Proyecto2

Laboratorio de Inteligencia Artificial

Nombre	Carnet
Edgardo Andrés Nil Guzmán	201801119

Manual Tecnico

Estructura HTML

1. Título del Proyecto

```
<h1>Proyecto 2, 201801119, Modelos de Inteligencia Artificial</h1>
```

- Proporciona una introducción clara al propósito del proyecto.

2. Selector de Modelo

```
<section id="selector-container">
    <label for="model-selector">Seleccionar Modelo:</label>
    <select id="model-selector">...</select>
</section>
```

- Permite al usuario seleccionar el modelo de inteligencia artificial que desea utilizar.

3. Sección de Regresión Polinomial

- Contiene controles para cargar archivos, especificar el grado del polinomio y botones para realizar acciones.
- Presenta resultados de la adaptación del modelo, predicciones y coeficiente R^2.

```
<section id="polynomial-regression" class="model-section">...</section>
```

4. Sección de Árbol de Decisión ID3

- Similar a la sección de regresión polinomial, permite cargar datos y predecir usando un árbol de decisión.

```
<section id="decision-tree" class="model-section">...</section>
```

5. Sección de Clustering K-Means

- Proporciona opciones para cargar archivos y ejecutar el algoritmo de K-means.

```
<section id="kmeans-section" class="model-section">...</section>
```

6. Scripts:

- Incluye referencias a bibliotecas externas y scripts que gestionan la lógica y el comportamiento del modelo.

```
<script src="..."></script>
```

Estructura TYTUSJS

Arbol de Decisiones

1. Clase `NodeTree`

```
class NodeTree {
    constructor(_value = null, _tag="", _childs = []){
        this.id = Math.random().toString(15).substr(3,12);
        this.tag = _tag;
        this.value = _value;
        this.childs = _childs;
    }
}
```

Descripción: Esta clase representa un nodo en el árbol de decisión. Cada nodo tiene un ID único, una etiqueta (`tag`), un valor (`value`), y una lista de nodos hijos (`childs`). El ID se genera aleatoriamente.

2. Clase `Feature`

```
class Feature {
    constructor(_attribute, _primaryPosibility, _secondPosibility){
        this.attribute = _attribute;
        this.entropy = -1;
```

```
        this.gain = -1;
        this.primaryCount = 0;
        this.secondaryCount = 0;
        this.primaryPosibility = _primaryPosibility;
        this.secondPosibility = _secondPosibility;
    }

    updateFeature(_posibility){
        if(_posibility === this.primaryPosibility){
            this.primaryCount += 1;
        }else if(_posibility === this.secondPosibility) {
            this.secondaryCount += 1;
        }else {
            return false;
        }
        this.entropy = this.calculateEntropy(this.primaryCount,
this.secondaryCount);
        return true;
    }

    calculateEntropy(_p, _n) {
        let entropy = -1;
        if(_p == 0 || _n == 0){
            return 0;
        }
        entropy = -(_p/(_p+_n))*Math.log2(_p/(_p+_n))
        entropy += -(_n/(_p+_n))*Math.log2(_n/(_p+_n))
        return entropy;
    }
}
```

Descripción: Esta clase representa una característica (o atributo) del conjunto de datos. Mantiene información sobre el atributo, las cuentas de las posibilidades primarias y secundarias, así como la entropía y la ganancia. La función `updateFeature` actualiza la cuenta basada en la posibilidad observada y recalcula la entropía usando `calculateEntropy`.

3. Clase `Attribute`

```
class Attribute {
    constructor(_attribute){
        this.attribute = _attribute;
        this.features = [];
        this.infoEntropy = -1;
        this.gain = -1;
        this.index = -1;
    }
}
```

Descripción: Representa un atributo en el conjunto de datos. Almacena el nombre del atributo, una lista de características (`features`), la entropía informativa y la ganancia asociada, así como el índice del atributo en el

conjunto de datos.

4. Clase `DecisionTreeID3`

4.1 Constructor

```
class DecisionTreeID3 {
    constructor(_dataSet = []) {
        this.dataset = _dataSet;
        this.generalEntropy = -1;
        this.primaryCount = -1;
        this.secondaryCount = -1;
        this.primaryPosibility = "";
        this.secondPosibility = "";
        this.root = null;
    }
}
```

Descripción: Esta clase es la principal para la creación del árbol de decisión. El constructor inicializa el conjunto de datos y variables para la entropía general y las cuentas de las posibilidades.

4.2 Método `train`

```
train(_dataset, _start = 0){
    let columnResult = _dataset[0].length - 1;
    this.calculateGeneralEntropy(_dataset, columnResult);

    let numberAttributes = _dataset[0].length;
    let gainAttribute = [];
    for (let i = _start; i < numberAttributes; i++){
        if(i === columnResult) continue;
        let attribute = new Attribute(_dataset[0][i]);
        attribute.index = i;
        attribute.features = this.classifierFeatures(_dataset, i, columnResult);
        attribute.infoEntropy =
this.calculateInformationEntropy(attribute.features);
        attribute.gain = this.calculateGain(this.generalEntropy,
attribute.infoEntropy);
        gainAttribute.push(attribute);
    }
    // ... Resto del código
}
```

Descripción: Este método entrena el árbol de decisión usando el conjunto de datos. Calcula la entropía general y evalúa cada atributo para determinar su ganancia. La lista `gainAttribute` almacena los atributos junto con su ganancia. Si no hay atributos, el método retorna `null`.

4.3 Método `calculateGeneralEntropy`

```
calculateGeneralEntropy(_dataset, indexResult){
    let att1 = { tag: "", count: 0 };
    let att2 = { tag: "", count: 0 };
    let header = false;
    _dataset.map(f => {
        if (header){
            if(!att1.tag){
                att1.tag = f[indexResult];
                att1.count += 1;
            } else if(!att2.tag && f[indexResult] != att1.tag) {
                att2.tag = f[indexResult];
                att2.count += 1;
            } else if(att1.tag === f[indexResult]){
                att1.count += 1;
            } else if(att2.tag === f[indexResult]){
                att2.count += 1;
            }
        } else {
            header = true;
        }
    });
    this.primaryPosibility = att1.tag;
    this.secondPosibility = att2.tag;
    this.primaryCount = att1.count;
    this.secondaryCount = att2.count;
    this.generalEntropy = this.calculateEntropy(att1.count, att2.count);
    return this.generalEntropy;
}
```

Descripción: Calcula la entropía general del conjunto de datos al contar las instancias de cada posibilidad en la columna de resultados. Actualiza las variables para las posibilidades primarias y secundarias y calcula la entropía usando el método `calculateEntropy`.

4.4 Método `predict`

```
predict(_predict, _root){
    return this.recursivePredict(_predict, _root);
}
```

Descripción: Este método realiza una predicción sobre una nueva instancia utilizando el árbol entrenado. Llama a `recursivePredict`, que navega por el árbol en función de los atributos de la entrada.

4.5 Método `generateDotString`

```
generateDotString(_root){
    let dotStr = "{";
    dotStr += this.recursiveDotString(_root);
```

```
    dotStr += "}";
    return dotStr;
}
```

Descripción: Genera una representación en formato DOT del árbol de decisión, que se puede utilizar para visualizar el árbol utilizando herramientas como Graphviz.

Kmeans

(`LinearKMeans`) y una versión bidimensional (`_2DKMeans`). Cada sección del código va acompañada de una descripción de su funcionalidad.

Clase: `KMeans`

```
class KMeans {
    constructor() {
        this.k = 3
    }
}
```

- **Descripción**: Esta es la clase base para el algoritmo de agrupamiento K-Means. Inicializa el número de clústeres `k` en 3. Otras clases pueden extender esta clase base para implementar funcionalidades específicas de K-Means.

Clase: `LinearKMeans`

```
class LinearKMeans extends KMeans {
    constructor() {
        super()
        this.data = []
    }
```

- **Descripción**: Esta clase extiende `KMeans` para manejar datos unidimensionales. Inicializa un array vacío `data` para almacenar los puntos de datos de entrada.

Método: `clusterize`

```
clusterize(k, data, iterations) {
    let clusters_proposals = []
    this.data = data
```

- **Descripción**: Este método agrupa los datos en `k` grupos durante un número especificado de `iteraciones`. Inicializa un array `clusters_proposals` para almacenar los resultados de agrupamiento potenciales.

Proceso de Agrupamiento

- Selección aleatoria de centros de clúster iniciales:

```
for (let i = 0; i < k; i++) {
    let c = data[Math.floor(Math.random() * data.length)]
    while (clusters.findIndex(x => x === c) != -1) {
        c = data[Math.floor(Math.random() * data.length)]
    }
    clusters.push(c)
}
```

- **Descripción**: Este bloque de código selecciona aleatoriamente `k` puntos del conjunto de datos para servir como centros iniciales de los clústeres. Se asegura de que los centros seleccionados sean únicos.

Cálculo de Distancias

```
data.forEach(point => {
    closer_cluster = 0, closer_distance = 0, first_distance = true;

    clusters.forEach(c => {
        let distance = this.distance(point, c)
        if (first_distance) {
            closer_distance = Math.abs(distance)
            closer_cluster = c
            first_distance = !first_distance
        } else {
            if (Math.abs(distance) < closer_distance) {
                closer_distance = Math.abs(distance)
                closer_cluster = c
            }
        }
        distances.push([point, c, distance])
    })
    clustered_data.push([point, closer_cluster, closer_distance])
});
```

- **Descripción**: Para cada punto de datos, se calcula la distancia a cada centro de clúster. Se determina el clúster más cercano y se almacena en `clustered_data`.

Cálculo de Medias y Varianza

```
clusters.forEach((c, i) => {
    let data = clustered_data.filter((cd) => cd[1] == c)
    let data_points = data.map((dp) => dp[0])
    cluster_stats.push(this.calculateMeanVariance(data_points))
    total_variance += data_points[1]
    clusters[i] = cluster_stats[i][0]
});
```

- **Descripción**: Para cada clúster, se calcula la media y la varianza de los puntos de datos que le pertenecen. Estos valores se utilizan para actualizar la posición del centro del clúster.

Clase: `_2DKMeans`

```
class _2DKMeans extends KMeans {
    constructor() {
        super()
        this.data = []
    }
```

- **Descripción**: Esta clase extiende `KMeans` para manejar datos bidimensionales. Al igual que en la clase anterior, se inicializa un array vacío `data`.

Método: `clusterize` (2D)

```
clusterize(k, data, iterations) {
    let clusters_proposals = []
    this.data = data
```

- **Descripción**: Al igual que en `LinearKMeans`, este método agrupa los datos bidimensionales en `k` grupos.

Cálculo de Distancias en 2D

```
distance(point_a, point_b) {
    let x1 = point_a[0]
    let y1 = point_a[1]
    let x2 = point_b[0]
    let y2 = point_b[1]

    return Math.sqrt(Math.pow((x2 - x1), 2) + Math.pow((y2 - y1), 2))
}
```

- **Descripción**: Este método calcula la distancia euclidiana entre dos puntos en un espacio bidimensional.

Cálculo de Media y Varianza

```
calculateMeanVariance(arr) {
    // Código para calcular media y varianza
}
```

- **Descripción**: Este método calcula la media y la varianza de un array de números, que es útil para analizar la distribución de los puntos en cada clúster.

PolynomialRegression

Clase: `PolynomialModel`

```
class PolynomialModel {
    // Modelo polinómico que será heredado por Polynomial Regression
    constructor() {
        this.isFit = false;
    }
}
```

- **Descripción**: Esta clase base representa un modelo polinómico. Tiene un atributo `isFit` que indica si el modelo ha sido ajustado (entrenado) o no. Inicialmente, este atributo se establece en `false`.

Clase: `PolynomialRegression`

```
class PolynomialRegression extends PolynomialModel {
    // Código de PolynomialRegression
    constructor() {
        super();
        this.solutions = [];
        this.error = 0;
    }
```

- **Descripción**: Esta clase extiende `PolynomialModel` para implementar la regresión polinómica. En el constructor, se inicializan un array `solutions` para almacenar los coeficientes del polinomio y un valor `error` para evaluar la calidad del ajuste.

Método: `fit`

```
fit(xArray, yArray, degree) {
    // Tamaño de la matriz de ecuaciones basado en el grado y el número de
elementos
```

```
    let equationSize = degree + 1;
    let nElements = degree + 2;

    // Matriz de ecuaciones a resolver
    let equations = new Array(equationSize);
    for (let i = 0; i < equationSize; i++) {
        equations[i] = new Array(nElements);
    }
```

- **Descripción**: Este método entrena el modelo creando la matriz de ecuaciones necesaria para calcular los coeficientes de la regresión polinómica. Se define el tamaño de la matriz en función del grado del polinomio.

Construcción de la Matriz de Ecuaciones

```
for (let i = 0; i < equationSize; i++) {
    for (let j = 0; j < nElements; j++) {
        let sum = 0;
        if (i == 0 && j == 0) {
            sum = xArray.length;
        }
        else if (j == nElements - 1) {
            for (let k = 0; k < xArray.length; k++) {
                sum += Math.pow(xArray[k], i) * yArray[k];
            }
        }
        else {
            for (let k = 0; k < xArray.length; k++) {
                sum += Math.pow(xArray[k], (j + i));
            }
        }
        equations[i][j] = sum;
    }
}
```

- **Descripción**: Este bloque de código construye la matriz de ecuaciones que se utilizará para resolver el sistema lineal. Se calcula la suma de potencias de los valores de `x` y la suma de los productos de `y` por las potencias correspondientes.

Eliminación de Gauss

```
for (let i = 1; i < equationSize; i++) {
    for (let j = 0; j <= i - 1; j++) {
        let factor = equations[i][j] / equations[j][j];
        for (let k = j; k < nElements; k++) {
            equations[i][k] -= factor * equations[j][k];
        }
```

```
    }
}
```

- **Descripción**: Se aplica la eliminación de Gauss para transformar la matriz de ecuaciones en una forma escalonada. Esto facilita la resolución del sistema lineal.

Resolución de la Matriz

```
for (let i = equationSize - 1; i > -1; i--) {
    for (let j = equationSize - 1; j > -1; j--) {
        if (i == j) {
            equations[i][nElements - 1] /= equations[i][j];
        }
        else if (equations[i][j] != 0) {
            equations[i][nElements - 1] -= equations[i][j] * equations[j]
[nElements - 1];
        }
    }
}
```

- **Descripción**: Este bloque de código resuelve la matriz utilizando la sustitución regresiva. Los coeficientes del polinomio se almacenan en la última columna de la matriz.

Almacenamiento de Soluciones y Cálculo del Error

```
this.solutions = new Array(equationSize);
for (let i = 0; i < equationSize; i++) {
    this.solutions[i] = equations[i][nElements - 1];
}

// Configurando el modelo como entrenado
this.isFit = true;

// Estableciendo el error
this.calculateR2(xArray, yArray);
```

- **Descripción**: Se almacenan los coeficientes calculados en el atributo `solutions`, se marca el modelo como entrenado (`isFit = true`) y se calcula el error utilizando el método `calculateR2`.

Método: `predict`

```
predict(xArray) {
    let yArray = [];
    // Verificando si el modelo ya está entrenado
    if (this.isFit) {
        // Generando las predicciones basadas en la entrada y las soluciones
```

```
        for (let i = 0; i < xArray.length; i++) {
            let yprediction = 0;
            for (let j = 0; j < this.solutions.length; j++) {
                yprediction += this.solutions[j] * Math.pow(xArray[i], j);
            }
            yArray.push(yprediction);
        }
    }

    // Devolviendo la predicción
    return yArray;
}
```

- **Descripción**: Este método genera predicciones basadas en el modelo entrenado. Si el modelo ha sido ajustado, calcula la salida `y` para cada entrada `x` utilizando los coeficientes del polinomio.

Método: `calculateR2`

```
calculateR2(xArray, yArray) {
    // Estableciendo el array de errores y predicciones
    let errors = new Array(xArray.length);
    let prediction = this.predict(xArray);
    let sumY = 0;

    // Calculando errores
    for (let i = 0; i < xArray.length; i++) {
        sumY += yArray[i];
        errors[i] = Math.pow(yArray[i] - prediction[i], 2);
    }

    let sr = 0;
    let st = 0;
    for (let i = 0; i < xArray.length; i++) {
        sr += errors[i];
        st += Math.pow(yArray[i] - (sumY / xArray.length), 2);
    }
    let r2 = (st - sr) / st;
    this.error = r2;
}
```

- **Descripción**: Este método calcula el coeficiente de determinación (R^2), que es una medida de la calidad del ajuste del modelo. Almacena el error calculado en el atributo `error`.

Método: `getError`

```
getError() {
    return this.error;
}
```

- **Descripción**: Este método devuelve el valor del error calculado, que se puede utilizar para evaluar la calidad del modelo.

SCRIPTS

Árbol de Decisión ID3

1. Variables Globales

```
let datosEntrenamiento = []; // Almacena los datos de entrenamiento
let datosPrediccion = [];    // Almacena los datos de predicción
```

Descripción:

- **datosEntrenamiento**: Array que almacenará los datos utilizados para entrenar el modelo.
- **datosPrediccion**: Array que contendrá los datos para hacer predicciones.

---

2. Función `leerCSVID3`

```
const leerCSVID3 = async (archivo) => {
    return new Promise((resolve, reject) => {
        const lector = new FileReader();
        lector.onload = (evento) => {
            const datos = evento.target.result;
            const filas = datos.trim().split("\n").map(fila => fila.split(","));
            resolve(filas);
        };
        lector.onerror = (error) => reject(error);
        lector.readAsText(archivo);
    });
};
```

Descripción:

- **Objetivo**: Leer un archivo CSV y devolver su contenido como un array de arrays.
- **Parámetros**:
  - `archivo`: El archivo CSV que se va a leer.
- **Funcionalidad**:
  - Utiliza `FileReader` para leer el contenido del archivo.
  - Separa el contenido en filas y columnas, devolviendo un array de filas.

---

3. Función `cargarDatosEntrenamiento`

```
const cargarDatosEntrenamiento = async () => {
    const archivo = document.getElementById("training-file").files[0];
    if (!archivo) {
        alert("Por favor selecciona un archivo CSV de entrenamiento!!");
        return;
    }
    datosEntrenamiento = await leerCSVID3(archivo);
};
```

Descripción:

- **Objetivo**: Cargar los datos de entrenamiento desde un archivo CSV.
- **Funcionalidad**:
  - Verifica si se ha seleccionado un archivo.
  - Llama a `leerCSVID3` para leer el archivo y almacenar los datos en `datosEntrenamiento`.

---

4. Función `cargarDatosPrediccion`

```
const cargarDatosPrediccion = async () => {
    const archivo = document.getElementById("predict-file").files[0];
    if (archivo) {
        datosPrediccion = await leerCSVID3(archivo);
    }
};
```

Descripción:

- **Objetivo**: Cargar los datos de predicción desde un archivo CSV (opcional).
- **Funcionalidad**:
  - Si se selecciona un archivo, llama a `leerCSVID3` para almacenar los datos en `datosPrediccion`.

---

5. Función `generarArbol`

```
const generarArbol = () => {
    if (datosEntrenamiento.length === 0) {
        alert("Por favor carga los datos de entrenamiento antes de generar el
árbol!!");
        return;
    }

    const arbolDecision = new DecisionTreeID3(datosEntrenamiento);
    const raiz = arbolDecision.train(arbolDecision.dataset);
```

```
    let prediccion = null;
    if (datosPrediccion.length > 0) {
        const cabeceraPrediccion = datosEntrenamiento[0].slice(0, -1);
        prediccion = arbolDecision.predict([cabeceraPrediccion,
datosPrediccion[0]], raiz);
    }

    return {
        dotStr: arbolDecision.generateDotString(raiz),
        nodoPrediccion: prediccion
    };
};
```

Descripción:

- **Objetivo**: Generar el árbol de decisión y realizar la predicción.
- **Funcionalidad**:
  - Verifica si hay datos de entrenamiento.
  - Crea una instancia del árbol de decisión y lo entrena.
  - Si hay datos de predicción, realiza una predicción utilizando la raíz del árbol.

---

6. Eventos de Interfaz de Usuario

Evento `click` para el botón de predicción

```
document.getElementById('predict').onclick = async () => {
    await cargarDatosPrediccion(); // Carga el archivo de predicción
    const grafico = document.getElementById("treed");
    const { dotStr, nodoPrediccion } = generarArbol();

    if (nodoPrediccion != null) {
        const cabecera = datosEntrenamiento[0];
        document.getElementById('prediction').innerText =
`${cabecera[cabecera.length - 1]}: ${nodoPrediccion.value}`;
    } else {
        document.getElementById('prediction').innerText = "No hay predicción
disponible.";
    }

    const parsDot = vis.network.convertDot(dotStr);
    const datos = {
        nodes: parsDot.nodes,
        edges: parsDot.edges
    };

    const opciones = {
        layout: {
            hierarchical: {
                levelSeparation: 100,
```

```
                nodeSpacing: 100,
                parentCentralization: true,
                direction: 'UD',
                sortMethod: 'directed'
            }
        }
    };

    new vis.Network(grafico, datos, opciones);
};
```

Descripción:

- **Objetivo**: Manejar la lógica de la interfaz de usuario para mostrar la predicción y visualizar el árbol.
- **Funcionalidad**:
    - Carga los datos de predicción.
    - Genera el árbol de decisión y actualiza la interfaz con la predicción, si está disponible.
    - Utiliza la biblioteca `vis.js` para visualizar el árbol.

Evento `click` para el botón de generación

```
document.getElementById('btngenerate').onclick = async () => {
    await cargarDatosEntrenamiento(); // Carga el archivo de entrenamiento
    const grafico = document.getElementById("treed");
    const { dotStr, nodoPrediccion } = generarArbol();

    if (nodoPrediccion != null) {
        const cabecera = datosEntrenamiento[0];
        document.getElementById('prediction').innerText =
`${cabecera[cabecera.length - 1]}: ${nodoPrediccion.value}`;
    } else {
        document.getElementById('prediction').innerText = "No hay predicción
disponible.";
    }

    const parsDot = vis.network.convertDot(dotStr);
    const datos = {
        nodes: parsDot.nodes,
        edges: parsDot.edges
    };

    const opciones = {
        layout: {
            hierarchical: {
                levelSeparation: 100,
                nodeSpacing: 100,
                parentCentralization: true,
                direction: 'UD',
                sortMethod: 'directed'
            }
        }
```

```
    };

    new vis.Network(grafico, datos, opciones);
};
```

Descripción:

- **Objetivo**: Manejar la lógica para cargar los datos de entrenamiento y generar el árbol.
- **Funcionalidad**:
  - Carga los datos de entrenamiento y genera el árbol de decisión.
  - Muestra la predicción (si está disponible) y visualiza el árbol usando `vis.js`.

Clustering K-Means

1. Variables Globales

```
let datos = [];
let k = 3; // Número de clusters por defecto
let iteraciones = 100; // Número de iteraciones por defecto
let grafico; // Variable para almacenar la instancia de Chart.js
```

Descripción:

- **datos**: Array que almacenará los puntos de datos leídos desde un archivo CSV.
- **k**: Número de clusters a formar, inicializado en 3.
- **iteraciones**: Número de iteraciones del algoritmo, inicializado en 100.
- **grafico**: Variable que almacenará la instancia del gráfico generado con Chart.js.

---

2. Función `leerCSVKMeans`

```
const leerCSVKMeans = (archivo) => {
  return new Promise((resolver, rechazar) => {
    const lector = new FileReader();
    lector.onload = (evento) => {
      const filas = evento.target.result
        .trim()
        .split("\n")
        .map(fila => fila.split(",").map(Number));
      resolver(filas);
    };
    lector.onerror = (error) => rechazar("Error de lectura del archivo: " +
error.message);
    lector.readAsText(archivo);
```

```
  });
};
```

Descripción:

- **Objetivo**: Leer un archivo CSV y devolver su contenido como un array de arrays de números.
- **Parámetros**:
  - `archivo`: El archivo CSV que se va a leer.
- **Funcionalidad**:
  - Utiliza `FileReader` para leer el contenido del archivo.
  - Separa el contenido en filas y columnas, convirtiendo los valores a números.

---

3. Cargar Archivo de Datos

```
document.getElementById("data-file").addEventListener("change", async (evento) =>
{
  try {
    datos = (await leerCSVKMeans(evento.target.files[0])).flat();
    console.log("Datos cargados:", datos);
  } catch (error) {
    console.error(error);
  }
});
```

Descripción:

- **Objetivo**: Cargar los datos desde un archivo CSV.
- **Funcionalidad**:
  - Escucha el evento de cambio en el input del archivo.
  - Llama a `leerCSVKMeans` para leer el archivo y almacenar los datos en la variable `datos`.

---

4. Cargar Archivo de Configuración

```
document.getElementById("config-file").addEventListener("change", async (evento)
=> {
  try {
    const config = (await leerCSVKMeans(evento.target.files[0]))[0];
    k = config[0];
    iteraciones = config[1];
    console.log(`Configuración: Clusters: ${k}, Iteraciones: ${iteraciones}`);
  } catch (error) {
    console.error(error);
  }
});
```

Descripción:

- **Objetivo**: Cargar la configuración de clustering desde un archivo CSV.
- **Funcionalidad**:
  - Escucha el evento de cambio en el input del archivo de configuración.
  - Establece `k` y `iteraciones` a partir de los valores leídos del archivo.

---

5. Función para Ejecutar Clustering y Graficar Resultados

```
document.getElementById("btnLineal").onclick = function () {
  if (datos.length === 0 || k <= 0 || iteraciones <= 0) {
    alert("Por favor, carga los archivos de datos y configuración.");
    return;
  }

  // Ejecutar el clustering
  const es2D = Array.isArray(datos[0]);
  const kmeans = es2D ? new _2DKMeans() : new LinearKMeans();
  const datosAgrupados = kmeans.clusterize(k, datos, iteraciones);

  // Asignar colores a los clusters
  const clusters = Array.from(new Set(datosAgrupados.map(a => a[1]))).map(cluster
=> [
    cluster,
    `#${Math.floor(Math.random() * 16777215).toString(16)}` // Color aleatorio
  ]);

  // Visualizar resultados
  dibujarGrafico(datosAgrupados, clusters, es2D);
};
```

Descripción:

- **Objetivo**: Ejecutar el algoritmo de clustering K-Means y graficar los resultados.
- **Funcionalidad**:
  - Verifica que los datos y parámetros sean válidos.
  - Crea una instancia de K-Means (2D o Lineal) y agrupa los datos.
  - Asigna colores aleatorios a cada cluster.
  - Llama a `dibujarGrafico` para visualizar los resultados.

---

6. Función `dibujarGrafico`

```
function dibujarGrafico(datosAgrupados, clusters, es2D) {
  // Destruir gráfico anterior si existe
  grafico?.destroy();
```

```
    // Preparar los datos para Chart.js
    const conjuntosDatos = clusters.map(([cluster, color]) => ({
      label: `Cluster ${cluster}`,
      data: datosAgrupados
        .filter(([_, idCluster]) => idCluster === cluster)
        .map(([punto]) => ({
          x: es2D ? punto[0] : punto,
          y: es2D ? punto[1] : 0,
        })),
      backgroundColor: color,
      pointRadius: 5,
    }));

    // Configuración del gráfico
    const ctx = document.getElementById("chartkmean").getContext("2d");
    grafico = new Chart(ctx, {
      type: "scatter",
      data: { datasets: conjuntosDatos },
      options: {
        title: { display: true, text: "Clustering K-Means" },
        scales: {
          x: {
            title: { display: true, text: "X" },
            min: Math.min(...datos.flat()) - 10,
            max: Math.max(...datos.flat()) + 10,
          },
          y: {
            title: { display: true, text: "Y" },
            min: es2D ? Math.min(...datos.flat()) - 10 : -1,
            max: es2D ? Math.max(...datos.flat()) + 10 : 1,
          },
        },
      },
    });
  }
```

Descripción:

- **Objetivo**: Graficar los resultados del clustering utilizando Chart.js.
- **Funcionalidad**:
  - Destruye el gráfico anterior si existe.
  - Prepara los datos agrupados y los clusters para el gráfico.
  - Configura y crea un nuevo gráfico de dispersión con los resultados.

Polynomial

Parte 1: Selección de Elementos del DOM

```
const archivoPolyInput = document.querySelector("#polynomial--file");
const gradoPolyInput = document.querySelector("#polynomial-degree");
const btnAjustarPoly = document.querySelector("#polynomial--btn-fit");
const btnPredecirPoly = document.querySelector("#polynomial--btn-predict");
const btnMSEPoly = document.querySelector("#polynomial--btn-mse");
const btnR2Poly = document.querySelector("#polynomial--btn-r2");

const resultadoAjustePoly = document.querySelector("#poly-fit-result");
const resultadoPrediccionPoly = document.querySelector("#poly-predict-result");
const resultadoMSEPoly = document.querySelector("#poly-mse-result");
const resultadoR2Poly = document.querySelector("#poly-r2-result");
```

Descripción:
Esta sección selecciona elementos del DOM (Document Object Model) usando `querySelector`, que se usarán más tarde para cargar archivos, configurar el modelo, mostrar resultados y realizar predicciones.

Parte 2: Variables Iniciales

```
let varsXPoli = [];
let varsYPoli = [];
let instanciaPolinomial = new PolynomialRegression();
let prediccionesPoly = [];
```

Descripción:
Se declaran variables para almacenar los datos de entrada (`varsXPoli` y `varsYPoli`), crear una nueva instancia de la clase `PolynomialRegression` (`instanciaPolinomial`), y almacenar las predicciones (`prediccionesPoly`).

Parte 3: Función para Leer el Archivo CSV

```
const leerCSVPoli = async (archivo) => {
  return new Promise((resolver, rechazar) => {
    const lector = new FileReader();
    lector.onload = (evento) => {
      const datos = evento.target.result;
      const filas = datos.split("\n").slice(1);
      const datosX = [];
      const datosY = [];

      filas.forEach((fila) => {
        const [x, y] = fila.split(",").map(Number);
        if (!isNaN(x) && !isNaN(y)) {
          datosX.push(x);
          datosY.push(y);
        }
      });
```

```
      resolver({ varsX: datosX, varsY: datosY });
    };
    lector.onerror = (error) => rechazar(error);
    lector.readAsText(archivo);
  });
};
```

Descripción:
Esta función `leerCSVPoli` lee un archivo CSV y procesa su contenido. Utiliza `FileReader` para leer el archivo de forma asíncrona y divide los datos en filas, extrayendo las variables `x` e `y` para almacenarlas en arreglos. Finalmente, devuelve un objeto con ambas variables.

Parte 4: Ajustar el Modelo Polinomial

```
const ajustarModeloPoli = async () => {
  if (!archivoPolyInput.files.length || !gradoPolyInput.value) {
    resultadoAjustePoly.textContent =
      "Por favor selecciona un archivo CSV y el grado del polinomio.";
    return;
  }

  const grado = parseInt(gradoPolyInput.value);
  const datos = await leerCSVPoli(archivoPolyInput.files[0]);
  varsXPoli = datos.varsX;
  varsYPoli = datos.varsY;

  instanciaPolinomial.fit(varsXPoli, varsYPoli, grado);

  btnPredecirPoly.disabled = false;
  resultadoAjustePoly.textContent = "Modelo polinomial ajustado correctamente.";
};
```

Descripción:
La función `ajustarModeloPoli` valida que se haya seleccionado un archivo y un grado de polinomio. Si es así, lee los datos del archivo CSV y ajusta el modelo polinomial utilizando los datos. Habilita el botón de predicción una vez ajustado el modelo.

Parte 5: Predecir Usando el Modelo Polinomial

```
const predecirModeloPoli = () => {
  prediccionesPoly = instanciaPolinomial.predict(varsXPoli);

  btnR2Poly.disabled = false;

  renderizarGraficoPoli();
```

```
  resultadoPrediccionPoly.textContent = "Predicciones realizadas.";
};
```

Descripción:
Esta función `predecirModeloPoli` realiza predicciones utilizando el modelo ajustado. Genera predicciones basadas en los datos de entrada y llama a la función para renderizar el gráfico. También habilita el botón para calcular el coeficiente R^2 y muestra un mensaje de confirmación.

Parte 6: Calcular el Coeficiente R^2

```
const calcularR2Poli = () => {
  const r2 = instanciaPolinomial.getError();
  resultadoR2Poly.textContent = `Coeficiente R2 del modelo es: ${r2.toFixed(4)}`;
};
```

Descripción:
La función `calcularR2Poli` calcula el coeficiente de determinación R^2 del modelo ajustado, que indica qué tan bien se ajusta el modelo a los datos. Muestra el resultado en el elemento correspondiente del DOM.

Parte 7: Renderizar el Gráfico Polinomial

```
const renderizarGraficoPoli = () => {
  const datosLineaPoli = varsXPoli.map((x, index) => ({
    x,
    y: prediccionesPoly[index],
  }));
  const datosPuntoPoli = varsXPoli.map((x, index) => ({
    x,
    y: varsYPoli[index],
  }));

  const ctx = document.querySelector("#polynomial--canva").getContext("2d");

  if (graficoPoli) {
    graficoPoli.destroy();
  }

  graficoPoli = new Chart(ctx, {
    type: "scatter",
    data: {
      datasets: [
        {
          label: "Regresión Polinomial",
          data: datosLineaPoli,
          type: "line",
          borderColor: "rgba(75, 192, 192, 1)",
```

```
            borderWidth: 2,
            fill: false,
            pointRadius: 0,
          },
          {
            label: "Datos Originales",
            data: datosPuntoPoli,
            backgroundColor: "rgba(255, 99, 132, 1)",
            pointRadius: 5,
          },
        ],
      },
      options: {
        scales: {
          x: {
            type: "linear",
            position: "bottom",
          },
          y: {
            beginAtZero: true,
          },
        },
        plugins: {
          legend: {
            display: true,
          },
        },
      },
    });
};
```

Descripción:
La función `renderizarGraficoPoli` prepara los datos para graficar la regresión polinomial y los datos originales. Utiliza `Chart.js` para crear un gráfico en un elemento `<canvas>`, mostrando tanto la línea de regresión como los puntos de los datos originales. Si ya existe un gráfico, lo destruye antes de crear uno nuevo.

Parte 8: Eventos de los Botones

```
btnAjustarPoly.addEventListener("click", ajustarModeloPoli);
btnPredecirPoly.addEventListener("click", predecirModeloPoli);
btnR2Poly.addEventListener("click", calcularR2Poli);
```

Descripción:
Se añaden manejadores de eventos a los botones. Cuando se hace clic en un botón, se ejecuta la función correspondiente para ajustar el modelo, predecir valores o calcular el coeficiente R^2.

Seleccionar Modelo

Parte 1: Escuchar el Evento `DOMContentLoaded`

```
document.addEventListener("DOMContentLoaded", () => {
  // Código a ejecutar una vez que el documento esté completamente cargado
});
```

Descripción:
Este evento asegura que el código dentro de la función se ejecute solo después de que todo el contenido del documento HTML haya sido completamente cargado y parseado, lo que garantiza que todos los elementos del DOM estén disponibles para manipulación.

Parte 2: Selección de Elementos del DOM

```
const selector = document.getElementById("model-selector");
const sections = document.querySelectorAll(".model-section");
```

Descripción:
Aquí se seleccionan dos elementos del DOM:

- `selector`: se refiere al elemento que permite al usuario seleccionar un modelo (generalmente un elemento `<select>`).
- `sections`: se refiere a todos los elementos que representan secciones de modelos (probablemente `<div>`s con la clase `model-section`).

Parte 3: Escuchar Cambios en el Selector

```
selector.addEventListener("change", () => {
  // Ocultar todas las secciones
  sections.forEach((section) => {
    section.style.display = "none";
  });

  // Mostrar la sección seleccionada
  const selectedModel = selector.value;
  if (selectedModel) {
    document.getElementById(selectedModel).style.display = "block";
  }
});
```

Descripción:
Se añade un manejador de eventos para el evento `change` del `selector`. Cuando el usuario selecciona un modelo diferente:

1. **Oculta todas las secciones:** Itera sobre cada `section` y establece su estilo `display` en `none`, lo que las oculta.
2. **Muestra la sección seleccionada:** Obtiene el valor del modelo seleccionado (`selectedModel`) y utiliza ese valor para mostrar la sección correspondiente cambiando su estilo `display` a `block`. Esto permite que solo se muestre la sección del modelo que el usuario ha elegido.
