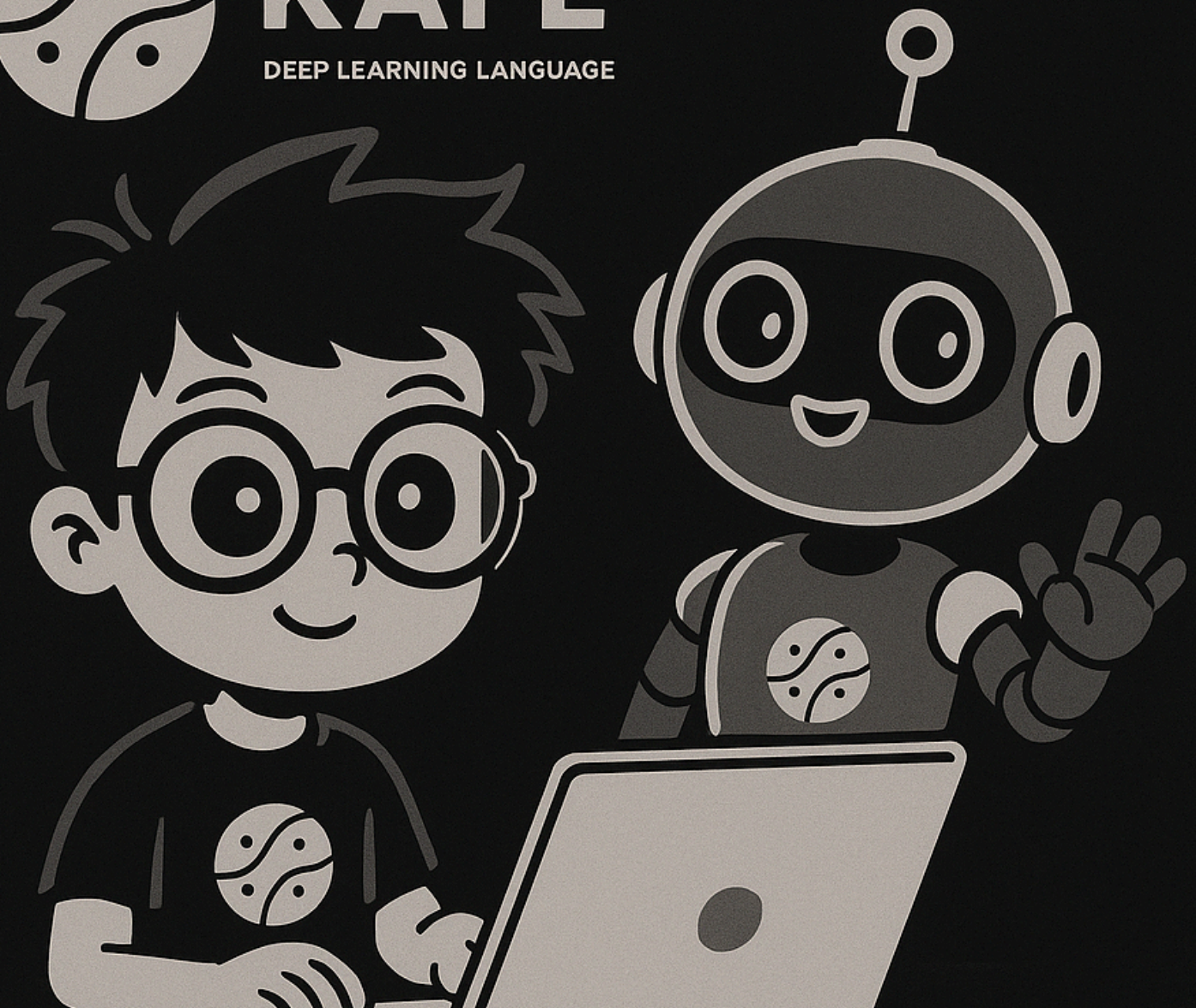
DEEP LEARNING LANGUAGE for DUMMIES

KAFE

DEEP LEARNING LANGUAGE



KAFE

DEEP LEARNING LANGUAGE

Andres Felipe Sindicue Alvarado

Luis Felipe Valencia Ramirez

Emanuel Felipe Molina Triana

Karen Yireth Castañeda Castro

Agradecimientos:

Queremos expresar nuestro más sincero agradecimiento a todas las personas que hicieron parte de este proyecto.

En especial, agradecemos profundamente a nuestro profesor Joaquín Sánchez, quien nos acompañó con paciencia, conocimiento y visión durante todo el proceso de diseño y desarrollo de KAFE.

Introducción

KAFE es un lenguaje de programación diseñado para la comunidad académica. Domain-Specific Language (DSL) enfocado en el desarrollo de redes neuronales, con una orientación funcional que permite el uso de funciones currificadas, composición funcional y estructuras declarativas.

Este lenguaje ha sido creado para facilitar el aprendizaje de conceptos fundamentales en programación y bases de Deep Learning. Nuestro objetivo es proporcionar una herramienta simple, clara y educativa que promueva una comprensión profunda del funcionamiento interno de los modelos neuronales.

KAFE es desarrollado por un equipo de cuatro estudiantes de Ciencias de la Computación e Inteligencia Artificial, con el propósito de contribuir al ecosistema educativo. Nos alegra que estés explorando y utilizando este proyecto.

Inicialmente, está pensado para ejecutarse en este entorno web: puedes escribir tu código, compilarlo, descargar los archivos generados y, si deseas continuar más adelante, simplemente vuelve a subirlos para seguir trabajando donde lo dejaste.

Guía Rápida para Empezar

KAFE es tu entorno de desarrollo online para aprender y experimentar con programación funcional de manera simple, intuitiva y sin necesidad de instalaciones complicadas.

Ya sea que estés comenzando tu camino en el mundo del código o que quieras explorar conceptos avanzados como lambdas, currificación o visualización de grafos Y MÁS!!! , KAFE te lo pone fácil.

Descarga el proyecto completo

¿Prefieres trabajar localmente o contribuir al desarrollo?

Dirígete a nuestro repositorio oficial y descarga KAFE:

🔗 *https://github.com/pipe2711/KAFE.git*

Ahí encontrarás documentación y un video para el uso de KAFE, recuerda que esto es un proyecto que sigue en desarrollo, puedes hacer parte del mismo, creando nuevas funcionalidades y más.

FUNDAMENTOS DEL LENGUAJE KAFE

COMENTARIOS

En KAFE, los comentarios permiten incluir anotaciones o explicaciones dentro del código, sin que afecten su ejecución. Son útiles para describir la intención del código, explicar secciones complejas o dejar recordatorios.

Tipos de Comentarios

Comentario de una sola línea:

Se escribe iniciando la línea con el símbolo --. Todo lo que siga en esa línea será ignorado por el compilador.

```
--Este es un comentario de una línea

INT x = 10; --También puede ir al final de una línea de código
```

Comentario de múltiples líneas:

Se delimita entre -> y <-, y puede ocupar varias líneas.

```
->
Este es un comentario
de varias líneas en KAFE.
<-
INT y = x + 5;
```

Los comentarios no afectan la ejecución ni el resultado del programa.

Se recomienda usarlos para mejorar la legibilidad del código.

VARIABLES Y TIPOS

En KAFE, las variables son contenedores donde puedes guardar información para usarla más adelante. Cada variable tiene un tipo de dato, que define el tipo de valor que puede almacenar, como números , texto o valores lógicos.

Los tipos primitivos en KAFE son simples pero poderosos, permitiéndote construir programas claros y seguros. Declarar una variable en KAFE es directo, y el lenguaje se asegura de que los tipos se usan correctamente durante la ejecución.

Enteros

Para declarar un número entero, se utiliza el tipo INT. Los enteros pueden ser positivos o negativos, y se definen de la siguiente manera:

```
INT a = 5;
```

Booleanos

Si deseas trabajar con valores booleanos (verdadero o falso), se utiliza el tipo BOOL. Es útil para controlar flujos lógicos, condiciones y ciclos:

```
BOOL b = True;
```

Strings

Las cadenas de texto se representan con el tipo STR. Puedes usar tanto comillas simples como dobles para definirlas. Ambas son equivalentes:

```
STR nombre_de_tu_string = 'Hola este es un String';
```

Flotantes

Para trabajar con números reales o decimales, se utiliza el tipo FLOAT. Este tipo es ideal para representar valores con parte fraccionaria, como porcentajes, promedios o resultados matemáticos más precisos:

```
FLOAT e = 234.234;
```

Recuerda que todas las instrucciones en KAFE deben finalizar con punto y coma (;). Además, el lenguaje es sensible a mayúsculas y minúsculas, así que True no es true.

OPERACIONES MATEMÁTICAS

En KAFE, puedes realizar operaciones matemáticas de forma sencilla utilizando los operadores aritméticos estándar. Estas operaciones son compatibles con los tipos de datos numéricos primitivos.

Operadores Aritméticos

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
^	Potencia
%	Módulo (residuo)

En **KAFE**, para que veas el resultado de tu código en la consola, usamos una palabrita mágica SHOW

Piensa en show como el equivalente a decirle al lenguaje:

👉 *"¡Ey! Quiero que esto se vea en pantalla."*

Así que cada vez que quieras imprimir algo, solo escribe show(); con lo que quieras mostrar entre paréntesis.

```
show(5 + 4 * 2);        -- Resultado: 13
show((5 + 4) * 2);      -- Resultado: 18
show(5 ^ 2 * 2);        -- Resultado: 50
show(-3 - -3);          -- Resultado: 0
show(5 % 4);            -- Resultado: 1
```

Notas importantes

- KAFE respeta la **precedencia de operadores**.
- Puedes usar **paréntesis ()** para modificar el orden de evaluación de las expresiones.
- El operador ^ se utiliza para elevar un número a una potencia.
- El operador % devuelve el **residuo** de una división entera.

Estas operaciones permiten construir cálculos complejos de forma clara y expresiva dentro del lenguaje, conservando la elegancia y simplicidad de la programación funcional.

LISTAS Y MATRICES

En KAFE, las listas son estructuras de datos muy versátiles que te permiten almacenar múltiples elementos del mismo tipo, ya sea números, cadenas, booleanos, o incluso otras listas. Esto significa que puedes representar desde colecciones simples hasta estructuras de datos complejas como matrices o listas multidimensionales.

¿Cómo se crea una lista?

Muy fácil. Mira este ejemplo de una lista vacía:

```
List[INT] numeros = [];
```

Y también puedes **inicializarla directamente** con algunos elementos:

```
List[BOOL] g = [True, False, True];
```

¿Y si quiero una lista de listas?

Aquí puedes tener una lista que guarda otras listas:

```
List[List[FLOAT]] h = [[234.234, 532.32], [234.4], []];
```

Y además una matriz bien formadita con enteros se puede crear asi:

```
List[List[INT]] matriz = [
  [234, 234],
  [2341, 1234]
];
```

¿Cómo veo lo que hay en mi lista?

Para visualizar lo que hay en tu lista es tan sencillo como utilizar show, además al finalizar el libro te daremos unos códigos ejemplo de cómo utilizar KAFE MEJOR.

```
show(numeros);
show(h);
```

KAFE también incorpora funciones especiales para el manejo de listas como el append, el len y el remove acá te daremos ejemplos.

Agregar elementos con append

```
append(numeros, 99);
append(letras, "z");
append(flags, False);
append(cadenas, [["asdf"]]);
```

Después, puedes mostrarlos otra vez:

```
show(numeros);
show(letras);
show(flags);
show(cadenas);
```

Eliminar elementos con remove

```
remove(numeros, 2);   -- Elimina el número 2
remove(letras, "a");  -- Elimina la letra "a"
```

Y sí, también puedes mostrar los cambios:

```
show(numeros);
show(letras);
```

Conocer la longitud de una lista con len

```
show(len(numeros));
```

Acceder o modificar elementos por índice

Acá te mostramos un ejemplo completo de cómo se utilizan las listas en KAFE

```
List[INT] number = [1, 2, 3];
List[STR] hola = ["h", "o"];
```

```
List[List[List[BOOL]]] booleanos = [[[True, False]]];

show(booleanos);
show(number);
show(hola);

hola[0] = "f";
number[0] = 100;
booleanos[0] = [[False]];

show(booleanos);
show(number);
show(hola);
```

CONDICIONALES

En KAFE, las estructuras condicionales permiten ejecutar diferentes bloques de código según se cumplan o no ciertas condiciones. Se utilizan las palabras clave if, elif y else, seguidas de dos puntos : y un bloque indentado.

Te mostraremos un pequeño ejemplo, para que veas cómo se utilizan y cómo se estructuran correctamente:

```
INT edad = 25;

BOOL tieneLicencia = True;

INT puntos = 3;

if (edad >= 18) :

  if (tieneLicencia) :

      if (puntos < 5) :

          show("Puede conducir con normalidad");

      ; else :
```

```
            show("Puede conducir pero debe tener precaución");
        ;
    ; else :
        show("No puede conducir, no tiene licencia");
    ;
; else :
    if (edad >= 16) :
        show("Puede conducir con permiso de aprendizaje");
    ; else :
        show("Es menor de edad, no puede conducir");
    ;
;
```

Las condiciones van entre paréntesis: if (condición).
Los bloques se delimitan por indentación y deben terminar con ;.
Se permiten condicionales anidados para tomar decisiones complejas.

BUCLES EN KAFE

En KAFE, los bucles permiten repetir bloques de código mientras se cumplan ciertas condiciones. Existen dos estructuras principales:

while y el for, cada una adaptada a diferentes tipos de tareas repetitivas.

Bucle while

El bucle while evalúa una condición antes de ejecutar su bloque. Si la condición es verdadera, se ejecuta el contenido indentado hasta que la condición deje de cumplirse. Es ideal para situaciones donde no se conoce de antemano cuántas veces se repetirá el ciclo.

Sintaxis básica

```
while (condición):
        // instrucciones
    ;
```

Te mostraremos unos ejemplos para que sea más fácil de entender.

Ejemplo 1: Contador simple

En este ejemplo, una variable i se incrementa de 0 a 4, mostrando cada valor con show.

```
INT i = 0;
    while (i < 5):
        show(i);
        i = i + 1;
    ;
```

Ejemplo 2: Comparación de dos variables

Aquí se demuestra cómo el ciclo while puede manejar múltiples condiciones lógicas para decidir si continuar.

```
INT a = 0;
    INT b = 5;
    while (a < b && b > 0):
        show(a);
        a = a + 1;
    ;
```

Bucle for

El bucle for es usado para iterar sobre secuencias como listas, rangos, u otras colecciones. Su sintaxis es más compacta y legible cuando sabes cuántas veces deseas repetir una tarea.

Sintaxis básica

```
for (elemento in colección):
        // instrucciones
    ;
```

Ejemplo 1: Iterar sobre una lista de cadenas

Se recorre una lista de letras y se imprime cada una. Es una forma directa y clara de procesar secuencias.

```
List[STR] letras = ['K', 'A', 'F', 'E'];
    for (l in letras):
        show(l);
    ;
```

Ejemplo 2: Rango descendente con range

KAFE soporta la función range(inicio, fin, paso) para crear listas de números. En este caso se genera una lista descendente desde 0 hasta -9 con paso -1.

```
List[INT] a = range(0, -10, -1);
show(a);
```

Ejemplo 3: Iteración sobre un rango

Puedes iterar directamente sobre un range en el bucle for. Este es el método más común para repeticiones fijas.

```
for(i in range(0, 10)):
        show(i);
    ;
```

Ejemplo 4: Rango corto

Cuando solo necesitas un número fijo de repeticiones (por ejemplo, 4 veces), puedes usar simplemente range(4).

```
for(i in range(4)):
```

```
        show(i);

    ;
```

Estas estructuras de bucle son fundamentales para controlar el flujo de ejecución repetitiva en KAFE. Su sintaxis está inspirada en lenguajes modernos pero simplificada para mantener claridad en contextos educativos.

FUNCIONES

En KAFE, las funciones permiten encapsular lógica reutilizable. Se definen usando la palabra clave drip, seguidas del nombre de la función, sus parámetros con tipo, el operador =>, el tipo de retorno, y finalmente el cuerpo de la función.

Sintaxis Básica

```
drip nombreFuncion (param1: TIPO, param2: TIPO) => TIPO_RETORNO :

        // cuerpo de la función

        return valor;

    ;
```

Ejemplo de una función:

Este solo es un pequeño ejemplo, si vas al apartado final de algoritmos podrás ver mejor y experimentar más ejemplos.

```
INT a = 9;

INT b = 6;

drip suma (a: INT, b: INT) => INT :

  return a + b;

;

show(suma(a, b));  -- Imprime 15
```

Termina con ;.
La instrucción return se usa para devolver el resultado.
El tipo de retorno debe declararse después del operador =>.

Las funciones pueden ser llamadas desde cualquier parte del programa después de ser definidas.

FUNCIONES CURRIFICABLES

En KAFE, las funciones currificables son funciones que pueden invocarse parcialmente. Este enfoque proviene del paradigma funcional, y permite que una función que recibe múltiples parámetros pueda ser ejecutada parcialmente con uno o más argumentos, devolviendo una nueva función que espera los restantes.

Esta característica es especialmente útil cuando quieres reutilizar una función con ciertos parámetros ya definidos, creando funciones más especializadas a partir de otras más generales.

Sintaxis Básica

```
drip nombreFuncion(param1: TIPO1, param2: TIPO2) => TIPO_RETORNO:
        // cuerpo
    ;
```

La palabra clave drip indica que la función puede currificarse. Puedes invocar la función pasando todos los argumentos de una vez o parcialmente, dejando que se genere una nueva función intermedia.

Ejemplo

A continuación se define una función sumar que suma dos enteros:

```
drip sumar(a: INT, b: INT) => VOID:
        show(a + b);
    ;
```

Puedes invocarla de las siguientes formas:

```
sumar(10)(5);   // Currificación — devuelve 15
sumar(10, 5);   // Llamada directa — también devuelve 15
```

La primera llamada es una aplicación parcial: al ejecutar sumar(10) se genera una función intermedia que espera un segundo parámetro. Luego, (5) se aplica a esa función.

Asignación como Función de Orden Superior

Puedes guardar una función ya parcialmente aplicada en una variable, para luego reutilizarla múltiples veces con diferentes argumentos:

```
FUNC(INT)=>INT sumarQuince = sumar(15);
```

En este caso, sumarQuince es una función que espera un INT y devuelve un INT. Internamente, es equivalente a sumar(15)(x) para cualquier valor x.

Luego puedes hacer:

```
sumarQuince(5);    // Imprime 20

sumarQuince(100);  // Imprime 115
```

Las Ventajas son:

Reutilización de lógica: permite definir funciones base y luego especializarse con valores parciales.

Modularidad: promueve un estilo de programación más declarativo y composicional.

Expresividad: permite escribir código más claro y directo en contextos funcionales.

Las funciones currificables en KAFE combinan claridad sintáctica con poder expresivo, facilitando la creación de código reusable y elegante.

FUNCIONES DE ALTO NIVEL

En KAFE, las funciones de alto nivel (también llamadas funciones de orden superior) son aquellas que aceptan otras funciones como parámetros o que retornan funciones. Este tipo de programación proviene del paradigma funcional y es fundamental para la composición, reutilización de lógica y abstracción de patrones.

Sintaxis Básica

La estructura general de una función de alto nivel en KAFE es:

```
drip nombreFuncion(f: FUNC(TIPO_IN) => TIPO_OUT, otro_param: TIPO)
=> TIPO_RETORNO:

    // cuerpo

;
```

Aquí, f es una función que puede ser pasada como argumento. El tipo FUNC(...) declara el tipo de entrada y salida de esa función.

Ejemplo con Función Nombrada

Veamos un ejemplo con una función llamada inc que incrementa un valor entero:

```
drip aplicar(f: FUNC(INT) => INT, n: INT) => INT:

    return f(n);

;


drip inc(x: INT) => INT:

    return x + 1;

;

show(aplicar(inc, 5));   // 6
```

En este ejemplo, la función aplicar recibe otra función f y un valor n, y retorna el resultado de aplicar f(n). Como estamos pasando inc, se incrementa el valor 5.

Ejemplo con Lambda

También puedes pasar funciones anónimas (lambdas) directamente como argumento, sin necesidad de definirlas antes:

```
show(aplicar((y: INT) => y*y , 4));   // 16
```

Aquí, la función lambda se pasa directamente a aplicar, elevando al cuadrado el número 4.

Abstracción: puedes encapsular patrones de ejecución y lógica genérica sin repetir código.

Reutilización: permite separar la lógica del comportamiento, pasando distintas funciones según el contexto.

Composición funcional: facilita trabajar con funciones que se combinan o modifican.

Las funciones de alto nivel son fundamentales para escribir código más flexible, expresivo y modular. Su uso se potencia aún más al combinarse con funciones currificables y lambdas, como se muestra en los ejemplos anteriores.

FUNCIONES RECURSIVAS

En KAFE, es posible definir funciones recursivas, es decir, funciones que se llaman a sí mismas. Un ejemplo clásico de recursividad es el cálculo del factorial de un número.

¿Qué es el factorial?

El factorial de un número entero positivo n (denotado como n!) se define como:

n! = n * (n - 1) * (n - 2) * ... * 1

Por ejemplo: 5! = 5 * 4 * 3 * 2 * 1 = 120

Sintaxis de la Función en KAFE

```
drip factorial(n: INT) => INT:
    if (n <= 1):
        return 1;
    ;
    return n * factorial(n - 1);
;
```

La función factorial recibe un entero n. Si n es menor o igual a 1, retorna 1 (caso base). En caso contrario, se llama a sí misma con n - 1, construyendo la multiplicación de forma descendente.

Ejemplo de Uso

```
INT result = factorial(5);

show("Factorial de 5:");

show(result);
```

Este código muestra el resultado del cálculo de factorial(5), que es 120.

¿Cómo funciona paso a paso?

factorial(5) → 5 × factorial(4)

factorial(4) → 4 × factorial(3)

factorial(3) → 3 × factorial(2)

factorial(2) → 2 × factorial(1)

factorial(1) → 1 (caso base)

Al ir resolviendo cada llamada recursiva, el resultado final se calcula como 5 * 4 * 3 * 2 * 1 = 120.

Ventajas

Permite escribir soluciones elegantes para problemas que siguen un patrón repetitivo.

Evita el uso de bucles explícitos.

Ayuda a pensar de forma declarativa: qué hacer, no cómo hacerlo paso a paso.

Aunque la recursión es poderosa, hay que usarla con cuidado para evitar llamadas infinitas y errores de pila. Siempre asegúrate de tener un caso base bien definido.

LA CAJA DE HERRAMIENTAS DE KAFE

Aquí es donde KAFE realmente brilla. Además de las funciones primitivas que ya aprendiste, el lenguaje cuenta con potentes librerías que te permiten hacer mucho más sin complicarte la vida.

KAFE incluye 6 librerías integradas, diseñadas para facilitarte el trabajo en áreas como matemáticas, visualización, inteligencia artificial y manejo de datos.

A continuación te las presentamos:

KafeNUMK
Una librería inspirada en NumPy, ideal para trabajar con **álgebra lineal**, matrices, vectores y operaciones matemáticas avanzadas.

Plot
Basada en Matplotlib. Con esta librería puedes **crear gráficos, diagramas y visualizaciones interactivas** para entender mejor tus datos.

GESHA DEEP
Tu herramienta para el **Deep Learning**. Diseñada para modelar redes neuronales de forma sencilla pero poderosa. ¡Ideal para tus primeros pasos en inteligencia artificial!

Math
Una librería con **funciones matemáticas utilitarias**, como raíces cuadradas, funciones trigonométricas, exponenciales y más. Un clásico que no puede faltar.

Files
Con esta librería puedes **leer, escribir y manipular archivos** directamente desde el entorno de KAFE. Perfecta para proyectos que necesitan persistencia de datos.

Pardos
Especializada en el **manejo de archivos CSV**. Perfecta para quienes trabajan con hojas de cálculo, reportes o datasets tabulares.

NUMK

Numk es la librería de álgebra lineal en KAFE, y una de las más útiles cuando se trata de trabajar con números, vectores y matrices.

Con Numk, puedes hacer cosas como sumar matrices, multiplicarlas, transponerlas o incluso invertirlas… ¡y todo con pocas líneas de código!

Al igual que en todas las librerías si las quieres usar debes importarla en tu archivo:

```
import numk;
```

1. Suma de Matrices

Puedes sumar dos matrices del mismo tamaño, y Numk se encargará del resto:

```
-- Prueba: Sumar Matrices

import numk;

List[List[INT]] A = [[1, 2], [3, 4]];

List[List[INT]] B = [[5, 6], [7, 8]];


List[List[INT]] C = numk.add(A, B);

show(C);  -- Resultado: [[6, 8], [10, 12]]
```

2. Multiplicación de Matrices

Para multiplicar matrices, asegúrate de que el número de columnas de la primera sea igual al número de filas de la segunda:

```
-- Prueba: Multiplicar Matrices

import numk;

List[List[INT]] A = [[1, 2]];

List[List[INT]] B = [[2], [3]];
```

```
List[List[INT]] C = numk.mul(A, B);

show(C);  -- Resultado: [[8]]
```

3. Transpuesta de una Matriz

Numk puede intercambiar filas por columnas con la función

```
-- Prueba: Transpuesta

import numk;


List[List[INT]] A = [

  [1, 2],

  [3, 4],

  [5, 6]

];


show(numk.transpose(A));

-- Resultado: [[1, 3, 5], [2, 4, 6]]
```

4. Inversa de una Matriz

La función inv permite obtener la matriz inversa, útil en muchas aplicaciones matemáticas. Solo funciona si la matriz es cuadrada (mismo número de filas y columnas) y es invertible:

```
-- Prueba: Inversa

import numk;

List[List[INT]] A = [

  [2, 1],
```

```
  [7, 4]
];
show(numk.inv(A));
-- Resultado: [[4.0, -1.0], [-7.0, 2.0]]
```

Tipos soportados: INT y FLOAT.

Dimensiones compatibles:

Para add y mul, las matrices deben tener dimensiones compatibles (suma: mismo tamaño; multiplicación: cols A = rows B).

Para inv, la matriz debe ser cuadrada e invertible.

transpose acepta cualquier matriz filas×columnas.
Si las dimensiones no encajan, Numk lanzará un error de dimensión.

PLOT

Una imagen vale más que mil líneas de código, ¿cierto?

Con la librería Plot en KAFE puedes transformar tus datos en gráficas claras, estéticas y fáciles de interpretar.

Plot te permite crear:

- Gráficas de líneas
- Gráficas de barras
- Gráficas de pastel
- Con títulos, etiquetas, leyendas, rejillas y colores personalizados

Primero, importa la librería:

```
import plot;
```

Y luego, inicializa una nueva figura donde colocarás tu visualización:

```
plot.figure();
```

Títulos y etiquetas

Dale contexto a tu gráfico usando:

```
plot.title("Mi Gráfica");     -- Título general
plot.xlabel("Eje X");         -- Etiqueta eje horizontal
plot.ylabel("Eje Y");         -- Etiqueta eje vertical
```

Rejilla

Activa la rejilla para hacer más legibles tus gráficos:

```
plot.grid(true);
```

Colores

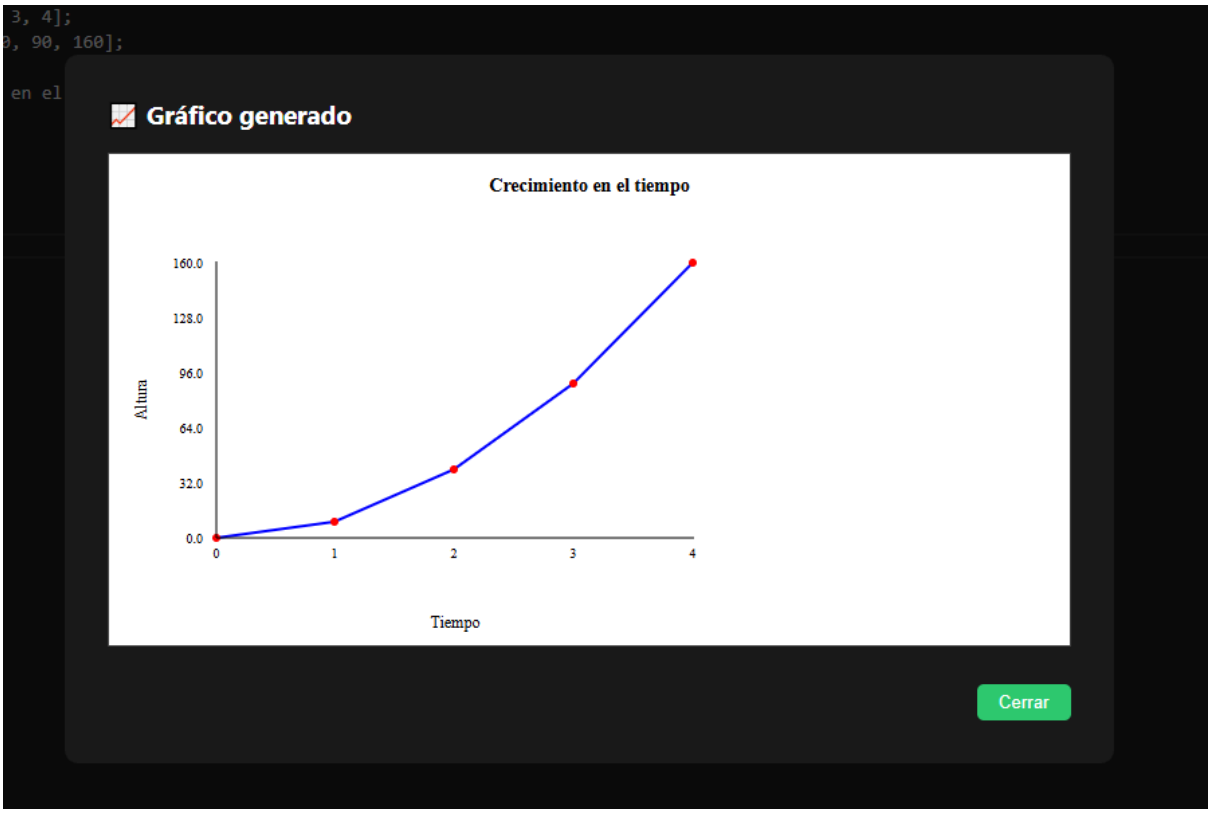
Puedes personalizar el color de cada serie o tipo de gráfico:

```
plot.color("blue");         -- Color de líneas o barras
plot.pointColor("orange");  -- Color de puntos (opcional)
```

Ejemplos de Visualización

1. Gráfica de Líneas (Line Chart)

```
List[INT] t = [0, 1, 2, 3, 4];
List[INT] h = [0, 10, 40, 90, 160];
plot.figure();
plot.title("Crecimiento en el tiempo");
plot.xlabel("Tiempo");
plot.ylabel("Altura");
plot.color("blue");
plot.graph(t, h);
plot.render();
```



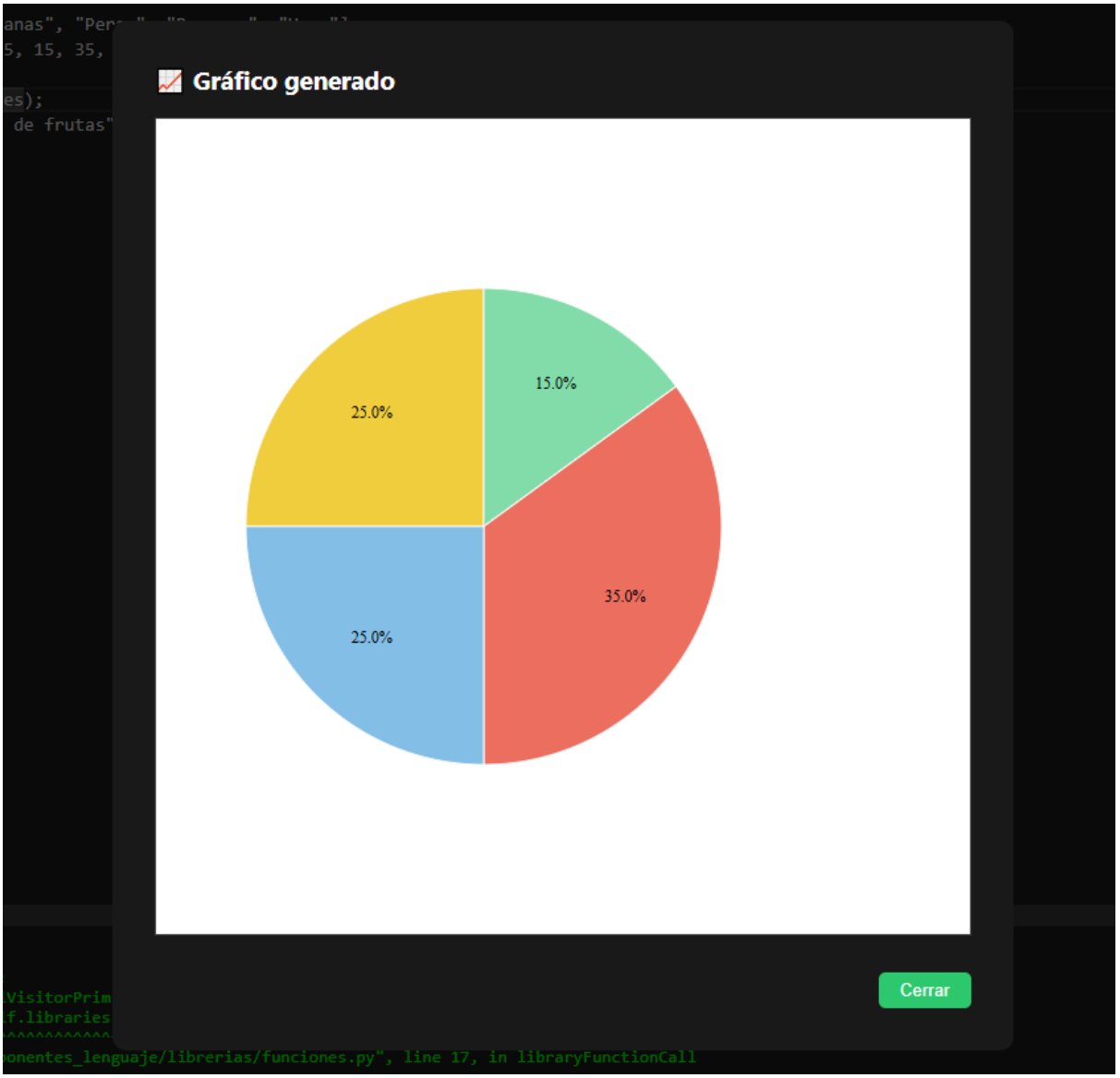
Importante, si estás trabajando desde el compilador web, y necesitas la gráfica recuerda tomarle foto antes, aun la opcion de descarga no esta disponible y si le das cerrar esta ya no la tendras.

2. Gráfica de Barras (Bar Chart)

```
List[STR] libs = ["Matplotlib", "Seaborn", "Plotly", "Plotnine"];

List[INT] users = [2500, 1800, 3000, 2200];

plot.figure();

plot.bar(libs, users);

plot.title("Bibliotecas más populares");

plot.ylabel("Usuarios");

plot.color("purple");

plot.render();
```

3. Gráfica de Pastel (Pie Chart)

```
List[STR] frutas = ["Manzanas", "Peras", "Bananas", "Uvas"];

List[INT] cantidades = [25, 15, 35, 25];

plot.figure();

plot.pie(frutas, cantidades);

plot.legend("Distribución de frutas");

plot.render();
```

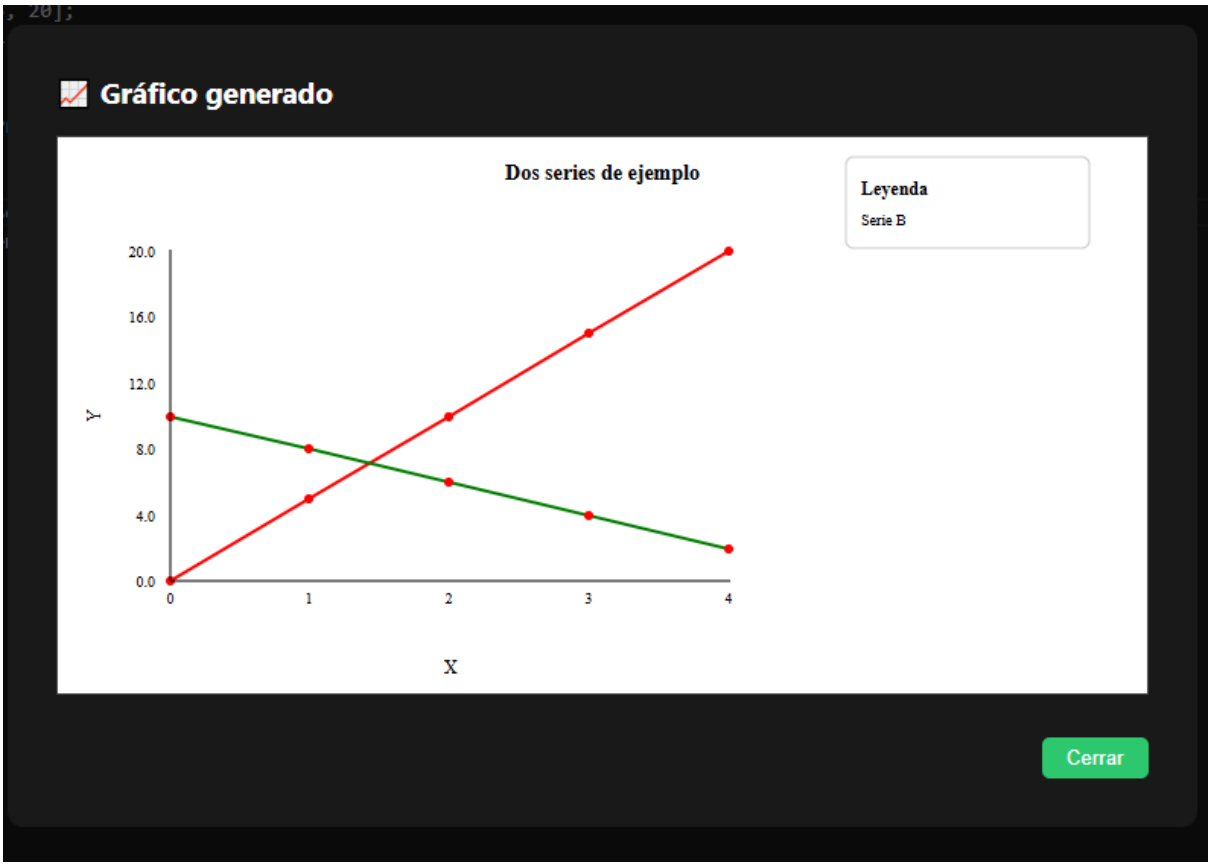


4. Múltiples Series en una Sola Gráfica

Si quieres comparar varias curvas en un mismo gráfico, puedes trazar múltiples series antes de llamar a render():

```
List[INT] x = [0, 1, 2, 3, 4];

List[INT] y1 = [0,  5, 10, 15, 20];

List[INT] y2 = [10, 8,  6,  4,  2];

plot.figure();

plot.color("red");

plot.graph(x, y1);       -- Primera serie en rojo
```

```
plot.color("green");
plot.graph(x, y2);       -- Segunda serie en verde
plot.title("Dos series de ejemplo");
plot.xlabel("X");
plot.ylabel("Y");
plot.legend("Serie A");
plot.legend("Serie B");
plot.render();
```



MATH

La librería MATH en KAFE proporciona una colección completa de funciones y constantes matemáticas, abarcando desde operaciones básicas hasta funciones avanzadas de análisis, combinatoria y precisión numérica. Puedes usarla mediante:

```
import math;
```

1. Constantes Matemáticas

Estas constantes son útiles en trigonometría, análisis y casos especiales.

```
show(math.pi);     -- π ≈ 3.14159
show(math.e);      -- e ≈ 2.71828
show(math.tau);    -- τ = 2π
show(math.inf);    -- Infinito
show(math.nan);    -- Not a Number
```

2. Funciones Trigonométricas

Todas las funciones trigonométricas trabajan en radianes.

```
show(math.sin(math.pi/2));  -- 1
show(math.cos(0));          -- 1
show(math.tan(math.pi/4));  -- ≈ 1
```

3. Funciones Trigonométricas Inversas

```
show(math.asin(1));    -- π/2
show(math.acos(1));    -- 0
show(math.atan(1));    -- π/4
```

4. Funciones Hiperbólicas

```
show(math.sinh(0));  -- 0
show(math.cosh(0));  -- 1
show(math.tanh(0));  -- 0
```

5. Exponenciales y Logaritmos

```
show(math.exp(1));        -- e^1
```

```
show(math.expm1(1));      -- e^x - 1
show(math.log(math.e));   -- log base e
show(math.log(8, 2));     -- log base 2 de 8
show(math.log2(8));       -- log2
show(math.log10(1000));   -- log10
show(math.exp2(3));       -- 2^3
show(math.cbrt(27));      -- raíz cúbica
```

6. Potencias y Raíces

```
show(math.sqrt(16));    -- 4
show(math.pow(2, 8));   -- 256
```

7. Combinatoria

```
show(math.factorial(5));  -- 120
show(math.comb(5, 2));    -- 10 (combinaciones)
show(math.perm(5, 2));    -- 20 (permutaciones)
```

8. Funciones Número-Teóricas

```
show(math.gcd(48, 18));             -- MCD
show(math.gcd(48, 18, 30));         -- MCD de 3 números
show(math.lcm(12, 15));             -- mcm
show(math.lcm(12, 15, 20));         -- mcm de 3 números
```

9. Aritmética de Punto Flotante

```
show(math.abs(-42));                -- Valor absoluto
show(math.trunc(3.7));              -- Truncado
show(math.fmod(7.3, 2.5));          -- Resto flotante
show(math.remainder(7.3, 2.5));     -- Resto con redondeo
show(math.copysign(3, -0.0));       -- Copia el signo
show(math.isclose(1.0000001, 1));   -- ¿Cercanos?
show(math.isfinite(math.inf));      -- ¿Finito?
show(math.isinf(math.inf));         -- ¿Infinito?
show(math.isnan(math.nan));         -- ¿NaN?
show(math.ulp(1.0));            -- Unidad menor representable
```

10. Sumas y Productos

```
show(math.sum(1, 5));              -- Suma del 1 al 5
show(math.sum(range(6)));         -- Suma de [0..5]
show(math.prod(1, 5));            -- Producto 1*2*3*4*5
show(math.sumprod([1,2,3], [4,5,6]));  -- 1*4 + 2*5 + 3*6
```

11. Distancias y Precisión

```
show(math.dist([0, 0], [3, 4]));    -- Distancia Euclidiana
show(math.fsum([0.1]*10));          -- Suma precisa
show(math.hypot(3, 4));             -- Hipotenusa
```

Notas

- Las funciones trigonométricas usan **radianes**.
- `log(x, base)` permite especificar la base del logaritmo.
- `gcd`, `lcm`, `factorial`, `comb`, y `perm` esperan **valores enteros**.
- `isnan`, `isinf` e `isfinite` son útiles para detección de errores en cálculos.
- `fsum` y `hypot` dan mayor precisión en operaciones con decimales.

FILES

Es la librería estándar de manejo de archivos de texto en KAFE. Permite realizar operaciones esenciales como crear, escribir, leer, anexar y eliminar archivos mediante una interfaz sencilla y directa.

Antes de usar cualquier función de esta librería, debes importarla:

```
import files;
```

Crear un archivo

Crea un archivo de texto vacío si no existe. Si el archivo ya existe, no lo sobrescribe.

```
files.create("mi_archivo.txt");
```

Escribir en un archivo

Escribe contenido en el archivo. Si el archivo ya existe, su contenido será sobrescrito completamente.

```
files.write("mi_archivo.txt", "Hola desde KAFE");
```

Leer un archivo

Lee todo el contenido del archivo y lo devuelve como STR.

```
STR texto = files.read("mi_archivo.txt");
show(texto);
```

Anexar contenido

Agrega texto al final del archivo sin sobrescribir el contenido existente.

```
files.append("mi_archivo.txt", "
Más texto en la siguiente línea");
```

Eliminar un archivo

Borra el archivo especificado del sistema de archivos.

```
files.delete("mi_archivo.txt");
```

Ejemplo completo

```
import files;

-- Crear y escribir en un archivo
files.create("notas.txt");
files.write("notas.txt", "Primera línea");

-- Anexar otra línea
files.append("notas.txt", "
Segunda línea");

-- Leer el contenido
STR contenido = files.read("notas.txt");
show(contenido);  -- Muestra ambas líneas

-- Eliminar el archivo
```

```
files.delete("notas.txt");
```

- Rutas relativas: Todos los archivos se gestionan en relación con el directorio actual de ejecución.
- `files.write(...)` sobrescribe todo el contenido previo. Si deseas conservar el contenido y agregar más, usa `files.append(...)`.
- `files.read(...)` devuelve un único STR con el contenido completo del archivo.
- Si intentas leer un archivo que no existe, se lanzará un error de ejecución.
- El tipo STR en KAFE equivale a una cadena de texto.

GESHADEEP

GeshaDeep es la librería de deep learning de KAFE. Soporta modelos de clasificación binaria, multiclase, regresión lineal y clustering, con una API unificada basada en "modelos" y "capas".

```
import geshaDeep;
```

1. Clasificación AND (binaria)

Este ejemplo entrena una red de una sola neurona con activación sigmoidal para resolver la función lógica AND. Tras el entrenamiento, predice la probabilidad y la etiqueta (0 o 1) para cada combinación de entrada.

```
-- Dataset AND
      List[List[INT]] x_train = [
         [0, 0],
         [0, 1],
```

```
    [1, 0],
    [1, 1]
];
List[INT] y_train = [0, 0, 0, 1];

GESHA model = geshaDeep.binary();
GESHA layer = geshaDeep.create_dense(1, "sigmoid", [2], 0.0, 42);
model.add(layer);

model.compile("sgd", "binary_crossentropy", ["accuracy"]);
model.fit(x_train, y_train, 1000, 1, [], []);

-- Probamos los 4 puntos
for (p in x_train):
    FLOAT prob = model.predict_proba(p);  -- probabilidad clase 1
    INT   lbl  = model.predict_label(p);  -- etiqueta 0/1
    show(str(p) + " -> prob=" + str(prob) + ", label=" + str(lbl));
;
```

2. Clasificación OR (multiclase)

Aquí usamos una red con capa oculta ReLU y capa de salida softmax para la función OR, codificada one-hot. Se incluye también un pequeño conjunto de validación y al final obtenemos las probabilidades para un nuevo ejemplo y el resumen del modelo.

```
-- classification_predict.kf
    import geshaDeep;

    -- OR lógico:  class0=[1,0], class1=[0,1]
    List[List[INT]] x_train = [
        [0,0],
        [0,1],
        [1,0],
        [1,1]
    ];
    List[List[INT]] y_train = [
        [1,0],
        [0,1],
        [0,1],
        [0,1]
```

```
];

-- pequeña validación
List[List[INT]] x_val = [[0,0],[1,1]];
List[List[INT]] y_val = [[1,0],[0,1]];

GESHA model = geshaDeep.categorical();
GESHA h  = geshaDeep.create_dense(4, "relu",  [2], 0.0, 42);
GESHA out = geshaDeep.create_dense(2, "softmax", [4], 0.0, 42);
model.add(h);
model.add(out);

model.compile("sgd", "categorical_crossentropy", ["accuracy"]);
model.fit(x_train, y_train, 100, 2, x_val, y_val);

-- predicción para [1,0]  (esperamos class1)
List[INT] sample = [1,0];
List[FLOAT] probs = model.predict(sample);
show("Probabilidades: " + str(probs));

model.summary();
```

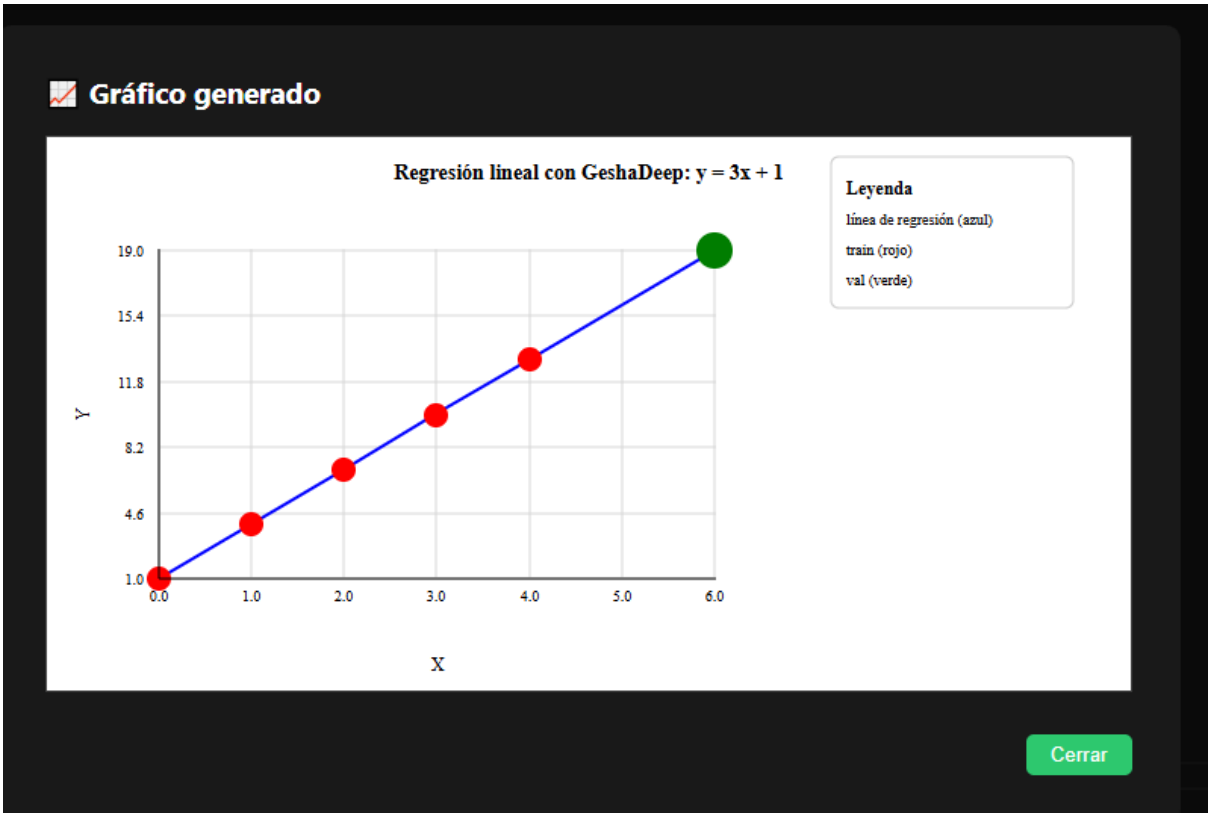
3. Regresión Lineal

Este ejemplo ajusta una regresión lineal simple con una única neurona de activación lineal. Tras entrenar con MSE, predice un nuevo valor y usa Plot para mostrar la línea de regresión junto a los puntos de entrenamiento y validación.

```
import geshaDeep;
import plot;

List[List[INT]] x_train = [[0], [1], [2], [3], [4]];
List[FLOAT]     y_train = [1.0,  4.0,  7.0, 10.0, 13.0];

List[List[INT]] x_val   = [[6]];
List[FLOAT]     y_val   = [19.0];

GESHA model = geshaDeep.regression();

GESHA dense = geshaDeep.create_dense(1, "linear", [1], 0.0, 42);
```

```
model.add(dense);

geshaDeep.compile(model, "sgd", "mse", []);

model.fit(x_train, y_train, 200, 1, x_val, y_val);

List[INT] sample = [6];
FLOAT y_pred = model.predict(sample)[0];
show("Predicción para 6: " + str(y_pred));

model.summary();

List[List[FLOAT]] train_points = [];
for (i in range(0, len(x_train))):
    FLOAT xi = float(x_train[i][0]);
    FLOAT yi = y_train[i];
    append(train_points, [xi, yi]);
;

List[List[FLOAT]] val_points = [];
for (i in range(0, len(x_val))):
    FLOAT xv = float(x_val[i][0]);
    FLOAT yv = y_val[i];
    append(val_points, [xv, yv]);
;

List[List[FLOAT]] line_points = [];
for (i in range(0, 7)):
    List[INT] xi_int = [i];
    FLOAT yi_pred = model.predict(xi_int)[0];
    append(line_points, [float(i), yi_pred]);
;

plot.figure();
plot.title("Regresión lineal con GeshaDeep: y = 3x + 1");
plot.xlabel("X");
plot.ylabel("Y");
plot.grid(True);

if (len(line_points) > 0):
    plot.pointColor("blue");
```

```
    plot.pointSize(2);
    plot.graph(line_points, "line");
;

if (len(train_points) > 0):
    plot.pointColor("red");
    plot.pointSize(8);
    plot.graph(train_points, "point");
;

if (len(val_points) > 0):
    plot.pointColor("green");
    plot.pointSize(12);
    plot.graph(val_points, "point");
;

plot.legend("línea de regresión (azul) ; train (rojo) ; val (verde)");
plot.render();
```



4. Clustering

El modelo de clustering utiliza la función k-means. Tras añadir dos capas densas y compilar con “adam”, el modelo ajusta los datos sin etiquetas. Luego, asigna cada punto a un clúster, calcula sus centroides y visualiza resultados con Plot.

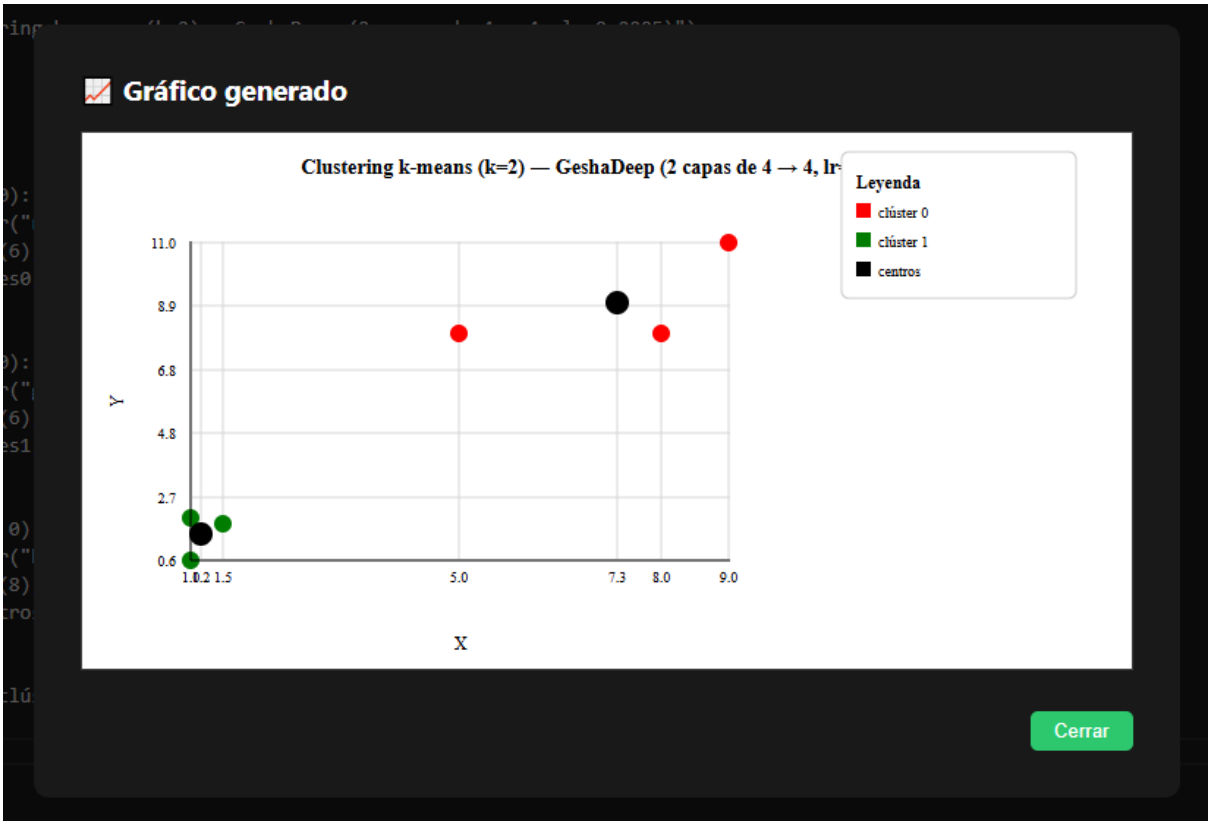
```
import geshaDeep;
	import plot;


	List[List[FLOAT]] datos = [
	   [1.0, 2.0],
	   [1.5, 1.8],
	   [5.0, 8.0],
	   [8.0, 8.0],
	   [1.0, 0.6],
	   [9.0, 11.0]
	];

	List[INT] y_dummy = [];


	GESHA modelo = geshaDeep.clustering();


	GESHA capa1 = geshaDeep.create_dense(
	   8,        -- 4 neuronas en primera capa oculta
	   "relu",
	   [2],      -- entrada 2D
	   0.0,
	   12        -- semilla distinta
	);
	modelo.add(capa1);


	GESHA capa1b = geshaDeep.create_dense(
	   4,
	   "relu",
	   [],
	   0.0,
	   99
```

```
    );
    modelo.add(capa1b);


    GESHA capa2 = geshaDeep.create_dense(
        2, "softmax", [], 0.0, 42
    );
    modelo.add(capa2);


        geshaDeep.compile(modelo, "adam", "categorical_crossentropy",
[]);

    geshaDeep.set_lr(modelo, 0.0005);


    modelo.fit(datos, y_dummy, 300, 2);


    List[FLOAT] X0 = [];
    List[FLOAT] Y0 = [];
    List[FLOAT] X1 = [];
    List[FLOAT] Y1 = [];

    List[INT] indices = [0,1,2,3,4,5];
    for (i in indices):
        List[FLOAT] punto = datos[i];
        List[FLOAT] probs = modelo.predict(punto);
        show(probs);  -- imprimo probabilidades para ver evolución

        FLOAT p0 = probs[0];
        FLOAT p1 = probs[1];
        INT etiqueta;
        if (p0 >= p1):
            etiqueta = 0;
        else:
            etiqueta = 1;
        ;

        FLOAT xi = punto[0];
        FLOAT yi = punto[1];
```

```
    if (etiqueta == 0):
        append(X0, xi);
        append(Y0, yi);
    else:
        append(X1, xi);
        append(Y1, yi);
    ;
;


FLOAT sumx0 = 0.0; FLOAT sumy0 = 0.0; INT count0 = 0;
FLOAT sumx1 = 0.0; FLOAT sumy1 = 0.0; INT count1 = 0;

for (i in indices):
    List[FLOAT] punto = datos[i];
    List[FLOAT] probs = modelo.predict(punto);
    FLOAT p0 = probs[0];
    FLOAT p1 = probs[1];
    INT etiqueta;
    if (p0 >= p1):
        etiqueta = 0;
    else:
        etiqueta = 1;
    ;
    FLOAT xi = punto[0];
    FLOAT yi = punto[1];
    if (etiqueta == 0):
        sumx0 = sumx0 + xi;
        sumy0 = sumy0 + yi;
        count0 = count0 + 1;
    else:
        sumx1 = sumx1 + xi;
        sumy1 = sumy1 + yi;
        count1 = count1 + 1;
    ;
;

FLOAT cx0;
if (count0 == 0):
    cx0 = 0.0;
else:
```

```
    cx0 = sumx0 / count0;
;

FLOAT cy0;
if (count0 == 0):
    cy0 = 0.0;
else:
    cy0 = sumy0 / count0;
;

FLOAT cx1;
if (count1 == 0):
    cx1 = 0.0;
else:
    cx1 = sumx1 / count1;
;

FLOAT cy1;
if (count1 == 0):
    cy1 = 0.0;
else:
    cy1 = sumy1 / count1;
;

List[FLOAT] CX = [cx0, cx1];
List[FLOAT] CY = [cy0, cy1];


List[List[FLOAT]] pares0 = [];
for (i in range(0, len(X0))):
    append(pares0, [X0[i], Y0[i]]);
;

List[List[FLOAT]] pares1 = [];
for (i in range(0, len(X1))):
    append(pares1, [X1[i], Y1[i]]);
;

List[List[FLOAT]] centros = [];
for (i in range(0, len(CX))):
    append(centros, [CX[i], CY[i]]);
```

```
        ;


        plot.figure();
         plot.title("Clustering k-means (k=2) — GeshaDeep (2 capas de 4 →
4, lr=0.0005)");
        plot.xlabel("X");
        plot.ylabel("Y");
        plot.grid(True);


        if (len(pares0) > 0):
            plot.pointColor("red");
            plot.pointSize(6);
            plot.graph(pares0, "point");
        ;

        if (len(pares1) > 0):
            plot.pointColor("green");
            plot.pointSize(6);
            plot.graph(pares1, "point");
        ;

        if (len(centros) > 0):
            plot.pointColor("black");
            plot.pointSize(8);
            plot.graph(centros, "point");
        ;

        plot.legend("red: clúster 0 ; green: clúster 1 ; black: centros");
        plot.render();
```



Para clasificación usa binary() (sigmoid) o categorical() (softmax).

En regresión lineal, utiliza regression() y capa “linear” con pérdida MSE.

Añade capas con create_dense antes de compile y fit.

Usa Plot para visualizar aprendizaje y resultados de forma rápida.

PARDOS

Esta aún no tiene soporte en nuestra API web sin embargo en el repositorio podrás encontrar tests de como probar localmente.

JUGUEMOS UN RATO , ALGORITMOS BÁSICOS EN KAFE

Binary search

```
drip binarySearch(lst: List[INT], target: INT) => INT:
    INT left = 0;
    INT right = len(lst) - 1;

    while (left <= right):
        INT mid = int((left + right) / 2);

        if (lst[mid] == target):
            return mid;
        ;
        if (lst[mid] < target):
            left = mid + 1;
        else:
            right = mid - 1;
        ;
    ;

    return -1;
;


List[INT] data = [5, 10, 15, 20, 25, 30, 35];
INT index = binarySearch(data, 25);
show("Índice encontrado:");
show(index);
```

BubbleSort

```
drip bubbleSort(arr: List[INT]) => List[INT]:
    INT n = len(arr);
    INT i = 0;
```

```
    while (i < n):
        INT j = 0;
        while (j < n - i - 1):
            if (arr[j] > arr[j + 1]):
                INT temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            ;
            j = j + 1;
        ;
        i = i + 1;
    ;
    return(arr);
;

List[INT] miLista = [5, 4324, 8, 4, 2];
List[INT] ordenada = bubbleSort(miLista);
show(ordenada);
```

Fibonacci

```
drip fibonacci(n: INT) => INT:
    if (n <= 1):
        return n;
    ;
    return fibonacci(n - 1) + fibonacci(n - 2);
;

INT result = fibonacci(7);
show("Fibonacci de 7:");
show(result);
```

Lineal Search

```
drip linearSearch(lst: List[INT], target: INT) => INT:
    INT i = 0;
```

```
    while (i < len(lst)):
        if (lst[i] == target):
            return i;
        ;
        i = i + 1;
    ;
    return -1;
;


List[INT] data = [10, 20, 30, 40, 50];
INT result = linearSearch(data, 30);
show("Índice encontrado:");
show(result);
```

MERGE SORT

```
drip mergeSort(lst: List[INT]) => List[INT]:
    List[INT] sortedList = [];
    List[INT] aux = [];
    INT width;
    INT n_len;

    INT i = 0;
    while (i < len(lst)):
        append(sortedList, lst[i]);
        append(aux, lst[i]);
        i = i + 1;
    ;

    width = 1;
    n_len = len(sortedList);

    while (width < n_len):
        INT i = 0;

        while (i < n_len):
            INT left = i;
            INT mid = left + width;
```

```
    INT right = left + 2 * width;

    if (mid > n_len):
       mid = n_len;
    ;

    if (right > n_len):
       right = n_len;
    ;

    INT l = left;
    INT r = mid;
    INT m = left;

    while (l < mid && r < right):
       if (sortedList[l] <= sortedList[r]):
          aux[m] = sortedList[l];
          l = l + 1;
       else:
          aux[m] = sortedList[r];
          r = r + 1;
       ;
       m = m + 1;
    ;

    while (l < mid):
       aux[m] = sortedList[l];
       l = l + 1;
       m = m + 1;
    ;

    while (r < right):
       aux[m] = sortedList[r];
       r = r + 1;
       m = m + 1;
    ;

    i = i + 2 * width;
;
```

```
        i = 0;
        while (i < n_len):
            sortedList[i] = aux[i];
            i = i + 1;
        ;

        width = width * 2;
    ;

    return sortedList;
;

List[INT] unsorted = [3, 400, 32432, 46, 223, 9];
List[INT] sorted = mergeSort(unsorted);
show("Lista ordenada con Merge Sort:");
show(sorted);
```

RECURSIÓN

```
drip factorial(n: INT) => INT:
    if (n <= 1):
        return 1;
    ;
    return n * factorial(n - 1);
;

INT result = factorial(5);
show("Factorial de 5:");
show(result);
```

INSERTION SORT

```
List[INT] lst = [3, 400, 32432, 46, 223, 9];

drip insertionSort(lst: List[INT]) => List[INT]:
        List[INT] sortedList;

        for (i in lst):
```

```
            append(sortedList, i);
        ;

        INT i = 1;
        while (i < len(sortedList)):
            INT key = sortedList[i];

            INT j = i - 1;
            while (j >= 0 && sortedList[j] > key):
                sortedList[j + 1] = sortedList[j];
                j = j - 1;
            ;

            sortedList[j + 1] = key;
            i = i + 1;
        ;

        return sortedList;
;

show("Lista ordenada:");
show(insertionSort(lst));
```

QUICKSORT

```
List[INT] less;
List[INT] greater;
List[INT] leftSorted;
List[INT] rightSorted;
List[INT] result;
INT i;
INT j;

drip quicksort(lst: List[INT]) => List[INT]:
    if (len(lst) <= 1):
        return lst;
    ;
    INT pivot = lst[0];
    less = [];
```

```
    greater = [];

    i = 1;
    while (i < len(lst)):
        if (lst[i] < pivot):
            append(less, lst[i]);
        else:
            append(greater, lst[i]);
        ;
        i = i + 1;
    ;

    leftSorted = quicksort(less);
    rightSorted = quicksort(greater);
    result = [];

    j = 0;
    while (j < len(leftSorted)):
        append(result, leftSorted[j]);
        j = j + 1;
    ;

    append(result, pivot);

    j = 0;
    while (j < len(rightSorted)):
        append(result, rightSorted[j]);
        j = j + 1;
    ;

    return result;
;


List[INT] unsorted = [3, 400, 32432, 46, 223, 9];
List[INT] sorted = quicksort(unsorted);
show("Lista ordenada:");
show(sorted);
```

Pruebalos y rectifica que te funcione, ahora puedes programar o experimentar con KAFE, inventa, crea y renueva.

A nuestros mentores, docentes y colegas: gracias por los errores compartidos, las largas conversaciones técnicas, y por recordarnos que en la computación —como en la vida— compilar no es lo mismo que ejecutar.