Nombre: Juan Manuel Varela

Entrega Métodos de Monte Carlo

Unidad 3 – Sesión 07 - Ejercicio 7.1

Descripción del problema:

Supongamos que un programa de posgrado incluye un conjunto de *P* profesores y otro de *S* estudiantes, y que se desea asignar a cada estudiante un profesor para consultas, pero que todas las personas son de diferentes países y comprenden distintos idiomas. Si cada estudiante tiene asignado un profesor para consultas, la cantidad total de formas de asignar profesores a los estudiantes es P^S (hay *P* opciones para cada estudiante, y *S* estudiantes en total, la cantidad total de opciones es el producto sobre los estudiantes de las opciones de cada estudiante).

Supongamos que nos interesa determinar cuántas formas hay de asignar profesores a los estudiantes, respetando que cada profesor asignado a un estudiante tenga un idioma en común con él (para que puedan comunicarse). Si hay *L* lenguajes posibles, podemos tener para cada estudiante *s* un subconjunto *Id(s)* con los lenguajes que entiende, y lo mismo para cada profesor *p*; y una asignación sólo sería valida si para cada estudiante *s* y su profesor *p(s)*, se cumple que $Id(s) \cap Id(p(s)) \neq \emptyset$ (estos conjuntos también podrían representarse como matrices $Id_{s,l}$ e $Id_{p,l}$, con entradas 1 cuando ese estudiante o profesor entienden el lenguaje *l* y 0 si no).

Para estimar la cantidad de formas distintas de realizar estas asignaciones de profesores y estudiantes, es posible aplicar el método Monte Carlo. Se debe recibir en entrada el número de replicaciones a realizar, y el nivel de confianza; en salida, se debe dar la estimación del número de combinaciones N_C, así como la desviación estándar y un intervalo de confianza (del nivel especificado) calculado en base al criterio de Agresti-Coull.

• Parte a: escribir un programa para hacer el cálculo previamente descrito. Entregar seudocódigo y código.

• Parte b: sea el siguiente caso: Estudiantes/lenguajes: María: español, inglés; Sophie: inglés, francés; Liliana: español, portugués; Lucia: inglés, portugués; Monique: francés; Rodrigo: español, inglés, francés; John: inglés; Neymar: portugués, español; Jacques: francés, portugués; Juan: español. Profesores: Tom: inglés, francés, español; Luciana: inglés, portugués; Gerard: inglés, francés; Silvia: español, francés. Usando el programa anterior, y empleando 1000 replicaciones de Monte Carlo, estimar los valores de cuantas combinaciones N_C hay para asignar profesores a estudiantes y que tengan un idioma en común, con intervalos de confianza de nivel 95%.

• Parte c: adaptar el programa para calcular el número de combinaciones si además queremos agregar como restricción que ningún profesor atienda menos de un estudiante ni más de cuatro estudiantes. Usando el programa modificado, y empleando 1000 replicaciones de Monte Carlo, estimar los valores de cuantas combinaciones N_C hay para asignar profesores a estudiantes y que tengan un idioma en común y cumplan la restricción adicional mencionada, con intervalos de confianza de nivel 95%.

Solución a aplicar en Python (parte a):

1. Función generar_asignacion(estudiantes, profesores):
   1.1. Crear una lista vacía llamada asignaciones
   1.2. Para cada estudiante en estudiantes:

1.2.1. Asignar un profesor_aleatorio de profesores
1.2.2. Agregar profesor_aleatorio a asignaciones
1.3. Salida: asignaciones
2. Función es_valida(asignacion, estudiantes, profesores):
2.1. Para cada estudiante y profesor en zip(estudiantes, asignacion):
2.1.1.Si la intersección de los conjuntos de lenguajes de estudiante y profesor es vacía:
2.1.1.1. Salida: Falso
2.2. Salida: Verdadero
3. Función agresti_coull(confianza, replicaciones, exitos):
3.1. Calcular p_hat, p_tilde, error #fracción de exitos, centro y error del intervalo
3.2. Salida: p_hat, p_tilde, error
4. Función estimar_combinaciones(estudiantes, profesores, replicaciones, confianza):
4.1. Inicializar exitos en 0
4.2. Para i desde 1 hasta replicaciones:
4.2.1. Generar una asignación llamada asignacion usando
generar_asignacion(estudiantes, profesores)
4.2.2. Si es_valida(asignacion, estudiantes, profesores):
4.2.2.1. Incrementar exitos en 1
4.3. Calcular p_hat, p_tilde, error usando agresti_coull(confianza, replicaciones, exitos)
4.4. Calcular N_C y desviacion_estandar
4.5. Salida: N_C, desviacion_estandar y el intervalo de confianza

Resultados:
Plataforma de cómputo: PC de escritorio, procesador Intel I5-12400, RAM: 32 Gb, Windows 10

Parte b:

Se ejecuta el código de la parte a utilizando como entrada los datos del problema:

Semilla: 1234
Replicaciones: 1000
N_C: 128975
Desviación estándar: 10896
Intervalo de confianza Agresti-Coull (95%): (109076, 151899)
Tiempo de ejecución: 0.006974697113037109 segundos

Parte c:

Se adapta el código de la parte a para evaluar las condiciones adicionales:

Semilla: 1234
Replicaciones: 1000
N_C: 79692
Desviación estándar: 8791
Intervalo de confianza Agresti-Coull (95%): (64036, 98750)
Tiempo de ejecución: 0.004983425140380859 segundos

Este resultado es coherente con el anterior ya que al agregar una restricción el número de combinaciones posibles es menor.

Anexos:

Log de ejecuciones:

```
runcell(0, 'G:/…/Ej7.1.py')
Parte b:
Ingrese la cantidad de replicaciones a realizar:
1000
Ingrese el nivel de confianza deseado:
0.95
Estimación de combinaciones NC: 128974.848
Desviación estándar: 10896.060891109579
Intervalo de confianza AC(95%): (109076.34628120286, 151898.88563653224)
Tiempo de ejecución (s): 0.006974697113037109
Parte c:
Estimación de combinaciones NC: 79691.776
Desviación estándar: 8791.438673100365
Intervalo de confianza (95%): (64036.575721992456, 98749.70102368308)
Tiempo de ejecución (s): 0.004983425140380859
```

Código fuente:

```
import random
import math
from scipy.stats import norm
import time

def generar_asignacion(estudiantes, profesores):
    return [random.choice(profesores) for estudiante in estudiantes]

def es_valida(asignacion, estudiantes, profesores):
    for estudiante, profesor in zip(estudiantes, asignacion):
        if not set(estudiante[1]).intersection(set(profesor[1])):
            return False
    return True

def agresti_coull(confianza, replicaciones, exitos):
    k = norm.ppf(1-(1-confianza)/2)
    p_hat = exitos / replicaciones
    n_hat = replicaciones + k**2
    p_tilde = (exitos + (k**2) / 2) / n_hat
    error = k * math.sqrt(p_tilde * (1 - p_tilde) / n_hat)
    return p_hat, p_tilde, error

def estimar_combinaciones(estudiantes, profesores, replicaciones, confianza):
    exitos = 0
    for i in range(replicaciones):
        asignacion = generar_asignacion(estudiantes, profesores)
        if es_valida(asignacion, estudiantes, profesores):
            exitos += 1
    p_hat, p_tilde, error = agresti_coull(confianza, replicaciones, exitos)
    NC = p_hat * (len(profesores) ** len(estudiantes))
    desviacion_estandar = math.sqrt(NC * ((len(profesores) **
len(estudiantes)) - NC) / (replicaciones-1))
    return NC, desviacion_estandar, ((p_tilde - error)*(len(profesores) **
len(estudiantes)), (p_tilde + error)*(len(profesores) ** len(estudiantes)))

#Parte b
```

```
print("Parte b:")

estudiantes = [("Maria", ["español", "ingles"]),
               ("Sophie", ["ingles", "frances"]),
               ("Liliana", ["español", "portugues"]),
               ("Lucia", ["ingles", "portugues"]),
               ("Monique", ["frances"]),
               ("Rodrigo", ["español", "ingles", "frances"]),
               ("John", ["ingles"]),
               ("Neymar", ["portugues", "español"]),
               ("Jacques", ["frances", "portugues"]),
               ("Juan", ["español"])]

profesores = [("Tom", ["ingles", "frances", "español"]),
              ("Luciana", ["ingles", "portugues"]),
              ("Gerard", ["ingles", "frances"]),
              ("Silvia", ["español", "frances"])]



replicaciones=int(input('Ingrese la cantidad de replicaciones a realizar:\n'))
confianza=float(input('Ingrese el nivel de confianza deseado:\n'))

random.seed(1234)

inicio=time.time()
NC, desviacion_estandar, intervalo_confianza =
estimar_combinaciones(estudiantes, profesores, replicaciones, 0.95)
tiempo_ejecucion=time.time()-inicio

print("Estimación de combinaciones NC:", NC)
print("Desviación estándar:", desviacion_estandar)
print("Intervalo de confianza AC(95%):", intervalo_confianza)
print("Tiempo de ejecución (s):", tiempo_ejecucion)

#Parte c
print("Parte c:")
random.seed(1234)

def cumple_restriccion(asignacion, profesores):
    conteo = {p[0]: 0 for p in profesores}
    for profesor in asignacion:
        conteo[profesor[0]] += 1
    return all(1 <= v <= 4 for v in conteo.values())

def estimar_combinaciones_c(estudiantes, profesores, replicaciones,
confianza):
    exitos = 0
    for i in range(replicaciones):
        asignacion = generar_asignacion(estudiantes, profesores)
        if es_valida(asignacion, estudiantes, profesores) and
cumple_restriccion(asignacion, profesores):
            exitos += 1
    p_hat, p_tilde, error = agresti_coull(confianza, replicaciones, exitos)
    NC = p_hat * (len(profesores) ** len(estudiantes))
    desviacion_estandar = math.sqrt(NC * ((len(profesores) **
len(estudiantes)) - NC) / (replicaciones-1))
    return NC, desviacion_estandar, ((p_tilde - error)*(len(profesores) **
len(estudiantes)), (p_tilde + error)*(len(profesores) ** len(estudiantes)))
```

```
inicio=time.time()
NC_c, desviacion_estandar_c, intervalo_confianza_c =
estimar_combinaciones_c(estudiantes, profesores, replicaciones, confianza)
tiempo_ejecucion=time.time()-inicio

print("Estimación de combinaciones NC:", NC_c)
print("Desviación estándar:", desviacion_estandar_c)
print("Intervalo de confianza (95%):", intervalo_confianza_c)
print("Tiempo de ejecución (s):", tiempo_ejecucion)
```