

PROYECTO JAVA/C++: SISTEMA DE GESTION DE PRODUCTOS

DAVID ORJUELA//STEVEN
BOHORQUEZ//JULIO
USECHE//YOSEFH PEÑA

ARCHIVO REGRESION_LINEAL.H

La clase RegresionLineal en C++ permite realizar regresión lineal simple, calculando una línea de ajuste para un conjunto de datos. Incluye los métodos entrenar, que calcula la pendiente e intercepto de la línea; predecir, que estima un valor de y para un x dado; y calcularMSE, que evalúa la precisión del modelo mediante el Error Cuadrático Medio. Los atributos privados pendiente e intercepto almacenan los valores calculados de la línea de ajuste. Esta clase es útil para analizar tendencias y predecir valores en series de datos.

```
1  #ifndef REGRESION_LINEAL_H
2  #define REGRESION_LINEAL_H
3
4  #include <vector>
5
6  class RegresionLineal {
7  public:
8      RegresionLineal();
9      void entrenar(const std::vector<double>& x, const std::vector<double>& y);
10     double predecir(double valor);
11     double calcularMSE(const std::vector<double>& x, const std::vector<double>& y);
12
13 private:
14     double pendiente;
15     double intercepto;
16 };
17
18 #endif
```

ARCHIVO REGRESION_LINEAL.CPP

Implementa los métodos declarados en regresion_lineal.h. El método entrenar calcula la pendiente e intersección basándose en los datos de entrada, utilizando fórmulas estadísticas. predecir toma un valor y calcula la predicción basada en la línea de regresión, y calcularMSE evalúa la precisión del modelo.

```
#include "regresion_lineal.h"
#include <numeric>
#include <cmath>

RegresionLineal::RegresionLineal() : pendiente(0.0), intercepto(0.0) {}

▼ void RegresionLineal::entrenar(const std::vector<double>& x, const std::vector<double>& y)
    int n = x.size();
    double sumaX = std::accumulate(x.begin(), x.end(), 0.0);
    double sumaY = std::accumulate(y.begin(), y.end(), 0.0);
    double sumaXY = 0.0, sumaX2 = 0.0;

    for (int i = 0; i < n; ++i) {
        sumaXY += x[i] * y[i];
        sumaX2 += std::pow(x[i], 2);
    }

    pendiente = (n * sumaXY - sumaX * sumaY) / (n * sumaX2 - std::pow(sumaX, 2));
    intercepto = (sumaY - pendiente * sumaX) / n;
}

double RegresionLineal::predecir(double valor) {
    return pendiente * valor + intercepto;
}

▼ double RegresionLineal::calcularMSE(const std::vector<double>& x, const std::vector<double>& y)
    double mse = 0.0;
    for (int i = 0; i < x.size(); ++i) {
        mse += std::pow(predecir(x[i]) - y[i], 2);
    }
    return mse / x.size();
}
```

ARCHIVO MAIN.CPP

Este archivo es el punto de inicio en C++. Aquí se integran los métodos para cargar datos de prueba, entrenar el modelo y generar predicciones, verificando el funcionamiento de la regresión lineal.

```
#include <iostream>
#include <vector>
#include "regresion_lineal.h"

int main() {
    RegresionLineal modelo;
    std::vector<double> ventas = {10, 20, 30, 40, 50};
    std::vector<double> precios = {100, 200, 300, 400, 500};

    modelo.entrenar(ventas, precios);

    std::cout << "Precio predicho para 60 ventas: " << modelo.predecir(60) << std::endl;
    std::cout << "MSE: " << modelo.calcularMSE(ventas, precios) << std::endl;

    return 0;
}
```

1. CLASE MAIN.JAVA (PUNTO DE ENTRADA DE LA APLICACIÓN JAVAFX)

Propósito: Main.java es la clase principal que inicia la aplicación en JavaFX. Su función es crear y mostrar la interfaz gráfica de usuario (GUI) para el sistema de gestión de productos. Detalles del Código:
Método start: Este método se ejecuta cuando la aplicación inicia. Aquí se utiliza FXMLLoader para cargar el archivo VentasView.fxml, que define la estructura visual de la ventana principal de la aplicación.
Configuración de la Ventana: La línea primaryStage.setTitle("Sistema de Gestión de Productos") establece el título de la ventana, y luego primaryStage.show() muestra la interfaz al usuario. Este método prepara la escena y la hace visible, permitiendo la interacción con la aplicación.

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        // Cargar la vista desde el archivo FXML
        FXMLLoader loader = new FXMLLoader(getClass().getResource("/views/VentasView.fxml"));
        Scene scene = new Scene(loader.load());

        // Configurar la ventana principal
        primaryStage.setTitle("Sistema de Gestión de Productos");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        // Iniciar la aplicación JavaFX
        launch(args);
    }
}
```

3. MODELOS (MODELS): REPRESENTACIÓN Y PREDICCIÓN DE PRODUCTOS

Propósito: Las clases en la carpeta models representan los diferentes tipos de productos y su funcionalidad relacionada, como el cálculo de predicciones.

PrediccionPrecios.java: Función de Predicción: Esta clase encapsula la lógica para predecir el precio de un producto utilizando datos históricos o atributos específicos. Idealmente, este modelo puede llamar al código C++ para realizar cálculos avanzados, integrando la funcionalidad de regresión lineal. Métodos de Predicción: Puede contener métodos que reciben datos de entrada (como el precio anterior o la categoría del producto) y devuelven un precio estimado. Este es el núcleo que integra la lógica de predicción con la gestión de productos.

```
package models;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class PrediccionPrecios {
    public static double predecir(double valor) {
        try {
            Process process = new ProcessBuilder("./prediccion_cpp", String.valueOf(valor)).start();
            BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
            return Double.parseDouble(reader.readLine());
        } catch (Exception e) {
            e.printStackTrace();
            return 0.0;
        }
    }
}
```

3. MODELOS (MODELS): REPRESENTACIÓN Y PREDICCIÓN DE PRODUCTOS

Clases de Productos (Producto, ProductoAlimenticio, ProductoElectronico):

Producto.java: Es la clase base que define los atributos comunes para todos los productos, como nombre, precio, y cantidad. Al ser la clase padre, permite una estructura uniforme para todos los tipos de productos, facilitando la extensión y el manejo de productos en la aplicación.

```
package models;

import java.io.Serializable;

public class Producto implements Serializable {
    private String nombre;
    private double precioBase;
    private int cantidadDisponible;

    public Producto(String nombre, double precioBase, int cantidadDisponible) {
        this.nombre = nombre;
        this.precioBase = precioBase;
        this.cantidadDisponible = cantidadDisponible;
    }

    public String getNombre() { return nombre; }
    public double getPrecioBase() { return precioBase; }
    public int getCantidadDisponible() { return cantidadDisponible; }

    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setPrecioBase(double precioBase) { this.precioBase = precioBase; }
    public void setCantidadDisponible(int cantidadDisponible) {
        this.cantidadDisponible = cantidadDisponible;
    }

    @Override
    public String toString() {
        return nombre + " - Precio: " + precioBase + " - Cantidad: " + cantidadDisponible;
    }
}
```

3. MODELOS (MODELS): REPRESENTACIÓN Y PREDICCIÓN DE PRODUCTOS

ProductoAlimenticio.java y
ProductoElectronico.java: Estas clases heredan
de Producto y añaden características
específicas. Por ejemplo, ProductoAlimenticio
podría tener atributos adicionales como fecha de
vencimiento o composición nutricional, mientras
que ProductoElectronico podría incluir garantía y
marca. Esto permite a la aplicación gestionar
diferentes tipos de productos sin alterar la
estructura principal.

```
package models;

import java.time.LocalDate;

public class ProductoAlimenticio extends Producto {
    private LocalDate fechaExpiracion;

    public ProductoAlimenticio(String nombre, double precioBase, int cantidadDisponible, LocalDate fechaExpiracion) {
        super(nombre, precioBase, cantidadDisponible);
        this.fechaExpiracion = fechaExpiracion;
    }

    // Getter y Setter para fechaExpiracion
    public LocalDate getFechaExpiracion() { return fechaExpiracion; }
    public void setFechaExpiracion(LocalDate fechaExpiracion) {
        this.fechaExpiracion = fechaExpiracion;
    }

    @Override
    public String toString() {
        return super.toString() + " - Fecha de Expiración: " + fechaExpiracion;
    }
}

package models;

public class ProductoElectronico extends Producto {
    private int garantiaMeses;

    public ProductoElectronico(String nombre, double precioBase, int cantidadDisponible, int garantiaMeses) {
        super(nombre, precioBase, cantidadDisponible);
        this.garantiaMeses = garantiaMeses;
    }

    // Getter y Setter para garantiaMeses
    public int getGarantiaMeses() { return garantiaMeses; }
    public void setGarantiaMeses(int garantiaMeses) { this.garantiaMeses = garantiaMeses; }

    @Override
    public String toString() {
        return super.toString() + " - Garantía: " + garantiaMeses + " meses";
    }
}
```

2. CONTROLADOR VENTASCONTROLLER.JAVA (LÓGICA DE LA INTERFAZ)

Propósito: VentasController.java conecta la lógica de la aplicación con la interfaz definida en VentasView.fxml. Maneja la interacción del usuario, como registrar nuevos productos y visualizar resultados. Detalles del Código: Variables de la UI (Interfaz de Usuario): Declaraciones como nombreProducto, precioProducto, y cantidadProducto están vinculadas a elementos de la interfaz en VentasView.fxml. Estas variables permiten capturar la entrada del usuario (nombre, precio y cantidad de los productos). Método registrarProducto: Este método se ejecuta cuando el usuario ingresa datos y los confirma. Recoge la información introducida en los campos de texto (nombre, precio, y cantidad), crea un nuevo objeto Producto, y lo agrega a la lista productos. Esto permite mantener un registro de los productos añadidos por el usuario. Listas y ObservableList: Utiliza ObservableList<String> productos, que es una lista reactiva en JavaFX. Cuando se agregan productos, la interfaz gráfica se actualiza automáticamente para mostrar el nuevo producto en la lista de productos.

4. VISTA (VENTASVIEW.FXML): DISEÑO DE LA INTERFAZ

Propósito: Este archivo define la interfaz gráfica usando el lenguaje FXML. Proporciona los elementos visuales que el usuario ve e interactúa en la aplicación. Detalles de la Interfaz: Campos de Entrada: Incluye TextField para capturar el nombre, precio, y cantidad del producto. Estos campos están vinculados a las variables correspondientes en VentasController.java. Lista de Productos (ListView): Muestra todos los productos que el usuario ha registrado, proporcionando una representación visual de los productos almacenados en la lista productos. Botones: Incluye botones que disparan eventos, como registrar un producto y predecir el precio. Estos botones están vinculados a métodos en el controlador VentasController para ejecutar acciones específicas cuando el usuario interactúa con ellos.

```
package controllers;

import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import models.Producto;
import models.PrediccionPrecios;

public class VentasController {

    @FXML private TextField nombreProducto, precioProducto, cantidadProducto, ventasInput;
    @FXML private ListView<String> listaProductos;
    @FXML private Label resultadoPrediccion;

    private ObservableList<String> productos = FXCollections.observableArrayList();

    @FXML public void registrarProducto() {
        String nombre = nombreProducto.getText();
        double precio = Double.parseDouble(precioProducto.getText());
        int cantidad = Integer.parseInt(cantidadProducto.getText());
        productos.add(nombre + " - Precio: " + precio + " - Cantidad: " + cantidad);
    }

    @FXML public void predecirPrecio() {
        double ventas = Double.parseDouble(ventasInput.getText());
        double prediccion = PrediccionPrecios.predecir(ventas);
        resultadoPrediccion.setText("Predicción: " + prediccion);
    }
}
```