Manual Técnico

- Damián Ignacio Peña Afre
- 202110568

Index.js

1. Descripción General

Este archivo `index.js` actúa como controlador principal para un proyecto de aprendizaje automático en el navegador. Permite a los usuarios cargar un dataset y seleccionar entre varios modelos de predicción, incluyendo regresión lineal, regresión polinómica y árboles de decisión. Además, incluye funcionalidades para entrenamiento, predicción, visualización de gráficos e identificación de patrones en los datos cargados.

2. Dependencias

- **Google Charts**: Utilizado para renderizar gráficos de los resultados. Se carga mediante `google.charts.load('current', { packages: ['corechart'] })`.
- **Funciones de Modelos Externos**:
  - `performLinearRegression` de `./linear.js`
  - `performRegresionPolynomial` de `./poli.js`
  - `performTree` de `./tree.js`

  Estas funciones contienen las implementaciones específicas de los modelos de aprendizaje automático.

3. Estructura del Código

3.1. Importación de Módulos

```
import { performLinearRegression } from './linear.js';
import { performRegresionPolynomial } from './poli.js';
import { performTree } from './tree.js';
google.charts.load('current', { packages: ['corechart'] });
```

Cada módulo importado corresponde a un modelo específico de aprendizaje automático. La biblioteca `google.charts` se utiliza para generar gráficos de los resultados.

3.2. Elementos HTML

Se obtienen referencias a los elementos HTML necesarios mediante `getElementById`, los cuales permiten interactuar con el usuario:

- `modelSelect`: Elemento `<select>` para elegir el modelo a aplicar.
- `trainButton`: Botón para entrenar el modelo.
- `predictButton`: Botón para ejecutar predicciones.

- `showGraphButton`: Botón para visualizar los gráficos generados.
- `patternsButton`: Botón para identificar patrones en los datos.
- `datasetInput`: Input de archivo (`file-input`) para cargar el dataset.

3.3. Función `executeModel`

```
const executeModel = (modelName, action) => {
    console.log({ modelName, action });
    if (!datasetInput.files.length) return alert('Please select a dataset file');

    switch (modelName) {
        case 'linear-regression':
            performLinearRegression(action);
            break;
        case 'polynomial-regression':
            performRegresionPolynomial(action);
            break;
        case 'decision-tree':
            performTree(action);
            break;
        default:
            console.log('Invalid model name');
            break;
    }
};
```

- **Propósito**: Ejecuta la función del modelo seleccionado, aplicando la acción especificada (`train`, `predict`, `show`, `patterns`).
- **Parámetros**:
    - `modelName`: Nombre del modelo seleccionado.
    - `action`: Acción a realizar.
- **Flujo**:
    - Verifica que un archivo de datos esté cargado. Si no es así, muestra una alerta.
    - Según el valor de `modelName`, invoca la función correspondiente al modelo.
    - En caso de un nombre de modelo no válido, muestra un mensaje de error en la consola.

3.4. Eventos

Cada botón está enlazado a un evento específico, que desencadena una acción en el modelo seleccionado:

```
trainButton.addEventListener('click', () => {
    const modelName = modelSelect.value;
    executeModel(modelName, 'train');
});

predictButton.addEventListener('click', () => {
    const modelName = modelSelect.value;
    executeModel(modelName, 'predict');
});

showGraphButton.addEventListener('click', () => {
    const modelName = modelSelect.value;
    executeModel(modelName, 'show');
});

patternsButton.addEventListener('click', () => {
    const modelName = modelSelect.value;
    executeModel(modelName, 'patterns');
});
```

- **Entrenamiento (`train`)**: Ejecuta el entrenamiento del modelo con el dataset cargado.
- **Predicción (`predict`)**: Realiza una predicción en función del modelo entrenado.
- **Visualización de Gráfica (`show`)**: Muestra un gráfico que representa los resultados del modelo.
- **Identificación de Patrones (`patterns`)**: Detecta patrones en el conjunto de datos.

4. Funcionalidades Específicas de los Modelos

Cada uno de los archivos `linear.js`, `poli.js` y `tree.js` contiene la lógica de sus respectivos modelos. Estos módulos exportan funciones que ejecutan acciones particulares sobre los datos de entrada.

4.1. `performLinearRegression` (en `linear.js`)

- **Acciones**: Entrenar, predecir, visualizar gráfico, detectar patrones.
- **Interfaz**: `performLinearRegression(action)` recibe la acción y ejecuta la lógica correspondiente.

4.2. `performRegresionPolynomial` (en `poli.js`)

- **Acciones**: Entrenar, predecir, visualizar gráfico, detectar patrones.
- **Interfaz**: `performRegresionPolynomial(action)` realiza la acción indicada.

4.3. `performTree` (en `tree.js`)

- **Acciones**: Entrenar, predecir, visualizar gráfico, detectar patrones.
- **Interfaz**: `performTree(action)` maneja el flujo específico del árbol de decisión.

5. Validaciones y Manejo de Errores

1. **Carga de Dataset**: Antes de ejecutar cualquier acción, se verifica si el dataset ha sido cargado; de lo contrario, se muestra una alerta.
2. **Selección de Modelo Válido**: La función `executeModel` emite un mensaje en consola para modelos no válidos, lo que ayuda a depurar errores.

6. Extensibilidad

Para agregar nuevos modelos de aprendizaje automático:

1. Crear un archivo con la implementación del modelo.
2. Exportar una función que reciba el parámetro `action` para decidir la operación (`train`, `predict`, `show`, `patterns`).
3. Importar la función en `index.js` y añadir un caso en `executeModel`.

`linear.js`

Aquí tienes el manual técnico para el archivo `linear.js` en tu proyecto:

Manual Técnico para `linear.js`

1. Descripción General

El archivo `linear.js` contiene la función `performLinearRegression`, que implementa el flujo completo para aplicar regresión lineal sobre un dataset cargado por el usuario. Esta función ejecuta varias acciones como entrenar el modelo, realizar predicciones, mostrar una gráfica y detectar patrones. Utiliza métodos importados de `utils.js` para procesar el dataset y mostrar los resultados.

2. Dependencias

- **Modulos Importados**:
  - `convertCsvToJson`: Convierte el archivo CSV cargado en un formato JSON adecuado para trabajar con los datos.
  - `displayPatterns`: Función que muestra patrones en los datos procesados.
  - `displayTrend`: Función que visualiza la tendencia en los datos mediante un gráfico.
- **Elementos HTML**:
  - `datasetInput`: Input de tipo archivo (`file-input`) para cargar el dataset.
  - `results`: Elemento `div` donde se muestran resultados de entrenamiento y el coeficiente del modelo.

3. Estructura de la Función `performLinearRegression`

3.1. Definición y Preprocesamiento

```
export const performLinearRegression = async (action) => {
    const { xValues, yValues } = await convertCsvToJson(datasetInput.files[0]);
    const linearModel = new LinearRegression();
    linearModel.fit(xValues, yValues);
    const predictions = linearModel.predict(xValues);
```

- **Propósito**: La función `performLinearRegression` realiza una operación específica de regresión lineal basada en el parámetro `action`.
- **Carga y Transformación de Datos**: Utiliza `convertCsvToJson` para transformar el archivo CSV cargado en JSON, extrayendo `xValues` e `yValues`.
- **Creación del Modelo**: Instancia un objeto `LinearRegression` y ajusta el modelo usando `fit(xValues, yValues)`.
- **Predicciones**: Realiza predicciones en los valores de entrada mediante `linearModel.predict(xValues)`.

3.2. Opciones de Gráfico

```
const options = {
    title: 'Linear Regression Model',
    seriesType: 'scatter',
    series: { 1: { type: 'line' } },
    hAxis: { title: 'X' },
    vAxis: { title: 'Y' }
};
```

- **Propósito**: Configura las opciones de visualización para el gráfico de Google Charts.
- **Configuraciones**:
    - Título del gráfico.
    - Tipo de serie (`scatter` para los puntos de datos y `line` para la línea de regresión).
    - Etiquetas de los ejes `X` y `Y`.

3.3. Acciones

La función `performLinearRegression` realiza diferentes operaciones basadas en el valor de `action`:

- **Entrenamiento (`train`)**

```
if (action === 'train') {
    results.innerHTML = '';
    const linearResult = document.createElement('div');
    linearResult.innerHTML = `Intercepto: ${linearModel.b} Pendiente: ${linearModel.m}`;
    results.appendChild(linearResult);
    alert('Model trained');
}
```

    - **Propósito**: Ajusta el modelo y muestra los parámetros del modelo entrenado.
    - **Resultado**: Limpia el elemento `results` y muestra el intercepto (`b`) y la pendiente (`m`) de la recta de regresión.

- **Predicción (`predict`)**

```
if (action === 'predict') {
    const dataArray = [['X', 'Actual Y', 'Prediction']];
    xValues.forEach((x, i) => dataArray.push([x, yValues[i], predictions[i]]));

    const dataTable = google.visualization.arrayToDataTable(dataArray);
    const chart = new google.visualization.ComboChart(document.getElementById('chart'));
    chart.draw(dataTable, options);
}
```

  - **Propósito**: Crea un gráfico de predicción comparando valores reales (`Actual Y`) y predichos.
  - **Estructura de Datos**: Utiliza un array de arrays `dataArray` que incluye `X`, `Actual Y` y `Prediction` para cada punto.
  - **Visualización**: Utiliza `google.visualization.arrayToDataTable` para crear el gráfico y `google.visualization.ComboChart` para dibujar el gráfico de línea y dispersión.

- **Mostrar Tendencia (`show`)**

```
if (action === 'show') {
    displayTrend(xValues, yValues);
}
```

  - **Propósito**: Muestra una gráfica de tendencia de los datos.
  - **Lógica**: Llama a `displayTrend`, que se encarga de crear la gráfica utilizando los valores de entrada.

- **Identificación de Patrones (`patterns`)**

```
if (action === 'patterns') {
    displayPatterns(predictions);
}
```

  - **Propósito**: Muestra patrones detectados en las predicciones realizadas.
  - **Lógica**: Llama a `displayPatterns` con las predicciones para visualizar patrones relevantes en los datos.

4. Estructura de la Clase `LinearRegression`

La clase `LinearRegression` no está definida aquí, pero su uso en este archivo sugiere que tiene al menos dos métodos:

- **`fit(xValues, yValues)`**: Calcula y ajusta los coeficientes de la regresión lineal.
- **`predict(xValues)`**: Devuelve las predicciones basadas en los valores `x` ingresados.

5. Flujo de Ejecución

1. Carga el archivo y convierte los datos a JSON.
2. Ajusta el modelo si se selecciona la acción `train`.

3. Muestra las predicciones en un gráfico si se elige la acción `predict`.
4. Genera un gráfico de tendencia si la acción es `show`.
5. Detecta y visualiza patrones si se elige la acción `patterns`.

6. Extensibilidad y Modificación

Para modificar el comportamiento de `performLinearRegression`, puedes:

- **Agregar Nuevas Acciones**: Expandir la estructura `if` para manejar nuevas acciones según se requiera.
- **Ajustar Configuraciones de Gráficos**: Modificar el objeto `options` para cambiar el estilo o el tipo de visualización en Google Charts.
- **Modificar el Modelo**: Alterar la lógica de `LinearRegression` para probar diferentes algoritmos de regresión.

poli.js

Aquí tienes el manual técnico para el archivo `poli.js`, que contiene la función `performRegresionPolynomial` para aplicar una regresión polinómica en los datos cargados:

Manual Técnico para `poli.js`

1. Descripción General

El archivo `poli.js` implementa la función `performRegresionPolynomial`, que permite realizar una regresión polinómica sobre un conjunto de datos. La función admite cuatro tipos de acciones (`train`, `predict`, `show`, `patterns`) y muestra los resultados y patrones mediante gráficos y mensajes en la interfaz.

2. Dependencias

- **Modulos Importados**:
  - `convertCsvToJson`: Convierte un archivo CSV en formato JSON, extrayendo valores de `x` y `y`.
  - `displayPatterns`: Muestra patrones detectados en los datos procesados.
  - `displayTrend`: Visualiza la tendencia de los datos.
  - `getModelParams`: Recupera parámetros adicionales para el modelo, como el grado del polinomio.
- **Elementos HTML**:
  - `datasetInput`: Input de tipo archivo (`file-input`) para cargar el dataset.
  - `results`: Elemento `div` donde se muestran los resultados del entrenamiento.

3. Estructura de la Función `performRegresionPolynomial`

3.1. Definición y Preprocesamiento

```
export const performRegresionPolynomial = async (action) => {
    const { xValues, yValues } = await convertCsvToJson(datasetInput.files[0]);
    const polynomialModel = new PolynomialRegression();
    const { degree = 2 } = getModelParams();
    console.log({ action, xValues, yValues, degree });
    polynomialModel.fit(xValues, yValues, degree);
    const predictions = polynomialModel.predict(xValues);
```

- **Propósito**: Ejecuta una regresión polinómica en los datos ingresados según la acción seleccionada (`train`, `predict`, `show`, `patterns`).
- **Preprocesamiento**:
    - **Carga de datos**: `convertCsvToJson` transforma el archivo cargado en JSON y extrae `xValues` e `yValues`.
    - **Configuración de Grado**: `getModelParams` obtiene el grado del polinomio (por defecto `2`) que se usará en la regresión.
- **Creación del Modelo**: Instancia un objeto `PolynomialRegression` y lo ajusta con `fit(xValues, yValues, degree)` para entrenar el modelo.
- **Predicciones**: `polynomialModel.predict(xValues)` genera predicciones para cada valor en `xValues`.

3.2. Acciones

La función `performRegresionPolynomial` ejecuta diferentes operaciones en función del parámetro `action`:

- **Entrenamiento (`train`)**

```
if (action === 'train') {
    const resultadoPolinomial = document.createElement('resultadoPolinomial');
    results.innerHTML = '';
    resultadoPolinomial.innerHTML = `
        <div class="">
        Degree: ${degree} <br>
        Soluciones: ${JSON.stringify(polynomialModel.solutions.join(', '))}
        Error: ${polynomialModel.error}
        </div>
        `;
    results.appendChild(resultadoPolinomial);
    alert('Modelo entrenado');
    return;
}
```

    - **Propósito**: Ajusta el modelo y muestra los coeficientes y el error.
    - **Resultados**: Limpia el elemento `results` y muestra el grado (`degree`), las soluciones (coeficientes) y el error del modelo.
    - **Elementos HTML**: Crea un `div` (`resultadoPolinomial`) que muestra el grado del polinomio y el error del modelo.

- **Predicción (`predict`)**

```
if (action === 'predict') {
    const options = {
        title: 'Regresión Polinomial',
        seriesType: 'scatter',
        series: { 1: { type: 'line' } },
        hAxis: { title: 'X' },
        vAxis: { title: 'Y' }
    };
    const dataArray = [['X', 'Y real', 'Predicción']];
    xValues.forEach((x, i) => dataArray.push([x, yValues[i], predictions[i]]));
    const dataTable = google.visualization.arrayToDataTable(dataArray);
    const chart = new google.visualization.ComboChart(document.getElementById('chart'));
    chart.draw(dataTable, options);
    return;
}
```

- **Propósito**: Genera un gráfico que compara los valores reales (`Y real`) con las predicciones (`Predicción`).
- **Opciones de Visualización**: Configura un gráfico de dispersión con línea de tendencia para las predicciones.
- **Estructura de Datos**: Crea un array de arrays `dataArray` que contiene `X`, `Y real` y `Predicción` para cada punto.
- **Visualización**: Usa `google.visualization.ComboChart` para dibujar el gráfico con los datos.

- **Mostrar Tendencia (`show`)**

```
if (action === 'show') {
    displayTrend(xValues, yValues);
}
```

- **Propósito**: Muestra la tendencia de los datos originales.
- **Lógica**: Llama a `displayTrend`, que se encarga de crear un gráfico usando `xValues` e `yValues`.

- **Identificación de Patrones (`patterns`)**

```
if (action === 'patterns') {
    displayPatterns(predictions);
}
```

- **Propósito**: Muestra patrones detectados en las predicciones.
- **Lógica**: Llama a `displayPatterns` para procesar los valores predichos y mostrar los patrones detectados en los datos.

4. Estructura de la Clase `PolynomialRegression`

Aunque la clase `PolynomialRegression` no está definida en este archivo, su uso sugiere que tiene al menos los siguientes métodos:

- **`fit(xValues, yValues, degree)`**: Ajusta el modelo polinómico según los valores `x`, `y` y el grado especificado.
- **`predict(xValues)`**: Devuelve predicciones para los valores `x` dados, según el modelo ajustado.

5. Flujo de Ejecución

1. **Carga de Datos**: Carga el archivo CSV y convierte los datos a JSON.
2. **Configuración del Grado**: Determina el grado del polinomio.
3. **Entrenamiento**: Ajusta el modelo con los valores de `x` e `y`.
4. **Acciones**:
    - `train`: Muestra el grado, los coeficientes y el error del modelo.
    - `predict`: Dibuja un gráfico comparativo entre los valores reales y las predicciones.
    - `show`: Visualiza una gráfica de tendencia de los datos originales.
    - `patterns`: Detecta y muestra patrones en las predicciones generadas.

6. Extensibilidad y Modificación

Para expandir la funcionalidad de `performRegresionPolynomial`, puedes:

- **Agregar Nuevas Acciones**: Extender la estructura de control para manejar otras acciones según se necesiten.
- **Ajustar Configuraciones de Gráficos**: Modificar `options` para cambiar el estilo y la presentación de los gráficos.
- **Modificar el Grado del Modelo**: Adaptar `getModelParams` para obtener el grado desde otra fuente o agregar configuraciones adicionales.

`tree.js`

Aquí tienes el manual técnico para el archivo `tree.js`, que contiene la función `performTree` y otros componentes necesarios para aplicar un árbol de decisión ID3 en un conjunto de datos cargado:

Manual Técnico para `tree.js`

1. Descripción General

El archivo `tree.js` implementa la función `performTree`, que permite entrenar y realizar predicciones utilizando un modelo de árbol de decisión ID3. Adicionalmente, incluye funciones para visualizar el árbol de decisión generado.

2. Dependencias

- **Módulos Importados**:
    - `convertCsvToTreeJson`: Convierte un archivo CSV a JSON para el entrenamiento del árbol de decisión.
    - `getModelParams`: Obtiene parámetros de configuración, como el porcentaje de datos de entrenamiento.
- **Elementos HTML**:
    - `datasetInput`: Input de tipo archivo (`file-input`) para cargar el dataset.

- `chart`: Contenedor HTML (`div`) donde se visualiza el árbol de decisión.

- **Biblioteca de Visualización**:
  - `vis-network`: Biblioteca usada para crear la representación gráfica del árbol de decisión mediante un objeto `vis.Network`.

3. Estructura de la Función `performTree`

3.1. Definición y Preprocesamiento

```
export const performTree = async (action) => {
    console.log({ action });
    const { params } = await convertCsvToTreeJson(datasetInput.files[0]);
    const { train = 0.8 } = getModelParams();
    const headers = params[0];
    const trainData = params.slice(0, Math.floor(params.length * train));
    const predictionData = [
        [...headers.slice(0, headers.length - 1)],
        ...params.slice(Math.floor(params.length * train)).map(row => {
            const rowCopy = [...row];
            rowCopy.pop();
            return rowCopy;
        })
    ];
    console.log({ trainData, predictionData });
    const decisionTree = new DecisionTreeID3(trainData);
    const root = decisionTree.train(decisionTree.dataset);
    ...
```

- **Propósito**: Ejecuta un árbol de decisión ID3 en el conjunto de datos de entrada, permitiendo entrenar el modelo y realizar predicciones.
- **Preprocesamiento**:
  - **Carga de Datos**: `convertCsvToTreeJson` transforma el archivo CSV en JSON y extrae las filas para el procesamiento.
  - **Separación de Datos**: Divide el conjunto de datos en `trainData` y `predictionData` según el porcentaje de entrenamiento especificado en `getModelParams`.
- **Configuración del Modelo**: Crea una instancia de `DecisionTreeID3` con los datos de entrenamiento (`trainData`).
- **Entrenamiento del Modelo**: `decisionTree.train` genera la estructura del árbol basada en el dataset de entrenamiento.

3.2. Acciones

La función `performTree` ejecuta diferentes operaciones en función del parámetro `action`:

- **Entrenamiento (`train`)**

```
if (action === 'train') return alert('Training completed successfully!');
```

- **Propósito**: Entrena el modelo ID3 usando los datos de entrenamiento.
- **Salida**: Notificación en pantalla indicando que el entrenamiento fue exitoso.

- **Predicción (`predict`)**

```
if (action === 'predict') {
    decisionTree.predict(predictionData, root);
    const dot = decisionTree.generateDotString(root);
    console.log({ dot });
    showDecisionTreeGraph(dot);
}
```

- **Propósito**: Realiza predicciones utilizando el árbol de decisión entrenado.
- **Visualización**:
  - **Generación de Dot String**: `generateDotString` crea una representación en formato DOT del árbol.
  - **Llamada a `showDecisionTreeGraph`**: Dibuja el árbol de decisión en el contenedor `chart` usando `vis.Network`.

4. Visualización del Árbol de Decisión (`showDecisionTreeGraph`)

La función `showDecisionTreeGraph` recibe un `dotStr` que contiene la representación del árbol en formato DOT y lo convierte en un gráfico interactivo usando `vis.Network`:

```
const showDecisionTreeGraph = (dotStr) => {
    const chart = document.getElementById("chart");
    const parsDot = vis.network.convertDot(dotStr);
    const data = {
        nodes: parsDot.nodes,
        edges: parsDot.edges
    };
    const options = {
        layout: {
            hierarchical: {
                levelSeparation: 100,
                nodeSpacing: 100,
                parentCentralization: true,
                direction: 'UD',
                sortMethod: 'directed',
            },
        },
    };
    const network = new vis.Network(chart, data, options);
};
```

- **Propósito**: Generar un gráfico del árbol de decisión basado en la representación DOT.
- **Parámetros de Configuración**:
    - **Jerarquía del Gráfico**: `levelSeparation`, `nodeSpacing` y `direction` controlan el espaciado y la disposición del gráfico.
    - **Generación del Objeto `network`**: Crea un gráfico interactivo a partir de `data`, que contiene `nodes` y `edges` procesados desde el DOT.

5. Estructura de la Clase `DecisionTreeID3`

Aunque `DecisionTreeID3` no está definido en este archivo, se asume que incluye métodos clave:

- **`train(dataset)`**: Genera el árbol de decisión a partir del conjunto de datos de entrenamiento.
- **`predict(data, root)`**: Realiza predicciones en los datos de prueba utilizando el árbol entrenado.
- **`generateDotString(root)`**: Devuelve una representación en formato DOT del árbol de decisión para su visualización.

6. Flujo de Ejecución

1. **Carga de Datos**: Se lee el archivo CSV y se convierte a JSON.
2. **División de Datos**: El conjunto se divide en datos de entrenamiento y datos para predicción.
3. **Entrenamiento**:
    - `train`: Entrena el modelo y muestra una alerta confirmando el éxito del entrenamiento.
4. **Predicción y Visualización**:
    - `predict`: Realiza predicciones en los datos y muestra el árbol de decisión.

7. Extensibilidad y Modificación

Para ampliar la funcionalidad de `performTree`, puedes:

- **Agregar Nuevas Acciones**: Extender la estructura `switch` o `if` para manejar nuevas acciones, como `evaluate` para métricas de precisión.
- **Personalizar la Visualización**: Ajustar `options` en `showDecisionTreeGraph` para cambiar el estilo y disposición del gráfico.
- **Ajustes en los Parámetros del Árbol**: Adaptar `getModelParams` para agregar opciones de configuración avanzadas.