**OPTIMIZACIÓN EN PYTHON: INSTRUCTIVO BÁSICO**

Considere el siguiente problema de optimización:

Una compañía de *call centers* cuenta con dos servidores principales, localizados en San Francisco y Houston. Dichos servidores atienden las tres áreas geográficas en la que la compañía dividió el país: Costa Este, Centro y Costa Oeste. Los servidores reciben las llamadas de los clientes y las dirigen a los operadores telefónicos para que estos las atiendan. La red de banda ancha permite que cualquier llamada de cualquier área se conecte a cualquiera de los servidores. El problema que debe resolver la dirección de sistemas de la compañía tiene que ver con la asignación de los servidores a las llamadas de las áreas geográficas. En las tablas siguientes se puede apreciar la disponibilidad de procesamiento de cada servidor (en Terabytes), y las demandas de cada cliente (en Terabytes).

| Servidor | Capacidad de procesamiento [TB] |
|---|---|
| San Francisco | 350 |
| Houston | 600 |

| Área geográfica | Demanda [TB] |
|---|---|
| Costa Este | 325 |
| Centro | 300 |
| Costa Oeste | 275 |

Por cada Terabyte que se atienda en cada servidor de cada área geográfica se incurre en un costo (relacionado con la electricidad necesaria refrigerar los equipos de cómputo y mantener la infraestructura de red), que se muestra a continuación.

| Servidor / Región | Costa Este | Centro | Costa Oeste |
|---|---|---|---|
| San Francisco | 225 | 153 | 162 |
| Houston | 225 | 162 | 126 |

Se desea, con un modelo de optimización, minimizar el costo total de despacho de las llamadas, sujeto a las condiciones anteriormente descritas.

El anterior problema es un caso particular del problema general de encontrar la solución óptima de un problema de Programación Lineal.

Considere la siguiente notación para formular matemáticamente el problema:

*Conjuntos*

$I = Servidores = \{SF, H\}$

$J = Área\ geográfica = \{CE, C, CO\}$

*Parámetros*

$S_i = \begin{bmatrix} 350 \\ 600 \end{bmatrix}\ capacidad\ de\ procesamiento\ del\ servidor\ i\ [TB]$

$D_j = \begin{bmatrix} 325 \\ 300 \\ 275 \end{bmatrix}\ demanda\ del\ área\ geográfica\ j\ [TB]$

$C_{ij} = \begin{bmatrix} 225 & 153 & 162 \\ 225 & 162 & 126 \end{bmatrix}\ costo\ de\ enviar\ un\ TB\ al\ servidor\ i\ desde\ el\ área\ j\ [\frac{\$}{TB}]$

La formulación del problema de optimización utilizando la notación anterior es la siguiente:

*Variables de decisión*

$X_{ij}$: *número de terabytes que se atenderán en el servidor* $i$ *del área geográfica* $j$

*Función objetivo*

$$Min \sum_i \sum_j C_{ij} X_{ij} \quad minimizar\ el\ costo\ total\ de\ transmisión\ de\ datos\ de\ cada\ área\ a\ cada\ servidor$$

*Restricciones*

$$\sum_j X_{ij} \leq S_i \quad \forall i \quad cada\ servidor\ puede\ atender\ como\ máximo\ su\ capacidad$$

$$\sum_i X_{ij} \geq D_j \quad \forall j \quad la\ demanda\ de\ cada\ área\ geográfica\ debe\ ser\ atendida\ en\ su\ totalidad$$

$$X_{ij} \geq 0 \ \ \forall i \ \forall j \quad restricción\ de\ no\ negatividad$$

Para formular el anterior problema en Python, se utilizará el módulo ***pyomo***. La esencia de la optimización en pyomo es declarar un modelo como un objeto que contendrá todos los elementos algebraicos que lo componen (conjuntos, variables, vectores, matrices y ecuaciones).

Siga los pasos que se describen a continuación y que se ejemplifican con el problema inicial de este documento:

**PASO 0. Instalar el módulo pyomo**

Los objetos que se crearán y los algoritmos que se correrán para solucionar problemas de optimización no pertenecen a la librería estándar de Python. Por eso se hace necesario instalar el módulo pyomo. Para hacerlo, abra la ventana de comandos y escriba

```
python -m pip install pyomo
```

Realice este paso una sola vez. El módulo quedará instalado en su equipo y no es necesario que lo instale de nuevo para correr futuros programas

**PASO 1. Cargar el módulo de pyomo**

Una vez instalado el módulo y establecido en el ambiente en el que va a trabajar (Jupyter Notebook, por ejemplo), cargue el módulo pyomo.

```
from pyomo.environ import *
```

**PASO 2. Crear el modelo**

Como se mencionó anteriormente, al utilizar pyomo todos los elementos del problema de optimización estarán contenidos en un gran objeto (llamado `model`), que tendrá la asignación de un modelo concreto.

```
model = ConcreteModel()
```

**PASO 3. Declare los conjuntos del modelo**

Recuerde que los *sets* son la estructura de datos utilizada en Python para declarar conjuntos. Recuerde ingresar los strings encerrados en comillas. La sintaxis es la siguiente:

```
model.i = Set(initialize=['SF','H'], doc='Servidores')
model.j = Set(initialize=['CE','C','CO'], doc='Áreas geográficas')
```

El texto asignado a `doc` corresponde a la documentación del conjunto; es decir, a la explicación de lo que representan sus elementos.

**PASO 4. Declare los parámetros del modelo**

Los parámetros son los escalares, vectores y matrices que ayudarán a conformar las ecuaciones del modelo. Su declaración en Python es con un escalar o con un diccionario. En el problema del ejemplo, únicamente hay vectores y matrices, por lo que se definirán usando diccionarios.

```
#Vectores

model.S = Param(model.i, initialize={'SF':350,'H':600}, doc='Cap serv i')
model.D = Param(model.j, initialize={'CE':325,'C':300,'CO':275},
doc='Demanda del área j')


#Matrices

tablac = {
    ('SF', 'CE') : 225,
    ('SF', 'C')  : 153,
    ('SF', 'CO') : 162,
    ('H',  'CE') : 225,
    ('H',  'C')  : 162,
    ('H',  'CO') : 126,
    }

model.C = Param(model.i, model.j, initialize=tablac,
doc='Costo envío de un TB al servidor i desde el área j')
```

**PASO 5. Declare las variables de decisión**

La declaración de las variables de decisión puede incluir los límites superior e inferior. Al declarar el límite inferior como 0, la restricción de no negatividad queda cubierta. Si la variable no tiene límite superior, se escribe **`None`** como ese límite.

```
model.x = Var(model.i, model.j, bounds=(0.0,None),
doc='TB que se atenderán en el servidor i del área j')
```

**PASO 6. Declare las ecuaciones**

Las ecuaciones se definen utilizando funciones definidas por el usuario en las que se hace uso de las *list comprehension* de Python. Primero se escriben algebraicamente las ecuaciones y luego se convierten en un objeto del modelo.

```
def supply_rule(model, i):
    return sum(model.x[i,j] for j in model.j) <= model.S[i]
model.supply = Constraint(model.i, rule=supply_rule,
doc='No exceder la capacidad de cada servidor i')


def demand_rule(model, j):
    return sum(model.x[i,j] for i in model.i) >= model.D[j]
model.demand = Constraint(model.j, rule=demand_rule,
doc='Satisfacer la demanda de cada área j')
```

**PASO 7. Declare la función objetivo**

La declaración de la función objetivo (la ecuación que se desea maximizar o minimizar) es similar a la declaración de las restricciones, salvo algunos argumentos que varían para especificar la dirección de la optimización.

```
def objective_rule(model):
    return sum(model.C[i,j]*model.x[i,j] for i in model.i for j in model.j)
model.objective = Objective(rule=objective_rule, sense=minimize,
doc='Función objetivo')
```

**PASO 8. Corra la optimización y visualice los resultados**

Para correr la optimización escriba las siguientes líneas de comandos, que lo que hacen es enviar el modelo ya construido a la plataforma NEOS, para que utilizando solvers externos al equipo se encuentre la solución óptima del problema. Las últimas dos líneas permiten visualizar los valores de las variables de decisión y de la función objetivo.

```
instance = model
opt = SolverFactory("cbc")
solver_manager = SolverManagerFactory('neos')
results = solver_manager.solve(instance, opt=opt)
results.write()
model.x.display()
model.objective.display()
```

La siguiente es la forma como se muestran los resultados:

```
x : TB que se atenderán en el servidor i del área j
    Size=6, Index=x_index
    Key          : Lower : Value : Upper : Fixed : Stale : Domain
      ('H', 'C') :   0.0 :   0.0 :  None : False : False :  Reals
     ('H', 'CE') :   0.0 : 325.0 :  None : False : False :  Reals
     ('H', 'CO') :   0.0 : 275.0 :  None : False : False :  Reals
     ('SF', 'C') :   0.0 : 300.0 :  None : False : False :  Reals
    ('SF', 'CE') :   0.0 :   0.0 :  None : False : False :  Reals
    ('SF', 'CO') :   0.0 :   0.0 :  None : False : False :  Reals
objective : Size=1, Index=None, Active=True
    Key  : Active : Value
    None :   True : 153675.0
```

La solución óptima del problema indica que:

$$X_{H-C} = 0 \quad X_{H-CE} = 325 \quad X_{H-CO} = 275 \quad X_{SF-C} = 300 \quad X_{SF-CE} = 0 \quad X_{SF-CO} = 0$$

El costo de esta asignación óptima es de $153675.

**Importar los datos desde archivos csv**

Recuerde que en Python se tiene la posibilidad de importar datos desde archivos csv. Al leerlos, Python guarda los datos en un objeto que se puede convertir en una lista de listas. El siguiente comando puede serle útil para importar y transformar datos provenientes de archivos csv:

- Leer datos:
  - Ejemplo:

```
import csv
with open('Ruta del archivo csv que contiene los datos', 'r') as f:
    data = list(csv.reader(f, delimiter=','))
```

El objeto `reader` está en formato _csv.reader. El objeto `data` es una lista.

**Más información:**

Puede obtener más detalles sobre el funcionamiento del módulo `pyomo` consultando material en la web de los desarrolladores de él, en http://www.pyomo.org/ o en la web en general.