Pregunta 1

En la administración de servidores web, el número de “Hits” o peticiones que recibe un servidor por minuto es una variable de conteo (números enteros no negativos), en este sentido, el tráfico web se modela mejor con una distribución de Poisson.

Deseamos predecir la tasa media de peticiones (λ) en función del número de usuarios activos simultáneos.

Utilizaremos un modelo lineal generalizado (GLM). La probabilidad de observar y peticiones dado un vector de características x es:

$$P(Y = y|x;\beta) = \frac{e^{-\lambda} * \lambda^{y}}{y!}$$

Donde la tasa media λ se relaciona con la entrada mediante la función de enlace logarítmica (para garantizar $\lambda > 0$):

$$\lambda = e^{B_0 + B_1 * X}$$

Actividades:

1. Derivar matemáticamente el gradiente $\nabla J(\beta)$.
2. Implementar el método de descenso de gradiente para encontrar β.
3. Predecir la tasa de peticiones para una carga de usuarios inédita.

Se pide que el código tenga:

```
# Datos simulados pregunta 1
#------------------------
import numpy as np
import pandas as pd
np.random.seed(2025)
m = 100
# Variable independiente (usuarios)
usuarios = np.random.uniform(10, 100, m)
# Variable dependiente (peticiones)
lambda_real = np.exp(0.5 + 0.03 * usuarios)
peticiones = np.random.poisson(lambda_real)
datos_pregunta_1 = pd.DataFrame({"Usuarios": usuarios, "Peticiones": peticiones})
print(datos_pregunta_1.head())
```

SOLUCIÓN:

```python
# Datos simulados pregunta 1
#-------------------------
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math as mate
import scipy.special
# semilla
np.random.seed(2025)
m = 100
# Variable independiente (usuarios)
# NUMERO DE USUARIOS   = X
usuarios = np.random.uniform(10, 100, m)
# Variable dependiente (peticiones)
lambda_real = np.exp(0.5 + 0.03 * usuarios)
# NUMERO DE PETICIONES = Y
peticiones = np.random.poisson(lambda_real)
datos_pregunta_1 = pd.DataFrame({"Usuarios": usuarios,"Peticiones": peticiones})
print(datos_pregunta_1.head())
```

Esta imagen contiene el código pedido con algunas líneas más de comentario y algunas librerías extras para desarrollar el ejercicio.

Solucion de Actividad 1:

```python
# Calculo de la derevida
def grand(beta, usuarios, peticiones):

    y = np.array(peticiones)
    x = np.array(usuarios)
    Lambda= Landa(x, beta)
    error =Lambda - y
    grad_beta0 = np.sum(error)
    grad_beta1 = np.sum(x * error)

    return grad_beta0, grad_beta1
```

Solucion de Actividad 2:

```python
# Descenso del gradiente
def desonse_grand(usuarios, peticiones, beta, alpha, iteraciones, tolerancia, epsilon):

    y = np.array(peticiones)
    x = np.array(usuarios)
    historial_costo = []

    for i in range(iteraciones):
        # Guardar valores actuales
        costo_actual = Costo_J(y, x, beta, epsilon)
        historial_costo.append(costo_actual)

        # Calcular gradiente
        grad_beta0, grad_beta1 = grand(beta, x, y)

        # Actualizar parámetros
        beta0_nuevo = beta[0] - alpha * grad_beta0
        beta1_nuevo = beta[1] - alpha * grad_beta1

        beta_nuevo = np.array([beta0_nuevo, beta1_nuevo])

        # Imprimir progreso cada 1000 iteraciones
        if i % (iteraciones*0.1) == 0:
            print(f"Iteración {i:5d} | Costo: {costo_actual:10.4f} | "
                  f"β₀: {beta[0]:8.5f} | β₁: {beta[1]:8.5f}")

        # Verificar convergencia
        cambio_beta0 = abs(beta_nuevo[0] - beta[0])
        cambio_beta1 = abs(beta_nuevo[1] - beta[1])

        if cambio_beta0 < tolerancia and cambio_beta1 < tolerancia:
            print(f"\n¡Convergencia alcanzada en iteración {i}!")
            break

        beta[0] = beta_nuevo[0]
        beta[1] = beta_nuevo[1]

    return beta, historial_costo
```

Solucion de Actividad 3:

```
El valor del gradiente es:  (-1077.0, -83066.60732692113)
/n
El valor del costo es:  99.99882300058859
/n
Iteración     0 | Costo:    99.9988 | β₀:  0.00000 | β₁:  0.00000
Iteración 10000 | Costo: -1876.5256 | β₀:  0.18754 | β₁:  0.03958
Iteración 20000 | Costo: -1883.4974 | β₀:  0.32912 | β₁:  0.03787
Iteración 30000 | Costo: -1887.6214 | β₀:  0.43401 | β₁:  0.03659
Iteración 40000 | Costo: -1890.0386 | β₀:  0.51038 | β₁:  0.03565
Iteración 50000 | Costo: -1891.4593 | β₀:  0.56526 | β₁:  0.03498
Iteración 60000 | Costo: -1892.3049 | β₀:  0.60429 | β₁:  0.03450
Iteración 70000 | Costo: -1892.8178 | β₀:  0.63186 | β₁:  0.03415
Iteración 80000 | Costo: -1893.1358 | β₀:  0.65122 | β₁:  0.03391
Iteración 90000 | Costo: -1893.3375 | β₀:  0.66477 | β₁:  0.03375
El valor de beta es:  [0.67422812 0.03362826]
Usuarios:  25 → Tasa predicha (λ):     4.55 peticiones/min
Usuarios:  50 → Tasa predicha (λ):    10.54 peticiones/min
Usuarios:  75 → Tasa predicha (λ):    24.44 peticiones/min
Usuarios: 100 → Tasa predicha (λ):    56.66 peticiones/min
```

Conclusión:

Fue el código más fácil de desarrollar, con único problema de calcular correctamente el gradiente de F según beta0 y beta1, y por eso mismo el descenso del gradiente da resultados incorrectos.
El código se empezó ingresando el código pedido y luego desarrollando en lenguaje python las funciones dadas para luego desarrollar un algoritmo para las actividades.

Ejercicio 2

En Internet de las Cosas (IoT), a menudo necesitamos calibrar sensores baratos. Supongamos un sensor de distancia basado en la intensidad de señal WiFi (RSSI). La relación teórica sigue una ley de potencia inversa con un término de ruido base. El modelo físico propuesto es:

$$h(x;\theta) = \frac{\theta_0}{X^{\theta_1}} + \theta_2$$

Donde x es la distancia real y $h(x)$ es la lectura del sensor.
El objetivo de optimización es minimizar el error cuadrático medio

$$f(\theta) = \sum_{i=0}^{n} \left(y_i - \left(\frac{\theta_0}{X^{\theta_0}} + \theta_2\right)\right)^2$$

Utilice el método BFGS provisto por scipy para manejar la curvatura sin calcular la Hessiana manualmente.

Actividades:

1. Definir la función objetivo.
2. Utilizar scipy.optimize.minimize con el método BFGS
3. Comparar el resultado con diferentes inicializaciones aleatorias para observar la sensibilidad a los puntos de partida.

Se pide que el código tenga:

```
# Datos simulados pregunta
#------------------------
import numpy as np
import pandas as pd
np.random.seed(2025)
# Variable independiente (metros)
distancia = np.linspace(1, 20, 50)
# Variable dependiente (lectura sensor)
lectura_sensor = (50 / (distancia ** 1.5)) + 2 + np.random.normal(0, 0.5, 50)

datos_pregunta_2 = pd.DataFrame({"Distancia": distancia,
                                 "Lectura_Sensor":lectura_sensor})
print(datos_pregunta_2.head())
```

Solución

```python
# Datos simulados pregunta 2
#------------------------
import numpy as np
import pandas as pd
from scipy.optimize import minimize
import matplotlib.pyplot as plt

np.random.seed(2025)
# Variable independiente (metros)
#Distancia       = x
distancia = np.linspace(1, 20, 50)
# Variable dependiente (lectura sensor)
#lesctura_sensor = y
lectura_sensor = (50 / (distancia ** 1.5)) + 2 + np.random.normal(0, 0.5, 50)
datos_pregunta_2 = pd.DataFrame({"Distancia": distancia,
"Lectura_Sensor": lectura_sensor})
print(datos_pregunta_2.head())
```

Esta imagen contiene el código pedido con algunas líneas más de comentario y algunas librerías extras para desarrollar el ejercicio.

Solucion de Actividad 1:

```python
def F(beta, distancia, lectura_sensor):
    y = np.array(lectura_sensor)
    x = np.array(distancia)
    h = div(beta, x)
    f = y - h
    f_fin = np.sum(f**2)
    return f_fin
```

Solucion de Actividad 2:

```python
# Llamada a la función de optimización
resultado = minimize(
    fun=F,                                  # Función a minimizar
    x0=beta,                                # Parámetros iniciales
    args=(distancia, lectura_sensor),       # Argumentos adicionales
    method='BFGS',                          # Método de optimización
    options={'disp': True})                  # Mostrar progreso

```

Solucion de Actividad 3:

```
experimento: 1
beta0_inicial: 18.735460164798326
beta1_inicial: 2.5860440347268105
beta2_inicial: -4.621662226977463
beta0_optimo: 49.93129973158694
beta1_optimo: 1.5051646023147 66
beta2_optimo: 2.070207892259479
costo_final: 10.30524686555783
exito: True
n_iteraciones: 19
n_evaluaciones: 116
experimento: 2
beta0_inicial: 31.489141496187308
beta1_inicial: 2.6427708423237384
beta2_inicial: -4.243093934130878
beta0_optimo: 49.93129950128451
beta1_optimo: 1.5051645576063781
beta2_optimo: 2.0702076388311887
costo_final: 10.305246865560122
exito: True
n_iteraciones: 17
n_evaluaciones: 108
experimento: 3
beta0_inicial: 27.46982656433918
beta1_inicial: 1.689764185724136
beta2_inicial: -1.4722451787201516
beta0_optimo: 49.93129933957373
beta1_optimo: 1.5051645944393182
beta2_optimo: 2.0702078986347994
costo_final: 10.305246865558109
exito: True
n_iteraciones: 16
n_evaluaciones: 92
```

Conclusión:

El código fue relativamente sencillo si supiera leer bien el problema ya que gran parte del tiempo se perdió implementando un código de BFGS, pero resulta que se tenía que hacer con respecto a una función de la librería scipy
El código se empezó ingresando el código pedido y luego desarrollando en lenguaje python las funciones dadas para luego desarrollar un algoritmo para las actividades.

Ejercicio 3

Imagine un administrador de contenedores (como Kubernetes). Tiene una lista de N microservicios, cada uno con una carga computacional estimada (CPU load). Debe asignar cada servicio a uno de los K servidores disponibles. El objetivo es minimizar el desequilibrio (varianza) de la carga entre los servidores.

Sea S ={s1,s2,…,sN} el conjunto de cargas de los servicios. Sea A un vector de asignación donde Ai ∈ {0,1,…,K −1} indica el servidor del servicio i. La carga total del servidor k es:

$$L_k = \sum_{i:A_i=K} (s_i)$$

La función objetivo minimiza la desviación estándar de las cargas de los servidores:

$$f(A) = \sqrt{\frac{1}{k} * \sum_{k=0}^{k} (L_k - \overline{L})^2}$$

Actividades:

1. Implemetar la función objetivo basada en la desviación estándar
2. Adaptar el algoritmo simulated annealing
    a. Estado: Vector de enteros (asignación)
    b. Vecino: Mover un servicio aleatorio de un servidor a otro.
3. Analizar la reducción del desequilibrio.

Se pide que el código tenga:

```
# Datos simulados pregunta
#------------------------
import numpy as np
np.random.seed(2025)
n_servicios = 50
n_servidores = 4
cargas_cpu = np.random.randint(1, 20, n_servicios)
```

Solución

```python
# Datos simulados pregunta 3
#-------------------------
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math as mate
import scipy.special

# semilla
np.random.seed(2025)
# S = servicios
n_servicios = 50
# K = servidor
n_servidores = 4
#conjunto de carga de los servicios
cargas_cpu = np.random.randint(1, 20, n_servicios)
# Asignacion
A = np.random.randint(0, n_servidores, n_servicios)
t_inicial=1000
t_final=0.1
alpha=0.3
max_iter=1000
```

Esta imagen contiene el código pedido con algunas líneas más de comentario y algunas librerías extras para desarrollar el ejercicio.

Solucion de Actividad 2:

```python
def f_objetico(cargas_cpu, n_servidores,A):

    s = np.array(cargas_cpu)
    k = n_servidores

    lk = Lk(s, k, A)

    # Calcular media de cargas
    carga_media = np.mean(lk)

    lk_diferencia= lk - carga_media
    lk_cuadrado = lk_diferencia**2
    f = (1/n_servidores) * np.sum(lk_cuadrado)
    f = np.sqrt(f)
    print(f)
    return f
```

Solucion de Actividad 2:

```python
def generar_vecino(A_actual, n_servidores):

    A_vecino = A_actual.copy()
    # Seleccionar un servicio aleatorio
    servicio = np.random.randint(0, len(A_vecino))

    # Seleccionar un servidor diferente al actual
    servidor_actual = A_vecino[servicio]
    servidores_disponibles = [s for s in range(n_servidores) if s != servidor_actual]
    nuevo_servidor = np.random.choice(servidores_disponibles)

    # Mover el servicio
    A_vecino[servicio] = nuevo_servidor

    return A_vecino
```

```python
def simulated_annealing(cargas_cpu, n_servidores, t_inicial, t_final, alpha, max_iter,A):

    # Asignación inicial aleatoria
    A_actual = A.copy()
    f_actual = f_objetico(cargas_cpu, n_servidores, A_actual)

    # Mejor solución
    A_mejor = A_actual.copy()
    f_mejor = f_actual

    # Variables de control
    temperatura = t_inicial
    iteraciones = 0

    # Historial
    historial_costo = []
    historial_temp = []
    while temperatura > t_final and iteraciones < max_iter:
        # Generar vecino
        A_nuevo = generar_vecino(A_actual, n_servidores)
        f_nuevo = f_objetico(cargas_cpu, n_servidores, A_nuevo)

        # Calcular delta
        delta = f_nuevo - f_actual

```

```
        # Criterio de aceptación
        if delta < 0:
            # Mejor solución: aceptar
            A_actual = A_nuevo
            f_actual = f_nuevo

            if f_actual < f_mejor:
                A_mejor = A_actual.copy()
                f_mejor = f_actual
        else:
            # Peor solución: aceptar con probabilidad
            probabilidad = np.exp(-delta / temperatura)
            if np.random.rand() < probabilidad:
                A_actual = A_nuevo
                f_actual = f_nuevo

        # Registrar historial
        historial_costo.append(f_actual)
        historial_temp.append(temperatura)
        # Enfriar
        temperatura = alpha * temperatura
        iteraciones += 1

        # Mostrar progreso cada 1000 iteraciones
        if iteraciones % 1000 == 0:
            print(f"Iter {iteraciones:5d} | T={temperatura:7.2f} | "
                  f"Actual={f_actual:7.4f} | Mejor={f_mejor:7.4f}")
```

```

    print()
    print(f"✓ Convergencia alcanzada!")
    print(f"Iteraciones totales: {iteraciones}")
    print(f"Desviación final: {f_mejor:.4f}")
    print()

    return A_mejor, f_mejor, iteraciones, historial_costo, historial_temp
```

Conclusión:

Por falt de tiempo esto es todo lo que se pudo hacer (problemas en la gestión de tiempo del estudiante no de la fecha

Resultó ser el ejercicio más estresante y emocionante en mucho tiempo, por 3 razones
1- Se empezó a resolver el ejercicio 3 el dia lunes 15 (dia limite de entrega) a las 4 de la tarde
2- Se se logro comprender bien lo que pedía la actividad 2
3- El entendimiento de S,A y K tomó más tiempo de lo estimado por confusión del estudiante.

Los más ejercicios fueron realizados días anteriores teniendo como horas trabajadas para cada ejercicio 4 y 6 para ejercicio 1 y 2 respectivamente, pero en la mente del alumno este 3 ejercicio se podía hacer en menos tiempo.

Se agrega un último gráfico de enfriamiento, solo porque esta bonito y tomo alrededor de 1 hora que resultó correcto.