OPTIMIZACIÓN: PRÁCTICA 1

Implementación del algoritmo Símplex primal

Alumnos: ***Victor Gesiarz y Noel Nathan Planell***

Tutor/a: ***Mari Paz Linares Herreros***

Index

Introducción

La programación lineal es una técnica matemática que se utiliza para resolver problemas de optimización en los que se busca maximizar o minimizar una función lineal sujeta a ciertas restricciones. Para resolver este tipo de problemas, el algoritmo más empleado es simplex, debido a su gran eficiencia y a su facilidad para implementarlo.

En esta práctica implementaremos el algoritmo del Símplex, sólo la parte primal, tanto la primera fase como la segunda. En nuestro caso hemos decidido implementarlo utilizando *Python* y para lograr una mayor eficiencia en los cálculos, hemos utilizado la librería de *Numpy*.

Implementación

Para implementar el algoritmo hemos decidido crear una clase de *Python*, la cual recibe como valores iniciales la matriz de restricciones A junto a su vector de inecuaciones y el vector de la función objetivo.

```
class Simplex():
   def __init__(self, A, b, c) -> None:
        self.A = np.array(A)
        self.b = np.array(b)
        self.c = np.array(c)

        self.n = len(self.c)
        self.m = len(self.A)

        self.iteration_info = {}
        self.output = ""
```

También guardamos las variables iteration_info y output que nos sirven para guardar información para imprimir el proceso de la ejecución del algoritmo y los resultados. Para estos también podemos encontrar en el código las funciones auxiliares: print, __repr__, __repr_matrix, __repr_list y _print_iter.

Fase 2

Empezaremos explicando la fase 2 primero, dado a que la primera fase hace uso de esta. La función para esta fase recibe como *inputs* una base inicial, tanto la B como la N, la solución inicial de las variables y la z inicial.

```
def fase_2(self, i_B, i_N, x_B, z, inv_B):

    cb = self.c[i_B]
    cn = self.c[i_N]
    An = self.A.take(i_N, axis=1)
```

A parte de esto, empezamos calculando variables que necesitaremos posteriormente para los cálculos como el vector de la función objetivo que forma parte de la base, la matriz B y su inversa, etc.

A continuación empezamos las iteraciones del algoritmo, en cada una de ellas calcularemos los **costes reducidos**, los cuales tendremos que comprobar si son todos positivos para encontrar el óptimo, posteriormente la **variable de entrada** *q*, el **vector de direcciones**, la **theta** y la **variable de salida** *p*, y realizaremos los cambios necesarios a nuestros datos. En nuestra implementación hemos conseguido aplicar el cambio de la inversa para no tener que recalcular esta en cada iteración.

```
while True:
    rn, stop = self.reduced_costs(cn, cb, inv_B, An)
    if stop:
        break

    _q, q = self.input_variable(i_N, rn)
    db = self.calculate_db(inv_B, q)

    if all(db >= 0):
        self.print('Unbounded Problem')
        break

    theta, _p, p = self.theta_and_p(i_B, x_B, db)

    self.swap(i_B, i_N, _q, _p)
    B, An, z, cb, cn, x_B = self.actualize_variables(i_N, i_B, x_B, z,
                            theta, db, rn, _q, _p)
    inv_B = self.actualize_inverse_better(inv_B, db, _p) if inverse \
            else self.actualize_inverse(B)

    return i_B, i_N, x_B, z
```

Costes reducidos

La función que calcula los costes reducidos simplemente implementa la fórmula de estos, la cual es:

$$r_n = c_n - c_b \cdot B^{-1} \cdot A_n$$

Además, la función devuelve `True` en caso de que todos sean positivos, por lo que la fase 2 parará y devolverá el resultado obtenido.

```
def reduced_costs(self, cn, cb, inv_B, An):
    rn = cn - cb.dot(inv_B).dot(An)
    if all(rn >= 0):
        return rn, True
    return rn, False
```

Variable de entrada

La variable de entrada *q* es la que usaremos para cambiar de base, esta será la que entre en ella y tendremos que calcular una que salga de ella.

Se calcula cogiendo el vector de costes reducidos y buscando el primer valor negativo que haya. En el código podemos ver que tenemos una *_q* y una *q*, la diferencia entre las dos es que la *q* representa la variable de entrada real, es decir, de todas las variables que tenemos entre 1 y n, cual de estas estamos cogiendo, y la *_q* representa el índice de la variable en el vector de variables no básicas.

```
def input_variable(self, i_N, rn):
    for _q, x in enumerate(rn):
        if x < 0:
            q = i_N[_q]
            return _q, q
```

Vector de direcciones

El vector de direcciones nos sirve para más adelante poder calcular la *theta* y conseguir la variable de salida *p*. Esto consiste simplemente en implementar la siguiente fórmula:

$$d_b = \ -B^{-1} \cdot A_q$$

```
def calculate_db(self, inv_B, q):
    db = - inv_B.dot(self.A.take(q, axis=1))
    return db
```

Posteriormente, al volver a la función fase_2 comprobaremos que exista algún valor negativo en este vector, porque de lo contrario significaría que el problema **no está acotado**.

Theta y variable de salida

En este paso calculamos la variable que saldrá de la base. La *theta* se calcula mediante:

$$\theta = Min_{d_b(i)<0}\left\{\frac{-x_b(i)}{d_b(i)}\right\}$$

Esta fórmula lo que nos dice es que, miramos todos los valores negativos del vector de direcciones y por cada uno calculamos la división, dónde $x_b(i)$ es la solución anterior de esa variable. La *p* será el índice que haya obtenido la menor *theta*.

```
def theta_and_p(self, i_B, x_B, db):
    theta = np.inf
    p = np.inf
    i_p = np.inf
    for i, x in enumerate(db):
        if x < 0:
            new_theta = - x_B[i]/x
            new_p = i_B[i]
            if new_theta < theta:
                theta = new_theta
                p = i_B[i]
                i_p = i
            if new_theta == theta:
                if new_p < p:
                    p = new_p
```

```
                    i_p = i
    return theta, i_p, p
```

Aquí nos volvemos a encontrar con lo mismo que pasaba en la variable de entrada, tenemos *_p* que nos indica el índice de la variable en el vector de variables básicas y la *p* indica la variable real.

Actualizar nuestras variables

Una vez hechos todos estos cálculos para saber qué variables salen y entran de la base, tenemos que formalizar este cambio. Para esto tenemos 3 funciones que veremos a continuación.

Primero intercambiamos las variables entre el vector de las variables básicas y las del vector de variables no básicas. Para que el cambio sea lo más efectivo posible y poder aplicar posteriormente la actualización de la inversa, las variables básicas tienen que intercambiar posiciones con las variables no básicas y quedarse en ese orden en el vector de variables básicas, en cambio en el vector de las no básicas, las variables tienen que quedar ordenadas.

```
def swap(self, i_B, i_N, _q, _p):
    into = i_N[_q]
    outof = i_B[_p]
    i_B[_p] = into
    del i_N[_q]

    for i, x in enumerate(i_N):
        if x > outof:
            i_N.insert(i, outof)
            break
```

Cabe destacar que, como las listas de *Python* funcionan con punteros, no necesitamos devolver los valores actualizados.

Posteriormente, tenemos que actualizar el valor de x_b y *z*, y obtener la nueva matriz B, A_n, etc. Esto es bastante sencillo debido a que tenemos el valor de *theta* y podemos aplicar las siguientes fórmulas:

$$x_b = x_b + \theta \cdot d_b \mid x_b(p) = \theta \qquad\qquad z = z + \theta \cdot r_q$$

```
def actualize_variables(self, i_N, i_B, x_B, z,theta, db, rn, q, p):
       x_B = x_B + theta * db
       x_B[p] = theta
       B = self.A.take(i_B, axis=1)
       An = self.A.take(i_N, axis=1)
       cn = self.c[i_N]
       cb = self.c[i_B]
```

```
        z += theta * rn[q]
        return B, An, z, cb, cn, x_Bç
```

Y por último, nos queda actualizar la inversa de B. Para ello, se puede o bien volverla a calcular usando el método Gauss-Jordan o bien hay una fórmula alternativa para este caso concreto (ya que conocemos la matriz inversa anterior) que es más eficiente computacionalmente y que es la que hemos acabado utilizando.

La fórmula para actualizar la inversa es:

$$B^{-1}_{actual} = B^{-1}_{anterior} \cdot E \Rightarrow E = (e_1, e_2, \dots, e_{p-1}, \eta_p, e_{p+1}, \dots, e_m)$$

$$\eta_p = \left\{ \frac{-d_{iq}}{d_{pq}}, i \neq p \mid \frac{-1}{d_{pq}}, i = p \right\}$$

```
def actualize_inverse(self, inv_B, db, p):
        Np = - db / db[p]
        Np[p] = - 1 / db[p]
        E = np.eye(len(inv_B))
        E[:, p] = Np
        new_inv_b = E.dot(inv_B)
        return new_inv_b
```

Fase 1

La primera fase del algoritmo, la cual consiste en encontrar una *solución básica factible inicial* para poder empezar con la segunda fase.

El método consiste en añadir variables artificiales, en concreto la misma cantidad que restricciones, y en cambiar la función objetivo a: “minimizar la suma de las variables artificiales”. Posteriormente aplicamos los mismos pasos que en la segunda fase, por lo que podemos llamarla también, pero con los datos artificiales.

```
def fase_1(self, verbose=0):
    copy_A = self.A.copy()
    copy_c = self.c.copy()

    self.A = np.concatenate((self.A.copy(), np.identity(self.m)), axis=1)
    self.c = np.array([0 for i in range(self.n)] + [1 for i in range(self.m)])
    i_B = [self.n + i for i in range(self.m)]
    i_N = [i for i in range(self.n)]
    x_B = self.b
    z = sum(x_B)
    inv_B = np.identity(self.m)

    i_B, i_N, x_B, z, inv_B = self.fase_2(i_B, i_N, x_B, z, inv_B, verbose)
```

```
    self.A = copy_A
    self.c = copy_c
    z = self.c[i_B].dot(x_B)

    return i_B, i_N, x_B, z
```

Cabe recalcar que en esta fase 1, como la matriz B es la matriz identidad, su inversa es la propia matriz identidad. Por ende, en ningún momento de nuestro código debemos calcular la inversa de la matriz B, sino que en cada iteración la actualizaremos.

Solve

La función solve simplemente sirve para unir ambas fases y ejecutarlo todo a través de un comando.

```
def solve(self, verbose=0):
        i_B, i_N, x_B, z = self.fase_1(verbose)
        i_B, i_N, x_B, z = self.fase_2(i_B, i_N, x_B, z, verbose)
        return i_B, i_N, x_B, z
```

Hemos añadido también una variable llamada *verbose*, que nos sirve para controlar cuánta información mostramos al ejecutar el código: 0 significa que no se muestra nada, 1 significa que se muestra poca información, 2 muestra la ejecución bastante más detalladamente y 3 muestra la máxima información.

Un ejemplo de cómo se mostraría la salida con *verbose = 0* y las siguientes matrices:

- A = [[2, 1, 1, 0], [1, 1, 0, 1]]
- b = [3, 2]
- c = [-1, -2, 0, 0]

```
- - - - - - - - - - - - - - - F A S E: 1 - - - - - - - - - - - - - - - - - - - -

Iteration  1: q =  0, p =  4, theta = 1.5,  z = 0.5
Iteration  2: q =  1, p =  5, theta = 1.0,  z = 0.0

- - - - - - - - - - - - - - - F A S E: 2 - - - - - - - - - - - - - - - - - - - -

Iteration  1: q =  2, p =  0, theta = 1.0,  z = -4.0

Optim solution:
vb = 2  1
xb = 1.0  2.0
z = -4.0
r = 1.0  2.0
```

Resultados

Los conjuntos de datos con los que hemos trabajado, tal y como indican las instrucciones, han sido el 12 y el 34. A continuación se muestra un resumen de los resultados obtenidos, en los archivos adjuntos a este pdf, están los archivos completos de los resultados.

12.1 **Fase 1:** 12 iteraciones. **Fase 2:** 15 iteraciones **Óptimo:** **vb** = 11 4 17 2 12 15 19 16 18 5 **xb** = 3.68 1.56 44.28 1.28 2.77 249.89 311.7 244.47 43.18 2.51 **z** = -469.31249 **r** = 163.9 161.54 30.06 44.3 62.58 2.02 73.07 31.88 81.47 0.48	12.2 **Fase 1:** 18 iteraciones. **Fase 2:** 7 iteraciones **Óptimo:** **vb** = 3 2 8 5 13 4 7 15 0 9 **xb** = 1.75 1.16 0.26 0.35 3.61 1.19 0.33 664.24 1.23 3.61 **z** = -352.7137 **r** = 60.58 22.13 103.52 145.14 120.47 0.26 0.02 0.18 0.01 0.4
12.3 **Fase 1:** 10 iteraciones. **Problema no factible**	12.4 **Fase 1:** 16 iteraciones. **Fase 2:** 32 iteraciones **Problema no acotado**
34.1 **Fase 1:** 18 iteraciones. **Fase 2:** 14 iteraciones **Óptimo:** **vb** = 18 4 6 9 19 7 0 11 15 2 **xb** = 764.42 2.21 1.48 2.47 291.82 3.6 0.0 2.21 322.82 0.52 **z** = -787.4866 **r** = 31.36 109.99 192.25 176.13 5.52 114.31 186.01 0.37 0.01 0.53	34.2 **Fase 1:** 18 iteraciones. **Fase 2:** 14 iteraciones **Óptimo:** **vb** = 11 0 10 15 13 1 12 18 16 5 **xb** = 1.4 2.36 0.3 293.96 4.32 0.15 3.03 444.75 520.01 2.74 **z** = -250.1337 **r** = 90.27 16.02 32.73 189.23 91.76 48.41 57.56 0.24 0.51 0.41
34.3 **Fase 1:** 13 iteraciones. **Problema no factible**	34.4 **Fase 1:** 15 iteraciones. **Fase 2:** 22 iteraciones **Problema no acotado**