Desarrollo de un Sistema Distribuido Para Procesamiento de Datos Utilizando el Algoritmo de Árboles de Decisión

Estudiantes

Joel Jhotan Chavez Chico

Franklin Espinoza Pari

Universidad Nacional de Ingeniería
Facultad de Ciencias

Contacto
franklin.espinoza.p@uni.pe
joel.chavez.c@uni.pe

Curso

CC4PA Programación Concurrente y Distribuida

Examen Parcial

Resumen

En este informe se presenta el desarrollo detallado de un sistema distribuido para el procesamiento de datos mediante el algoritmo de árboles de decisión. El sistema se compone de un Productor y

múltiples Nodos, los cuales colaboran en el procesamiento de tareas de manera paralela y concurrente. La comunicación entre el Productor y los Nodos se lleva a cabo mediante el uso de Sockets, asegurando una conexión robusta y confiable. Para facilitar la serialización y deserialización de conjuntos de datos durante la comunicación, se emplea la librería Gson. Se describe en profundidad la arquitectura diseñada para el sistema, destacando la eficiencia en la transmisión de datos y la escalabilidad en entornos de cluster de datos. Además, se presenta un diagrama de protocolo que ilustra el flujo de comunicación entre los componentes del sistema. Se proporciona una explicación detallada del desarrollo del Productor y los Nodos, incluyendo la implementación de la comunicación por sockets y el uso de la librería Gson para el manejo de datos. Se discuten las lecciones aprendidas durante el desarrollo, y se identifican áreas de mejora para futuras iteraciones del sistema distribuido.

Palabras clave: árboles de decisión, sistema distribuido, procesamiento paralelo, comunicación por sockets, Gson, serialización de datos, deserialización de datos.

Contenido

1 Introducción

El procesamiento de grandes volúmenes de datos de manera eficiente y escalable es un desafío fundamental en el ámbito de la ciencia de datos y la inteligencia artificial. En este contexto, los sistemas distribuidos juegan un papel crucial al permitir la ejecución de tareas en paralelo y concurrente en múltiples nodos, lo que conduce a un procesamiento más rápido y una mayor capacidad de respuesta.

El presente informe describe el desarrollo de un sistema distribuido para el procesamiento de datos utilizando el algoritmo de árboles de decisión. Este sistema se compone de un Productor y varios Nodos, donde el Productor envía tareas a los Nodos para su procesamiento en paralelo y concurrente. La comunicación entre el Productor y los Nodos se realiza a través de sockets, garantizando una conexión robusta y confiable. Además, se emplea la librería Gson para la serialización y deserialización de conjuntos de datos durante la comunicación, facilitando así el intercambio de información entre los componentes del sistema.

El objetivo principal de este proyecto es desarrollar un sistema distribuido eficiente y escalable que permita el procesamiento rápido de datos utilizando árboles de decisión. Se busca no solo implementar un sistema funcional, sino también comprender en profundidad los desafíos y consideraciones involucrados en el desarrollo de sistemas distribuidos para el procesamiento de datos.

A lo largo de este informe, se presentará en detalle la arquitectura diseñada para el sistema, se describirá el protocolo de comunicación utilizado, y se proporcionarán explicaciones detalladas sobre el desarrollo del Productor y los Nodos. Además, se discutirán las lecciones aprendidas durante el proceso de desarrollo y se identificarán posibles áreas de mejora para futuras iteraciones del sistema distribuido.

2 Marco Teórico

Árboles de Decisión

Los árboles de decisión son modelos de aprendizaje automático utilizados para la clasificación. Estos modelos representan un conjunto de reglas de decisión en forma de árbol, donde cada nodo interno representa una característica del conjunto de datos, cada rama representa una decisión basada en esa característica, y cada hoja representa la clasificación final.

Sockets

El sistema operativo incluye el recurso de comunicación por entre procesos (IPC) de Berkeley Software Distribution (BSD) conocido como *sockets {IBM}*. Los sockets son conductos de comunicación que posibilitan el intercambio de datos entre procesos independientes tanto a nivel local como en redes. Cada socket representa un extremo de un canal de comunicación bidireccional. En el sistema operativo, los sockets tienen las características siguientes:

1. Un socket existe mientras un proceso lo esté usando.
2. Los sockets son referenciados por descriptores de archivo y se comportan como dispositivos especiales de caracteres. Puedes leer, escribir y seleccionar acciones en ellos usando las funciones apropiadas.
3. Puedes crear sockets en grupos, asignarles nombres o usarlos para conectarte con otros sockets en una red, aceptando conexiones entrantes o enviando mensajes.

Comunicación vía Sockets

El mecanismo de comunicación vía sockets tiene los siguientes pasos básicamente:

1. Creación de sockets: El servidor crea un socket con un nombre y espera conexiones entrantes. El cliente crea un socket sin nombre.
2. Solicitud de conexión: El cliente realiza una solicitud de conexión al socket del servidor.
3. Establecimiento de conexión: El servidor acepta la solicitud de conexión del cliente y establece una conexión, lo que permite la comunicación bidireccional entre el cliente y el servidor.
4. Comunicación: Una vez establecida la conexión, tanto el cliente como el servidor pueden enviar y recibir datos a través de los sockets.
5. Cierre de conexión: Cuando la comunicación ha finalizado, ya sea porque se ha completado la tarea o por otros motivos, se cierra la conexión. Esto libera los recursos asociados con los sockets y finaliza la comunicación entre el cliente y el servidor.

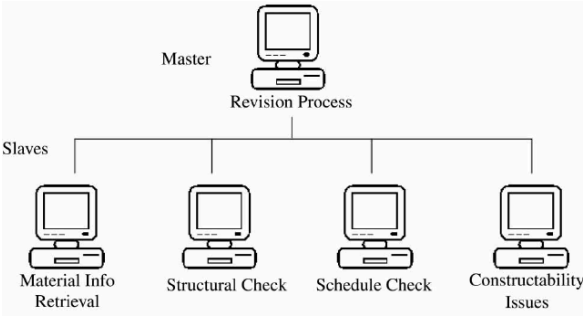
Serialización y Deserialización JSON

La serialización es el proceso de convertir el estado de un objeto, es decir, los valores de sus propiedades, en un formulario que se puede almacenar o transmitir. El formulario serializado no incluye información sobre los métodos asociados de un objeto. La deserialización reconstruye un objeto a partir del formulario serializado.
JSON fue diseñado para ser un lenguaje de intercambio de datos comprensible para los seres humanos, fácil de interpretar y usar por las computadoras. JSON es directamente soportado por el lenguaje de programación JavaScript y es la mejor opción para la serialización/deserialización e intercambio de datos.

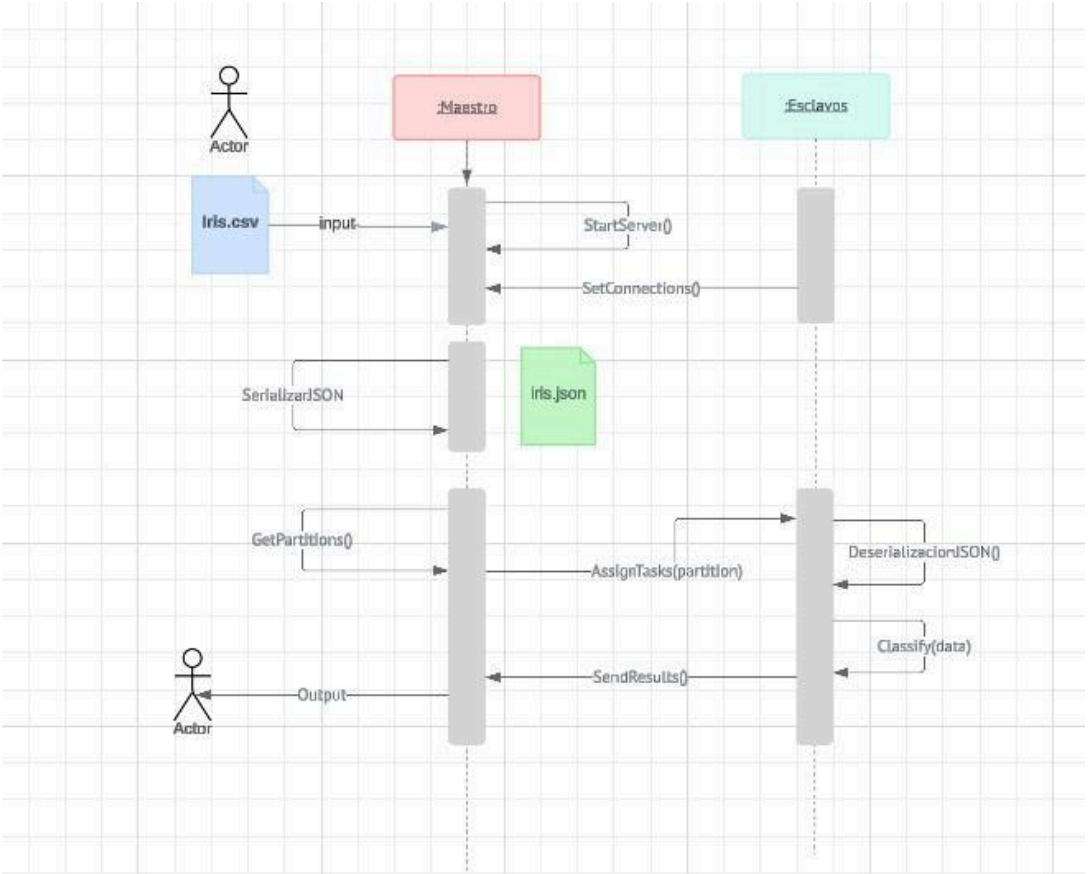
3 Arquitectura Maestro-Esclavo

En nuestra arquitectura diseñada, implementamos un modelo de arquitectura maestro-esclavo para coordinar las actividades de procesamiento distribuido. En este modelo, un nodo maestro centraliza la coordinación de uno o más nodos esclavos. El maestro asigna tareas a los esclavos y recopila los resultados para su procesamiento final.

La interacción en este modelo está más centrada en una dirección, con el maestro coordinando las actividades y los esclavos realizando tareas específicas en respuesta a las instrucciones del maestro. En nuestro programa, el maestro particiona el conjunto de datos iris y distribuye estas particiones entre los nodos esclavos para su procesamiento. Adicionalmente el maestro le envía el modelo, donde cada nodo esclavo realizará la predicción con el modelo del árbol de predicción sobre la partición de datos que fueron asignados y este devolverá los resultados al maestro.



4 Diagrama de Protocolo



5 Metodología

Recopilación y Preprocesamiento de datos:

Para demostrar el funcionamiento de los árboles de decisión hemos seleccionado el conjunto de datos Iris. Este conjunto de datos contiene información sobre tres especies de Iris (Setosa, Versicolor y Virginica) y cuatro características medidas para cada muestra: longitud y anchura del pétalo y sépalo. El objetivo es predecir la especie de iris a partir de estas características utilizando un árbol de decisión. Durante el desarrollo del programa vamos a explicar a más detalle sobre este conjunto de datos y como enviamos, recibimos, y predecimos aquellos datos.

Manejo de Datos y Procesamiento Distribuido

Hasta ahora, hemos delineado la idea fundamental de que el maestro enviará tanto el modelo como las particiones de datos a cada nodo esclavo. Sin embargo, surge una pregunta crucial: ¿cómo logramos enviar estos datos de manera eficiente y efectiva? Para resolver este desafío, implementamos la serialización y deserialización JSON.

El proceso comienza con el maestro, quien serializa los datos a un formato JSON. Esta serialización convierte los datos y el modelo en una representación legible por máquina que puede ser fácilmente transmitida a través de la red. Una vez serializados, estos datos son particionados y enviados a cada nodo esclavo.

En el lado del nodo esclavo, se lleva a cabo el proceso inverso. El nodo recibe los datos serializados, y a través de la deserialización JSON, los convierte nuevamente en una estructura de datos utilizable. Este proceso de deserialización es crucial para garantizar que los nodos esclavos puedan trabajar con los datos recibidos de manera efectiva.

Implementación de la Arquitectura Maestro-Esclavo

Para la implementación de la arquitectura maestro-esclavo en nuestro sistema, desarrollamos dos componentes principales: el nodo maestro y los nodos esclavos. Cada uno de estos componentes desempeña un papel crucial en la coordinación y ejecución del procesamiento distribuido. A continuación, se detalla cómo se implementaron estos componentes:

Nodo Esclavo (Cliente)

El nodo esclavo, representado por la clase Client, desempeña el papel de recibir tareas del nodo maestro, realizar el procesamiento de datos asignado y devolver los resultados al nodo maestro. Aquí están los principales aspectos de su implementación:

1. Conexión con el Nodo Maestro: El nodo esclavo establece una conexión TCP/IP con el nodo maestro utilizando la dirección IP y el puerto correspondiente.
2. Recepción de Datos: Una vez establecida la conexión, el nodo esclavo recibe los datos y el modelo serializados del nodo maestro.
3. Procesamiento de Datos: Utilizando el modelo de árbol de decisión proporcionado, el nodo esclavo realiza la predicción sobre las particiones de datos asignadas.
4. Envío de Resultados: Después de realizar las predicciones, el nodo esclavo serializa los resultados en formato JSON y los envía de vuelta al nodo maestro a través de la conexión establecida.
5. Cierre de la Conexión: Una vez completado el procesamiento de datos, el nodo esclavo cierra la conexión con el nodo maestro.

Nodo Maestro (Servidor)

El nodo maestro, representado por la clase Server, tiene la responsabilidad de coordinar las actividades de procesamiento distribuido y recopilar los resultados de los nodos esclavos. A continuación se detallan los aspectos principales de su implementación:

1. Espera de Conexiones: El nodo maestro escucha las conexiones entrantes de los nodos esclavos utilizando un socket de servidor.
2. Asignación de Tareas: Una vez que se establece la conexión con los nodos esclavos, el nodo maestro asigna las particiones de datos y el modelo a cada nodo esclavo para su procesamiento.

3. Recolección de Resultados: Después de que los nodos esclavos han completado el procesamiento de datos, el nodo maestro recibe los resultados de cada nodo esclavo a través de la conexión establecida.
4. Almacenamiento de Resultados: Los resultados recibidos del nodo esclavo se almacenan en archivos de texto en el sistema de archivos local para su posterior análisis y procesamiento.
5. Finalización de la Conexión: Una vez completada la recopilación de resultados, el nodo maestro cierra la conexión con los nodos esclavos y finaliza su ejecución.

Tecnologías Utilizadas

Lenguajes de Programación:
- LP1: Java: Utilizamos el lenguaje de programación Java para desarrollar tanto el Productor como los Nodos en nuestro sistema distribuido. Java proporciona una plataforma sólida y confiable para el desarrollo de aplicaciones distribuidas.
- LP2: Python: Para el desarrollo del cliente que interactúa con el sistema distribuido, utilizamos Python. Python es conocido por su facilidad de uso y su amplia variedad de bibliotecas que facilitan el desarrollo de aplicaciones de red.

Entorno de Desarrollo Integrado (IDE):
- NetBeans: Utilizamos NetBeans como nuestro entorno de desarrollo integrado (IDE) para el desarrollo en Java. NetBeans ofrece características avanzadas de desarrollo y es ampliamente utilizado en el desarrollo de aplicaciones Java.
- Terminal: Para el desarrollo en Python, utilizamos el terminal estándar de Python. Esto nos permite escribir y ejecutar código Python de manera eficiente directamente desde la línea de comandos.

Bibliotecas y Dependencias:
Para el LP1: Java:
- Bibliotecas nativas usadas recurrentemente en el curso para la manipulación de sockets como java.net asimismo para la lectura de los flujos de stream se usaron las clases BufferedReader y OutputStreamWriter, proporcionando métodos para la lectura eficiente de datos desde un flujo de entrada y la escritura de datos en un flujo de salida respectivamente. Además, se emplearon estructuras de datos sencillas como ArrayList y List para el almacenamiento y manipulación de colecciones de datos de manera eficiente.
- Gson (Google Gson): Para la serialización y deserialización de datos en formato JSON en Java, utilizamos la biblioteca Gson de Google. Gson simplifica el intercambio de datos entre componentes del sistema distribuido al proporcionar métodos para convertir objetos Java en formato JSON y viceversa.

Para el LP2: Python:
- Socket: Utilizada para la comunicación entre los diferentes componentes del sistema distribuido mediante sockets.
- json: Utilizada para la serialización y deserialización de datos en formato JSON.
- socket: Proporciona una interfaz de bajo nivel para la comunicación de red.
- typing: Proporciona compatibilidad con tipos en Python.

Dependencia Maven para Gson en Java:
```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.10.1</version>
</dependency>
```

Estas tecnologías nos proporcionan una visión general de las herramientas y tecnologías utilizadas en el desarrollo de nuestro sistema distribuido para el procesamiento de datos con el algoritmo de árboles de decisión. Estas herramientas fueron seleccionadas por su idoneidad para el desarrollo de aplicaciones distribuidas y su capacidad para satisfacer los requisitos específicos del proyecto.

Caracteristicas de Laptop usada para el Desarrollo

Colección de productos	Procesadores Intel® Core™ i5 de 10ma Generación
Nombre de código	Productos anteriormente Ice Lake
Segmento vertical	Mobile
Número de procesador	i5-1035G1
Litografía	10 nm

Especificaciones de la CPU

Cantidad de núcleos	4
Total de subprocesos	8
Frecuencia turbo máxima	3.60 GHz
Frecuencia básica del procesador	1.00 GHz

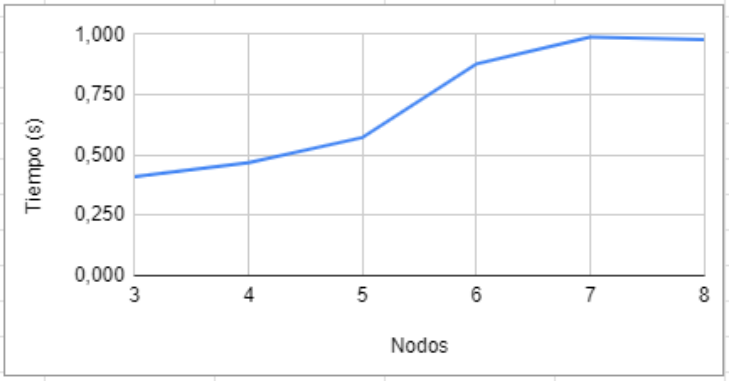
6 Conclusión

Se llevó a cabo un trabajo distribuido utilizando un conjunto de un millón de datos. Posteriormente, ejecutamos el proceso con varios hilos y observamos los tiempos de ejecución. Como se puede apreciar en la figura, el gráfico siguiente muestra lo siguiente para LP1 = LP2 = JAVA y para LP1!=LP2 JAVA y Python:

Cantidad de Datos: 1 Millon

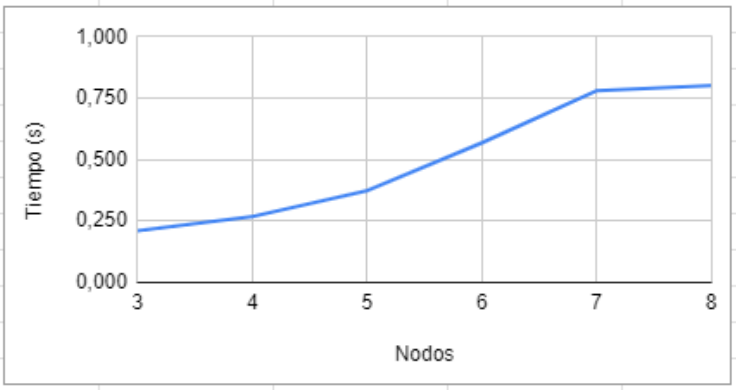
LP1 = LP2 = Java

Nodos productores	tiempo (s)
3	0,408
4	0,466
5	0,571
6	0,877
7	0,989
8	0,978



LP1 = Java LP2 = Python

Nodos productores	tiempo (s)
3	0,208
4	0,266
5	0,371
6	0,567
7	0,779
8	0,801



7 Anexo Código

Server.java

```
public class Server {
    public static final String END = "END";
    private static final int MAX_CONNECTIONS = 2;
    private static List<String[]> data_iris;
    private static String json;
    private static List<String> partitions;

    private static String[] assignTasks(ArrayList<StreamSocket> sockets, ExecutorService pool) {
        FileConverter fc = new FileConverter();
        String filePath =
"/home/jhozzel/NetBeansProjects/TaskAssigner/src/main/java/com/mycompany/taskassigner/resources/iris
.csv";
        data_iris = fc.readCSV(filePath);
        json = fc.convertToStringJson(data_iris);
        partitions = fc.partitionJson(json, MAX_CONNECTIONS);

        List<HandleConnection> connections = new ArrayList<>();

        for (int i = 0; i < MAX_CONNECTIONS; i++) {
            HandleConnection connection = new HandleConnection(sockets.get(i), partitions.get(i), i
+ 1);
            connection.start();
            connections.add(connection);
        }

        // Esperar a que todos los hilos terminen
        for (HandleConnection connection : connections) {
            try {
                connection.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Hacaer el testeo de que tan buena fue la prediccion con los datos reales
        StringBuilder all_predicted = new StringBuilder();
        for (HandleConnection connection : connections) {
            String filePathResult = connection.getFilePath();
            if (filePathResult != null) {
                String result = fc.readResultFile(filePathResult);
                all_predicted.append(result);
            }
        }
        String[] speciesPredicted = all_predicted.toString().split("\n");

        return speciesPredicted;
    }

    private static void listen(int port) {
        try (ServerSocket connection = new ServerSocket(port)) {
            System.out.printf("Server running on port %d\n", port);
            ExecutorService pool = Executors.newCachedThreadPool();
            ArrayList<StreamSocket> sockets = new ArrayList<>();
            int clientes = 0;
            while (clientes  < MAX_CONNECTIONS) {
                System.out.println("Waiting for connections...");
                StreamSocket socket = new StreamSocket(connection.accept());
                System.out.printf("New connection from %s\n", socket.host());
                System.out.println("Cliente: " + (clientes + 1) + " conectado....");
                sockets.add(socket);
                //pool.execute(new HandleConnection(socket));
                clientes++;
            }
```

```
            long startTime = System.currentTimeMillis();

            String[] speciesPredicted = assignTasks(sockets, pool);

            int len_data = data_iris.size();
            int cnt_ok_data = 0;
            for (int i = 0; i < len_data - 1; i++) {
                if (data_iris.get(i)[5].equals(speciesPredicted[i])) {
                    cnt_ok_data++;
                }

            }
            long endTime = System.currentTimeMillis();
            long elapsedTime = endTime - startTime;

            System.out.println("Tiempo transcurrido en milisegundos: " + (double)elapsedTime /
1000.0 + "s");
            double accuracy = (double) cnt_ok_data / len_data;
            System.out.println("=> Tasa de éxito de predicción: " + (accuracy * 100) + " %");
            sockets.clear();
        } catch (BindException e) {
            error("Port %d is not available\n", port);
        } catch (IOException e) {
            e.printStackTrace();
            error("Server crashed\n");
        }
    }

    private static void error(String msg, Object... args) {
        System.err.printf(msg, args);
        System.exit(-1);
    }

    public static void main(String[] args) {
        int PUERTO = 8084;
        try {
            listen(PUERTO);
        } catch(NumberFormatException e) {
            error("Failed parsing port number '%s'\n", args[0]);
        }
    }
}
```

Client.java

```
public class Client {

    public static FileConverter fc = new FileConverter();
    public static String modelPath =
"/home/jhozzel/NetBeansProjects/TaskAssigner/src/main/java/com/mycompany/taskassigner/resources/mode
l.json";
    public static List<Nodo> graph;

    public static String decisionTree(IrisFlor x, int nodeId) {
        Nodo node = graph.get(nodeId);
        if (!node.specie.equals("unknown")) {
            return node.specie;
        }
        double valueFeature;
        switch (node.feature) {
            case "sepal_length":
                valueFeature = x.sepal_length;
                break;
            case "sepal_width":
                valueFeature = x.sepal_width;
                break;
```

```
            case "petal_length":
                valueFeature = x.petal_length;
                break;
            case "petal_width":
                valueFeature = x.petal_width;
                break;
            default:
                throw new IllegalArgumentException("Feature desconocida: " + node.feature);
        }
        if (valueFeature <= node.threshold) {
            return decisionTree(x, node.id_left);
        } else {
            return decisionTree(x, node.id_right);
        }
    }

    public static String classify(String data_json) {
        StringBuilder sb = new StringBuilder();
        List<IrisFlor> flowers = fc.readStringJson(data_json);
        for (IrisFlor flower : flowers) {
            sb.append(decisionTree(flower, 0)).append("\n");
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        graph = fc.readModel(modelPath);

        // Connect to host
        String host = "127.0.0.1";
        int port = 8084;
        try {
            Socket socket = new Socket(host, port);
            System.out.println("Connected to " + host + ":" + port);

            BufferedReader inputReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter outputWriter = new PrintWriter(socket.getOutputStream(), true);

            StringBuilder responseBuilder = new StringBuilder();
            String line;
            while (true) {
                line = inputReader.readLine();
                responseBuilder.append(line).append("\n");
                if (line.equals("]")) {
                    break;
                }
            }
            String data_json = responseBuilder.toString();

            //System.out.println(data_json);

            String classifiedData = classify(data_json);
            outputWriter.println(classifiedData);

            socket.close();
            System.out.println("Connection closed");
        } catch (UnknownHostException e) {
            System.err.println("Unknown host: " + host);
            e.printStackTrace();
        } catch (IOException e) {
            System.err.println("Error connecting to server: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Client.py

```
import json
import socket
from typing import List

# Definición de las clases Flower y Nodo
class Flower:
    def __init__(self, id, sepal_length, sepal_width, petal_length, petal_width, specie):
        self.id = id
        self.sepal_length = sepal_length
        self.sepal_width = sepal_width
        self.petal_length = petal_length
        self.petal_width = petal_width
        self.specie = specie

class Nodo:
    def __init__(self, id, id_left, id_right, feature, threshold, specie):
        self.id = int(id)
        self.id_left = int(id_left)
        self.id_right = int(id_right)
        self.feature = feature
        self.threshold = threshold
        self.specie = specie

# Definición de la función para leer el modelo
graph = []

def read_model():
    global graph
    with open("model.json", "r") as file:
        data = json.load(file)
        for item in data:
            nodo = Nodo(
                id=item['id'],
                id_left=item['id_left'],
                id_right=item['id_right'],
                feature=item['feature'],
                threshold=item['threshold'],
                specie=item['specie']
            )
            graph.append(nodo)

# Función para clasificar los datos
def decision_tree(x: Flower, node_id: int) -> str:
    node = graph[node_id]
    if node.specie != "unknown":
        return node.specie
    value_feature = getattr(x, node.feature)
    if value_feature <= node.threshold:
        return decision_tree(x, node.id_left)
    else:
        return decision_tree(x, node.id_right)

def classify(data_json: str) -> str:
    result = []
    flowers = json.loads(data_json)
    for flower_data in flowers:
        flower = Flower(**flower_data)
        result.append(decision_tree(flower, 0))
    return "\n".join(result)

def main():
    read_model()
    host = "127.0.0.1"
    port = 8084
```

```
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((host, port))
            print(f"Connected to {host}:{port}")

            data_json = ""
            while True:
                chunk = s.recv(1024)
                if not chunk:
                    break
                data_json += chunk.decode("utf-8")
                if "]" in data_json:
                    break  # Salir del bucle cuando se encuentra ']'

            classified_data = classify(data_json)
            print("Data clasificada...")
            print("Enviando al server los resultados...")
            print(classified_data)
            s.sendall(classified_data.encode("utf-8"))

        print("Connection closed")
    except ConnectionRefusedError:
        print("Connection refused. Is the server running?")
    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    main()
```
