

PLAN DE TRABAJO DEL ESTUDIANTE

1. INFORMACIÓN GENERAL

Apellidos y Nombres:	Gandy William Humiri Quispe	ID:1546329@senati.pe
Dirección Zonal/CFP:	Tacna/moquegua	
Carrera:	Ingenieria de SoftwareconInteligenciaArtificial	
Curso/ Mód. Formativo	ARTIFICIAL INTELLIGENCE WITH MACHINE LEARNING IN JAVA (ORACLE)	
Tema del Trabajo:	Informe	

Entregable 1: Documento explicativo sobre el uso de la IA en el sistema de recomendación.

Tarea 01: Uso de la IA

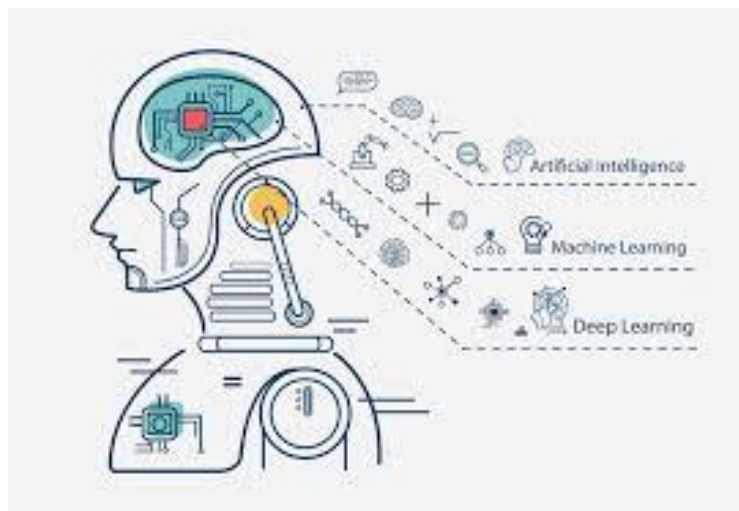
1 Ejemplo Y Aprendizaje Automatico en Recomendaciones

Un sistema de recomendacion utiliza historial de peliculas vistas (datos) y calificaciones para sugerir otras peliculas.

Ejemplo escribe un pseudocódigo para un sistema de filtradocolaborativo.

Encuentra usuarios con gustos similares (vecinos)

Recomienda peliculas populares entre los vecinos



2 Diferencia entre Aprendizaje Supervisado y No Supervisado

Supervisado Predice si te gustará una película (si/no) basándose en calificaciones pasadas.

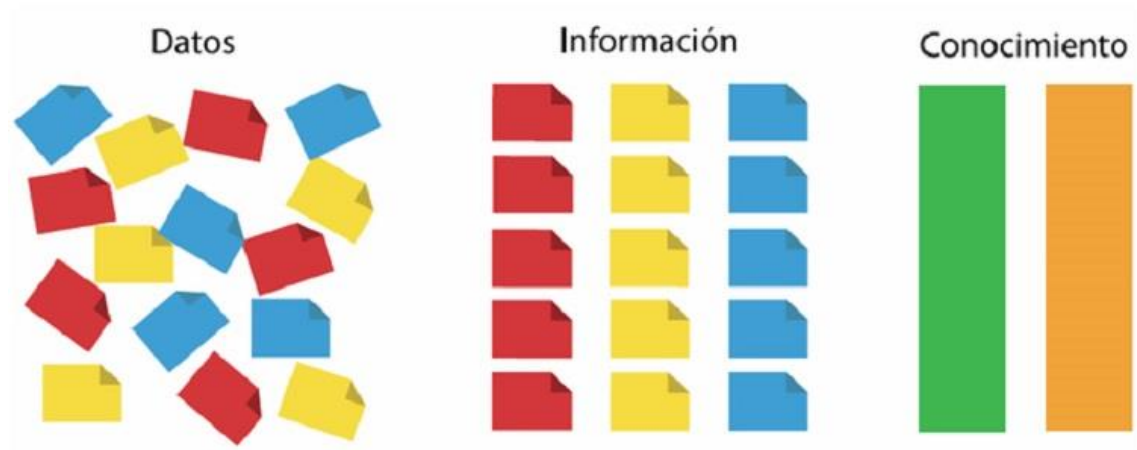
No Supervisado Agrupa usuarios por géneros preferidos.

Ejemplo un conjunto de datos con calificaciones de películas, identifica si usarias supervisado o no supervisado

3 Diferencia entre Datos e Informacion:

Datos "Usuario X vio la pelicula Y a las 8:00 PM"

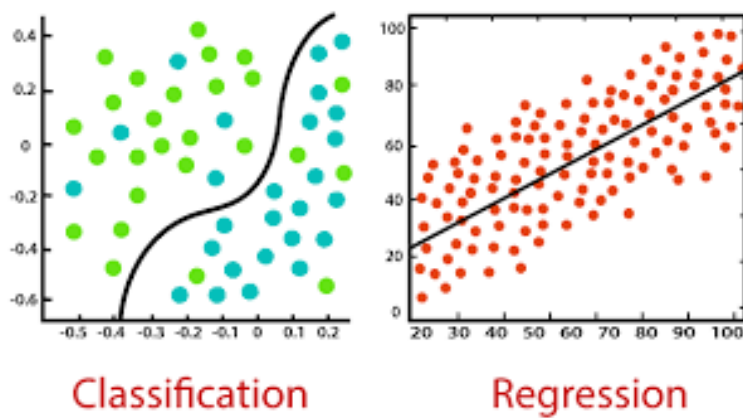
Información "El usuario X prefiere ver comedias por la noche"



4 Clasificacion vs Regresion:

Clasificacion: Predecir si un usuario disfrutará una pelicula (si/no)

Regresion: Estimar la calificacion esperada de una pelicula (de 1 a 5 estrellas).

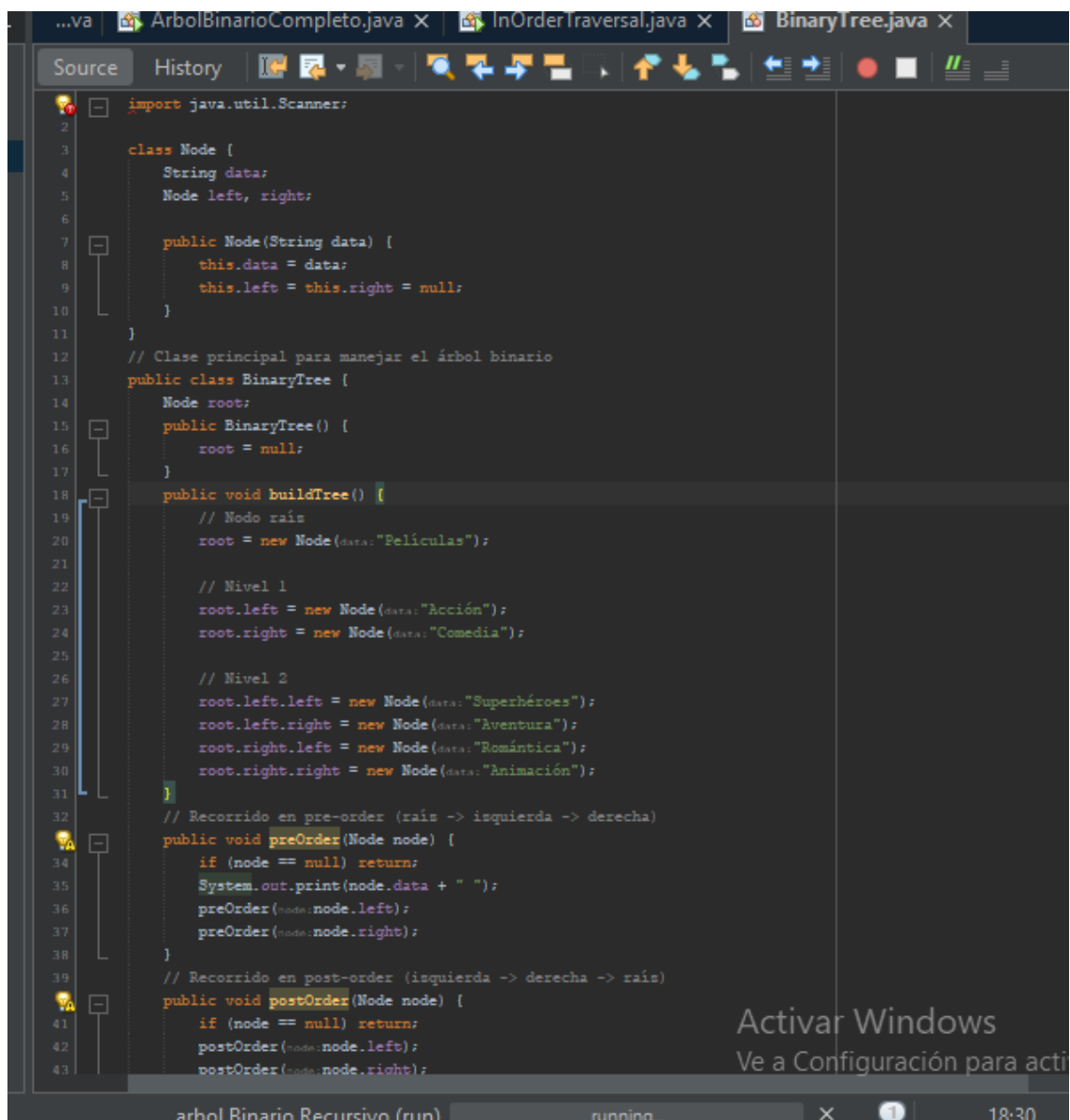


Tarea 02: Recursividad en Árboles

Aquí Ejemplo – Arbol Binario Recursivo:

Requerimientos de la Tarea

1. **Usar recursividad:** Definir los nodos y relaciones en un árbol binario.
2. **Definir y crear la estructura del árbol.**
3. **Implementar el recorrido pre-order traversal.**
4. **Implementar el recorrido post-order traversal.**



```
import java.util.Scanner;

class Node {
    String data;
    Node left, right;

    public Node(String data) {
        this.data = data;
        this.left = this.right = null;
    }
}

// Clase principal para manejar el árbol binario
public class BinaryTree {
    Node root;

    public BinaryTree() {
        root = null;
    }

    public void buildTree() {
        // Nodo raíz
        root = new Node(data: "Películas");

        // Nivel 1
        root.left = new Node(data: "Acción");
        root.right = new Node(data: "Comedia");

        // Nivel 2
        root.left.left = new Node(data: "Superhéroes");
        root.left.right = new Node(data: "Aventura");
        root.right.left = new Node(data: "Romántica");
        root.right.right = new Node(data: "Animación");
    }

    // Recorrido en pre-order (raíz -> izquierda -> derecha)
    public void preOrder(Node node) {
        if (node == null) return;
        System.out.print(node.data + " ");
        preOrder(node.left);
        preOrder(node.right);
    }

    // Recorrido en post-order (izquierda -> derecha -> raíz)
    public void postOrder(Node node) {
        if (node == null) return;
        postOrder(node.left);
        postOrder(node.right);
    }
}
```

Activar Windows
Ve a Configuración para activar Windows

Arbol Binario Recursivo (run) running... 18:30

```
Source History
30 root.right.right = new Node(data:"Animación");
31 }
32 // Recorrido en pre-order (raíz -> izquierda -> derecha)
33 public void preOrder(Node node) {
34     if (node == null) return;
35     System.out.print(node.data + " ");
36     preOrder(node.left);
37     preOrder(node.right);
38 }
39 // Recorrido en post-order (izquierda -> derecha -> raíz)
40 public void postOrder(Node node) {
41     if (node == null) return;
42     postOrder(node.left);
43     postOrder(node.right);
44     System.out.print(node.data + " ");
45 }
46 public static void main(String[] args) {
47     BinaryTree tree = new BinaryTree();
48     tree.buildTree(); // Construir un árbol binario de ejemplo
49     Scanner scanner = new Scanner(System.in);
50     System.out.println("Árbol de recomendaciones construido:");
51     System.out.println("      Películas");
52     System.out.println("      /      \\");
53     System.out.println("     Acción    Comedia");
54     System.out.println("    /  \\    /  \\");
55     System.out.println(" Superhéroes  Aventura Romántica Animación");
56     System.out.println("Seleccione el tipo de recorrido:");
57     System.out.println("1. Pre-order (Raíz -> Izquierda -> Derecha)");
58     System.out.println("2. Post-order (Izquierda -> Derecha -> Raíz)");
59     int choice = scanner.nextInt();
60     System.out.println("Resultado del recorrido:");
61     if (choice == 1) {
62         tree.preOrder(tree.root);
63     } else if (choice == 2) {
64         tree.postOrder(tree.root);
65     } else {
66         System.out.println("Opción inválida.");
67     }
68
69     scanner.close();
70 }
71 }
72 }
```

Activar Windows
Ve a Configuración para activar

```
run:
Árbol de recomendaciones construido:
      Películas
      /      \
     Acción    Comedia
    /  \    /  \
 Superhéroes  Aventura Romántica Animación

Seleccione el tipo de recorrido:
1. Pre-order (Raíz -> Izquierda -> Derecha)
2. Post-order (Izquierda -> Derecha -> Raíz)
```

Entregable 2: Implementación del Algoritmo de Árbol de Decisión Tarea 3

```
class Node {
    String attribute;
    Map<String, Node> children = new HashMap<>();
    boolean isLeaf;
    String classification;

    public Node(String attribute) {
        this.attribute = attribute;
    }

    public void addChild(String value, Node child) {
        children.put(key: value, value: child);
    }
}

// árbol de decisión
public class DecisionTreeID3 {

    public static double calculateEntropy(Map<String, Integer> counts) {
        int total = counts.values().stream().mapToInt(Integer::intValue).sum();
        double entropy = 0.0;

        for (int count : counts.values()) {
            if (count == 0) continue;
            double probability = (double) count / total;
            entropy -= probability * (Math.log(a: probability) / Math.log(a: 2));
        }

        return entropy;
    }

    public static double calculateInformationGain(List<Map<String, String>> data, String attribute,
        Map<String, Integer> totalCounts = new HashMap<>();
        Map<String, Map<String, Integer>> attributeCounts = new HashMap<>();
        for (Map<String, String> record : data) {
            String targetValue = record.get(key: target);
            String attributeValue = record.get(key: attribute);
            totalCounts.put(key: targetValue, totalCounts.getOrDefault(key: targetValue, defaultValue: 0) + 1);
            attributeCounts.get(key: attributeValue).put(key: targetValue, attributeCounts.get(key: attributeValue).getOrDefault(key: targetValue, defaultValue: 0) + 1);
        }

        double totalEntropy = calculateEntropy(counts: totalCounts);
        double weightedEntropy = 0.0;
        for (Map<String, Integer> subset : attributeCounts.values()) {
            int subsetSize = subset.values().stream().mapToInt(Integer::intValue).sum();
            double subsetEntropy = calculateEntropy(counts: subset);
            weightedEntropy += ((double) subsetSize / data.size()) * subsetEntropy;
        }

        return totalEntropy - weightedEntropy;
    }

    // construir el árbol de decisión
    public static Node buildTree(List<Map<String, String>> data, List<String> attributes, String target) {
        Map<String, Integer> targetCounts = new HashMap<>();
        for (Map<String, String> record : data) {
            targetCounts.put(key: record.get(key: target), targetCounts.getOrDefault(key: record.get(key: target), defaultValue: 0) + 1);
        }
        if (targetCounts.size() == 1) {
            Node leaf = new Node(attribute: null);
            leaf.isLeaf = true;
            return leaf;
        }
        // seleccionar el atributo con mayor ganancia de información
        String bestAttribute = null;
        double bestGain = 0.0;
        for (String attribute : attributes) {
            double gain = calculateInformationGain(data, attribute, target);
            if (gain > bestGain) {
                bestGain = gain;
                bestAttribute = attribute;
            }
        }
        if (bestAttribute == null) {
            // si no se encuentra un atributo con ganancia, se selecciona el primer atributo
            bestAttribute = attributes.get(0);
        }
        Node node = new Node(attribute: bestAttribute);
        List<String> remainingAttributes = new ArrayList<>(attributes);
        remainingAttributes.remove(bestAttribute);
        Map<String, Node> children = new HashMap<>();
        for (Map<String, String> record : data) {
            String attributeValue = record.get(key: bestAttribute);
            Node child = buildTree(data: data, attributes: remainingAttributes, target: target);
            children.put(key: attributeValue, value: child);
        }
        node.children = children;
        return node;
    }
}
```

```
return entropy;
}

public static double calculateInformationGain(List<Map<String, String>> data, String attribute, String target) {
    Map<String, Integer> totalCounts = new HashMap<>();
    Map<String, Map<String, Integer>> attributeCounts = new HashMap<>();
    for (Map<String, String> record : data) {
        String targetValue = record.get(key: target);
        String attributeValue = record.get(key: attribute);
        totalCounts.put(key: targetValue, totalCounts.getOrDefault(key: targetValue, defaultValue: 0) + 1);
        attributeCounts.get(key: attributeValue).put(key: targetValue, attributeCounts.get(key: attributeValue).getOrDefault(key: targetValue, defaultValue: 0) + 1);
    }

    double totalEntropy = calculateEntropy(counts: totalCounts);
    double weightedEntropy = 0.0;
    for (Map<String, Integer> subset : attributeCounts.values()) {
        int subsetSize = subset.values().stream().mapToInt(Integer::intValue).sum();
        double subsetEntropy = calculateEntropy(counts: subset);
        weightedEntropy += ((double) subsetSize / data.size()) * subsetEntropy;
    }

    return totalEntropy - weightedEntropy;
}

// construir el árbol de decisión
public static Node buildTree(List<Map<String, String>> data, List<String> attributes, String target) {
    Map<String, Integer> targetCounts = new HashMap<>();
    for (Map<String, String> record : data) {
        targetCounts.put(key: record.get(key: target), targetCounts.getOrDefault(key: record.get(key: target), defaultValue: 0) + 1);
    }
    if (targetCounts.size() == 1) {
        Node leaf = new Node(attribute: null);
        leaf.isLeaf = true;
        return leaf;
    }
    // seleccionar el atributo con mayor ganancia de información
    String bestAttribute = null;
    double bestGain = 0.0;
    for (String attribute : attributes) {
        double gain = calculateInformationGain(data, attribute, target);
        if (gain > bestGain) {
            bestGain = gain;
            bestAttribute = attribute;
        }
    }
    if (bestAttribute == null) {
        // si no se encuentra un atributo con ganancia, se selecciona el primer atributo
        bestAttribute = attributes.get(0);
    }
    Node node = new Node(attribute: bestAttribute);
    List<String> remainingAttributes = new ArrayList<>(attributes);
    remainingAttributes.remove(bestAttribute);
    Map<String, Node> children = new HashMap<>();
    for (Map<String, String> record : data) {
        String attributeValue = record.get(key: bestAttribute);
        Node child = buildTree(data: data, attributes: remainingAttributes, target: target);
        children.put(key: attributeValue, value: child);
    }
    node.children = children;
    return node;
}
```

```

Source History
public static Node buildTree(List<Map<String, String>> data, List<String> attributes, String target) {
    Map<String, Integer> targetCounts = new HashMap<>();
    for (Map<String, String> record : data) {
        targetCounts.put(key: record.get(key: target), targetCounts.getOrDefault(key: record.get(key: target), defaultValue: 0) + 1);
    }
    if (targetCounts.size() == 1) {
        Node leaf = new Node(attribute: null);
        leaf.isLeaf = true;
        leaf.classification = targetCounts.keySet().iterator().next();
        return leaf;
    }
    if (attributes.isEmpty()) {
        Node leaf = new Node(attribute: null);
        leaf.isLeaf = true;
        leaf.classification = targetCounts.entrySet().stream().max(comparator: Map.Entry.comparingByValue()).get().getKey();
        return leaf;
    }
    String bestAttribute = null;
    double bestGain = -1;
    for (String attribute : attributes) {
        double gain = calculateInformationGain(data, attribute, target);
        if (gain > bestGain) {
            bestGain = gain;
            bestAttribute = attribute;
        }
    }
    Node root = new Node(attribute: bestAttribute);
    Map<String, List<Map<String, String>>> partitions = new HashMap<>();
    for (Map<String, String> record : data) {
        String attributeValue = record.get(key: bestAttribute);
        partitions.putIfAbsent(key: attributeValue, new ArrayList<>());
        partitions.get(key: attributeValue).add(record);
    }
}

```

Output X

>>

Activ

Va 3 6

```

Source History
Node root = new Node(attribute: bestAttribute);
Map<String, List<Map<String, String>>> partitions = new HashMap<>();
for (Map<String, String> record : data) {
    String attributeValue = record.get(key: bestAttribute);
    partitions.putIfAbsent(key: attributeValue, new ArrayList<>());
    partitions.get(key: attributeValue).add(record);
}
List<String> remainingAttributes = new ArrayList<>(attributes);
remainingAttributes.remove(attribute: bestAttribute);
for (Map.Entry<String, List<Map<String, String>>> entry : partitions.entrySet()) {
    Node child = buildTree(data: entry.getValue(), attributes: remainingAttributes, target);
    root.addChild(key: entry.getKey(), child);
}
return root;
}
public static void printTree(Node node, String prefix) {
    if (node.isLeaf) {
        System.out.println(prefix + "-> " + node.classification);
        return;
    }
    for (Map.Entry<String, Node> entry : node.children.entrySet()) {
        System.out.println(prefix + node.attribute + " = " + entry.getKey());
        printTree(node: entry.getValue(), prefix + "    ");
    }
}
public static String recommend(Node tree, Map<String, String> inputs) {
    Node current = tree;
    while (!current.isLeaf) {
        String attribute = current.attribute;
        String value = inputs.get(key: attribute);
        if (!current.children.containsKey(key: value)) {
            return "No hay suficiente información para hacer una recomendación.";
        }
        current = current.children.get(key: value);
    }
    return current.classification;
}

```

Out X

```
Source History
95     return;
96 }
97 for (Map.Entry<String, Node> entry : node.children.entrySet()) {
98     System.out.println(prefix + node.attribute + " = " + entry.getKey());
99     printTree(node, entry.getValue(), prefix + " ");
100 }
101 }
102 public static String recommend(Node tree, Map<String, String> inputs) {
103     Node current = tree;
104     while (!current.isLeaf) {
105         String attribute = current.attribute;
106         String value = inputs.get(key: attribute);
107         if (!current.children.containsKey(key: value)) {
108             return "No hay suficiente información para hacer una recomendación.";
109         }
110         current = current.children.get(key: value);
111     }
112     return current.classification;
113 }
114 public static void main(String[] args) {
115     // Conjunto de datos sobre películas
116     List<Map<String, String>> data = new ArrayList<>();
117     data.add(m: Map.of(k1:"Género", v1:"Acción", k2:"Calificación", v2:"Alta", k3:"Ver", v3:"Sí"));
118     data.add(m: Map.of(k1:"Género", v1:"Acción", k2:"Calificación", v2:"Baja", k3:"Ver", v3:"No"));
119     data.add(m: Map.of(k1:"Género", v1:"Comedia", k2:"Calificación", v2:"Alta", k3:"Ver", v3:"Sí"));
120     data.add(m: Map.of(k1:"Género", v1:"Comedia", k2:"Calificación", v2:"Baja", k3:"Ver", v3:"No"));
121     data.add(m: Map.of(k1:"Género", v1:"Drama", k2:"Calificación", v2:"Alta", k3:"Ver", v3:"Sí"));
122     // Atributos
123     List<String> attributes = List.of(k1:"Género", k2:"Calificación");
124     String target = "Ver";
125     Node tree = buildTree(data, attributes, target);
126     System.out.println(x: "Árbol de Decisión:");
127 }
128 }
129 }
130 }
131 }
132 }
133 }
134 }
135 }
136 }
137 }
```

```
124     String target = "Ver";
125     Node tree = buildTree(data, attributes, target);
126     System.out.println(x: "Árbol de Decisión:");
127     printTree(node: tree, prefix: "");
128     Map<String, String> userInputs = Map.of(
129         k1:"Género", v1:"Acción",
130         k2:"Calificación", v2:"Alta"
131     );
132
133     String recommendation = recommend(tree, inputs: userInputs);
134     System.out.println("\nRecomendación: " + recommendation);
135 }
136 }
137 }
```

```
Output X
JavaApplication20 (debug) X Debugger Console X
debug:
Árbol de Decisión:
Calificación = Alta
-> Sí
Calificación = Baja
-> No

Recomendación: Sí
BUILD SUCCESSFUL (total time: 1 second)
```