los elementos que domina c. Puesto estamos corriendo sobre todos los elementos de costs entonces al finalizar is_efficient tendrá True en las posiciones que el respectivo elemento de costs que no son dominados.

- **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 no son dominados es decir los 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.narray de valores booleanos, nondominated_ point _mask[k] es False si cada entrada de cost[k] es mayor o igual que costs[next_ponit_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_ponit_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 dominan a costs[next_ponit_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_ponit_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_effcient y finalmente la función devolverá is_efficient_mask. Si return_mask = False entonces la función devuelve is_efficient.