Certamen - 1 - INF-285
SCT - Vi.11.04.25 - V.1.01

Instrucciones: *Usted tiene 120 minutos para responder el Certamen con letra clara y de forma ordenada.* ***Usted tiene que mostrar todo su trabajo para obtener todos los puntos****. Puntos parciales serán entregados a preguntas incompletas. Respuestas finales sin desarrollo,* ***sin nombre*** *o con lápiz rojo reciben 0 puntos. Buena letra, claridad, completitud,* ***ser conciso*** *y orden en* ***todo el certamen*** *recibe 10 puntos adicionales, excepcionalmente se considerarán puntos parciales. Copy-and-Paste de algoritmos reciben 0 puntos. ¡Éxito!*

- **Contexto Pregunta 1**: *Taylor*: Uno de los teoremas más importantes en el ámbito de la Computación Científica es el teorema de Taylor, en particular la versión con su residuo. Por completitud se incluye a continuación:

  Thm 1 (Teorema de Taylor con residuo)**.** *Sea x y x_0 números reales, y $f(x)$ una función $k+1$-veces continuamente diferenciable en el intervalo entre x y x_0, entonces existe un número c entre x y x_0 tal que:*

$$f(x) = f(x_0) + f'(x_0)\,(x - x_0) + \frac{f''(x_0)}{2!}\,(x - x_0)^2 + \cdots + \frac{f^{(k)}(x_0)}{k!}\,(x - x_0)^k + \frac{f^{(k+1)}(c)}{(k+1)!}\,(x - x_0)^{k+1}.$$

  El teorema indica es que si uno tiene una función $k+1$-veces diferenciable, es decir, sus $k+1$-derivadas son continuas y diferenciables, se puede encontrar una relación entre la función $f(x)$, los coeficientes conocidos x y x_0, y **el coeficiente desconocido** c. En específico se conoce que $c \in [\text{mín}(x, x_0), \text{máx}(x, x_0)]$. Lo interesante de este teorema es que asegura la existencia de c en el intervalo indicado tal que la igualdad de la expresión anterior se cumple. Desafortunadamente no se conoce el valor de c de forma explícita.

- **Contexto Pregunta 2**: *Similitud Coseno*: En el contexto de la Inteligencia Artificial, en particular en el área asociada a Máquinas de Aprendizaje, se usa tradicionalmente la pseudo-norma llamada *similitud coseno*. La cual se define de la siguiente forma:

$$\text{SimilitudCoseno}(\mathbf{x}, \mathbf{y}) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|_2 \, \|\mathbf{y}\|_2},$$

  donde "$\langle \mathbf{x}, \mathbf{y} \rangle$" representa el producto interno y $\|\mathbf{x}\|_2$ representa la norma 2. El resultado anterior también representa el coseno del ángulo θ entre los vectores $\mathbf{x}$ e $\mathbf{y}$, es decir,

$$\frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|_2 \, \|\mathbf{y}\|_2} = \cos(\theta).$$

  Para su análisis inicial, considere la siguiente implementación:

```
'''
input:
x   : (ndarray) Input vector 'x'.
y   : (ndarray) Input vector 'y'.

output:
cs  : (float) The cosine similarity - first approximation.
'''
def cosine_similarity_v1(x,y):
    # Computing the dot-product
    dot_product = np.dot(x,y)

    # Computing the 2-norm of x
    n_x = my_norm_2(x)

    # Computing the first division
    out = dot_product/n_x

    # Computing the 2-norm of y
    n_y = my_norm_2(y)

    # Computing the second division
    cs = out/n_y

    # Returning the computed cosine similarity
    return cs
```

donde la función `my_norm_2` se define a continuación,

```
'''
input:
x   : (ndarray) Input vector 'x'.

output:
norm_2  : (float) norm-2 of the vector 'x'
'''
def my_norm_2(x):
    # Getting the max absolute value
    max_abs_value = np.max(np.abs(x))

    # Scaling the vector
    x_tilde = x/max_abs_value

    # Computing the 2-norm of the scaled vector
    x_tilde_norm = np.linalg.norm(x_tilde)

    # Computing the actual 2-norm of 'x'
    norm_2 = max_abs_value*x_tilde_norm

    # Return the computed norm-2
    return norm_2
```

Note que la implementación anterior está tomando ventaja de la propiedad de que uno puede escalar convenientemente un vector, obtener la norma del vector escalado y finalmente multiplicar el resultado por el recíproco del coeficiente en valor absoluto. Por ejemplo, si denotamos el escalamiento como α entonces el escalamiento se interpreta como:

$$\begin{aligned} \|\mathbf{x}\| &= \left\| \alpha \frac{1}{\alpha} \mathbf{x} \right\| \\ &= |\alpha| \left\| \frac{\mathbf{x}}{\alpha} \right\|. \end{aligned}$$

NOMBRE:_____ ROL:_____ PARALELO:_____

- **Desarrollo Pregunta 1**:

(a) **[30 puntos]** Proponga un algoritmo que exhiba convergencia cuadrática para obtener el coeficiente c de una expansión de Taylor de $k+1$ términos dado los coeficientes conocidos x y x_0, y la función $f(x)$. Adicionalmente considere que usted tiene acceso a evaluar todas las derivadas que requiera de $f(x)$, por ejemplo la k-ésima derivada de $f(x)$ se denota como $f^{(k)}(x)$ para $k \in 0, 1, 2, 3, \ldots$, donde $f^{(0)}(x)$ se interpreta simplemente como la evaluación de la función $f(x)$.

En resumen, usted tiene acceso a x, x_0, k, $f(x)$, y $f^{(k)}(x)$, y debe obtener c tal que la siguiente ecuación se cumpla:

$$f(x) = f(x_0) + f'(x_0)\,(x - x_0) + \frac{f''(x_0)}{2!}\,(x - x_0)^2 + \cdots + \frac{f^{(k)}(x_0)}{k!}\,(x - x_0)^k + \frac{f^{(k+1)}(c)}{(k+1)!}\,(x - x_0)^{k+1}.$$

Por simplicidad de análisis considere que la raíz tiene multiplicidad 1.

(b) **[20 puntos]** Implemente en Python utilizando adecuadamente la librería NumPy (**en especial su capacidad de vectorización**) el algoritmo propuesto en la Pregunta 1 ítem (a), es decir, en la pregunta anterior. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos, condicionales propios de Python y la función `range`:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`. Por ejemplo si `np.arange(3)` retorna el vector `[0,1,2]`.
- `np.abs(x)`: Entrega el valor absoluto de `x`.
- `np.power(x,n)`: Evalúa la expresión x^n si `x` y `n` son escalares. En caso de que `x` e `n` sean vectores, deben tener la misma dimensión y entrega la evaluación elemento a elemento. Si solo uno de los términos es un vector, entrega el vector donde el término constante se consideró para cada término de vector. Por ejemplo si `np.power(3,[0,1,2])` retorna el vector `[1,3,9]`.
- `np.sqrt(x)`: Entrega la evaluación de la raíz cuadrada no negativa de un vector o escalar `x`.
- `bisection(f,a,b)`: Implementa el método de la Bisección para la búsqueda de la raíz de la función `f` recibida como entrada en el intervalo [`a`,`b`]. Por simplicidad se omite criterio de detención.
- `Newton1D(f,fp,x0,m=1)`: Implementa el método de Newton en 1D que entrega la aproximación de la raíz de `f`. Esta función recibe como parámetros requeridos la función `f`, la derivada `fp` de `f` y el *initial guess* `x0`. Adicionalmente puede recibir como parámetro adicional la multiplicidad de la raíz, es decir `m`, el cual está definido por defecto en `1`. Por simplicidad se omite criterio de detención.
- `np.dot(a,b)`: Obtiene el producto interno entre el vector `a` y `b`. En caso de que `a` sea una matriz, entrega el producto matriz-vector respectivo. Para esto último también es posible utilizar el operador `@`.
- `k_derivative_of_f(x,k)`: Esta función entrega la k-ésima derivada de la función $f(x)$ evaluada en x. Notar que si $k = 0$, entonces simplemente se evalúa la función $f(x)$ en x. Esta función también puede recibir un vector como parámetro `k`, y retorna las derivadas indicadas en el vector. Por ejemplo si `k_derivative_of_f(1.0,[0,1,2])` retorna el vector numérico que representa $\langle f(1.0), f'(1.0), f^{(2)}(1.0)\rangle$.
- `my_factorial(n)`: Calcula el factorial de número natural `n`, es decir si $n = 5$ entonces la función entrega $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. En caso de que `n` sea un vector de números naturales, entrega el factorial de cada elemento, por ejemplo si `[3,5]` entonces la función entrega `my_factorial([3,5])=[6,120]`.

Notar que al momento de implementar usted **debe decidir** qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
'''
input :
x       : (float) value of $x$.
x0      : (float) value of $x_0$
k       : (integer) This defines the number of terms to be used, which is $k+1$.

output:
c       : (float) the estimated value for $c$.
'''
def find_c(x,x0,k):
    # Your own code.




    return c
```

NOMBRE:_____ ROL:_____ PARALELO:_____

- **Desarrollo Pregunta 2**:

(a) **[15 puntos]** Considere los siguientes vectores:

$$\mathbf{x} = \left[2^{-800}, 2^{-800}\right],$$
$$\mathbf{y} = \left[2^{-700}, 2^{-700}\right].$$

Ejecute la función `cosine_similarity_v1` con los vectores $\mathbf{x}$ e $\mathbf{y}$ definidos anteriormente y considerando *double precision*.

Detalle explícitamente cada paso en la ejecución de su algoritmo. Incluso cuando ejecuta la función `my_norm_2` que es parte de `cosine_similarity_v1`.

(b) **[20 puntos]** Proponga un algoritmo alternativo para obtener la similitud coseno tal que entregue el resultado correcto para los vectores antes mencionados y obtenga el resultado con su propuesta (simplifique lo que más pueda, no se requiere uso de calculadora).

En resumen:

a) Proponga un algoritmo alternativo para obtener la similitud coseno tal que entregue el resultado correcto.

b) Obtenga el resultado con su propuesta (simplifique lo que más pueda, no se requiere uso de calculadora).

(c) [**15 puntos**] Implemente en Python utilizando adecuadamente la librería NumPy (**en especial su capacidad de vectorización**) el algoritmo propuesto en la Pregunta 2 ítem (b), es decir, en la pregunta anterior. Para su implementación, considere que **solo** tiene a su disposición las siguientes funciones de la librería NumPy, además de las operaciones elementales, ciclos, condicionales propios de Python y la función `range`:

- `np.arange(n)`: Para `n` un número entero positivo entrega un vector de largo `n` con números enteros desde `0` a `n-1`. Por ejemplo si `np.arange(3)` retorna el vector `[0,1,2]`.
- `np.abs(x)`: Entrega el valor absoluto de `x`.
- `np.max(x)`: Entrega el valor máximo del vector `x`.
- `np.min(x)`: Entrega el valor máximo del vector `x`.
- `np.power(x,n)`: Evalúa la expresión x^n si `x` y `n` son escalares. En caso de que `x` e `n` sean vectores, deben tener la misma dimensión y entrega la evaluación elemento a elemento. Si solo uno de los términos es un vector, entrega el vector donde el término constante se consideró para cada término de vector. Por ejemplo si `np.power(3,[0,1,2])` retorna el vector `[1,3,9]`.
- `np.sqrt(x)`: Entrega la evaluación de la raíz cuadrada no negativa de un vector o escalar `x`.
- `np.dot(a,b)`: Obtiene el producto interno entre el vector `a` y `b`. En caso de que `a` sea una matriz, entrega el producto matriz-vector respectivo. Para esto último también es posible utilizar el operador `@`.
- `norma_2_direct(x)`: Obtiene la norma 2 del vector `x` por medio de sumar cada elemento al cuadrado y luego obtener su raíz cuadrada.
- `my_norm_2(x)`: Obtiene la norma 2 del vector `x` considerando la explicación antes entregada. Ver implementación en el contexto.

Notar que al momento de implementar usted **debe decidir** qué componentes se deben vectorizar y qué componentes no, considerando las funciones de NumPy antes mencionadas. Considere la siguiente firma:

```
'''
input :
x       : (ndarray) Input vector 'x'.
y       : (ndarray) Input vector 'y'.

output:
cs  : (float) The cosine similarity - second approximation.
'''
def cosine_similarity_v2(x,y):
    # Your own code.
```

```
    return cs
```