Universidad Zaragoza
1542

Trabajo Fin de Grado

Sistema inteligente para la optimización del razonamiento semántico en dispositivos móviles

"An intelligent system for the optimization of semantic reasoning on mobile devices"

Autor/es

Marta Lanau Coronas

Director/es

Fernando Bobillo Ortega

Escuela de Ingeniería y Arquitectura

Año: 2018

Resumen

En nuestras vidas cotidianas es cada vez más frecuente el uso de aplicaciones específicamente diseñadas para su ejecución en dispositivos móviles. Si bien dichas aplicaciones proporcionan servicios útiles, existen muchos ejemplos de escenarios donde la incorporación de tecnologías semánticas podría mejorar la calidad de los servicios y la experiencia del usuario.

El uso de información semántica específicamente representada y de un razonador semántico para inferir nuevo conocimiento implícito plantea ciertos retos cuando el hardware es un dispositivo móvil, con limitaciones en cuanto a capacidad de procesamiento, memoria, batería...

En este proyecto se ha creado una plataforma inteligente para el razonamiento semántico en dispositivos móviles que permite decidir en tiempo de ejecución dónde procesar los datos: si en el propio dispositivo móvil o en un servidor externo.

Para ello, se han utilizado algoritmos de aprendizaje automático que permiten predecir el coste del razonamiento de acuerdo a diferentes criterios y, finalmente, decidir el lugar del razonamiento teniendo en cuenta el hardware del dispositivo móvil.

Índice

1. **Introducción** 1
   1.1 Marco y objetivo del proyecto 1
   1.2 Organización de la memoria 2

2. **Contexto tecnológico** 3
   2.1 Ontologías y razonadores 3
   2.2 Redes Neuronales Artificiales 5
   2.3 Bosques aleatorios 7

3. **Diseño** 8
   3.1 Elección y cálculo de las métricas de una ontología 8
   3.2 Consultas mediante un razonador semántico 8
   3.3 Aprendizaje automático 9
   3.4 Toma de decisiones 9

4. **Implementación y experimentación** 10
   4.1 Aplicación para obtener las métricas de una ontología 10
      4.1.1 Identificación de métricas 10
      4.1.2 Desarrollo de la aplicación 11
      4.1.3 Análisis de los resultados 14
   4.2 Aplicación para consultas con razonador 15
      4.2.1 Elección de razonadores 15
      4.2.2 Desarrollo de la aplicación 16
      4.2.3 Análisis de los resultados y comparativa entre razonadores 22
   4.3 Algoritmo de aprendizaje automático 23
      4.3.1 Redes Neuronales Artificiales 24
      4.3.2 Bosques Aleatorios 27
   4.4 Algoritmo de toma de decisiones 29
   4.5 Sistema Inteligente completo 32

5. **Conclusiones y trabajo futuro** 34

6. **Referencias** 36

1. Introducción

1.1 Marco y objetivo del proyecto

En los últimos años se ha incrementado considerablemente el desarrollo de tecnología móvil. El uso de *smartphones* está completamente integrado en la sociedad y, con ello, el uso de aplicaciones para dispositivos móviles (*apps*). Estas apps podrían mejorar la calidad de sus servicios y la experiencia del usuario mediante la incorporación de razonadores semánticos. Estos son capaces de descubrir conocimiento implícito en un conjunto de información agrupada en lo que se conoce como ontologías, lo que también facilitaría el intercambio y reutilización de información. Un tipo de aplicaciones que se beneficiaría del uso de razonadores serían las proveedoras de servicios según la ubicación del usuario. Podrían recomendar un transporte u otro partiendo de una información inicial sobre los medios y teniendo en cuenta la hora, ubicación o preferencias del usuario (p.ej. transporte público/privado).

El principal entorno donde se ha desarrollado esta herramienta ha sido en PC ya que, por desgracia, el uso de razonadores en dispositivos móviles plantea ciertos retos debido a las limitaciones que supone tener menor capacidad computacional.

El objetivo de este TFG es la creación de una plataforma inteligente para el razonamiento semántico en dispositivos móviles Android, que permita decidir en tiempo de ejecución dónde procesar los datos: si en el propio dispositivo móvil o en un servidor externo con más recursos. Para esta decisión se han utilizado varios algoritmos de aprendizaje automático (“Redes Neuronales Artificiales” y “Bosques Aleatorios”), que tienen en cuenta el consumo en tiempo, batería y memoria durante el razonamiento.

Existen trabajos previos sobre la predicción de tiempo (en PC) y de batería (en móviles), aunque ambos consideran sólo los “Bosques Aleatorios”. Además, ninguno de ellos tiene en cuenta la memoria, ni utiliza varios criterios simultáneamente.

1.2 Organización de la memoria

La memoria está organizada en los siguientes apartados:

- **Contexto tecnológico:** explicación detallada de conceptos y métodos utilizados a lo largo del proyecto.
- **Diseño:** se decide cuáles son las fases requeridas para una correcta implementación y se detallan las necesidades que estas deben cumplir.
- **Implementación y experimentación:** se desarrollan las ideas expuestas en las etapas de diseño, elaborando distintas aplicaciones y algoritmos para conseguir el objetivo final. También se evalúan las pruebas realizadas en el proceso.
- **Conclusiones y trabajo futuro:** se comenta la validez y el alcance de los resultados. También se hace referencia al trabajo futuro en este campo.
- **Referencias**.

2. Contexto tecnológico

A continuación se abordan con detalle algunos conceptos y métodos fundamentales en el desarrollo del proyecto.

2.1 Ontologías y razonadores

Las ontologías son agrupaciones de conocimiento dentro de un dominio de interés. En ellas se describen conceptos pertenecientes a este dominio y las relaciones que se forman entre ellos. Para su modelado se utilizan Lógicas de Descripción (DL), una familia de lenguajes para la representación de conocimiento. Supone uno de los principales fundamentos de OWL 2 Web Ontology Language (informalmente OWL 2), estandarizado por el World Wide Web Consortium (W3C) [1] y utilizado en las ontologías de este proyecto. Es totalmente compatible con *Protégé* [2], un software especializado para su edición. Para la manipulación de ontologías en aplicaciones Java se ha utilizado la biblioteca OWL API 3.4.1 [3].

Las DLs están equipadas por una semántica formal que permite intercambiar ontologías sin ambigüedad respecto a su significado. Además, tienen la capacidad de inferir conocimiento adicional sobre las ontologías, mecanismo conocido como razonamiento. Un razonador semántico es un software capaz de inferir conocimiento dentro de una ontología. Puede verificar si las afirmaciones y definiciones de una ontología son consistentes, así como reconocer qué conceptos se ajustan a las definiciones, entre otras posibles tareas de razonamiento. [4] Uno de los principales servicios que ofrecen los razonadores semánticos es la "clasificación", utilizada en el desarrollo del proyecto. Esta tarea comprueba si una clase es subclase de otra, es decir, computa la jerarquía de clases de la ontología en cuestión.

Las ontologías se componen de clases, propiedades, individuos y axiomas:

- Las **clases** son las ideas básicas para representar el conocimiento de algún dominio. Se especifican las condiciones que un individuo debe satisfacer para pertenecer a una clase determinada. Pueden estar organizadas con una jerarquía de superclase-subclase, también conocido como taxonomía. Las subclases especializan a las superclases.

- Las **propiedades** representan relaciones. Hay dos tipos principales: *Object properties* y *Datatype properties*. Las primeras enlazan dos individuos y pueden ser funcionales, inversas, transitivas, simétricas, reflexivas… Las segundas relacionan individuos con valores de datos.
- Los **individuos** son instancias de las clases. Representan objetos en el dominio en que estamos interesados, pudiendo dos nombres distintos referirse al mismo individuo.
- Los **axiomas** son teoremas que describen relaciones que deben cumplir los elementos de la ontología. Por ejemplo: "*Si A y B son de la clase C, entonces A no es subclase de B*". [5][6]

En la Figura 1 hay un esquema que ejemplifica estos componentes: "*Gemma*" e "*England*" son individuos, "*Person*" y "*Country*" son clases y "*hasPet*" y "*livesInCountry*" propiedades.

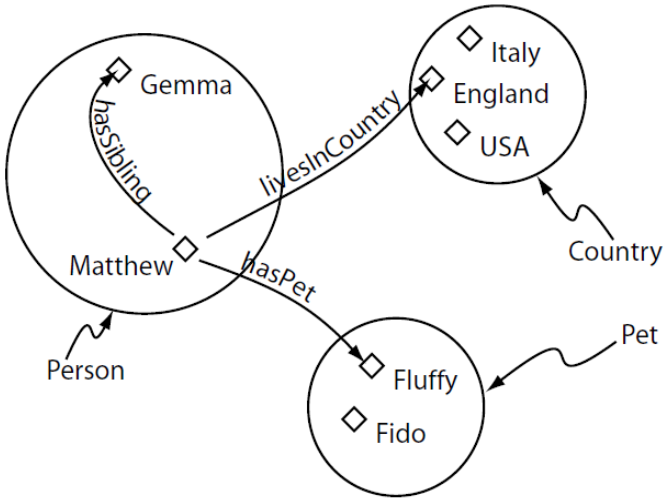

Figura 1. Representación de fragmento de una ontología [6]

Una ontología consiste en un conjunto de axiomas que capturan conocimiento de una situación descrita. Estos axiomas se dividen comúnmente en dos grupos: axiomas asertivos (ABox) y terminológicos (TBox) [4].

- Axiomas ABox: almacenan conocimiento sobre los individuos nombrados, especialmente las clases a los que pertenecen y las relaciones que mantienen. Por ejemplo, "*Fido(pet)*" afirma que Fido es una mascota y "*livesInCountry(Matthew, England)*" afirma que Matthew vive en Inglaterra.

- Axiomas TBox: describen relaciones entre clases o entre propiedades. Dentro de este tipo de axiomas, destacan los *General Concept Inclusion axioms (GCIs)*, que establecen que un concepto es subclase de otro. [7] Por ejemplo, "Mascota ⊂ Animal" quiere decir que todas las mascotas son animales.

Algunas referencias para obtener más información sobre ontologías y DLs son "*Handbook on ontologies*" y "*The Description Logic Handbook*", respectivamente.

2.2 Redes Neuronales Artificiales

"Una Red Neuronal Artificial es un modelo matemático inspirado en el comportamiento biológico de las neuronas y en la estructura del cerebro, y que es utilizada para resolver un amplio rango de problemas. […] Esta también puede ser vista como un sistema inteligente que lleva a cabo tareas de manera distinta a como lo hacen las computadoras actuales." [8]

Este modelo es capaz de aprender a realizar tareas basadas en una experiencia inicial (conocida como entrenamiento) y de crear su propia organización de la información según lo aprendido. En la Figura 2 se muestra un modelo de RNA.

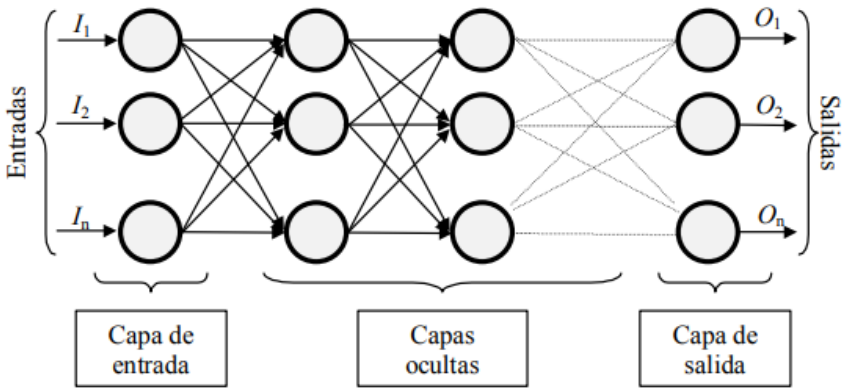

Figura 2. Modelo de Red Neuronal Artificial [9]

Está constituido por un conjunto de neuronas interconectadas agrupadas en distintas capas. Como su nombre indica, los valores son introducidos a la red por la "capa de entrada" y las salidas se obtienen de la "capa de salida", siendo variable el número de "capas ocultas".

La función de entrada a una RNA se calcula a partir del vector de entrada y los pesos, que son modificados a medida que la red va aprendiendo, para así controlar la influencia que tienen las entradas.

Se ha utilizado un aprendizaje supervisado por corrección de error. Este consiste en ajustar los pesos de las conexiones de la red en función de la diferencia entre los valores deseados y los obtenidos a la salida de la red, es decir, en función del error cometido en la salida. [9] El aprendizaje divide los datos en dos conjuntos: el de entrenamiento (80% de la muestra) y el de test (20%). Con el primero se fijan los pesos y, posteriormente, con el segundo se valida si la red neuronal es capaz de resolver los problemas para los que ha sido entrenada.

Para entrenar la red es común utilizar el algoritmo de propagación hacia atrás (*backpropagation*), proceso en el que el error se propaga hacia atrás desde la capa de salida. De esta manera los pesos de las neuronas de las capas ocultas se van modificando durante el entrenamiento, lo cual influye en la función de activación de las neuronas (que se detallará un poco más adelante) y, en consecuencia, en la salida de la neurona. [9]

Para realizar el entrenamiento, es necesario determinar algunos parámetros:

- Tasa de aprendizaje: amortigua el cambio de los valores de los pesos. [10]
- *Momentum*: evita mínimos locales en la propagación hacia atrás ya que introduce un sumando adicional en la actualización de los pesos, el cual tiene en cuenta el cambio que se hizo en la iteración anterior. Adquiere valores entre 0 y 1. [11]
- Función de activación: calcula el estado de actividad de una neurona; transformando la entrada en un valor de 0 (totalmente inactiva) a 1 (activa). [12] La función utilizada ha sido la sigmoide (Figura 3).

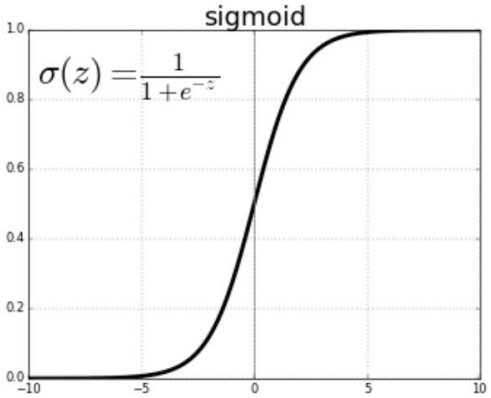

Figura 3. Función sigmoide [12]

En los últimos años, Google ha puesto de moda este tipo de Inteligencia Artificial con proyectos exitosos como AlphaZero o AlphaGo, los cuales consiguen ganar a los humanos jugando al ajedrez y al go, respectivamente.

2.3 Bosques aleatorios

Los bosques aleatorios (del inglés, "*Random forests*") son una combinación de árboles de decisión. Un árbol de decisión es un modelo de predicción utilizado en diversos ámbitos, la Inteligencia Artificial entre ellos. Dado un conjunto de datos, se crean construcciones lógicas que representan una serie de condiciones. Estas ocurren de forma sucesiva y concluyen con la resolución de un problema. [13]

En los bosques aleatorios cada árbol depende de los valores de un vector aleatorio probado independientemente y con la misma distribución para cada uno de estos. [14] En la figura 4 se muestra un ejemplo de Bosque aleatorio.

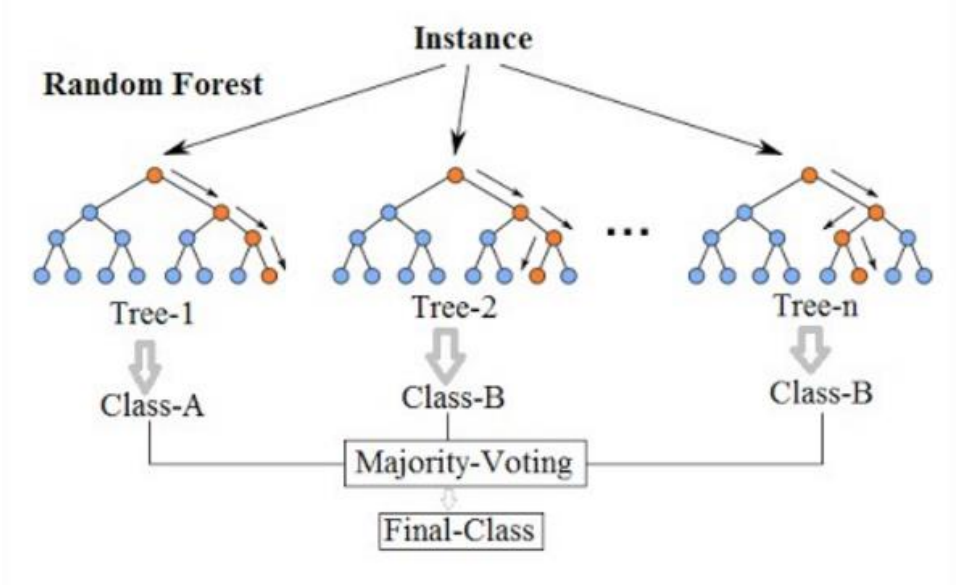

Figura 4. Modelo de Bosque aleatorio [15]

Como en el caso de las RNA, el conjunto de datos debe dividirse en dos subconjuntos aleatoriamente: 80% de datos para el entrenamiento, 20% para el test. Este modelo ha sido implementado en R [16], lenguaje de programación con enfoque estadístico. Para ello ha sido necesario incluir las bibliotecas: "randomForest", para hacer el entrenamiento y el test; "MASS", para utilizar métodos estadísticos y "XLConnect" para cargar el conjunto de datos de un archivo *Excel*.

3. Diseño

Para llevar a cabo la implementación del sistema inteligente para la optimización del razonamiento semántico, es necesaria una etapa previa de diseño. En ella se deciden cuáles son las fases requeridas para un correcto desarrollo y se detallan las necesidades que estas deben cumplir.

3.1 Elección y cálculo de las métricas de una ontología

El motivo de querer conocer las métricas de una ontología es poder caracterizarla para posteriormente realizar predicciones en función de estas. Por ejemplo, si llevamos a cabo una consulta con un razonador sobre una ontología y obtenemos el tiempo que dura esta tarea, podemos establecer relaciones que ayudarán a aproximar este coste cuando sea otra la ontología de entrada.

Por ello, el primer paso es decidir la manera óptima de caracterizarla. Tras un proceso de documentación, se ha llegado a la conclusión de utilizar las 51 métricas propuestas por el artículo de *IJSWIS* [7], como el número de axiomas TBox y ABox. En este artículo se elabora la predicción del tiempo que llevará a un razonador realizar determinada tarea en PC. Otro trabajo previo útil han sido los scripts para calcular las métricas utilizadas en el artículo nombrado. Para llevar a cabo la obtención de estas métricas se ha utilizado el software Android Studio, ya que debía ejecutarse en un dispositivo móvil porque se ha querido medir el tiempo que cuesta realizar esta tarea.

3.2 Consultas mediante un razonador semántico

En esta aplicación se introduce el uso de los razonadores. Se ha hecho una selección de los existentes y utilizado para realizar consultas sobre las ontologías. Para elegir qué tarea iban a llevar a cabo los razonadores, se ha tenido en cuenta que su duración debía ser suficiente porque mientras tanto se querían medir tiempo, batería y memoria consumidas. Por ello, se decidió realizar la "clasificación" de la ontología, tarea que comprueba la relación jerárquica entre conceptos, es decir, para cada par de conceptos decide si uno es una subclase de otro.

Basándonos en estos resultados, he realizado una comparación de qué razonador da mejores resultados y, por tanto, ha sido el utilizado para la posterior predicción de los costes.

3.3 Aprendizaje automático

Este algoritmo utiliza una técnica conocida como "aprendizaje automático" (del inglés, "*machine learning*"). Es una rama de la Inteligencia Artificial cuyo objetivo es desarrollar técnicas que permitan que las computadoras aprendan. Concretamente, trata de crear programas capaces de generalizar comportamientos. [17]

Existen diversos tipos de algoritmos con este cometido, de los cuales en este proyecto se han utilizado dos: Redes Neuronales Artificiales (RNA) y Bosques Aleatorios. La utilización de estos algoritmos se ha necesitado para realizar la predicción de los costes del razonamiento. El algoritmo aprende a partir de los datos obtenidos con la aplicación del cálculo de métricas y la de los costes del razonador. Una vez entrenado, se ha realizado un test en el que realiza una predicción de las salidas (costes) a partir de unas entradas (métricas), y se mide el error de la predicción. Los parámetros de los algoritmos deben ajustarse hasta conseguir minimizar el error.

3.4 Toma de decisiones

Se ha desarrollado un algoritmo que decide dónde realizar el razonamiento (local o remoto) a partir de los resultados obtenidos con el algoritmo de aprendizaje automático. Si los valores de predicción del algoritmo superan determinado límite, el razonamiento deberá realizarse en remoto, y, en caso contrario, en local. Estos límites pueden cambiar en función de la capacidad del dispositivo móvil utilizado, por ello son parámetros que deben introducirse en el algoritmo de toma de decisión. Además, se ha tenido en cuenta el nivel de batería restante en el dispositivo.

4. Implementación y experimentación

En esta etapa del proyecto se desarrollan las ideas expuestas en la etapa de diseño. A lo largo de este apartado se implementan varias aplicaciones, todas ellas necesarias para el desarrollo de la aplicación definitiva, el sistema inteligente completo. Además, se describen los experimentos realizados con un conjunto de ontologías.

4.1 Aplicación para obtener las métricas de una ontología

Se ha desarrollado una aplicación que, dada una o varias ontologías, obtiene las métricas que la caracterizan. Esta sección consta de tres subsecciones: la primera consiste en el trabajo previo realizado, le sigue el desarrollo de la aplicación y, por último, el análisis de los resultados.

4.1.1 Identificación de métricas

Las métricas escogidas son las propuestas por el artículo *IJSWIS* [7], en el cual se realiza la predicción del tiempo en PC. Estas se pueden agrupar en 5 conjuntos, el primero incluye 3 métricas y todos los demás 12 cada uno:

- *Intensity Metrics* (IM): tamaño de TBox, ABox y ratio.
- *Concept Complexity Assertions with GCIs applied* (CCA)
- *Concept Complexity Assertions without GCIs applied* (CCA_WO)
- *Object Property Complexity Assertions* (OPCA)
- *Datatype Property Complexity Assertions* (DPCA)

Cada uno de los cuatro últimos conjuntos se puede dividir a su vez en dos subconjuntos:

- Subconjunto 1: Agrega la estimación de la complejidad de cada elemento de la ontología.
- Subconjunto 2: Considera que cada axioma observa a un elemento asociado y agrega los valores ponderados.

Una vez obtenidas las estimaciones de complejidad y los testigos de cada elemento, estos valores son agregados para obtener los valores finales de las métricas.

En primer lugar, para cada uno de los conjuntos de estimaciones, se calcula su suma total, valor promedio, valores máximo y mínimo, desviación estándar y entropía de la distribución de complejidad.

En segundo lugar, se obtienen los mismos valores agregados ponderados según los conteos de testigos de cada elemento considerado.

4.1.2 **Desarrollo de la aplicación**

En un primer periodo de prueba y hasta que se terminó de perfeccionar el código, el cálculo de métricas se llevó a cabo con un conjunto reducido de 9 ontologías. Posteriormente, en el estudio han participado 139 ontologías de tamaños variados. Estas ontologías las he obtenido de los *workshops* ORE 2013 [18] y ORE 2015 [19], bases de datos cuyos archivos son ontologías destinadas a usarse con razonadores.

Dentro del lenguaje OWL 2 existen varios perfiles o sub-lenguajes que ofrecen diferentes ventajas dependiendo del ámbito de aplicación. En este estudio se ha utilizado el perfil OWL 2 EL, tal y como recomienda W3C para trabajar con ontologías con gran cantidad de clases y propiedades. [1] Los ORE 2013 y 2015 están organizados de forma que facilitan escoger este subconjunto de ontologías. Además, están separadas en función de la tarea que se quiera realizar con ellas, en mi caso la clasificación.

Los resultados de la Tabla 1 siguen la categorización de ontologías en función de su tamaño según se propone en [20]. En ella se clasifican las 139 ontologías utilizadas dependiendo del número de axiomas que contienen.

	Pequeñas (<500)	**Medianas (500-5000)**	**Grandes (>5000)**
Nº ontologías	40	50	49

Tabla 1. Clasificación de ontologías según su número de axiomas

La realización del código para la obtención de métricas ha tenido como referencia los scripts de Carlos Bobed [21], perteneciente al Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza.

Se ha estudiado el desempeño de la aplicación en Android y se ha adaptado para un funcionamiento adecuado. En concreto, se ha descartado la parte del código en la que se

elabora un grafo de la ontología (para calcular la relación entre los axiomas TBox y Abox), ya que el cálculo de las métricas va a realizarse en dispositivos móviles. Esto se debe a que, por su capacidad computacional limitada, cuesta demasiado tiempo y memoria procesarlos y por ello no son eficientes en estos dispositivos.

Para la implementación de la App he utilizado Android Studio [22], IDE para el desarrollo de aplicaciones Android en un entorno de lenguaje Java. El dispositivo en el que se han realizado los experimentos ha sido un Xiaomi Redmi 4 con versión Android 6.0.1 (*Marshmallow*), el cual dispone de 3 GB de RAM y un procesador *Qualcomm Snapdragon 625*. El nivel de API mínimo para poder utilizar la aplicación es 19, que equivale a la versión Android 4.4 (*KitKat*).

Se ha diseñado una actividad que inicia el cálculo de métricas cuando se lanza la aplicación:

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   MetricsCalculator();
}
```

Además, ha sido necesario dar los siguientes permisos en el Archivo de Manifiesto para lectura/escritura de ficheros y para que la aplicación tenga acceso a Internet.

```
<uses-permission
   android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission
   android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission
   android:name="android.permission.INTERNET" />
<uses-permission
   android:name="android.permission.ACCESS_NETWORK_STATE" />
```

La aplicación para la obtención de métricas contiene un código extenso. A continuación, se muestran algunos fragmentos.

```
// Cálculo del número de axiomas y ratio
TBoxAxioms = ontology.getTBoxAxioms(true).size();
ABoxAxioms = ontology.getABoxAxioms(true).size();
outResults.print(ontology.getAxiomCount() + "\t" + TBoxAxioms + "\t" +
      ABoxAxioms + "\t" + ABoxAxioms/TBoxAxioms + "\t");
```

```
// Cálculo del resto de métricas
complexityCalculator = new
      AssertionsComplexityMetricsCalculatorEL(ontology);
complexityCalculator.calculateAllMetrics();
outResults.println(
      complexityCalculator.getClassComplexities().toString() +
      complexityCalculator.getClassComplexitiesWithoutGCI().toString()
      + complexityCalculator.getObjectPropComplexities().toString()
      + complexityCalculator.getDataPropComplexities().toString());
```

Como he comentado al principio del apartado, primero se ha tratado con un conjunto reducido de 9 ontologías de características diversas para sacar algunas conclusiones generales. Los archivos "*.owl*" se han almacenado dentro de la aplicación y el programa los llama uno a uno.

Descartando una de las ontologías por dar problemas al parsear el fichero y teniendo en cuenta el criterio de tamaño explicado, podemos clasificar las 8 restantes en: 4 pequeñas (0-500 axiomas), 2 medianas (500-5000) y 2 grandes (+5000).
Cabe decir que, mientras se intenta procesar la ontología de mayor tamaño (46940 axiomas, 34.5 MB), el log de mensajes avisa de la falta de memoria, aunque finalmente consiga obtener sus métricas. De lo cual podemos conjeturar que, probablemente, con una ontología mayor no podría hacerlo. Esta restricción se debe a que se ha realizado en un dispositivo Android, ya que esta misma actividad desarrollada en PC no tiene este tipo de limitaciones.

Tras esta introducción en el cálculo de las métricas, el siguiente paso es realizar este proceso pero en un conjunto mayor: las 139 ontologías pertenecientes a los conjuntos ORE 2013 Y 2015. Como es un conjunto de gran tamaño, es inviable almacenar estos archivos dentro de la aplicación. Por ello, las he subido al servidor web gratuito *webhost000* [23], y he dado permisos a la aplicación para poder acceder a Internet. La primera idea, por sencillez, fue utilizar Dropbox y Google Drive. Esta idea se descartó rápidamente, ya que los enlaces no siguen un patrón lógico (no hay correspondencia entre nombre del fichero y URL) y suponía escribir en el código todos los enlaces de las ontologías para acceder a ellos. En cambio, en *webhost000*, estos enlaces siguen el mismo formato para todas las ontologías:

"*https://mlanau.000webhostapp.com + /nombreCarpeta/ + nombreOntologia*"

De esta manera, al programa se le pasa un fichero con los nombres de las ontologías, las va llamando una a una, calcula sus métricas y crea dos ficheros de texto: en uno almacena estos valores en forma de tabla, y en otro el tiempo que cuesta hallarlos. Estos ficheros se guardan en el almacenamiento externo del dispositivo Android. Para que todo ello pueda realizarse es necesario dar permisos de lectura y escritura de ficheros.

A continuación, se muestra un fragmento de código de la actividad principal:

```
// Cada elemento del vector ontologyNames contiene un String con el
// nombre de una ontología, p.ej. "ore_ont_508.owl"
IRI ontologyIRI = IRI.create("https://mlanau.000webhostapp.com/files/"
      + ontologyNames[n]);

// Se carga la ontología con la OWL API
OWLOntologyManager om = OWLManager.createOWLOntologyManager();
OWLOntology ontology =
      om.loadOntologyFromOntologyDocument(ontologyIRI);

startTime = System.nanoTime();
// Calcula las métricas
// ...
finishTime = System.nanoTime();

Log.i(TAG, "... has runned for " + (finishTime - startTime) / 1e9 + "
seconds");
```

4.1.3 **Análisis de los resultados**

Analizando los datos obtenidos en relación al tiempo, se observa que a partir de las ontologías de más de 2.500 axiomas, hay una tendencia ascendente en cuanto al tiempo que cuesta calcular sus métricas. Para ontologías con menos de 2.500 axiomas, este cálculo no llega en ningún caso a la décima de segundo, mientras para las de más de 100.000 axiomas, esta cifra no baja de los 20 segundos.

Se ha hecho una división de los datos teniendo en cuenta la duración del cálculo las métricas y se ha obtenido el promedio en cada subconjunto para observar la dependencia con el número de axiomas de la ontología. En la Tabla 2 pueden verse estos resultados.

		Tiempo (s)
Número de axiomas	< 2.500	0,026123724
	2.500 - 25.000	0,433249145
	25.000 - 65.000	3,121878496
	100.000 - 215.000	43,46248782

Tabla 2. Dependencia del número de axiomas de una ontología con el tiempo que cuesta calcular sus 51 métricas establecidas.

Se aprecia que para las ontologías de más de 100.000 axiomas, la obtención de sus métricas en dispositivos móviles no es eficiente, por un excesivo coste temporal. En cambio, para valores inferiores a 65.000 axiomas, esta duración es breve. Por carencia de ontologías con axiomas entre 65.000 y 100.000 no se ha podido analizar ese intervalo.

4.2 Aplicación para consultas con razonador

Se ha desarrollado una aplicación que utiliza varios razonadores semánticos para realizar consultas sobre un conjunto de ontologías. El objetivo de esto es obtener el tiempo de ejecución, potencia y memoria utilizada durante las consultas.

Esta sección, como la anterior, consta de 3 subsecciones: trabajo previo realizado, desarrollo de la aplicación y, finalmente, análisis de los resultados obtenidos.

4.2.1 Elección de razonadores

En primer lugar se debía elegir qué razonador utilizar para realizar la posterior predicción de los costes del razonamiento. Existen numerosos razonadores desarrollados para su uso en PC, pero no ocurre lo mismo con su uso en dispositivos Android. Por ello, he utilizado varios de los razonadores portados por el grupo de Sistemas de Información Distribuidos de la Universidad de Zaragoza [24]: *HermiT*, *JFact* y *TrOWL*.

- ***HermiT*** [25]**:** es un razonador implementado en Java que soporta completamente el lenguaje OWL 2. También está incluido en *Protégé*, el editor de ontologías más conocido. A diferencia de otros razonadores, HermiT 1.3.8 no puede ser convertido directamente a Dalvik, la máquina virtual utilizada en dispositivos Android.
- ***JFact*** [26]**:** nació como el resultado de portar el razonador FaCT++ en lenguaje Java y soporta completamente OWL 2. JFact 1.2.1 puede ser importado directamente en un proyecto Android, ya que no necesita ninguna modificación para convertirlo a Dalvik.
- ***TrOWL*** [27]**:** es una infraestructura de razonamiento para OWL 2. TrOWL 1.4 puede ser convertido a Dalvik sin modificación alguna y ser importado en un proyecto Android directamente. Este razonador solo da resultados exactos para el perfil OWL 2 EL (el utilizado en el proyecto), mientras que para OWL 2 DL son aproximados.

Además de los razonadores comentados he decidido probar con *Mini-ME*, un prototipo de razonador para dispositivos móviles. El objetivo de su desarrollo es superar los problemas de rendimiento que suceden con los razonadores diseñados para PC.

- ***Mini-ME*** [28]**:** es compatible con el lenguaje OWL 2, está desarrollado en Java y se ejecuta en Android.

4.2.2 **Desarrollo de la aplicación**

Una vez seleccionados los razonadores con los que se harán las pruebas, el siguiente paso es comparar cual tiene un mejor funcionamiento en cuanto a tiempo, potencia y memoria.

Como en el caso de las métricas, para la implementación de la *app* he utilizado Android Studio y se ha ejecutado en un Xiaomi Redmi 4. El nivel de API mínimo requerido es 21 (por el uso de propiedades implementadas a partir de ese nivel de API).

En un mismo programa y haciendo pequeñas modificaciones para utilizar cada razonador, he realizado consultas en las ontologías midiendo los costes. Se ha creado una aplicación distinta a la de las métricas para evitar lo máximo posible errores de memoria, ya que cuando se probó a ejecutar ambas tareas en el mismo programa, en muchos casos surgieron este tipo de problemas y el proceso no llegaba a terminar.

En primer lugar, se tuvo que decidir qué tarea iba a realizar el razonador. Como se ha comentado anteriormente, debía ser una consulta que durase lo suficiente para que las medidas que se iban a calcular fueran fiables. Por ese motivo, se decidió realizar la "clasificación" de la ontología.

Como en la aplicación de las métricas, se ha seguido un desarrollo incremental: antes de pasar a un conjunto mayor de ontologías, las primeras pruebas se han realizado con una agrupación reducida (9 ontologías), para ver qué posibles fallos podían suceder.

Al realizar esta clasificación en un dispositivo móvil he obtenido numerosos errores debidos a distintas causas. Es importante diferenciar los que sólo suceden en el móvil de los que también sucederían en PC, para conocer las limitaciones del dispositivo móvil.

La Tabla 3 muestra el número de ontologías en las que no se ha llegado a realizar la clasificación, causado por distintos errores. En el caso del razonador *Mini-ME*, carece de sentido decir los errores que ocurrirían en PC por ser específico para dispositivos móviles. Aun así se han anotado los errores no causados por las limitaciones de estos dispositivos (memoria, tiempo).

	Sólo en móvil		**También en PC**			
	Falta de memoria	**Demasiado tiempo ejecución**	**Sintaxis incorrecta**	**Característ-ica no soportada**	**Stack overflow**	**Correctas**
HermiT	2	1 (37 s)	1	1		4
JFacT	2		1	1		5
TrOWL	1		1			7
MiniME	2		1		1	5

Tabla 3. Número de errores según el razonador producidos durante la clasificación

Vemos que todos los razonadores han tenido algún problema para clasificar alguna ontología, siendo la limitación en memoria el error que más se repite. También vemos

que el razonador *TrOWL* es el que mejor resultados ha obtenido, aunque es un dato poco significativo debido al tamaño reducido del conjunto de ontologías.

Seguidamente, se pasó a hacer las pruebas de los razonadores con un conjunto de ontologías mucho mayor, como en el caso de las métricas. Se trata de una fusión de los conjuntos ORE 2013 y ORE 2015, seleccionando el perfil EL, como se ha comentado anteriormente.

Teniendo en cuenta los resultados obtenidos con el conjunto reducido, se puede predecir que las ontologías que sean mayores que un determinado tamaño darán problemas de ejecución en la aplicación. En un principio, el límite de tamaño máximo fue el de la ontología que la aplicación no pudo ejecutar en ninguno de los 4 casos en la primera prueba (34.5 MB), pero cuando comencé las pruebas tuve que ir reduciendo aún más este margen, el cual finalmente rondó los 20 MB. De hecho, la idea inicial fue realizar la clasificación sólo con el conjunto ORE 2013, pero como se fue reduciendo por los descartes realizados, se optó por añadir también el conjunto ORE 2015, ya que cuanto mayor y más variada fuera la muestra de ontologías, mejores serían las predicciones.

Esta aplicación ha sido más problemática que la anterior, debido a errores de memoria para realizar la clasificación de ontologías de las que se habían obtenido sus métricas. Por ello, muchas de las métricas halladas han quedado inutilizadas para poder predecir los costes del razonamiento y, finalmente, solo han podido utilizarse 102 ontologías.

Para realizar las consultas he subido los archivos "*.owl*" de las ontologías a *000webhost*, como en el caso de las métricas. El programa va descargando una a una cada ontología, crea el razonador, realiza su tarea y mide los costes. Estos son almacenados en un fichero "*.txt*" en forma de tabla.

Los distintos costes que he medido han sido el tiempo que le cuesta realizar la clasificación, la potencia y la memoria que ocupa.

A continuación se muestran algunos fragmentos del código implementado:

- Se crea un razonador u otro dependiendo del parámetro de entrada *reasoner_type*. Para poder utilizarlos, se debe incluir en el proyecto sus correspondientes librerías.

```
switch (reasoner_type) {
    case REASONER_JFACT:
       return new JFactFactory().createNonBufferingReasoner(ontology);
    case REASONER_HERMIT:
       Configuration conf = new Configuration();
       conf.ignoreUnsupportedDatatypes = true;
       return new Reasoner(conf, ontology);
    case REASONER_MINIME:
       MicroReasonerFactory reasonerFactory = new
      MicroReasonerFactory();
       OWLReasoner reasoner =
            reasonerFactory.createMicroReasoner(ontology);
       return reasoner;
    case REASONER_TROWL:
       RELReasonerFactory relfactory = new RELReasonerFactory();
       RELReasoner reasoner = relfactory.createReasoner(ontology);
       return reasoner;
    default:
       throw new IllegalArgumentException("No valid reasoner");
       }
}
```

- Clasificación de una ontología con el razonador *JFacT*:

```
// Carga la ontología
IRI ontologyIRI = IRI.create("https://mlanau.000webhostapp.com/files/"
      + ontologyNames[n]);
OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
OWLOntology ontology =
      manager.loadOntologyFromOntologyDocument(ontologyIRI);

// Crea un razonador para dicha ontología
OWLReasoner reasoner = createOWLReasoner(ontology, REASONER_JFACT);

// Realiza su clasificación
reasoner.precomputeInferences(InferenceType.CLASS_HIERARCHY);
```

Este procedimiento es el mismo para todos los razonadores probados excepto para *Mini-ME*, ya que su clasificación se realiza directamente cuando es creado.

La obtención de los costes se ha realizado de la siguiente manera:

- **Tiempo [s]:** Una variable toma el tiempo en el instante anterior de realizar la clasificación y otra lo toma justo después. Mediante la resta de ambas se obtiene el resultado. Este es en nanosegundos pero se ha pasado a segundos.

  Fragmento del código:

```
startTime = System.nanoTime();
// tarea a medir
// ...
finishTime = System.nanoTime();
time = (finishTime-startTime) / 1e9;
Log.i(TAG, " Time: " + time + " s");
```

  La clase *System* puede utilizarse en cualquier versión de Android ya que está disponible desde el nivel de API 1.

- **Potencia [W]:** Se ha realizado por software, a través de la clase *BatteryManager* [29]. He obtenido el valor de la corriente y la tensión en el instante que termina la tarea y realizando el producto de ambas resulta la potencia en Vatios.

  Antes de llegar a este resultado se han considerado varias opciones. Una de ellas fue calcular la energía consumida en vez de la potencia. Esto suponía tener en cuenta el tiempo que dura la tarea para obtener el consumo (Ws), lo cual se descartó para que las salidas no dependieran entre sí. Otra opción considerada fue la propuesta por un artículo sobre el cálculo de consumo energético en dispositivos móviles [30]: ir midiendo el valor de la tensión con un *BroadcastReceiver* por si se producen cambios durante la clasificación. Se decidió comprobar si esto realmente sucedía mediante una App para el manejo de la batería (AccuBattery [31]). Esta aplicación actualiza los valores en tensión de la batería cada pocos segundos, y se observó que, ejecutando la aplicación a la vez que la consulta, las variaciones de tensión eran mínimas (unos pocos mV). Por ello también se descartó esta opción. Además, la fórmula para el cálculo de la energía consumida que propone el artículo no es válida ya que utiliza el tiempo dividiendo en vez de multiplicando a la tensión y la corriente.

  Para que las medidas sean válidas se debe tener el dispositivo móvil desconectado de cualquier fuente de energía.

Fragmento del código:

```
// tarea a medir
// ...
powerConsumption = new
      PowerConsumption(getApplicationContext());
double power = powerConsumption.getPower();
double current = powerConsumption.getCurrentNow();
double voltage = powerConsumption.getVoltage();
Log.i(TAG, "Power: " + power + " W
             Current: " + current + " A
             Voltage: " + voltage);
```

En la clase "*powerConsumption*":

```
// Cálculo corriente
public double getCurrentNow(){
   //...
   Integer currentNow =
      mBatteryManager.getIntProperty(BatteryManager.BATTERY
      _PROPERTY_CURRENT_NOW);
   return currentNow/(1e6);
}

// Cálculo tensión
public double getVoltage(){
   //...
   int voltage =
      batteryStatus.getIntExtra(BatteryManager.EXTRA_VOLTAG
      E, -1);
   return voltage/(1e3);
}

// Cálculo potencia
public double getPower(){
   return (currentNow()*voltage());
}
```

La clase *BatteryManager* se añadió en la primera versión de Android, pero ha sufrido numerosas modificaciones. Su uso en esta aplicación requiere un nivel de API superior a 21, lo que equivale a la versión Android 5.0 (Lollipop).

- **Memoria [KB/MB]:** Se ha obtenido la memoria ocupada (memoria total – memoria libre) antes y después de realizar la clasificación, calculando el resultado mediante la diferencia de ambas medidas. Justo antes de la tarea del razonador se llama al recolector de basura para mejorar la predicción del resultado. De todos modos, este es aproximado ya que no se puede evitar que durante la clasificación se libere memoria. El resultado se obtiene en bytes, pero es convertido a KB/MB dependiendo del razonador utilizado.

  Fragmento del código:

```
System.gc();
Runtime runtime = Runtime.getRuntime();
memoryUsed1 = (runtime.totalMemory() - runtime.freeMemory()) /
1024;
// tarea a medir
// ...
memoryUsed2 = (runtime.totalMemory() - runtime.freeMemory()) /
1024;
Log.i(TAG, "The memory used is " + (memoryUsed2 - memoryUsed1) +
" KB");
```

  Recordemos que la clase *Runtime* puede utilizarse en cualquier versión de Android ya que está disponible desde el nivel de API 1.

4.2.3 Análisis de los resultados y comparativa entre razonadores

Durante la ejecución de la aplicación se genera un fichero de texto en el que se van escribiendo los costes de las ontologías en forma de tabla.

Como se ha visto en la prueba con el conjunto reducido de ontologías, cada razonador tenía diferentes problemas para realizar la clasificación. De las 139 ontologías disponibles, el número de ontologías que ha sido capaz de clasificar cada razonador se muestra en la última columna de la Tabla 4. La mayor diferencia se observa con el razonador *JFacT*.

Para comparar los costes en tiempo, potencia y memoria entre los 4 razonadores, se han tenido en cuenta sólo las ontologías que todos los razonadores han sabido clasificar,

es decir, 102 ontologías. En la Tabla 4 se muestran la media de estos costes para cada razonador.

	Tiempo (s)	Potencia (W)	Memoria	Nº ontologías
JFacT	6.16	1.216	3.91 MB	111
Mini-ME	3E-4	1.206	3.53 KB	129
TrOWL	1.10	1.513	5.07 MB	131
HermiT	3.05	1.082	15.53 MB	124

Tabla 4. Coste medio de la clasificación según el razonador (para 102 ontologías) y nº de ontologías que ha sabido clasificar cada razonador

Se aprecia una gran diferencia en cuanto al tiempo y la memoria consumida con el razonador *Mini-ME* respecto a los demás, ya que es el único específicamente diseñado para su uso en dispositivos móviles, es decir, adaptado a sus limitaciones.

El razonador *HermiT* ocupa una gran cantidad de memoria de media y, junto con el razonador *JFacT*, son los que más tiempo tardan en realizar la consulta. Además, *HermiT* y *JFacT* son los que menos ontologías han sabido clasificar. Por todo ello, se ha llegado a la conclusión que los razonadores que dan mejores resultados son *TrOWL* y *Mini-ME*.

4.3 Algoritmo de aprendizaje automático

Una vez adquiridas las métricas de la ontología y los costes que conlleva realizar su clasificación, se necesita la implementación de un algoritmo para predecir estos costes, dadas unas métricas de entrada.

Como se ha comentado en la etapa de diseño, se trata de un algoritmo de "aprendizaje automático". Existen distintos tipos de algoritmos que realizan esta

función, pero según hemos comprobado, su eficiencia varía de unos casos a otros. En el artículo de *IJSWIS* [7] en el que se realizan predicciones sobre conjuntos de ontologías se decide utilizar "Bosques aleatorios". En el artículo "Trees vs neurons" [32] comparan el uso de Redes Neuronales Artificiales y de Bosques Aleatorios. La conclusión a la que llega es que en el caso de predicción de consumo energético en edificios es mejor el uso de Redes Neuronales. Es por ello que se ha decidido implementar ambos algoritmos y comparar cuál ofrece mejores predicciones.

El razonador para el que se ha desarrollado el aprendizaje es *TrOWL* debido a que, aunque con *Mini-ME* se hayan obtenido buenos resultados, este razonador aún está en desarrollo.

4.3.1 **Redes Neuronales Artificiales**

El objetivo es predecir, con el mínimo error, el valor de las salidas (tiempo, potencia, memoria) partiendo del valor de las entradas (51 métricas).

En primer lugar, se deben normalizar los datos que utilizará la RNA. Para la normalización de los datos, en algunas métricas no se ha utilizado su máximo real por tener valores muy superiores al resto; por ello, se ha escogido como máximo un número inferior.

Para poder realizar las predicciones hay que entrenar la red neuronal. Para ello se ha utilizado el 80 % de la muestra de los datos normalizados. Se deben establecer varios parámetros como la tasa de aprendizaje, el *momentum*, el número máximo de iteraciones y la norma de aprendizaje. La norma de aprendizaje que he establecido para entrenar la red es *Momentum Backpropagation* (propagación hacia atrás con momentum).

Después del entrenamiento, la red es capaz de deducir el valor de las salidas con un error de mayor o menor grado. Por ello, el siguiente paso es realizar un test para saber cuál es el error de estas predicciones. Para ello he utilizado el 20 % restante de la muestra de datos. El test calcula las salidas a partir de las entradas y compara estos resultados con las salidas deseadas.

De esta manera se va calculando y acumulando el error, el cual se muestra al final del proceso. Para calcular el error he utilizado el "Error cuadrático medio" [33] entre los valores deseados y los de la predicción.

$$\mathrm{ECM} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Este error es dependiente de los parámetros establecidos en el proceso de entrenamiento. Por ello, para minimizar el error lo máximo posible, he ido variando estos parámetros hasta conseguir un error muy pequeño. En la Tabla 5 se muestran los valores de error según diferentes parámetros.

		Momentum		
		0.6	**0.7**	**0.8**
Tasa de aprendizaje	**0.4**	0.2061299	0.2592956	0.3576868
	0.5	0.3107389	0.2614297	0.3714283
	0.6	0.4420598	0.2485813	0.1927562

Tabla 5. Error en función de la tasa de aprendizaje y el *momentum*

Debido a causas de aleatoriedad en el proceso, para los mismos parámetros no siempre resulta el mismo valor de error. Por ello, el programa realiza el cálculo 20 veces y después se obtiene la media para una mayor fiabilidad de los resultados.

Una vez minimizado el error, esos parámetros son los definitivos. En este caso, el error mínimo (0.1927562) se consigue para una tasa de aprendizaje de 0.6 y un *momentum* de 0.8.

Este programa ha sido desarrollado en el lenguaje Java utilizando el entorno Eclipse [34] y posteriormente se ha portado a Android. Las ventajas de utilizar Eclipse han sido la comodidad de programar y realizar las numerosas pruebas del entrenamiento y test modificando los parámetros (tasa de aprendizaje y *momentum*). Una vez ajustados, se ha entrenado la red definitiva en Android y guardado en un archivo "*.nnet*" para poder utilizarla sin tener que volver a ejecutar el código de entrenamiento. El problema de realizar el entrenamiento en Eclipse, guardar el archivo e importarlo desde

un proyecto Android es que para escribir en el fichero, el programa utiliza el mecanismo de "serialización". Este permite escribir objetos Java en un fichero, pero realmente lo que escribe es el "bytecode" de Java que no es directamente compatible con el de Android.

Para trabajar con redes neuronales se ha incluido en el proyecto la biblioteca Neuroph [35]. Esta biblioteca tiene clases incompatibles con Android, y por ello ha habido que modificarla para poder utilizarla en dispositivos móviles. En concreto se han eliminado un par de atributos de la clase *Listener* que se heredan de la biblioteca gráfica para Java *swing*, no permitida en Android.

A continuación, se muestran algunos fragmentos del código desarrollado:

- Se definen el número de neuronas por capa, se crea la red neuronal y se definen los parámetros para el entrenamiento.

```
String dataSetFileName = "DatosNormalizados.txt";
int inputsCount = 51;
int outputsCount = 3;
int hiddensCount = Math.round(inputsCount * outputsCount);
String separator = ",";

DataSet dataSet =
      TrainingSetImport.importFromFile(dataSetFileName,
      inputsCount, outputsCount, separator);

// Crea red neuronal
MultiLayerPerceptron neuralNet = new
      MultiLayerPerceptron(TransferFunctionType.SIGMOID,
      inputsCount, hiddensCount, outputsCount);

// Introducción de parámetros
MomentumBackpropagation learningRule = new
      MomentumBackpropagation();
learningRule.setLearningRate(0.6);
learningRule.setMomentum(0.8);
learningRule.setMaxIterations(1000000);
neuralNet.setLearningRule(learningRule);
```

- Entrenamiento y test de la red

```
// División del DataSet para entrenamiento y test
DataSet [] dataSet8020 =
      trainingSet.createTrainingAndTestSubsets(80,20);

// Entrenamiento (con el 80% de la muestra original)
neuralNet.learn(dataSet8020[0]);

// Test  (con el 20% restante)
testCostesRazonador(neuralNet, dataSet8020[1]);
```

- Fragmento del método *testCostesRazonador:*

```
for (DataSetRow trainingElement : dset.getRows()) {
      counter+=1;
      nnet.setInput(trainingElement.getInput());
      nnet.calculate();
      double[] networkOutput = nnet.getOutput();
      double[] desiredOutput =
            trainingElement.getDesiredOutput();
      //cálculo del numerador de la fórmula del ECM
      errorNUM += difSquared(networkOutput, desiredOutput);
}
error = errorNUM/counter;
```

- Fragmento del método *difSquared:*

```
for(int i = 0; i < N; i++)
      diff += Math.pow(networkOutput[i] - desiredOutput[i], 2);
```

4.3.2 **Bosques Aleatorios**

Como en el caso de las RNA, contamos con 51 métricas de entrada y queremos obtener el valor de las salidas (tiempo, potencia y memoria).

Para realizar el entrenamiento se consideraron dos opciones: calcular de una vez las tres salidas (predicción multivariable) o, con un mismo conjunto de entrenamiento calcular las 3 salidas de una en una. La opción multivariable sería necesaria si los valores de salida tuvieran influencia entre ellos, pero se ha decidido escoger la otra opción por similitud a la RNA, ya que en este caso no se utilizan las salidas como variable de

entrada. Por ello se han creado 3 conjuntos de entrenamiento, uno para cada salida. Estos conjuntos son idénticos entre sí, exceptuando que se han eliminado las columnas con las salidas no utilizadas en cada caso.

A continuación, se muestra parte del código desarrollado en R:

```
# Semilla para el generador de números aleatorios
set.seed(101)

# Definición de los conjuntos de entrenamiento (80% de los datos) y
# test (el resto)
samp <- sample(nrow(dataset), 0.8 * nrow(dataset))
train <- dataset[samp, ]
test <- dataset[-samp, ]

# Se crea un conjunto de entrenamiento para cada salida
train1 <- train[1:nrow(train), 1:(ncol(train) - 2)]
train2 <- train[1:nrow(train), c(1:(ncol(train) - 3),ncol(train)-1)]
train3 <- train[1:nrow(train), c(1:(ncol(train) - 3),ncol(train))]

# Algoritmo de aprendizaje, para predecir Time, Power y Memory
result1 <- randomForest(Time~., data=train1)
result2 <- randomForest(Power~., data=train2)
result3 <- randomForest(Memory~., data=train3)

# Predicciones sobre el conjunto de test
pred1 <- predict(result1, test)
pred2 <- predict(result2, test)
pred3 <- predict(result3, test)

# Cálculo del Mean Squared Error
error1 <- with(test, mean((Time - pred1)^2))
print(error1)
error2 <- with(test, mean((Power - pred2)^2))
print(error2)
error3 <- with(test, mean((Memory - pred3)^2))
print(error3)
```

En la Tabla 6 se muestra el error obtenido para cada variable así como el error total.

	Error
Tiempo	0.01312135
Potencia	0.07773907
Memoria	0.06839997
Total	**0.15926039**

Tabla 6. Error en la predicción de cada variable y total

Comparando ambos algoritmos de aprendizaje automático, vemos que el error es menor en los Bosques Aleatorios que en las RNA (0.15926039< 0.1927562), aunque con una diferencia mínima.

4.4 Algoritmo de toma de decisiones

El siguiente paso es implementar un algoritmo de toma de decisiones que diga si el razonamiento debe realizarse en remoto o en local, en función de las métricas obtenidas de una ontología. Para ello se utiliza un algoritmo de predicción (RNA) ya entrenado en el que se introducen unas entradas (métricas) y obtiene las salidas aproximadas (costes).

A partir del valor de estas salidas toma la decisión de dónde se debería realizar el razonamiento.

Para realizar la predicción, las métricas no pueden ser introducidas directamente en la red neuronal. Por este motivo, una vez obtenidas se deben normalizar utilizando los valores empleados en la normalización del conjunto de datos del entrenamiento y test. Si el valor de alguna métrica es superior al tomado como máximo en el conjunto de datos, su valor normalizado será 1. De la misma manera, si un valor es inferior al valor mínimo del conjunto, su valor normalizado será 0.

Es necesario decidir cuáles son los valores límite para los cuales, si el resultado es menor, se aconsejará realizar el razonamiento en local, y si es mayor, en un dispositivo externo con más recursos. Estos valores son parámetros del algoritmo, que podrían

variar en función del dispositivo móvil en el que se quiera realizar el razonamiento (siempre entre 0 y 1 por ser cifras normalizadas). Por ejemplo, la capacidad del móvil podría influir en el tiempo y la memoria que cuesta, y para la potencia podría tenerse en cuenta el nivel de batería restante.

En mi caso, he decidido utilizar los siguientes parámetros:

- t (tiempo) = 0.25: puesto que 1 equivale a 47 segundos, he considerado que un usuario podría esperar una cuarta parte de este máximo a que se realizara el razonamiento.
- p (potencia) = 0.9: es un valor cercano a la unidad ya que también se ha decidido tener en cuenta el nivel de batería restante en el dispositivo móvil. Si tiene un valor inferior al 15% de la batería total, el sistema aconsejará que el razonamiento se ejecute en remoto.
- m (memoria) = 0.95: se ha optado por un valor elevado ya que en vez de utilizar para la normalización de los datos el valor máximo de la muestra (82 MB) se utilizó uno inferior (18 MB). El motivo de esta decisión es conseguir una mayor precisión, es decir, un error inferior en la predicción debido a que en la muestra solo había dos valores superiores a 18 MB, muy alejados del resto.

El algoritmo actual es muy sencillo pero ilustrativo. Para generalizarlo se necesitarían datos de más dispositivos móviles. El diseño de aplicación permite ejecutarlo en otros dispositivos de manera inmediata.

A continuación, se muestra un fragmento del código desarrollado con redes neuronales:

```
// Introducción de entradas: ontologyMetrics() devuelve un vector con
// las métricas normalizadas a partir del fichero de las métricas
double networkInput[] = ontologyMetrics();
System.out.println("Input: " + Arrays.toString(networkInput));

// Se carga la red entrenada del archivo "neuralnet.nnet", almacenado
// en los recursos de la aplicación
NeuralNetwork neuralNet =
        load(getResources().openRawResource(R.raw.neuralnet));
```

```
// Predicción de salidas
double [] networkOutput = outputPrediction(neuralNet, networkInput);

// toma de decisiones
int battery = batteryLevel.getBatteryPercentage();
double t = 0.25 ;
double p = 0.9 ;
double m = 0.98 ;

if (networkOutput[0] > t || networkOutput[1] > p || battery < 15 ||
      networkOutput[2] > m)
    System.out.println("Realizar en remoto");
else
    System.out.println("Realizar en local");
```

- Obtención de salidas de la red:

```
public double[] outputPrediction(NeuralNetwork nnet,
                                        double [] input) {
      nnet.setInput(input);
      nnet.calculate();
      double[] networkOutput = nnet.getOutput();
      System.out.println("Output: " +
            Arrays.toString(networkOutput));
      return networkOutput;
}
```

- Normalización de métricas:

```
switch (n_metric) {
    case 0:
        min = 30;
        max = 35531;
        break;
    // ...
    case 50:
        min = 0;
        max = 6.46;
        break;
    // ...
}
```

```
if(metricValue <= min)
    normalMetric = 0;
    else{
        if(metricValue >= max)
            normalMetric = 1;
        else
            normalMetric = (metricValue-min)/(max-min);
    }
```

- Obtención del nivel de batería, en la clase *BatteryLevel*

```
public int getBatteryPercentage() {
// ...
int level =
    batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale =
    batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
float batteryPct = level / (float) scale;
return Math.round(batteryPct * 100);
}
```

El desarrollo de la toma de decisiones mediante Bosques Aleatorios es interesante ya que su nivel de precisión es ligeramente mayor que en las Redes Neuronales Artificiales. Sin embargo, queda pendiente como trabajo futuro, del cual se habla en la sección 5.

4.5 Sistema Inteligente completo

Para obtener el Sistema Inteligente completo en Android, se ha creado una aplicación que reúne todo lo explicado en la Sección 4: dada una ontología de entrada, calcula sus métricas y, en función de las salidas obtenidas con un algoritmo de predicción (RNA), decide dónde realizar el razonamiento.

Esta aplicación final puede ejecutarse en dispositivos móviles con un nivel de API superior a 19 (versión Android 4.4).

Su funcionamiento es el siguiente:

- Introducción de la ontología, para lo cual se dan dos opciones: almacenarla en la carpeta "*Downloads*" del almacenamiento externo del dispositivo o cargarla mediante una URL.
- Se calculan sus métricas, las cuales se escriben en un fichero llamado "*MetricasOntologias.txt*" que se guarda en "*Downloads*".
- La aplicación lee el fichero, normaliza estos valores y los introduce en un vector.
- Este vector se utiliza como entrada en el algoritmo de predicción y obtiene tres salidas (tiempo, potencia y memoria).
- Se toma la decisión de dónde realizar el razonamiento mediante la comparación del valor de las salidas y unos parámetros. Además, se tiene en cuenta la batería restante en el dispositivo.

5. Conclusiones y trabajo futuro

Una vez finalizada la elaboración del proyecto, es el momento de evaluar algunos aspectos como el alcance de los resultados, las dificultades afrontadas, el aprendizaje adquirido... Además, de cara a un futuro, cabe comentar algunas ideas que sería interesante introducir para la mejora del proyecto.

Como resultado de este trabajo, se ha obtenido una aplicación para dispositivos Android que recomienda el lugar óptimo para el razonamiento de una determinada ontología en función de sus características. Las distintas formas de uso del razonador tienen sus ventajas e inconvenientes. Por ejemplo, la ventaja principal de realizar el razonamiento externamente es que uno puede considerar un servidor tan potente como requiera la aplicación. Sin embargo, el problema que surge es que el envío masivo de datos a través de la red se ve limitado por la conectividad. En cambio, un razonador local puede ser utilizado sin conexión a la red, donde también se ve beneficiada la privacidad de datos, importante en algunos casos. [36] Algunas de las limitaciones de este tipo de razonamiento son las probadas en el proyecto, como por ejemplo, un excesivo tiempo de razonamiento.

Durante la realización del proyecto he afrontado numerosas dificultades por ser un campo desconocido para mí. Desde la introducción en el ámbito de las ontologías, (utilizando *Protégé* para una mejor comprensión) hasta el aprendizaje sobre nuevos lenguajes de programación como Android y R. Además, este proyecto me ha permitido conocer algunos métodos de *"machine learning"*, como las Redes Neuronales Artificiales o los Bosques Aleatorios. Como base para la elaboración de este trabajo he tenido los conocimientos adquiridos durante el grado, en especial, la programación en lenguaje Java.

Cabe decir que es la primera vez que me enfrento a la elaboración de un proyecto de estas dimensiones y que necesita tantas horas de dedicación. Esto ha requerido de una planificación y administración del tiempo, aunque siempre aproximada debido a algunas dudas en las que he empleado más tiempo del previsto.

En el proyecto hay algunos aspectos mejorables a tener en cuenta como trabajo futuro. Sería interesante hacer pruebas con ontologías DL además del perfil EL utilizado en la aplicación, así como experimentar con más razonadores. Otra idea es realizar pruebas con más dispositivos móviles para poder generalizar los resultados obtenidos.

Otro aspecto a introducir en la aplicación final sería tener en cuenta los Bosques Aleatorios en la toma de decisiones. Una manera de hacerlo podría ser guardar el *random forest* ya entrenado en un archivo y posteriormente cargarlo en Android para que realice la predicción según la ontología introducida y, en consecuencia, se tome la decisión sobre el lugar de razonamiento. Sin embargo, el uso de R en Android presenta ciertas dificultades.

6. Referencias

[1] OWL 2. http://www.w3.org/TR/owl2-overview. Fecha de acceso: noviembre 2018

[2] Protégé. http://protege.stanford.edu. Fecha de acceso: noviembre 2018

[3] OWL API. http://owlapi.sourceforge.net. Fecha de acceso: 12 de noviembre de 2018

[4] Krotzsch, M., Simancik, F., & Horrocks, I. (2014). Description logics. *IEEE Intelligent Systems*, *29*(1), 12-19.

[5] Tello, A. L. (2001). Ontologías en la Web semántica. *España: Universidad De Extremadura*.

[6] Horridge, M., Jupp, S., Moulton, G., Rector, A., Stevens, R., & Wroe, C. (2009). A practical guide to building owl ontologies using protégé 4 and co-ode tools edition1. 2. *The university of Manchester*, *107*.

[7] Pan, J. Z., Bobed, C., Guclu, I., Bobillo, F., Kollingbaum, M. J., Mena, E., & Li, Y. F. (2018). Predicting Reasoner Performance on ABox Intensive OWL 2 EL Ontologies. International Journal on Semantic Web and Information Systems (IJSWIS), 14(1), 1-30.

[8] Tablada, C. J., & Torres, G. A. (2009). Redes Neuronales Artificiales. *Revista de Educación Matemática*, *24*(3).

[9] Matich, D. J. (2001). Redes Neuronales: Conceptos básicos y aplicaciones. *Universidad Tecnológica Nacional, México*.

[10] Tasa de aprendizaje. https://es.wikipedia.org/wiki/Perceptr%C3%B3n. Fecha de acceso: noviembre 2018

[11] Momentum. https://www.cs.us.es/cursos/ia2/temas/tema-05.pdf. Fecha de acceso: noviembre 2018

[12] Función de activación sigmoide. https://ml4a.github.io/ml4a/es/neural_networks/. Fecha de acceso: noviembre 2018

[13] Árbol de decisión. https://es.wikipedia.org/wiki/%C3%81rbol_de_decisi%C3%B3n. Fecha de acceso: noviembre 2018

[14] Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5-32.

[15] Modelo de *Random forest*. https://commons.wikimedia.org/wiki/File:Random_forest_diagram_complete.png. Fecha de acceso: noviembre 2018

[16] R. https://www.r-project.org/. Fecha de acceso: noviembre 2018

[17] Aprendizaje automático. https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico. Fecha de acceso: noviembre 2018

[18] ORE 2013. http://mowl-power.cs.man.ac.uk/ore2013/. Fecha de acceso: noviembre 2018

[19] ORE 2015. https://zenodo.org/record/18578#.W9Q3Z3szbIW. Fecha de acceso: noviembre 2018

[20] ORE 2013. http://curation.cs.manchester.ac.uk/ore2013/ore2013.cs.manchester.ac.uk/index.html. Fecha de acceso: noviembre 2018

[21] Métricas. https://github.com/cbobed/OWL2Predictions. Fecha de acceso: noviembre 2018

[22] Android Studio. https://developer.android.com/studio/intro/?hl=es-419. Fecha de acceso: noviembre 2018

[23] 000webhost. https://mlanau.000webhostapp.com/. Fecha de acceso: noviembre 2018

[24] Razonadores semánticos para Android. http://sid.cps.unizar.es/AndroidSemantic/Reasoners/reasoners.html. Fecha de acceso: noviembre 2018

[25] HermiT. http://www.hermit-reasoner.com. Fecha de acceso: noviembre 2018

[26] JFacT. http://jfact.sourceforge.net. Fecha de acceso: noviembre 2018

[27] TrOWL. http://trowl.org. Fecha de acceso: noviembre 2018

[28] Mini-ME. http://sisinflab.poliba.it/swottools/minime/. Fecha de acceso: noviembre 2018

[29] BatteryManager. https://developer.android.com/reference/android/os/BatteryManager. Fecha de acceso: noviembre 2018

[30] Valincius, E., Nguyen, H. H., & Pan, J. Z. (2015, June). A Power Consumption Benchmark Framework for Ontology Reasoning on Android Devices. In *ORE* (pp. 80-86).

[31] AccuBattery. https://play.google.com/store/apps/details?id=com.digibites.accubattery&hl=es. Fecha de acceso: noviembre 2018

[32] Ahmad, M. W., Mourshed, M., & Rezgui, Y. (2017). Trees vs Neurons: Comparison between random forest and ANN for high-resolution prediction of building energy consumption. *Energy and Buildings*, *147*, 77-89.

[33] Error cuadrático medio. https://es.wikipedia.org/wiki/Error_cuadr%C3%A1tico_medio. Fecha de acceso: noviembre 2018

[34] Eclipse. https://www.eclipse.org/ide/. Fecha de acceso: noviembre 2018

[35] Neuroph. http://neuroph.sourceforge.net/documentation.html, Fecha de acceso: noviembre 2018

[36] Bobed, C., Bobillo, F., Mena, E., Pan, J. Z., Bernad, J., Bobed, C., ... & Yus, R. (2017, December). On Serializable Incremental Semantic Reasoners. In *Proceedings of the Knowledge Capture Conference* (p. 27). ACM.