EXPLORACIONES SOBRE EL SOPORTE MULTI-AGENTE BDI EN EL PROCESO DE DESCUBRIMIENTO DE CONOCIMIENTO EN BASES DE DATOS

ROSIBELDA MONDRAGÓN BECERRA

Departamento de Inteligencia Artificial

APROBADA:

Dr. Nicandro Cruz Ramírez

M. C. Virginia Angélica García Vega

Dr. Alejandro Guerra Hernández

MIA. Carlos Rubén de la Mora Basañez
Director de la Facultad de Física e Inteligencia Artificial

a mi

FAMILIA

con amor, por los momentos compartidos y su apoyo incondicional...

EXPLORACIONES SOBRE EL SOPORTE MULTI-AGENTE BDI EN EL PROCESO DE DESCUBRIMIENTO DE CONOCIMIENTO EN BASES DE DATOS

por

ROSIBELDA MONDRAGÓN BECERRA

TESIS

Presentada ante la Facultad de Física e Inteligencia Artificial

UNIVERSIDAD VERACRUZANA

para obtener el Grado de

MAESTRA EN INTELIGENCIA ARTIFICIAL

Director: Dr. Alejandro Guerra Hernández

Codirector: Dr. Nicandro Cruz Ramírez

Diciembre 2007

Resumen

En este trabajo se explora cómo dar soporte al proceso de Descubrimiento de Conocimiento en Bases de Datos (KDD) con un Sistema Multi-Agente (MAS) conformado de agentes de Creencias, Deseos e Intenciones (BDI). El resultado de esta exploración se refleja en el diseño e implementación del proceso KDD como un MAS BDI, que consiste de seis agentes especializados en algunos pasos (preprocesamiento, minería de datos y evaluación de patrones) de este proceso. Coordinador se encarga de gestionar la entrada/salida del sistema y la base de datos; Preprocesador suple los valores faltantes en esta base de datos y la discretiza de forma supervisada y no supervisada; y los agentes aprendices (ID3 , C4.5, NB y TAN) a partir de esta base de datos preprocesada construyen sus respectivos modelos de minería de datos, que envían al Coordinador para que éste seleccione y reporte al ganador. Las tareas de minería de datos que el MAS tiene encomendadas son la clasificación de datos y el modelado de dependencias, que se implementaron con los algoritmos ID3 y C4.5, y NB y TAN, respectivamente. Los modelos aprendidos por los agentes son evaluados con la métrica de validación cruzada estratificada con 10 particiones, debido a que fue reportada en Kohavi [58] como la mejor. Para llevar a cabo el diseño de este MAS nos apegamos a la metodología Prometheus (que está dirigida al desarrollo de agentes inteligentes) y empleamos la herramienta PDT, que da soporte a esta metodología, mientra que el desarrollo lo realizamos en el lenguaje de programación de agentes AgentSpeak(L), mediante el intérprete Jason. Otra herramienta importante para este trabajo fue WEKA, debido a que proporcionó el código en Java que implementa los algoritmos que ejecutan los agentes del MAS.

Índice general

Página

Lista de Tablas

Lista de Figuras

Capítulo 1

Introducción

El proceso de *Descubrimiento de Conocimiento en Bases de Datos* (mejor conocido por sus siglas en inglés como KDD - Knowledge Discovery in Databases) [69] consiste en seleccionar, preprocesar y transformar un conjunto de datos (que puede ser obtenido a partir de varias fuentes heterogéneas como bases de datos, archivos planos, almacenes de datos, etc.), para facilitar la aplicación de algoritmos de minería de datos, que obtienen patrones que subyacen en este conjunto de datos. Estos patrones se interpretan y evalúan posteriormente, para seleccionar aquellos que representen conocimiento útil y novedoso[1] [41].

Para llevar a cabo cada una de las tareas del proceso KDD existen diversos algoritmos, que se pueden encontrar implementados en herramientas como Clementine [63], DBMiner [32], Ambiente de Waikato para Análisis de Conocimiento (WEKA - Waikato Environment for Knowledge Analysis) [31], entre otras. El inconveniente surge cuando el usuario común tiene que escoger los algoritmos apropiados para su problema, lo que depende entre otras cosas de la naturaleza de los datos, la tarea de minería de datos a realizar y la forma de representar los patrones.

Por lo tanto, KDD es un proceso complejo, que requiere que el usuario disponga de conocimiento variado y especializado (es decir, debe conocer diversos algoritmos para llevar a cabo de principio a fin el proceso KDD), para que pueda obtener los resultados deseados.

Una solución a los problemas encontrados en el proceso KDD la han ofrecido los *Sistemas Multi-Agentes* (por sus siglas en inglés MAS - Multi-Agent Systems) que disponen de

[1]Este conocimiento se refiere a hipótesis que deben ser comprobadas o refutadas a través de la experimentación, que es un paso del método científico.

agentes especializados en minería de datos, que algunas veces se encuentran distribuidos en una red [8, 27, 28, 62]. Sin embargo, el enfoque seguido por estos MAS no ofrece el nivel de abstracción (por ejemplo, un lenguaje que maneje algunos conceptos de actitudes mentales — creencias, deseos, intenciones, conocimiento, obligaciones, etc. — que describen nuestros actos cotidianos) requerido para efectuar un proceso tan complejo como KDD. Esto se debe a que los agentes son implementados de acuerdo a la noción de agencia débil, es decir, como sistemas de software que poseen ciertas propiedades como autonomía, reactividad, proactividad y habilidad social [47]. Estos conceptos se revisarán en el capítulo 2.

La agencia fuerte conceptualiza e implementa a los agentes con actitudes mentales que se relacionan usualmente con las personas, además, de atribuirles las propiedades de los agentes débiles [47]. Un modelo de agente que ha adquirido importancia en la comunidad de Inteligencia Artificial y que se apega a este tipo de agencia, es el de *Creencias, Deseos* e *Intenciones* (BDI - Beliefs, Desires and Intentions) [9]. Este paradigma de agentes proporciona un nivel de abstracción alto (representado con creencias, deseos, intenciones, planes, acciones, metas, eventos), que logra una comunicación más eficaz entre el desarrollador y el usuario, debido a que ambos entienden y emplean el mismo lenguaje que se usa para definir a este tipo de agentes.

Por lo tanto, cuando se concibe el proceso KDD con el enfoque BDI como una sociedad de agentes inteligentes que interactúan autónomamente para alcanzar una meta común (por ejemplo, encontrar conocimiento y presentarlo en forma de modelos o patrones), se apoya al usuario en la realización de este proceso, pues estos agentes sirven de guía para seleccionar los algoritmos (y sus parámetros) adecuados para cada paso del proceso KDD.

Nuestra propuesta se centra en explorar cómo efectuar el proceso KDD con el apoyo de un MAS, conformado de agentes BDI especializados en algunos pasos (preprocesamiento, minería de datos y evaluación de patrones) de este proceso. El resultado de esta exploración se refleja en el diseño e implementación del proceso KDD como un MAS BDI, que consta de seis agentes, *Coordinador*, *Preprocesador*, *ID3*, *C4.5*, *NB* y *TAN*. Coordinador se encarga de gestionar la entrada/salida del sistema, así como la base de datos. Preprocesador

suple los valores faltantes de esta base de datos y la discretiza de forma supervisada y no supervisada. Y el resto de los agentes (a partir de la base de datos preprocesada — sin valores faltantes y discretizada) aprenden sus respectivos modelos de minería de datos, que envían al Coordinador para que éste seleccione y reporte al ganador.

Las tareas de minería de datos que el MAS tiene encomendadas son la *clasificación* de datos y el *modelado de dependencias*. La primera consiste en aprender un modelo que predice la categoría a la que pertenece un ejemplo de datos (por ejemplo, si es conveniente o no ir a jugar tenis, tomando en cuenta que el día estará soleado, fresco, con humedad alta y viento débil), mientras que la segunda encuentra un modelo que describe las relaciones significativas entre las variables de una base de datos.

La clasificación y el modelado de dependencias se implementaron mediante los algoritmos de minería de datos que ejecutan los agentes aprendices del MAS (ID3, C4.5, NB y TAN), *ID3* [33], *C4.5* [34], *Bayes Ingenuo* (NB - Naive Bayes) [19, 65] y *Bayes Ingenuo Aumentado a Árbol* (TAN - Tree Augmented Naive Bayes) [48].

Los algoritmos ID3 y C4.5 representan sus modelos mediante *árboles de decisión*, mientras que NB y TAN lo hacen a través de *clasificadores Bayesianos*. Es importante estimar la exactitud de estos modelos, no solo para predecir su comportamiento futuro, sino también para seleccionar uno de éstos a partir de un conjunto dado (selección del modelo). Para este fin se eligió el método de *validación cruzada estratificada con 10 particiones* (Stratified 10-fold Cross Validation), debido a que ha sido reportado en Kohavi [58] como el mejor método para la selección del modelo.

Para realizar el diseño del MAS se siguió la metodología *Prometheus* [38], que está dirigida al desarrollo de agentes inteligentes, y se empleó la *herramienta de diseño Prometheus* (PDT - Prometheus Design Tool) [37], que da soporte a esta metodología. Mientras que la implementación de este MAS se llevó a cabo en el lenguaje de programación de agentes *AgentSpeak(L)*, a través del intérprete *Jason* [56]. Otra herramienta de gran utilidad fue *WEKA* [31], debido a que proporcionó el código en Java que implementa los algoritmos de preprocesamiento, minería de datos y evaluación de patrones, que ejecutan los agentes.

Una vez implementado el MAS se efectuó la etapa de experimentación, que consistió en probar este MAS con trece bases de datos obtenidas del sitio de la Universidad de California [1]. Los resultados de los experimentos nos revelan que para los dos tipos de discretización de datos (supervisada y no supervisada), los modelos aprendidos con TAN, en general, logran el mejor desempeño, mientras que los construidos con ID3 obtienen el porcentaje de clasificación correcta más bajo. Sin embargo, TAN toma más tiempo en construir sus modelos, mientras que NB lo hace en menor tiempo (debido a que solo aprende los parámetros del modelo, pues éste siempre es el mismo).

El resto del documento está organizado de la siguiente forma: el capítulo 2 presenta los antecedentes, en donde se abordan los fundamentos de este trabajo; el capítulo 3 introduce las herramientas y plataformas que se utilizaron en la realización del trabajo; el capítulo 4 aborda aspectos del diseño y de la implementación del proceso KDD como un MAS BDI; el capítulo 5 ofrece y discute los resultados de los experimentos realizados con este MAS; y por último el capítulo 6 expone las conclusiones y el trabajo futuro.

Capítulo 2

Antecedentes

En este capítulo se introducen los temas que permitirán entender la propuesta que se desarrolla en este trabajo de tesis. Esta propuesta consiste en explorar cómo dar soporte al proceso KDD mediante un MAS conformado de agentes BDI. Por lo tanto, es esencial comprender el *proceso KDD y la minería de datos* (un paso central de este proceso), que son presentados en la sección 2.1. Los patrones extraídos mediante los algoritmos de minería de datos (que son ejecutados por los agentes) toman la forma de *árboles de decisión y clasificadores Bayesianos*, por consiguiente éstos se abordan en la sección 2.2. La sección 2.3 explica los métodos *holdout* y *validación cruzada*, que se utilizan frecuentemente para evaluar qué tanto se ajustan los patrones a los datos. Una versión de este último método, que se conoce como *validación cruzada estratificada*, se empleó para validar los modelos aprendidos por los agentes. El MAS implementado en este trabajo está conformado de *agentes BDI*, dado que éstos constituyen uno de los enfoques más ampliamente utilizado para desarrollar agentes inteligentes [45]. Estos agentes BDI se exponen en la sección 2.4. El lenguaje que se usa para implementar a los agentes BDI es *AgentSpeak(L)*, debido a que proporciona un marco de trabajo abstracto y elegante para programar este tipo de agentes [4]. AgentSpeak(L) se presenta en la sección 2.5. Finalmente, la sección 2.6 define un *MAS*, proporciona sus características y habilidades, y lista algunos de ellos, que han sido concebidos para llevar a cabo el proceso KDD.

2.1. El Proceso de Descubrimiento de Conocimiento en Bases de Datos (KDD) y la Minería de Datos

El proceso KDD consiste en la ejecución de una serie de pasos que tienen como fin: (1) la verificación de conocimiento existente, o (2) la descripción/predicción de nuevo conocimiento [67]. Uno de los pasos relevantes del proceso KDD es la minería de datos, que consiste en la aplicación de algoritmos que extraen este conocimiento que se representa en forma de patrones. A continuación abordamos tanto el proceso KDD (sección 2.1.1) como la minería de datos (sección 2.1.2).

2.1.1. El Proceso KDD

Fayyad et al. [66, 67] definen a KDD como un proceso no trivial de identificación de patrones válidos, novedosos, potencialmente útiles, y entendibles en los datos. En este contexto, los *datos* se refieren a un conjunto de hechos (ejemplos en una base de datos), y los *patrones* son expresiones en algún lenguaje que describen de manera compacta los datos. El término *proceso* implica que KDD comprende muchos pasos, entre los que se encuentran: la preparación de datos, búsqueda de patrones, evaluación del conocimiento, y refinamiento; los cuales pueden ser repetidos en múltiples iteraciones. Por *no trivial*, debe entenderse que alguna búsqueda o inferencia es llevada a cabo; es decir, involucra la búsqueda de estructuras, modelos, patrones o parámetros. Los patrones descubiertos deben ser *válidos* sobre nuevos datos con algún grado de certeza, para que puedan describir y/o predecir confiablemente el comportamiento futuro de alguna entidad. También se desea que los patrones sean *novedosos* (al menos para el sistema y preferentemente para el usuario) y *potencialmente útiles*, es decir, que proporcionen algún beneficio al usuario o a la tarea. Por último, los patrones deben ser *entendibles*, en otro caso, será necesario algún postprocesamiento.

Es posible definir medidas cualitativas y cuantitativas para evaluar los patrones extraídos, por ejemplo: medidas de certeza, para conocer la precisión del modelo sobre nuevos

datos; o utilidad, tal vez ganancias debido a mejores predicciones o una mayor velocidad en tiempo de respuesta de un sistema. Los términos de novedad y entendimiento son más subjetivos. En algunos contextos, el entendimiento puede ser estimado por simplicidad, por ejemplo, mediante el número de bits para describir un patrón. Las medidas cualitativas se derivan de las creencias que el usuario tiene sobre el dominio del problema, mientras que las cuantitativas se basan en la estructura y propiedades estadísticas de los patrones descubiertos [32].

Una noción importante llamada *interés*, se toma usualmente como una medida global del valor de un patrón, al combinar validez, novedad, utilidad y simplicidad. Algunas funciones de interés que se emplean para evaluar un tipo especial de patrones, llamado reglas de asociación [52], son: *soporte de la regla* y *confianza*. La primera de éstas representa el porcentaje de ejemplos que la regla de asociación satisface. La confianza evalúa el grado de certeza de la asociación detectada [32].

Dado lo anterior, podemos decir que un patrón va a representar conocimiento si excede algún umbral de interés. En este caso el conocimiento es específico del dominio y está determinado por las funciones y umbrales que el usuario elija, es decir, está orientado al usuario.

Como puede observarse en la figura 2.1 (adaptada de Fayyad et al. [66]), el KDD es un proceso iterativo, debido a que cuando no se obtienen los resultados esperados, es posible regresar a los pasos anteriores para aplicar otros algoritmos. Además, es un proceso interactivo, debido a que el usuario tiene que tomar varias decisiones [66, 67]. Los pasos del proceso KDD se describen a continuación:

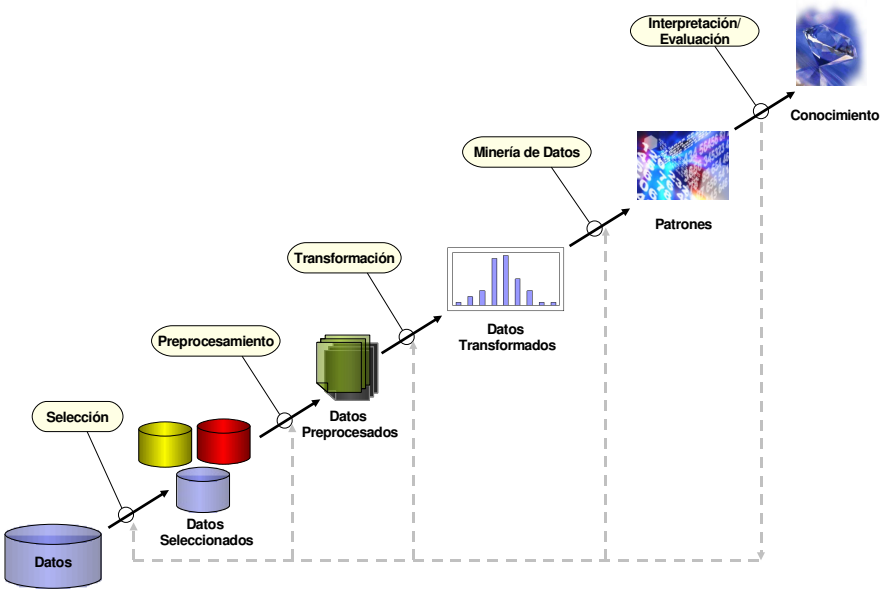

Figura 2.1: Un resumen de los pasos del proceso KDD.

1. Lograr un entendimiento del dominio de la aplicación (por ejemplo, la detección de cáncer) y del conocimiento a priori, así como identificar el objetivo del proceso KDD desde el punto de vista del usuario.

2. Crear un conjunto de datos objetivo (aquellos en los que se aplicará el proceso KDD), al enfocarse en un subconjunto de variables o muestra de datos, en los que se llevará a cabo el descubrimiento.

3. Realizar operaciones de limpieza y preprocesamiento de datos, por ejemplo: remover ruido si es necesario, reunir la información necesaria para modelar el ruido, decidir qué estrategias se usarán para manejar los datos faltantes, etc. En este contexto el ruido es un error aleatorio o varianza en una variable medida [32], es decir, son los valores que se desvían de lo esperado.

4. Reducir y proyectar los datos, a través de la selección de ejemplos y atributos que sean relevantes para el objetivo del proceso. Mediante métodos de reducción de la dimensionalidad o de transformación, es posible disminuir el número de variables bajo consideración; o encontrar representaciones reducidas del conjunto de datos, que son más pequeñas en volumen, pero que todavía producen los mismos (o casi los mismos) resultados, pero con un cómputo menor [32].

5. Seleccionar una tarea (método) de minería de datos de acuerdo a las metas del proceso KDD (paso 1). Algunas de estas tareas son: clasificación, regresión, agrupamiento y compactación. Éstas serán explicadas más adelante (sección 2.1.2).

6. Realizar un análisis exploratorio para seleccionar los algoritmos de minería de datos y las técnicas que se usarán para buscar los patrones en los datos. En este paso, el experto en minería de datos decide cuáles modelos y parámetros son más apropiados para satisfacer el objetivo del proceso KDD.

7. Ejecutar la minería de datos, es decir, buscar los patrones de interés en una forma de representación particular, como puede ser: reglas o árboles de clasificación, regresión, agrupamiento, entre otras. Es importante que el usuario realice correctamente los pasos anteriores, para asegurarse que los resultados que obtiene son confiables.

8. Interpretar y evaluar los patrones minados, posiblemente regresando a alguno de los pasos del 1 al 7 para volver a iterar. Este paso puede incluir la visualización de los patrones y modelos extraídos o la visualización de los datos dados los modelos extraídos.

9. Consolidar el conocimiento descubierto, a través de su incorporación dentro de otro sistema, o simplemente documentar y enviar este conocimiento a las partes interesadas. Este paso también incluye revisar y resolver conflictos potenciales con conocimiento previamente considerado (o extraído).

La mayoría del trabajo previo sobre KDD se ha enfocado en la minería de datos, debido a la creciente necesidad de crear técnicas de análisis de datos avanzadas, que nos permitan extraer el conocimiento (patrones) oculto en los datos. Sin embargo, los otros pasos del proceso KDD también son importantes, para la aplicación exitosa de este proceso [67]. Una vez que los conceptos básicos y el proceso KDD se han presentado, se explicará en mayor detalle el paso de minería de datos, debido a que representa una de las partes centrales de esta tesis.

2.1.2. Minería de Datos

La minería de datos es un paso del proceso KDD, que consiste en la aplicación de algoritmos que construyen modelos a partir de los datos, con el fin de describir o predecir el comportamiento de un cierto fenómeno [36].

Fayyad et al. [67] definen a la minería de datos como un paso del proceso KDD, que consiste en la aplicación de algoritmos de análisis y descubrimiento de datos, que bajo limitaciones computacionales aceptables, producen una enumeración de patrones sobre los datos.

La figura 2.2 muestra los objetivos del descubrimiento de conocimiento, los cuales se definen por el uso que se le dé al sistema KDD. Se pueden identificar dos tipos de objetivos: (1) *verificación* y (2) *descubrimiento*. En el primer caso, el sistema se limita a verificar las hipótesis del usuario, mientras que con el descubrimiento, el sistema autónomamente encuentra patrones nuevos, siempre que sea posible. El objetivo de descubrimiento se puede subdividir, en *descripción* y *predicción*. Con la descripción, el sistema obtiene patrones que presenta al usuario de forma entendible, y con la predicción, el sistema encuentra patrones para predecir el comportamiento futuro de alguna entidad [66].

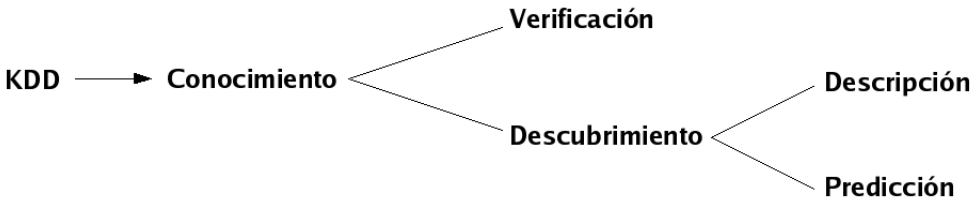

Figura 2.2: Objetivos del Descubrimiento de Conocimiento.

Aunque los límites entre descripción y predicción no son claros (algunos de los modelos predictivos también pueden ser descriptivos, en el sentido que pueden ser entendidos, y viceversa), es importante distinguirlos para lograr un entendimiento del objetivo del descubrimiento global. Por ejemplo, las redes neuronales [65] infieren modelos predictivos de ciertos fenómenos, siendo estos modelos difíciles de interpretar, mientras que las reglas de asociación [52] construyen modelos descriptivos que son fácilmente entendibles [14].

Los objetivos de descripción y predicción se alcanzan mediante los métodos de minería de datos (ver figura 2.3, modificada de Dunham [41]), que se presentan a continuación [32, 66, 67, 68]:

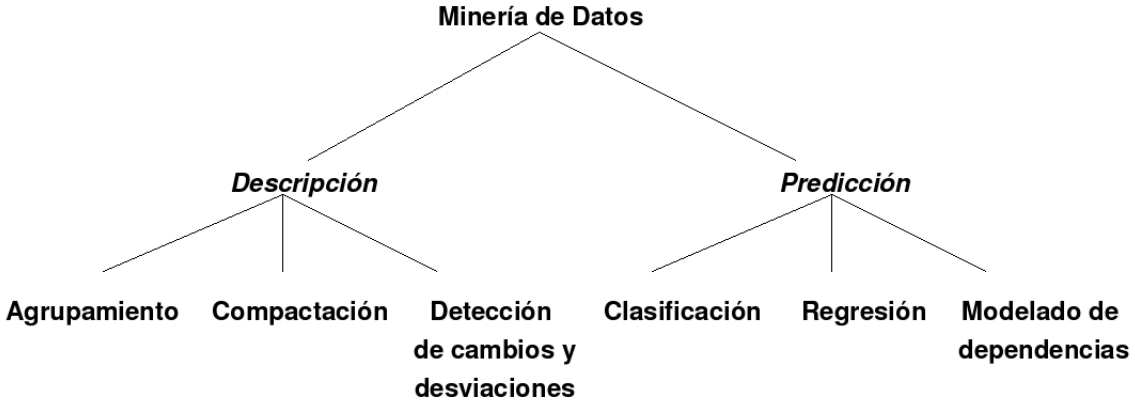

Figura 2.3: Métodos de Minería de Datos.

- *Agrupamiento*. Identifica un conjunto finito de categorías o grupos que describen los datos. Las categorías se definen al encontrar agrupamientos de ejemplos de datos

con base en métricas de similitud o modelos de densidad de probabilidad. Algunos algoritmos de minería de datos que se emplean para realizar esta tarea son: el vecino más cercano, K-Medias, el vecino más lejano, "K-medoids", CLARA, CLARANS, BEA, entre otros [41].

- *Compactación.* Encuentra una descripción compacta de un subconjunto de datos. Un ejemplo simple sería la media y la desviación estándar de todos los campos de la base de datos. Funciones más sofisticadas consisten de reglas sumarias, técnicas de visualización multivariadas, y relaciones funcionales entre variables. Las funciones de compactación frecuentemente son usadas en análisis de datos exploratorios y en la generación de reportes automatizados. HITS y PageRank son dos algoritmos que se utilizan para resumir documentos de texto, al extraer los enunciados más importantes (que permitan un entendimiento general) de éstos [59].

- *Detección de cambios y desviaciones.* Descubre los cambios más significativos en los datos a partir de valores anteriores o comunes. Para llevar a cabo esta tarea existen varios enfoques, entre los que se encuentran los que se basan en: estadísticas, distancias y desviaciones. Dentro del enfoque basado en distancias se han desarrollado algunos algoritmos eficientes: basado en índices, de ciclo anidado y basado en celdas [32].

- *Clasificación.* Aprende una función que mapea (clasifica) un ejemplo de datos en una de varias clases categóricas predefinidas. Los algoritmos ID3 (sección 2.2.1) y C4.5, que serán ejecutados por los agentes del MAS, aprenden árboles de decisión para clasificar los ejemplos de datos.

- *Regresión.* Aprende una función que mapea un ejemplo de datos a un valor de una variable predictiva numérica. La regresión asume que los datos se ajustan a algún tipo conocido de función (por ejemplo, lineal, logística, etc.) y después determina la mejor función de este tipo que modele los datos dados. Para encontrar la mejor función emplea algún tipo de análisis de error [41]. SECRET es un algoritmo que construye

árboles de regresión con modelos lineales en las hojas, lo que produce árboles de regresión precisos [3].

- *Modelado de dependencias*. Encuentra un modelo que describe dependencias significativas entre variables. Los modelos de dependencias existen en dos niveles: estructural y cuantitativo. El nivel estructural especifica (frecuentemente en forma gráfica) cuáles variables son localmente dependientes, y el nivel cuantitativo define la fuerza de las dependencias mediante el uso de alguna escala numérica. Por ejemplo, las redes Bayesianas usan independencia condicional para especificar el aspecto estructural del modelo y probabilidades para indicar la fuerza de las dependencias entre las variables. Los algoritmos NB (sección 2.2.2) y TAN (que serán ejecutados por los agentes del MAS) describen las relaciones de dependencia entre los diferentes atributos de la base de datos.

Los métodos comentados previamente son implementados a través de los algoritmos de minería de datos. En general, estos algoritmos consisten en tres componentes [67, 68]:

1. *El modelo*. Aquí hay que tomar en cuenta dos factores importantes: la *función del modelo*, es decir, para qué se empleará (por ejemplo, clasificación, modelado de dependencias, agrupamiento, etc.) y la *forma de representarlo* (que es el lenguaje que se usará para describir los patrones descubiertos, por ejemplo, como un árbol de decisión, una red Bayesiana, etc.). Un modelo también contiene parámetros que se determinan a partir de los datos o de la experiencia.

2. *El criterio de evaluación del modelo*. Usualmente es alguna forma de función de bondad de ajuste que nos indica qué tan bien un patrón particular (un modelo y sus parámetros) satisface la meta del proceso KDD. Los modelos predictivos frecuentemente son evaluados por la exactitud de la predicción empírica sobre algún conjunto de prueba. Los modelos descriptivos pueden ser evaluados por: la novedad, la utilidad y el entendimiento del modelo ajustado. Aunque no hay que olvidar que no todos los modelos novedosos son relevantes.

3. *El método de búsqueda.* Este paso puede verse como una tarea de optimización, donde una vez que el modelo y el criterio de evaluación de éste han sido determinados, el problema de minería de datos se reduce a encontrar los *parámetros* o *modelos* que optimizen el criterio de evaluación. En la *búsqueda de parámetros* el algoritmo debe hallar los parámetros que optimicen el criterio de evaluación del modelo dados los datos observados y la representación del modelo. En la *búsqueda del modelo* se modifica la representación del modelo para que se considere una familia de modelos (por ejemplo, los árboles de decisión), los cuales se diferenciarán en sus parámetros.

Un algoritmo de minería de datos particular, frecuentemente es una instanciación de los tres componentes descritos anteriormente, por ejemplo: un modelo de clasificación basado en una representación de árbol de decisión, con un criterio de evaluación apoyado en la probabilidad de los datos, y determinado por una búsqueda voraz usando una heurística particular.

En Fayyad et al. [69] y en Witten et al. [31] se encuentra una revisión breve de los algoritmos más populares de minería de datos.

2.2. Árboles de Decisión y Clasificadores Bayesianos

En esta sección se abordan las dos formas que se emplearon para representar los patrones extraídos de los datos: los árboles de decisión y los clasificadores Bayesianos. Éstos se eligieron debido a que representan las líneas de investigación del departamento de Inteligencia Artificial, donde se desarrolló este trabajo de tesis. Los árboles de decisión se presentan en la sección 2.2.1, mientras que los clasificadores Bayesianos se introducen en la sección 2.2.2.

2.2.1. Árboles de Decisión

Un *árbol de decisión* es una estructura de árbol parecida a un diagrama de flujo, donde cada *nodo interno* expresa una prueba sobre un atributo, cada *rama* representa una solución a la prueba, y los *nodos hoja* representan clases o distribuciones de clases. El nodo superior en el árbol se conoce como la *raíz* [32].

Para clasificar un ejemplo desconocido, los valores de los atributos de este ejemplo se evalúan en el árbol de decisión. Se traza un camino desde la raíz hasta un nodo hoja que tiene la predicción de la clase para el ejemplo.

Muchas de las ramas de un árbol de decisión pueden reflejar ruido o excepciones en los datos de entrenamiento. Técnicas como *prepoda* y *postpoda* intentan identificar y remover estas ramas, con el objetivo de mejorar la exactitud de la clasificación en datos nuevos [31, 32].

Con objeto de ganar legibilidad, los árboles de decisión pueden ser convertidos en reglas de clasificación, al seguir cada uno de los caminos del árbol desde la raíz hasta los nodos hoja. Por lo tanto, se obtendrán tantas reglas (formadas por la conjunción de las pruebas de los atributos y el nodo hoja correspondiente) como caminos tenga el árbol.

Los árboles de decisión han sido aplicados en una variedad de áreas que van desde la medicina hasta la teoría de juegos y los negocios. Estos árboles son la base de varios sistemas comerciales de inducción de reglas [32].

En esta sección se describe un algoritmo básico para aprender árboles de decisión. Este algoritmo se conoce como *ID3* (y es ejecutado por el MAS implementado en este trabajo). Posteriormente, se introduce la medida *ganancia de información*, que usa el algoritmo ID3 para seleccionar los atributos que formarán el árbol de decisión. Por último, se presenta un ejemplo de la inducción de un árbol de decisión con el algoritmo ID3.

ID3

ID3 [33] es un algoritmo básico para inducir árboles de decisión a partir de conjuntos de ejemplos de entrenamiento. Éste es un algoritmo voraz (debido a que los atributos seleccionados para formar el árbol de decisión son aquellos que minimizan o maximizan la función que se emplea para escoger estos atributos) que construye árboles de decisión en un estilo divide y vencerás [32]. Este paradigma descompone recursivamente un problema en dos o más subproblemas del mismo tipo, hasta que éstos llegan a ser suficientemente simples para poder resolverlos directamente. Posteriormente, las soluciones de los subproblemas son mezcladas para dar la solución al problema original.

El algoritmo 1, es una versión de ID3. Este algoritmo trabaja de la siguiente forma [32]:

Algoritmo 1 Genera-Árbol-Decisión. Genera un árbol de decisión a partir de los ejemplos de entrenamiento proporcionados.

Entrada: Los ejemplos de entrenamiento, *ejemplos*, representados por atributos discretos; el conjunto de atributos candidatos, *lista-atributos*.

Salida: Un árbol de decisión.

1: Crea un nodo N;
2: **si** *ejemplos* tienen la misma clase, C **entonces**
3: **regresa** N como un nodo hoja etiquetado con la clase C;
4: **fin si**
5: **si** *lista-atributos* está vacía **entonces**
6: **regresa** N como un nodo hoja etiquetado con la clase más común en *ejemplos*;
7: **fin si**
8: selecciona *atributo-prueba*, el atributo de *lista-atributos* con la ganancia de información más alta;
9: etiqueta el nodo N con *atributo-prueba*;
10: **para cada** valor a_i de *atributo-prueba* **hacer** // partición de los ejemplos
11: crea una rama desde el nodo N para la condición *atributo-prueba* $= a_i$;
12: sea s_i el conjunto de ejemplos en *ejemplos* para los cuales *atributo-prueba* $= a_i$;
13: **si** s_i está vacío **entonces**
14: agrega un nodo hoja etiquetado con la clase más común en *ejemplos*;
15: **en otro caso**
16: agrega el nodo que regresa Genera-Árbol-Decisión(s_i, *lista-atributos - atributo-prueba*);
17: **fin si**
18: **fin para**

- El árbol comienza como un nodo simple que representa los ejemplos de entrenamiento (paso 1).
- Si los ejemplos tienen el mismo valor de clase, entonces el nodo llega a ser una hoja y se etiqueta con ese valor de clase (pasos 2-4).
- De otra forma, el algoritmo usa una medida basada en la entropía [13] que se conoce como *ganancia de información*. Esta medida es una heurística que selecciona el atributo que separa mejor los ejemplos dentro de las clases (paso 8). Este atributo llega

a ser el atributo "prueba" del nodo (paso 9).

- Se crea una rama para cada valor del atributo prueba, y se particionan los ejemplos adecuadamente (pasos 10-12).
- El algoritmo usa el mismo proceso recursivamente para formar un árbol de decisión para los ejemplos de cada partición. Un atributo solamente puede ser escogido una vez a lo largo de una rama para formar un nodo (pasos 15-17).
- El particionamiento recursivo se detiene cuando cualquiera de las siguientes condiciones se cumple:
  (a) Todos los ejemplos para un nodo dado pertenecen a la misma clase (pasos 2-4), o
  (b) No hay atributos para seguir particionando los ejemplos (paso 5). En este caso, el nodo dado se convierte en hoja y se le asigna la clase que ocurre más frecuentemente en los ejemplos de entrenamiento (paso 6).
  (c) No existen ejemplos para la rama *atributo-prueba* $= a_i$ (paso 13). En este caso, se crea un nodo hoja y se etiqueta con la clase más común de los ejemplos de entrenamiento (paso 14).

Cabe mencionar que ID3 trabaja solamente con atributos discretos, incluyendo la clase, lo cual no representa un problema, dado que si existiecen atributos numéricos, éstos podrían ser discretizados antes de realizar el aprendizaje.

Medida para la Selección de Atributos

La medida ganancia de información evalúa qué tan bien un atributo dado divide los ejemplos de entrenamiento de acuerdo a su clasificación objetivo. Esta medida también se conoce como *medida de selección de atributos* o *medida de la bondad del particionamiento*.

Antes de definir la ganancia de información, se debe explicar la *entropía*, que es una medida que se usa comúnmente en teoría de la información. La entropía caracteriza la

impureza de un conjunto arbitrario de ejemplos. Dado un conjunto S, que contiene ejemplos positivos y negativos de algún concepto objetivo, la entropía de S relativa a esta clasificación booleana es

$$I(S) = -\, p_+ \, log_2 \, p_+ \; - \; p_- \, log_2 \, p_- \tag{2.1}$$

donde p_+ es la proporción de ejemplos positivos en S y p_- la proporción de ejemplos negativos en S.

Para ejemplificar, considera que S es un conjunto de 14 ejemplos de algún concepto booleano, que tiene 9 ejemplos positivos y 5 negativos (la notación [9+,5-] resume estos datos). La entropía de S relativa a esta clasificación booleana es

$$I([9+, 5-]) = -\,(9/14)\, log_2\,(9/14) \; - \; (5/14)\, log_2\,(5/14) = 0{,}940 \tag{2.2}$$

Si todos los ejemplos de S pertenecen a la misma clase la entropía es 0. Por ejemplo, si todos los ejemplos son positivos ($p_+ = 1$), entonces $p_- = 0$, y $I(S) = -\,1 \times log_2\,(1) - 0 \times log_2\, 0 = -1 \times 0 - 0 \times log_2 0 = 0$. Es importante aclarar, que para los cálculos de la entropía, se considera que $log_2 0$ es igual a 0. La entropía es 1 cuando el conjunto contiene el mismo número de ejemplos positivos y negativos, es decir, cuando $p_+ = 0{,}5$ y $p_- = 0{,}5$. Por el contrario, si el conjunto tiene un número diferente de ejemplos positivos y negativos, entonces la entropía está entre 0 y 1. Esto puede apreciarse en la figura 2.4 (adaptada de Mitchell [65]), que muestra la función de entropía relativa a una clasificación booleana, donde la proporción de ejemplos positivos, p_+, varía entre 0 y 1.

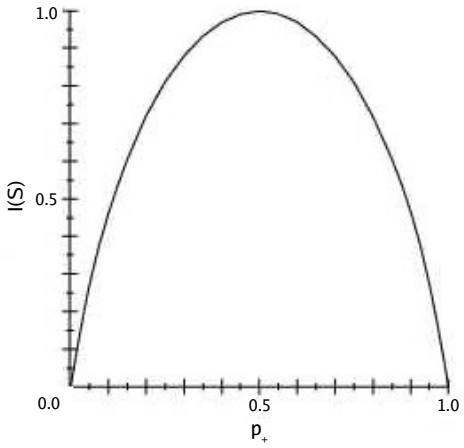

Figura 2.4: Función de entropía relativa a una clasificación booleana, donde la entropía (*I(S)*) varía entre 0 y 1, lo que depende de la proporción de ejemplos positivos (p_+), que también toma valores entre 0 y 1.

Una vez que se ha discutido la entropía para el caso especial donde la clasificación es booleana, se procede con el caso general. Sea S un conjunto de s ejemplos de datos. Suponer que el atributo clase tiene m valores distintos, $C_i\,(para\, i = 1, \ldots, m)$. Sea s_i el número de ejemplos de S de la clase C_i. Entonces la entropía de S relativa a esta clasificación de m valores se define como

$$I(S) = -\sum_{i=1}^{m} p_i \, log_2 \, p_i \tag{2.3}$$

donde p_i es la probabilidad que un ejemplo arbitrario pertenezca a la clase C_i y se calcula como s_i/s.

Dada la entropía como una medida de la impureza en un conjunto de ejemplos de entrenamiento, ahora se puede definir la ganancia de información. Esta medida se refiere a la reducción esperada en la entropía, que se consigue al particionar los ejemplos de acuerdo a un atributo. Específicamente, la ganancia de información de un atributo A relativa a un conjunto de ejemplos S, *Ganancia(A)*, se define como

$$Ganancia(A) = I(S) - \sum_{v \in Valores(A)} \frac{|S_v|}{|S|} I(S_v) \tag{2.4}$$

donde $Valores(A)$ es el conjunto de los valores posibles del atributo A, y S_v es el subconjunto de S que tiene el valor v en el atributo A. Cabe destacar que el primer término en la ecuación 2.4 es la entropía del conjunto original S, mientras que el segundo término es el valor esperado de la entropía después de que S se particiona de acuerdo a los valores del atributo A. La entropía esperada que describe este segundo término es simplemente la suma de las entropías de cada subconjunto S_v, ponderadas por la fracción de ejemplos $\frac{|S_v|}{|S|}$, donde $|S_v|$ es la cardinalidad de S_v. Por lo tanto, $Ganancia(A)$ es la disminución esperada en la entropía, que se obtiene al particionar el conjunto S de acuerdo a los valores del atributo A.

El algoritmo ID3 calcula la ganancia de información para cada atributo. El atributo con la mayor ganancia de información (o mayor reducción de entropía) se escoge como el atributo prueba para el conjunto dado S. Se crea un nodo y se etiqueta con el atributo, se generan ramas para cada valor del atributo, y se particionan los ejemplos adecuadamente. Después, el proceso se repite con cada subconjunto de datos, hasta que alguna de las condiciones de paro (comentadas en la sección anterior) del algoritmo se cumple.

Ejemplo del algoritmo ID3

Para ilustrar cómo funciona el algoritmo ID3, se considerarán los catorce ejemplos de entrenamiento de la tabla 2.1 (adaptada de Mitchell [65]). Estos ejemplos se conforman por los atributos *Pronóstico, Temperatura, Humedad* y *Viento*, que describen sábados por la mañana, y que predicen el atributo clase *JugarTenis* (es decir, si es conveniente o no jugar tenis).

Tabla 2.1: Ejemplos de entrenamiento que se emplearán para decidir si es conveniente o no Jugar Tenis.

Día	**Pronóstico**	**Temperatura**	**Humedad**	**Viento**	**JugarTenis**
D1	Soleado	Caliente	Alta	Débil	No
D2	Soleado	Caliente	Alta	Fuerte	No
D3	Nublado	Caliente	Alta	Débil	Si
D4	Lluvioso	Templado	Alta	Débil	Si
D5	Lluvioso	Fresco	Normal	Débil	Si
D6	Lluvioso	Fresco	Normal	Fuerte	No
D7	Nublado	Fresco	Normal	Fuerte	Si
D8	Soleado	Templado	Alta	Débil	No
D9	Soleado	Fresco	Normal	Débil	Si
D10	Lluvioso	Templado	Normal	Débil	Si
D11	Soleado	Templado	Normal	Fuerte	Si
D12	Nublado	Templado	Alta	Fuerte	Si
D13	Nublado	Caliente	Normal	Débil	Si
D14	Lluvioso	Templado	Alta	Fuerte	No

El primer paso consiste en seleccionar el nodo que será la raíz del árbol. Por lo tanto, ID3 aplica la ganancia de información (ecuación 2.4) a cada uno de los atributos (*Pronóstico, Temperatura, Humedad* y *Viento*), y después selecciona aquel que reporta la mayor ganancia. La figura 2.5 muestra el cálculo de la ganancia de información para dos de estos atributos.

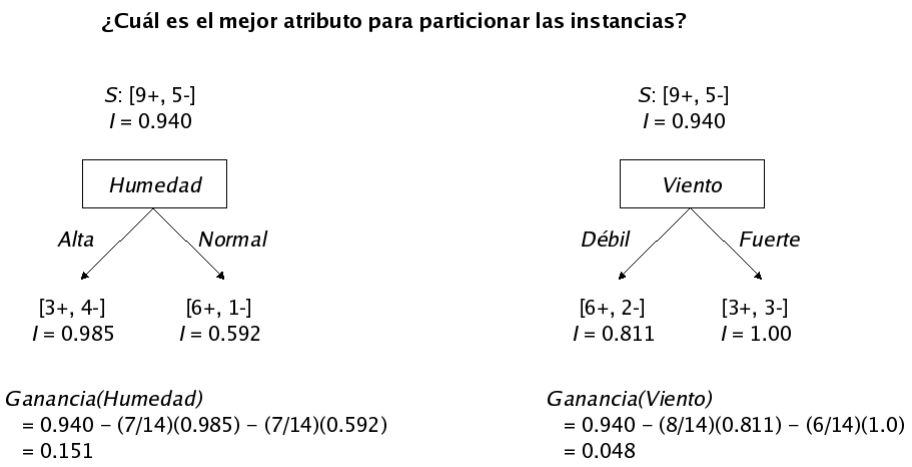

Figura 2.5: *Humedad* proporciona una mayor ganancia de información que *Viento*. Aquí, I significa entropía y S es el conjunto original de ejemplos. S contiene 9 ejemplos positivos (con valor de clase *Si*) y 5 negativos (valor de clase *No*), [9+, 5-], la ordenación de éstos de acuerdo a *Humedad* produce conjuntos de [3+, 4-] (*Humedad = Alta*) y [6+, 1-] (*Humedad = Normal*). La información que se gana con este particionamiento es de .151, comparada con una ganancia de .048 para el atributo *Viento*.

Los valores de la ganancia de información para los cuatro atributos son

Ganancia(Pronóstico) = 0.246

Ganancia(Humedad) = 0.151

Ganancia(Viento) = 0.048

Ganancia(Temperatura) = 0.029

De acuerdo a la ganancia de información, el atributo *Pronóstico* proporciona la mejor predicción para la clase, *JugarTenis*, sobre los ejemplos de entrenamiento. Por lo tanto, *Pronóstico* se selecciona como el atributo prueba para el nodo raíz, y se crean ramas para cada uno de sus valores posibles (*Soleado, Nublado,* y *Lluvioso*). El árbol de decisión parcial que resulta de este particionamiento se muestra en la figura 2.6, junto con los ejemplos de entrenamiento divididos entre cada nuevo nodo descendiente. Cada ejemplo para el que *Pronóstico = Nublado* pertenece a la clase *Si*. Por lo tanto, este nodo del árbol se convierte

en una hoja con la clasificación *JugarTenis = Si*. Por el contrario, los descendientes que corresponden a *Pronóstico = Soleado* y *Pronóstico = Lluvioso* aún no tienen entropía cero, por lo que el árbol de decisión tendrá que ser desarrollado para estos nodos.

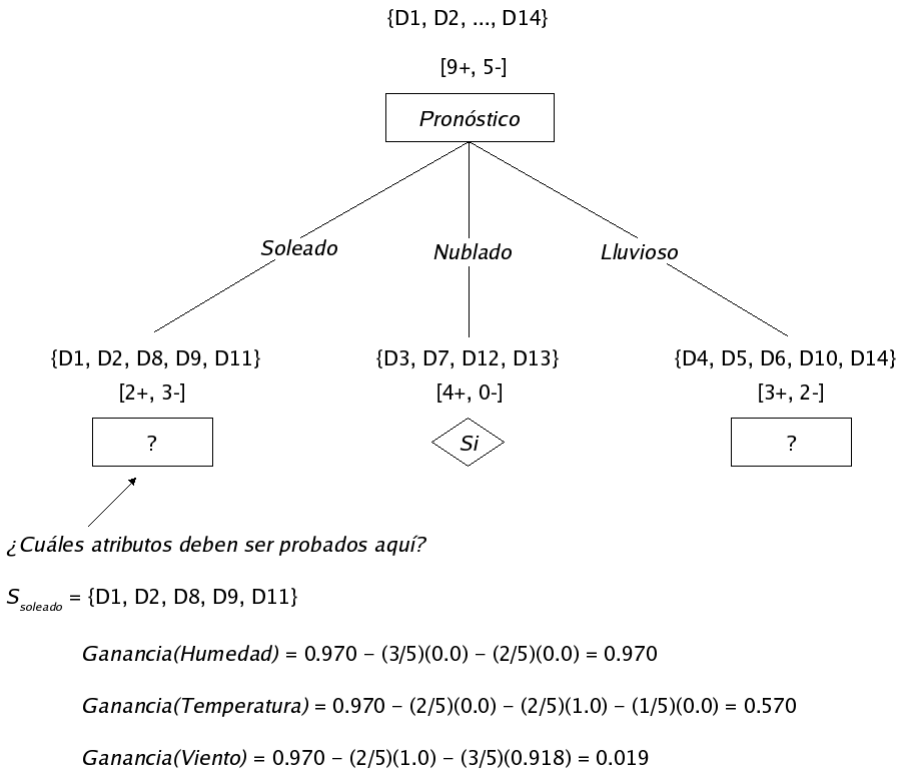

Figura 2.6: Árbol de decisión aprendido parcialmente que resultó de seleccionar el nodo raíz. Los ejemplos de entrenamiento están divididos en los nodos descendientes que les corresponden. El descendiente de *Nublado* solamente tiene ejemplos positivos, por lo tanto, se convierte en un nodo hoja con clasificación *Si*. Los otros dos nodos serán expandidos aún más, al seleccionar el atributo con la mayor ganancia de información con relación a los nuevos subconjuntos de ejemplos.

El proceso de seleccionar un nuevo atributo y particionar los ejemplos de entrenamiento se repite para cada nodo descendiente no terminal, aunque esta vez solamente se usarán los ejemplos de entrenamiento asociados con el nodo correspondiente. Los atributos que

han sido seleccionados como nodos del árbol deben ser descartados, para que no aparezcan más de una vez a lo largo de cualquier camino del árbol. La figura 2.7 muestra el árbol de decisión final aprendido por ID3 con los catorce ejemplos de entrenamiento de la tabla 2.1.

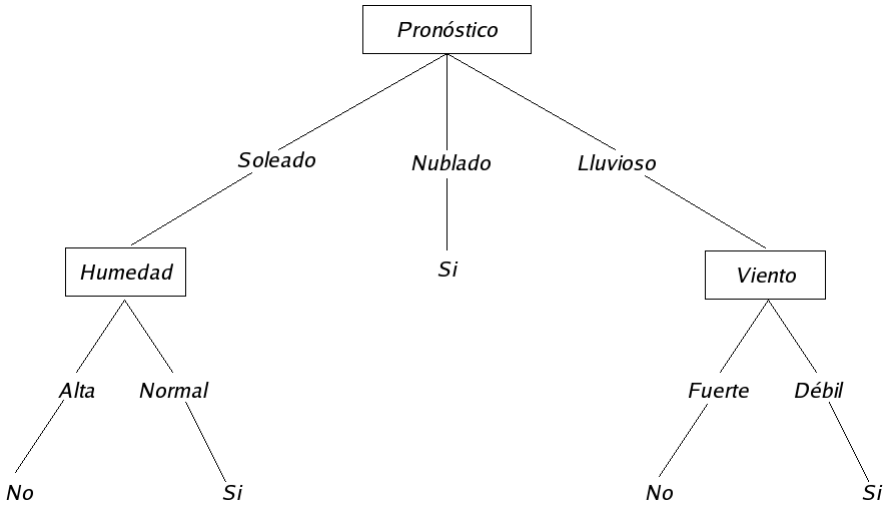

Figura 2.7: Árbol de decisión para decidir si es conveniente o no ir a jugar tenis un sábado por la mañana. Para conocer el valor de la clase (*Si* o *No*) que le corresponde a un ejemplo, éste debe recorrer el árbol desde la raíz hasta un nodo hoja, contestando cada una de las pruebas de los atributos de acuerdo a los valores de los atributos de este ejemplo.

2.2.2. Clasificadores Bayesianos

Los clasificadores Bayesianos son clasificadores estadísticos, que predicen las probabilidades de membresía de clase, como la probabilidad que un ejemplo dado pertenezca a una clase particular [32].

La clasificación Bayesiana se basa en el teorema de Bayes, que será descrito más adelante. Un clasificador Bayesiano simple llamado Bayes Ingenuo (NB - Naive Bayes) asume que el efecto del valor de un atributo sobre una clase dada es independiente de los valores de los otros atributos. Esta suposición se conoce como *independencia condicional de clase* y se hace para simplificar los cálculos involucrados. Las *Redes de Creencias Bayesianas* (BBN

- Bayesian Belief Networks) son modelos gráficos, que a diferencia de NB, permiten la dependencia entre los atributos.

En esta sección se presentan notaciones básicas de probabilidad y el teorema de Bayes. Posteriormente, se revisan el clasificador NB (que es ejecutado por el MAS) y las BBN.

Teorema de Bayes

Sea X un ejemplo de datos cuya etiqueta de clase se desconoce. H es la hipótesis que declara que el ejemplo X pertenece a una clase específica C. En problemas de clasificación, se desea determinar $P(H|X)$, la probabilidad que la hipótesis H se sostenga dado el ejemplo de datos observado X.

$P(H|X)$ es la *probabilidad a posterior*, de H condicionada sobre X. Por ejemplo, consideremos que los ejemplos de datos representan frutas, descritas por su color y su forma. Supongamos que X es roja y redonda, y que H es la hipótesis que X es una manzana. Entonces $P(H|X)$ refleja la confianza que X es una manzana dado que se ha visto que X es roja y redonda. Por el contrario, $P(H)$ es la *probabilidad a prior*, de H. Para el ejemplo, ésta es la probabilidad que cualquier ejemplo de datos dado es una manzana. La probabilidad a posterior, $P(H|X)$, requiere más información (como conocimiento del dominio) que la probabilidad a prior, $P(H)$, que es independiente de X.

De igual forma, $P(X|H)$ es la verosimilitud de X condicionada sobre H. Ésta es la probabilidad que X es roja y redonda dado que se conoce que X es una manzana. $P(X)$ es la probabilidad a prior de X. Ésta es la probabilidad que un ejemplo de datos desde el conjunto de frutas es roja y redonda.

$P(X)$, $P(H)$, y $P(X|H)$ pueden ser estimadas a partir de los datos dados, como se verá en la siguiente sección. $P(H|X)$ se calcula mediante el *teorema de Bayes* , al emplear $P(H)$, $P(X)$, y $P(X|H)$. El teorema de Bayes está dado por

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}. \tag{2.5}$$

En la siguiente sección, se ilustrará el uso del teorema de Bayes en el clasificador NB.

Bayes Ingenuo (NB)

Como ya se comentó, el clasificador NB asume la independencia condicional de clase (los atributos son independientes dada la clase), que le permite simplificar la tarea de aprendizaje. Aunque esta suposición de independencia generalmente es pobre, en la práctica NB compite frecuentemente bien con los clasificadores más sofisticados [30]. Varios estudios empíricos que analizan clasificadores de árboles de decisión, redes neuronales y NB han encontrado que éste último logra el mismo desempeño que los dos primeros en algunos dominios [32].

A continuación se describe el clasificador NB, o clasificador Bayesiano simple, haciendo referencia al algoritmo 2, que es una versión de este clasificador [32].

Algoritmo 2 Bayes-Ingenuo. Asigna la clase con la probabilidad a posterior más alta al ejemplo proporcionado.

Entrada: Los ejemplos de entrenamiento, *ejemplos*, representados por atributos discretos. El ejemplo que se va a clasificar, representado por un vector de n dimensiones, $X = (x_1, x_2, \ldots, x_n)$, donde x_k es el valor del atributo A_k, para $1 \leq k \leq n$.

Salida: La *clase* predicha para el ejemplo.

1: Sea s el número total de *ejemplos*;
2: **para cada** valor de clase C_i **hacer**
3: **si** es posible conocer la probabilidad a prior, $P(C_i)$, de la clase C_i **entonces**
4: Sea s_i el número de *ejemplos* de la clase C_i;
5: Calcular $P(C_i)$, como s_i/s;
6: **fin si**
7: Asignar a la probabilidad a posterior de X condicionada sobre C_i el valor de uno, $P(X|C_i) = 1$;
8: **para cada** atributo-valor, $A_k - x_k$, de X **hacer**
9: Sea s_{ik} el número de *ejemplos* de la clase C_i que tienen el valor x_k en el atributo A_k;
10: Calcular $P(x_k|C_i)$, como s_{ik}/s_i;
11: Asignar a $P(X|C_i)$ el resultado de multiplicar su valor anterior por $P(x_k|C_i)$, $P(X|C_i) = P(X|C_i) * P(x_k|C_i)$;
12: **fin para**
13: Obtener la probabilidad a posterior de C_i dado X al multiplicar $P(X|C_i)$ por $P(C_i)$, $P(C_i|X) = P(X|C_i) * P(C_i)$;
14: **fin para**
15: **regresa** la *clase* con la probabilidad a posterior más alta, condicionada sobre X.

1. Cada ejemplo de datos se representa por un vector de características n-dimensional, $X = (x_1, x_2, \ldots, x_n)$, que representa n medidas hechas sobre el ejemplo de n atributos, respectivamente, $A_1, A_2, \ldots, A_n$.

2. Suponer que hay m clases, $C_1, C_2, \ldots, C_m$. Dado un ejemplo de datos desconocido (que no tiene etiqueta de clase), X, el clasificador predecirá que X pertenece a la clase que tiene la probabilidad a posterior más alta, condicionada sobre X. Es decir, el clasificador Bayesiano ingenuo le asigna a un ejemplo desconocido X la clase C_i si y sólo si

$$P(C_i|X) > P(C_j|X) \ \ para\ 1 \leq j \leq m,\ j \neq i. \tag{2.6}$$

$P(C_i|X)$ se maximiza y la clase C_i se conoce como la *hipótesis a posterior máxima*. Por el teorema de Bayes (ecuación 2.5),

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}. \tag{2.7}$$

3. Debido a que $P(X)$ es constante para todas las clases, solamente $P(X|C_i)P(C_i)$ necesita ser maximizada. Si las probabilidades a prior de las clases son desconocidas, entonces comúnmente se asume que las clases son igualmente probables, es decir, $P(C_1) = P(C_2) = \ldots = P(C_m)$, y por lo tanto solo se maximizaría $P(X|C_i)$. De otra forma, se maximiza $P(X|C_i)P(C_i)$. Las probabilidades a prior pueden ser estimadas por $P(C_i) = \frac{s_i}{s}$ (paso 5), donde s_i es el número de ejemplos de entrenamiento de la clase C_i (paso 4), y s es el número total de ejemplos de entrenamiento (paso 1).

4. Dados conjuntos de datos con muchos atributos, sería computacionalmente caro calcular $P(X|C_i)$. Para reducir el cómputo en la evaluación de $P(X|C_i)$, se hace la suposición ingenua de independencia condicional de clase. Ésta supone que los valores de los atributos son condicionalmente independientes dada la clase del ejemplo, es decir, no hay relaciones de dependencia entre los atributos. Por lo tanto,

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i). \tag{2.8}$$

Las probabilidades $P(x_1|C_i), P(x_2|C_i), \ldots, P(x_n|C_i)$ pueden ser estimadas a partir de los ejemplos de entrenamiento, donde

- Si A_k es categórico, entonces $P(x_k|C_i) = \frac{s_{ik}}{s_i}$ (paso 10), donde s_{ik} es el número de ejemplos de entrenamiento de clase C_i que tienen el valor x_k para A_k (paso

9), y s_i es el número de ejemplos de entrenamiento que pertenecen a la clase C_i (paso 4).

- Si A_k es continuo, entonces típicamente se asume que el atributo tiene una distribución Gaussiana

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}) = \frac{1}{\sqrt{2\pi}\sigma_{C_i}} e^{-\frac{(x_k - \mu_{C_i})^2}{2\sigma_{C_i}^2}}, \tag{2.9}$$

donde $g(x_k, \mu_{C_i}, \sigma_{C_i})$ es la función de densidad Gaussiana (normal) para el atributo A_k, mientras que μ_{C_i} y σ_{C_i} son la media y la desviación estándar, respectivamente, dados los valores del atributo A_k para los ejemplos de entrenamiento de la clase C_i.

5. Para clasificar un ejemplo desconocido X, $P(X|C_i)P(C_i)$ se evalúa para cada clase C_i (paso 13). El ejemplo X se asigna a la clase C_i si y sólo si

$$P(X|C_i)P(C_i) > P(X|C_j)P(C_j) \ \ para\ 1 \leq j \leq m,\ j \neq i. \tag{2.10}$$

Es decir, se asigna a la clase C_i para la cual $P(X|C_i)P(C_i)$ es el máximo (paso 15).

En la siguiente sección se presenta un ejemplo del clasificador NB.

Ejemplo del clasificador NB

Se desea predecir con el clasificador NB si es conveniente o no ir a jugar tenis un sábado por la mañana, dados los ejemplos de entrenamiento (tabla 2.1) que se usaron para ejemplificar el algoritmo ID3. Recordemos que estos ejemplos se componen por los atributos *Pronóstico, Temperatura, Humedad y Viento*. El atributo clase, *JugarTenis*, tiene dos valores distintos, {*Si,No*}; por lo tanto, hay dos clases diferentes ($m = 2$). Sea C_1 la clase *JugarTenis = "Si"* y C_2 *JugarTenis = "No"*. Hay 9 ejemplos de clase *"Si"* y 5 de clase *"No"*. El ejemplo desconocido que se desea clasificar es

X = (Pronóstico = "Soleado", Temperatura = "Fresco", Humedad = "Alta", Viento = "Fuerte").

Se necesita maximizar $P(X|C_i)P(C_i)$, para $i = 1, 2$. $P(C_i)$, la probabilidad a prior de cada clase, puede ser calculada a partir de los ejemplos de entrenamiento:

P(JugarTenis = "Si") = 9/14 = 0.64

P(JugarTenis = "No") = 5/14 = 0.36

Para calcular $P(X|C_i)$, para $i = 1, 2$, se computan las siguientes probabilidades condicionales:

P(Pronóstico = "Soleado"|JugarTenis = "Si") = 2/9 = 0.22

P(Pronóstico = "Soleado"|JugarTenis = "No") = 3/5 = 0.60

P(Temperatura = "Fresco"|JugarTenis = "Si") = 3/9 = 0.33

P(Temperatura = "Fresco"|JugarTenis = "No") = 1/5 = 0.20

P(Humedad = "Alta"|JugarTenis = "Si") = 3/9 = 0.33

P(Humedad = "Alta"|JugarTenis = "No") = 4/5 = 0.80

P(Viento = "Fuerte"|JugarTenis = "Si") = 3/9 = 0.33

P(Viento = "Fuerte"|JugarTenis = "No") = 3/5 = 0.60

Usando las probabilidades de arriba, se obtiene:

P(X|JugarTenis = "Si") = 0.22 x 0.33 x 0.33 x 0.33 = 0.0079

P(X|JugarTenis = "No") = 0.60 x 0.20 x 0.80 x 0.60 = 0.0576

P(X|JugarTenis = "Si")P(JugarTenis = "Si") = 0.0079 x 0.64 = 0.0051

P(X|JugarTenis = "No")P(JugarTenis = "No") = 0.0576 x 0.36 = 0.0207

Por lo tanto, el clasificador NB predice *JugarTenis = "No"* para el ejemplo *X*. Al normalizar las últimas dos cantidades, de tal forma que su suma sea igual a uno, es posible calcular la probabilidad condicional que "*No*" es adecuado jugar tenis dado que *Pronósti-*

co = "Soleado", Temperatura = "Fresco", Humedad = "Alta", Viento = "Fuerte". Esta probabilidad es igual a $\frac{0{,}0207}{0{,}0051+0{,}0207} = 0{,}8023$.

Redes de Creencias Bayesianas (BBN)

Las BBN especifican distribuciones de probabilidad conjunta. Estas BBN permiten definir la independencia condicional de clase entre subconjuntos de variables. Esta independencia condicional de clase se puede representar mediante un modelo gráfico de relaciones probabílistas. Las BBN también son conocidas como *redes de creencias*, *redes Bayesianas*, y *redes probabílistas*.

Una BBN se define por dos componentes. En primer lugar, por un *grafo acíclico dirigido*, donde cada nodo representa una variable aleatoria (discreta o continua) y cada arco una dependencia probabílista. Si existe un arco que une el nodo Y con el Z, entonces Y es un padre o predecesor inmediato de Z, y Z es un descendiente de Y. Cada variable es condicionalmente independiente de sus no descendientes dados sus padres (condición de Markov) [21].

La figura 2.8(a) muestra una BBN (adaptada de Han y Kamber [32]) para seis variables booleanas. Los arcos permiten representar conocimiento causal. Por ejemplo, la historia familiar de cáncer de pulmón de una persona y si ésta es o no fumadora, influyen en que esta persona tenga cáncer de pulmón. Los arcos también muestran que la variable *Cáncer de Pulmón* es condicionalmente independiente de *Enfisema*, dados sus padres, *Historia Familiar* y *Fumador*. Es decir, una vez que los valores de *Historia Familiar* y *Fumador* se conocen, *Enfisema* no proporciona información adicional respecto a *Cáncer de Pulmón*.

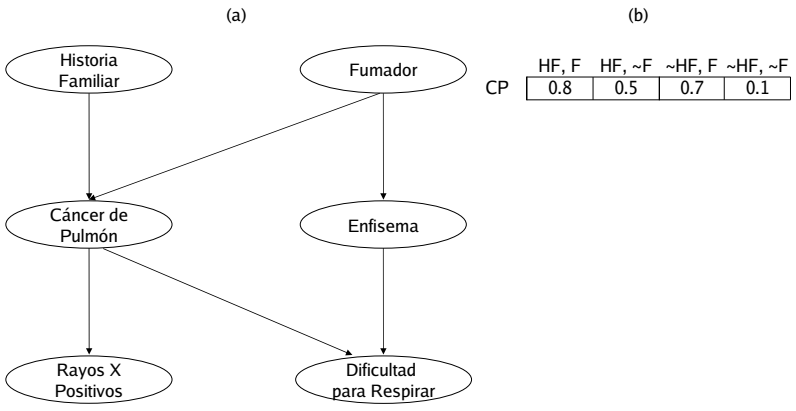

Figura 2.8: (a) Una red de creencias Bayesiana. (b) La tabla de probabilidad condicional para los valores de la variable *Cáncer de Pulmón (CP)*, en la primera fila se muestran las combinaciones posibles de los valores de sus padres, *Historia Familiar (HF)* y *Fumador (F)*.

El segundo componente que define una BBN consiste de una *tabla de probabilidad condicional (TPC)* para cada variable. La TPC para una variable Z especifica la distribución condicional $P(Z|Padres(Z))$, donde $Padres(Z)$ son las variables que tienen un arco apuntando hacia Z. La figura 2.8(b) muestra una TPC para *Cáncer de Pulmón*. La probabilidad condicional para cada valor de *Cáncer de Pulmón* está dada por cada combinación posible de valores de sus padres. Por ejemplo, el valor más a la izquierda se obtiene con la siguiente expresión

$$P(\textit{Cáncer de Pulmón} = \textit{“Si”}|\textit{Historia Familiar} = \textit{“Si”}, \textit{Fumador} = \textit{“Si”}) = 0.8$$

La probabilidad conjunta de cualquier ejemplo $(z_1, \ldots, z_n)$ que corresponde a las variables o atributos $Z_1, \ldots, Z_n$ se computa por

$$P(z_1, \ldots, z_n) = \prod_{i=1}^{n} P(z_i | Padres(Z_i)), \tag{2.11}$$

donde los valores para $P(z_i|Padres(Z_i)$ se obtienen de la TPC de Z_i.

Un nodo dentro de la red puede ser seleccionado como un nodo de "salida", que representa un atributo clase. Puede existir más de un nodo de salida [32], por lo tanto, el proceso de clasificación en vez de regresar una etiqueta de clase, devuelve una distribución de probabilidad para el atributo clase, es decir, predice la probabilidad de cada clase.

Existen varios algoritmos que aprenden una BBN a partir de los ejemplos de entrenamiento. Entre estos algoritmos se encuentran: *K2* [23], que aprende la estructura de una BBN, asumiendo que existe un ordenamiento en las variables que conforman los ejemplos de entrenamiento; *NB* [19] (presentado en la sección 2.2.2), que aprende únicamente los parámetros de una BBN, es decir, las probabilidades y *TAN* [48], que aprende tanto la estructura como los parámetros de una BBN de estructura restringida.

2.3. Evaluación de clasificadores

El siguiente paso después de construir un clasificador es conocer su capacidad para clasificar correctamente ejemplos que aún no ha visto (que no estuvieron presentes en la etapa de su construcción). Para este fin se emplean frecuentemente los métodos holdout y validación cruzada, que se presentan en las secciones 2.3.1 y 2.3.2, respectivamente. Una variación del último de estos métodos, que se conoce como validación cruzada estratificada, es el que se utilizó en este trabajo, para validar los modelos aprendidos por los agentes del MAS.

2.3.1. Método Holdout

El método "holdout" particiona aleatoriamente los datos en dos subconjuntos independientes llamados conjunto de entrenamiento y conjunto de prueba, o conjunto holdout.

Comúnmente dos terceras partes de los datos se designan al conjunto de entrenamiento y el tercio restante al conjunto de prueba. Con el primero de éstos se construye un clasificador, cuya exactitud se estima con el conjunto de prueba (figura 2.9, adaptada de Han y Kamber [32]). Esta exactitud se calcula al dividir el número de ejemplos clasificados correctamente entre el número de ejemplos del conjunto de prueba. Holdout es un estimador pesimista, debido a que solamente una porción de los datos (conjunto de entrenamiento) se usa para aprender el clasificador [32, 58].

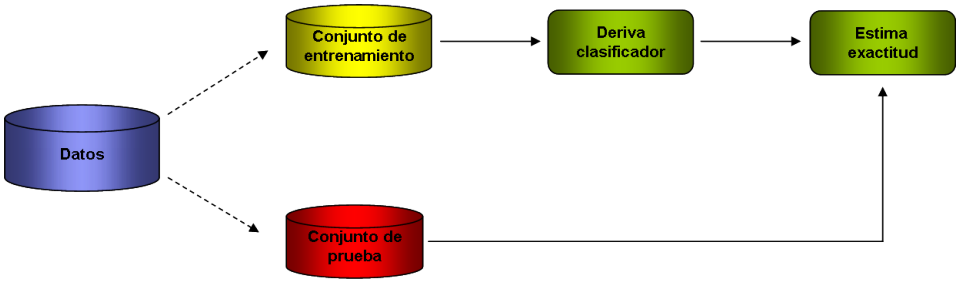

Figura 2.9: Estimación de la exactitud del clasificador con el método holdout.

Es importante que los conjuntos de datos que se usan para entrenamiento y prueba representen las características de la población bajo estudio. Una forma de conseguirlo es verificar que cada clase en el conjunto de datos original esté representada aproximadamente en la misma proporción en los conjuntos de entrenamiento y prueba. Este procedimiento se conoce como estratificación y en este caso particular, *holdout estratificado*. Aunque generalmente es valioso realizar este método, la estratificación proporciona solamente una garantía primitiva contra la representación irregular en los conjuntos de entrenamiento y prueba [31].

Una manera más general de mitigar cualquier sesgo causado por la muestra particular escogida para holdout es repetir el proceso completo, entrenamiento y prueba, varias veces con diferentes muestras aleatorias. En cada iteración una cierta proporción (dos terceras partes) de los datos se seleccionan aleatoriamente para entrenamiento, posiblemente con estratificación, y el resto de estos datos se usan para prueba. Las exactitudes estimadas de las diferentes iteraciones se promedian para obtener la exactitud estimada general. Este

método se conoce como *holdout repetido* [31].

2.3.2. Método de Validación Cruzada

La validación cruzada es un método que se emplea para prevenir el sobreajuste en los modelos aprendidos (un modelo sobreajustado es aquel que ha incorporado ciertas anomalías de los datos de entrenamiento, las cuales no están presentes en la población general [32]). La idea básica de este método es intentar estimar qué tan bien el modelo predecirá datos que aún no ha visto. La validación cruzada se puede usar con cualquier otro método para construcción de árboles (por ejemplo, con poda) para seleccionar un árbol con buen desempeño de predicción [64].

En un método de validación cruzada típico (k-hojas), un conjunto de datos S se particiona aleatoriamente dentro de k subconjuntos (hojas) disjuntos de aproximadamente igual tamaño, S_1, S_2, ..., S_k. Cada una de las k hojas después se usa en turno como el conjunto de prueba, las hojas restantes (k - 1) son usadas como el conjunto de entrenamiento. Posteriormente, se construye un clasificador desde el conjunto de entrenamiento y su exactitud se evalúa con el conjunto de prueba. Este proceso se repite k veces, usando una hoja diferente como el conjunto de prueba cada vez. La exactitud estimada por este método se obtiene al dividir el número total de ejemplos clasificados correctamente entre el número de ejemplos del conjunto de datos S [58, 72].

Una extensión de la validación cruzada regular es la validación cruzada *estratificada*. En la validación cruzada estratificada de k hojas, un conjunto S se particiona en k hojas de tal forma que cada clase se distribuye proporcionalmente dentro de las k hojas. De esta manera la distribución de clase en cada hoja es similar a la del conjunto de datos original S. En contraste, la validación cruzada regular particiona aleatoriamente S dentro de k hojas sin considerar las distribuciones de clase. Un escenario posible con la validación cruzada regular es que una cierta clase podría estar distribuida irregularmente - algunas hojas contienen más casos de esa clase que otras hojas. Esta distorsión en las distribuciones de clase puede causar una estimación de la exactitud menos confiable. Kohavi [58] reportó que la validación

cruzada estratificada tiene un mejor desempeño (un sesgo y una varianza más pequeños) que la validación cruzada regular.

Una dificultad de la validación cruzada surge cuando ésta se usa con métodos que requieren un intenso cómputo como las redes neuronales, debido a que la repetición (k) del ciclo de aprendizaje puede demandar mucho esfuerzo computacional [64].

A continuación ejemplificamos el método de validación cruzada con los ejemplos de entrenamiento del juego de tenis que se presentaron en la tabla 2.1. Supongamos que el número de hojas es igual a siete, $k = 7$, por lo tanto obtendremos siete subconjuntos de datos. $S_1 = \{D1, D3\}$, $S_2 = \{D2, D4\}$, $S_3 = \{D5, D6\}$, $S_4 = \{D7, D8\}$, $S_5 = \{D9, D14\}$, $S_6 = \{D10, D11\}$ y $S_7 = \{D12, D13\}$. Dado que $k = 7$, siete veces se construirá y probará el modelo. En la primera iteración, tomaremos a S_1 como el conjunto de prueba (tabla 2.3) y a las hojas restantes ($S_2, \ldots, S_7$) como el conjunto de entrenamiento (tabla 2.2). Con éste aprenderemos un árbol de decisión con ID3 (ver algoritmo 1).

Tabla 2.2: Conjunto de entrenamiento que se utilizará en la primera iteración del método de validación cruzada.

Día	**Pronóstico**	**Temperatura**	**Humedad**	**Viento**	**JugarTenis**
D2	Soleado	Caliente	Alta	Fuerte	No
D4	Lluvioso	Templado	Alta	Débil	Si
D5	Lluvioso	Fresco	Normal	Débil	Si
D6	Lluvioso	Fresco	Normal	Fuerte	No
D7	Nublado	Fresco	Normal	Fuerte	Si
D8	Soleado	Templado	Alta	Débil	No
D9	Soleado	Fresco	Normal	Débil	Si
D10	Lluvioso	Templado	Normal	Débil	Si
D11	Soleado	Templado	Normal	Fuerte	Si
D12	Nublado	Templado	Alta	Fuerte	Si
D13	Nublado	Caliente	Normal	Débil	Si
D14	Lluvioso	Templado	Alta	Fuerte	No

Tabla 2.3: Conjunto de prueba que se empleará en la primera iteración del método de validación cruzada.

Día	**Pronóstico**	**Temperatura**	**Humedad**	**Viento**	**JugarTenis**
D1	Soleado	Caliente	Alta	Débil	No
D3	Nublado	Caliente	Alta	Débil	Si

En primer lugar debemos seleccionar el nodo que será la raíz del árbol. Para esto calculamos la entropía del conjunto de entrenamiento (ecuación 2.1):

$$I([8+, 4-]) = -\,(8/12)\,log_2\,(8/12)\;-\;(4/12)\,log_2\,(4/12) = 0{,}918, \tag{2.12}$$

y posteriormente, obtenemos la ganancia de información (ecuación 2.4) para cada uno de los atributos, *Pronóstico*, *Temperatura*, *Humedad* y *Viento*. La figura 2.10 muestra este cálculo.

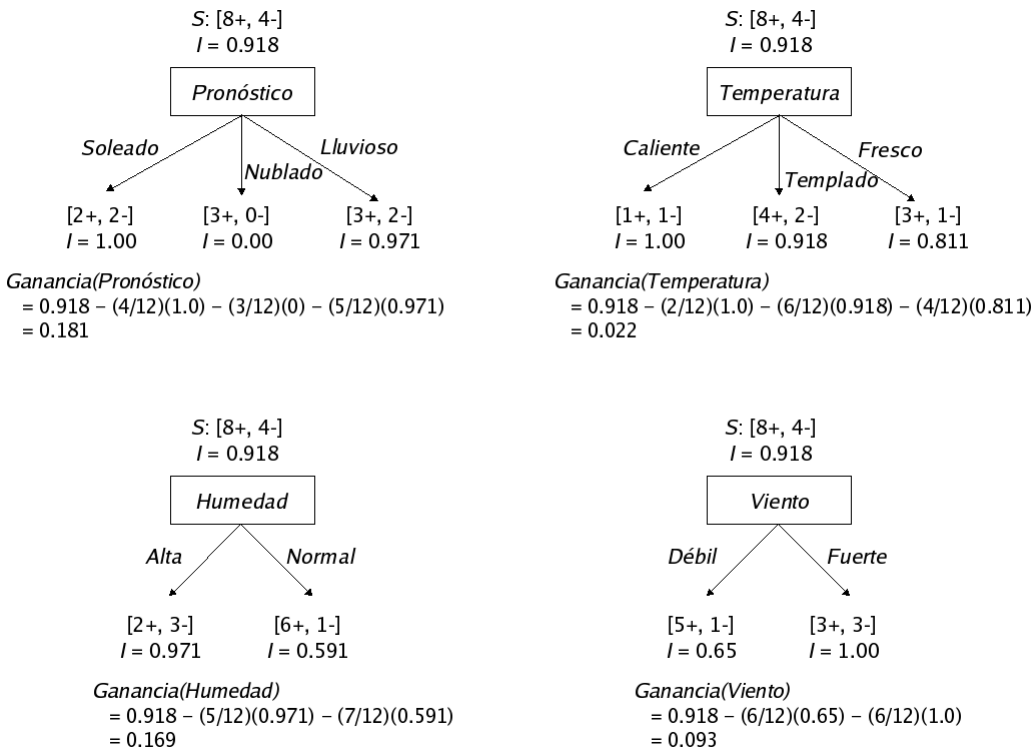

Figura 2.10: Cálculo de la ganancia de información para los atributos *Pronóstico*, *Temperatura*, *Humedad* y *Viento*.

Como recordaremos, en estos cálculos I significa entropía y S es el conjunto de ejemplos de entrenamiento. S contiene 8 ejemplos positivos y 4 negativos, [8+, 4-], y la ordenación de éstos de acuerdo a *Pronóstico* produce conjuntos de [2+, 2-] (*Pronóstico = Soleado*), [3+, 0-] (*Pronóstico = Nublado*) y [3+, 2-] (*Pronóstico = Lluvioso*). Como puede apreciarse en esta figura, este es el particionamiento con la mayor ganancia de información (0.181), por consiguiente, *Pronóstico* se selecciona como el nodo raíz (figura 2.11).

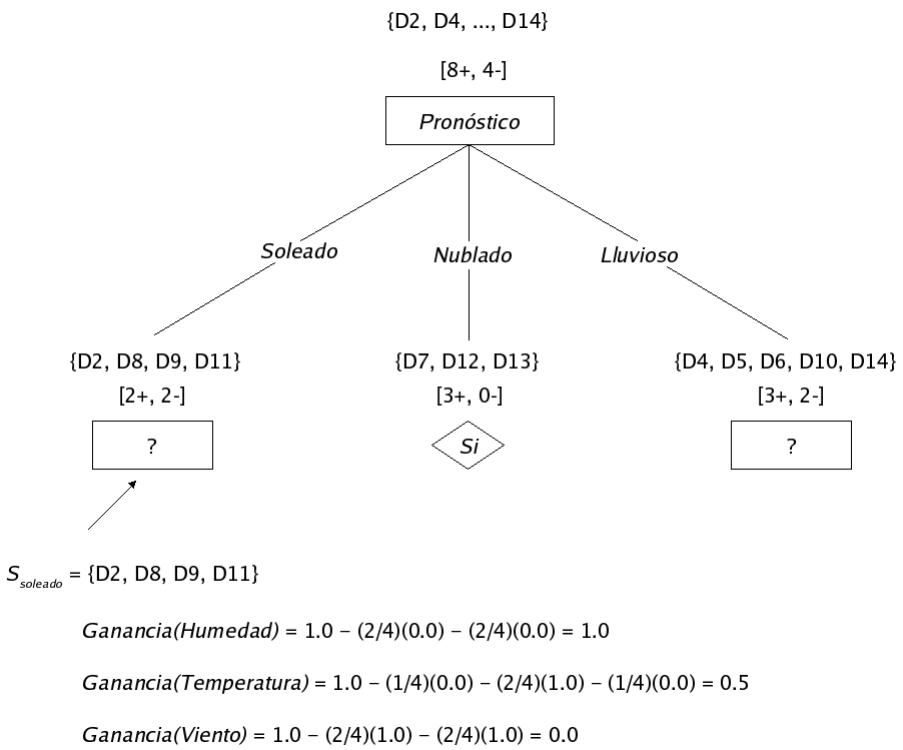

Figura 2.11: Árbol parcial que resulta de seleccionar *Pronóstico* como nodo raíz.

En esta figura podemos ver que los ejemplos de entrenamiento se dividen de acuerdo a los valores de *Pronóstico*: *Soleado*, *Nublado* y *Lluvioso*. Todos los ejemplos que tienen el valor *Nublado* son positivos, por lo tanto, el nodo que desciende de estos ejemplos se convierte en hoja con clasificación *Si*. Los otros dos nodos que se generan con los valores de *Soleado* y *Lluvioso* deben expandirse más, al seleccionar el atributo con la mayor ganancia de información con relación a los nuevos subconjuntos de ejemplos.

En la figura 2.12 se muestra el árbol final que se aprendió en la primera iteración del método de validación cruzada, con el algoritmo ID3 y los 12 ejemplos de entrenamiento de la tabla 2.2. Una vez terminada la etapa de aprendizaje del modelo, es necesario validarlo con el conjunto de prueba de la tabla 2.3.

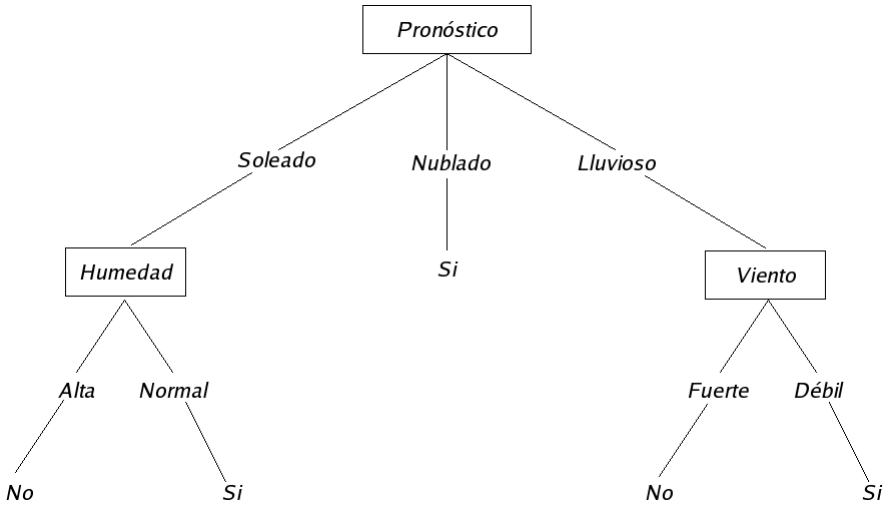

Figura 2.12: Árbol de decisión aprendido en la primera iteración del método de validación cruzada.

La validación consiste en obtener el porcentaje de ejemplos clasificados correctamente por este modelo. Para esto, se toma cada uno de los ejemplos del conjunto de prueba y con éste se recorre el árbol desde la raíz hasta las hojas, que dan la clasificación predicha del ejemplo en cuestión. Si esta clasificación coincide con el valor de la clase asignada al ejemplo, entonces éste ha sido clasificado correctamente. Dado lo anterior, el número de ejemplos clasificados correctamente por este árbol de decisión es igual a dos. Ahora es necesario repetir seis veces el mismo proceso llevado a cabo, sin olvidar que se debe tomar una hoja diferente como el conjunto de prueba cada vez. Al terminar todas las iteraciones se obtiene el número total de ejemplos clasificados correctamente, que se divide entre catorce (que es el número de ejemplos del conjunto de datos original), para conocer el porcentaje de ejemplos clasificados correctamente por el modelo aprendido.

2.4. Agentes de Creencias, Deseos e Intenciones (BDI)

Como ya se indicó, los agentes BDI conforman uno de los enfoques que se utiliza frecuentemente para desarrollar agentes inteligentes. Por esta razón, el MAS que se desarrolló en este trabajo está compuesto de este tipo de agentes, que ejecutan principalmente tareas de preprocesamiento (reemplazo de valores faltantes y discretización de la base de datos), minería de datos (aplicación de los algoritmos ID3, C4.5, NB y TAN) y evaluación de patrones (con el método de validación cruzada estratificada).

De acuerdo a Wooldridge [46], un agente es un sistema de cómputo que está inmerso en algún ambiente, y que actúa autónomamente en este ambiente para satisfacer sus objetivos de diseño. La figura 2.13 (adaptada de Wooldridge [44]) muestra de forma abstracta un agente. En este diagrama se puede apreciar que el agente modifica su ambiente mediante las acciones que realiza, y a su vez, el comportamiento de este agente se ve afectado por las entradas del ambiente, que percibe mediante sus sensores. Es relevante destacar que la interacción entre el agente y su ambiente, usualmente es continua.

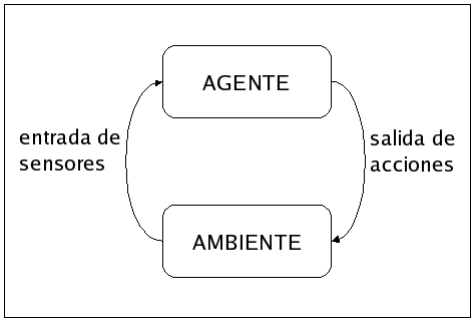

Figura 2.13: Agente.

A los agentes en general se les atribuyen una serie de propiedades, las cuales se enuncian a continuación [2, 11, 24, 47, 50]:

1. *Autonomía*: Un agente puede actuar sin vigilancia constante e intervención humana directa y puede tener algún tipo de control sobre sus acciones y estados internos.

2. *Interactividad*: Un agente puede interactuar con su ambiente y con otras entidades. La interactividad abarca todos los aspectos del comportamiento que un agente puede exhibir dentro de un ambiente. Las dos características dominantes que determinan unívocamente el estado del comportamiento de un agente son:

    a) *Reactividad*: Es la habilidad para responder a los cambios en el ambiente que se percibe, siempre y cuando sea necesario.

    b) *Proactividad*: La habilidad para tomar la iniciativa y actuar de tal forma que satisfaga la meta para la que ha sido utilizado.

3. *Sociabilidad*: Es la capacidad que tienen los agentes para interactuar con otros agentes (posiblemente humanos) para satisfacer sus objetivos de diseño.

4. *Adaptabilidad*: El agente se adecua automáticamente a los hábitos, métodos de trabajo y preferencias del usuario con base en la experiencia que este agente posee. El agente también se adapta de forma automática a los cambios que se producen en su ambiente.

5. *Cooperación*: Es la habilidad de colaborar para cumplir una meta común.

6. *Competitividad*: Un agente puede competir con otro agente, o negarle un servicio a un agente que percibe como su oponente. Este comportamiento es requerido en situaciones donde sólo un agente puede lograr el objetivo.

7. *Continuidad temporal*: Un agente es un proceso que se ejecuta continuamente, y no un programa que se corre una sola vez y después termina.

8. *Carácter*: Un agente puede exhibir características tal como personalidad y estado emocional.

9. *Movilidad*: Es la habilidad para transitar de un ambiente a otro, sin interrumpir su funcionamiento.

10. *Aprendizaje*: Un agente tiene la capacidad de modificar su comportamiento a través de la experiencia, de forma que logre un mejor desempeño en determinadas circunstancias.

Las propiedades que cada agente exhiba dependerá de las características de la aplicación en donde se use este agente. Para algunas aplicaciones puede ser más relevante que el agente aprenda a partir de su experiencia, mientras que para otras posiblemente importe que estos agentes sean competitivos (por ejemplo, en subastas) [2].

Es importante mencionar que en la literatura encontramos dos nociones del término agente, éstas son: *débil* y *fuerte* [47]. En el primer caso, un agente generalmente se implementa como un proceso de software que se ejecuta concurrentemente, que encapsula algún estado y que es capaz de comunicarse con otros agentes a través del paso de mensajes. Este tipo de agente cumple con las primeras tres propiedades (1-3) mencionadas anteriormente y se usa en las disciplinas de programación concurrente basada en objetos y en ingeniería de software basada en agentes. Mientras tanto, en el segundo caso, un agente se concibe e implementa como un sistema que posee ciertas actitudes mentales (como: conocimiento, creencias, intenciones y obligaciones [73]) que se aplican usualmente a los humanos; además de contar con algunas de las propiedades listadas previamente.

Por ejemplo, un softbot (robot de software) es un agente que satisface la noción débil de agencia. Este agente interactúa con un ambiente de software (por ejemplo UNIX) al emitir comandos e interpretar las respuestas del ambiente. El softbot sensa su ambiente mediante comandos (por ejemplo `ls` o `pwd` en UNIX) que le proporcionan información y lo modifica a través de comandos como `mv` o `compress`, en el caso de UNIX [47].

Una de las diferencias entre las agencias débil y fuerte es que ésta última expresa la noción de autonomía más intensamente, es decir, los agentes deciden por sí mismos si ejecutan o no una acción que ha sido requerida por otro agente. En cambio, debido a que los agentes débiles frecuentemente se implementan como objetos dentro del paradigma de programación orientada a objetos, estos agentes no tienen control sobre la ejecución de sus métodos públicos (procedimientos que ponen a disposición de cualquier agente) [24].

En [4], Guerra et al. comentan que uno de los modelos de agentes que ha adquirido gran relevancia en la comunidad de Inteligencia Artificial ha sido el de agentes intencionales BDI, debido a que cuenta con fundamentos filosóficos sólidos, que se basan en la actitud intencional de Dennett [16] y la teoría de planes, intenciones y razonamiento práctico de Bratman [39]. Además, este tipo de agentes cumple con la noción de agencia fuerte.

Los agentes BDI son sistemas que perciben continuamente su ambiente y que toman acciones para afectarlo, basándose en estas tres actitudes mentales: *creencias, deseos* e *intenciones* [9, 40]. Las creencias representan el estado actual del agente, es decir, la información que tiene de sí mismo, de su ambiente y de otros agentes. Los deseos o metas representan los estados que el agente desea alcanzar, al considerar sus estímulos internos o externos. Las intenciones representan el estado deliberativo del agente, es decir, son los deseos o metas con los que el agente se ha comprometido. Además de estos componentes, un agente BDI tiene una *biblioteca de planes*, que se pueden ver como recetas que representan el conocimiento procedimental del agente, y una *cola de eventos* donde se almacenan eventos (los cuales son percibidos desde el ambiente o generados por el agente para notificar una actualización de su base de creencias) y submetas internas (generadas por el agente cuando intenta alcanzar una meta). Con el fin de clarificar la noción de este tipo de agente se puede consultar la sección 2.5, donde se encuentra un ejemplo.

La arquitectura de un agente BDI se puede ver en la figura 2.14 y opera de la siguiente forma [71]:

1. Observar su ambiente y su estado interno, y si ha surgido un evento (externo o interno) actualiza la cola de eventos.

2. Generar nuevas instancias de planes cuyo evento disparador coincida con un evento en la cola de eventos (instancias de planes *relevantes*) y cuya precondición sea satisfecha mediante las creencias (instancias de planes *aplicables*).

3. Seleccionar una instancia del conjunto de instancias de planes aplicables.

4. Colocar la instancia seleccionada dentro de una *pila de intenciones* existente o nueva, dependiendo si el evento es una (sub)meta o no.

5. Seleccionar una pila de intenciones, tomar la instancia del plan que está en la cabeza de esta pila, y ejecutar el siguiente paso de esta instancia: si es una acción externa, la ejecuta, de otra forma, si es una acción interna (submeta), la inserta en la cola de eventos.

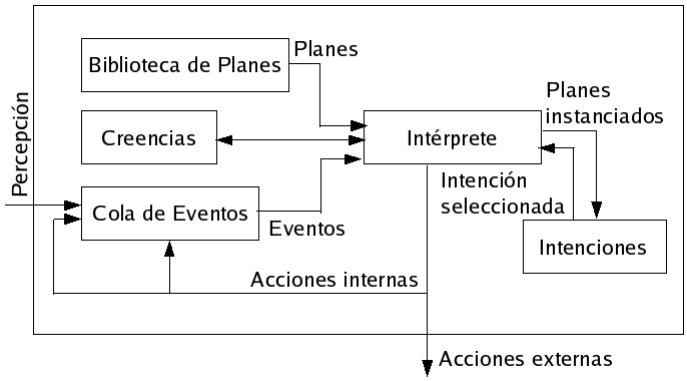

Figura 2.14: Arquitectura de un agente BDI.

El modelo BDI es interesante por varias razones. Primero, es intuitivo, es decir, todos tenemos un entendimiento informal de las nociones de creencias, deseos e intenciones. Segundo, se ha realizado un gran esfuerzo para formalizarlo. Rao y Georgeff han desarrollado lógicas BDI, que usan para axiomatizar las propiedades de los agentes BDI. Estas lógicas han sido extendidas por otros autores para tratar aspectos como la comunicación entre agentes [44].

2.5. AgentSpeak(L)

AgentSpeak(L) es el lenguaje que se eligió para implementar a los agentes BDI, del MAS desarrollado como parte de esta tesis, debido a que proporciona un marco de trabajo

abstracto y elegante para programar este tipo de agentes [4]. AgentSpeak(L) fue creado por Rao [9] y es una extensión de la programación lógica para la arquitectura de agentes BDI [45].

Un agente AgentSpeak(L) se define por un conjunto de creencias que proporcionan el estado inicial de la *base de creencias* del agente y un conjunto de planes que forman la *biblioteca de planes* [56]. Antes de explicar cómo se escribe un plan, es necesario introducir los conceptos de metas y eventos disparadores. En AgentSpeak(L) existen dos tipos de *metas*: metas de logro y metas de prueba. Las *metas de logro* son predicados (similares a las creencias) que se preceden con el operador '`!`', mientras que las *metas de prueba* se anteponen con el operador '`?`'. Una meta de logro declara que el agente desea alcanzar un estado del mundo donde el predicado asociado es verdadero. Una meta de prueba informa que el agente desea probar si el predicado asociado se encuentra en su base de creencias.

Un agente AgentSpeak(L) es un sistema de planeación reactivo. Los eventos a los que reacciona están relacionados a cambios en las creencias debido a la percepción del ambiente, o a cambios en las metas del agente que se originan desde la ejecución de planes disparados por eventos previos. Un *evento disparador* es aquel que puede iniciar la ejecución de un plan particular. Los planes son escritos por el programador para que sean disparados por la *adición* ('`+`') o *eliminación* ('`-`') de creencias o metas ("actitudes mentales" de los agentes AgentSpeak).

Un plan AgentSpeak(L) tiene la forma *cabeza* $\leftarrow$ *cuerpo*, donde la *cabeza* está formada por un evento disparador (que denota el propósito para ese plan), y una conjunción de literales de creencias que representan un *contexto*. La conjunción de literales en el contexto debe ser una consecuencia lógica de las creencias del agente, si el plan se va a *aplicar* en ese momento (solamente los planes aplicables pueden ser ejecutados). El *cuerpo* es una secuencia de acciones básicas o sub(metas) que el agente tiene que lograr (o probar) cuando el plan sea disparado. Las *acciones básicas* también se escriben como predicados, al emplear símbolos de predicados especiales llamados *símbolos de acción*.

A continuación se muestra un ejemplo de un agente BDI en AgentSpeak(L), el cual

fue tomado y modificado de Bordini y Hübner [54]. Este agente es instruido para detectar "manchas verdes" sobre rocas mientras se desplaza en Marte. En la 1ra. línea encontramos la creencia (`battery_charge(high)`) que informa que la batería está perfectamente cargada. De la 3era. a la 6ta. línea, tenemos el primer plan, el cual dice que cuando el agente perciba una mancha verde sobre una roca (una adición de creencia), debe intentar examinar esa roca, sin embargo, este plan solamente puede ser usado (sólo es aplicable) cuando las baterias no estén bajas. Para examinar la roca, tiene que recuperar desde su base de creencias las coordenadas que tiene asociadas con esa roca (`?location(Rock,Coordinates)`), después satisfacer la meta de alcanzar aquellas coordenadas y, una vez allá, examinar la roca. Los otros dos planes proporcionan cursos de acción alternativos que el agente tiene que tomar cuando debe satisfacer la meta de alcanzar ciertas coordenadas dadas. Si el agente cree que hay un camino seguro en esa dirección, entonces tiene que desplazarse hacia aquellas coordenadas (`move towards(Coords)`) (8va. y 9na. línea). El plan alternativo no es mostrado aquí, sin embargo, debe proporcionar los medios para que el agente alcance la roca mientras evita caminos inseguros.

```
1  battery_charge(high).
2
3  +green_patch(Rock) : not battery_charge(low)
4     <- ?location(Rock,Coordinates);
5        !traverse(Coordinates);
6        !examine(Rock).
7
8  +!traverse(Coords) : safe_path(Coords)
9     <- move_towards(Coords).
10
11 +!traverse(Coords) : not safe_path(Coords)
12    <- ...
```

En la siguientes secciones se revisa AgentSpeak(L) con mayor profundidad tomando como referencia el trabajo de Bordini et al. [56]. La sección 2.5.1 presenta algunas especi-

ficaciones y la sintax del lenguaje, mientras que la sección 2.5.2 aborda aspectos de la semántica de AgentSpeak(L).

2.5.1. Especificaciones y Aspectos Sintácticos

La gramática que se muestra a continuación proporciona la sintaxis de AgentSpeak como es requerida por el intérprete Jason (sección 3.2). ⟨ATOM⟩ es un identificador que comienza con una letra minúscula o un '.', ⟨VAR⟩ (una variable) es un identificador que empieza con una letra mayúscula, ⟨NUMBER⟩ es cualquier número entero o punto flotante, y ⟨STRING⟩ es cualquier cadena encerrada entre comillas dobles.

agent → beliefs plans

beliefs → (literal ".")*

plans → (plan)+

plan → ["@"atomic_formula]
  triggering_event ":" context "< −" body "."

triggering_event → "+" literal
| "-" literal
| "+" "!" literal
| "-" "!" literal
| "+" "?" literal
| "-" "?" literal

literal → atomic_formula
| "~" atomic_formula
| ⟨VAR⟩

default_literal → literal
| "not" literal
| "not" "(" literal ")"
| term ("<" | "<=" | ">" | ">=" | "==" | "\\ ==" | "=") term
| literal ("==" | "\\ ==" | "=") literal

context → "true"
| default_literal ("&" default_literal)*

body → "true"

```
                  |  body_formula ( ";" body_formula )*
body_formula      → literal
                  |  "!"  literal
                  |  "?"  literal
                  |  "+"  literal
                  |  "-"  literal
atomic_formula    → ⟨ATOM⟩ [ "(" list_of_terms ")" ] [ "[" list_of_terms "]" ]
list_of_terms     → term ( "," term )*
list              → "["
                     [ term ( ( "," term )*
                              | "|" ( list | ⟨VAR⟩ )
                              )
                     ] "]"
term              → atomic_formula
                  |  list
                  |  ⟨VAR⟩
                  |  ⟨NUMBER⟩
                  |  ⟨STRING⟩
```

$p(t_1, \ldots, t_n)$ $n \geq 0$, o $\tilde{}p(t_1, \ldots, t_n)$ es un predicado (literal) , donde '~' denota negación fuerte. La negación default se emplea en el contexto de los planes, y se denota al preceder una literal con la palabra '`not`'. El contexto es una conjunción de literales default. Los términos pueden ser variables, listas (con la misma sintaxis de Prolog), números enteros o punto flotante, cadenas (encerradas entre comillas dobles), y cualquier fórmula atómica. Las variables que ya han tomado valores pueden ser tratadas como literales. En el contexto de los planes se permiten operadores relacionales infijos (ejemplo, `Formato == ".csv"`).

Los predicados pueden tener "anotaciones". Ésta es una lista de términos encerrados dentro de paréntesis cuadrados que se coloca después del predicado. En la base de creencias, las anotaciones se usan, por ejemplo, para registrar las fuentes de la información. Para tal propósito se usa el término `source(`s`)` en las anotaciones, donde s puede ser el nombre de un agente (para denotar el agente que comunica la información), o dos átomos especiales,

`percept` y `self`, que se usan para indicar que una creencia surgió a través de la percepción del ambiente, o desde el agente cuando agrega explícitamente una creencia a su base de creencias mediante la ejecución de un plan, respectivamente. Las creencias iniciales que son parte del código fuente del agente AgentSpeak se asumen como creencias internas (ejemplo, como si tuvieran una anotación `[source(self)]`), a menos que el predicado tenga alguna anotación explícita dada por el usuario.

Los planes también tienen etiquetas. Sin embargo, la etiqueta de un plan puede ser cualquier predicado, con anotaciones, aunque se sugiere que estas etiquetas sean predicados de aridad 0 con anotaciones solo si es necesario, por ejemplo, `unaEtiqueta` o `otraEtiqueta[probabilidadExito(0.7),recompensaEsperada(0.9)]`. Las anotaciones de los predicados que se usan como etiquetas de planes pueden ser usadas para implementar funciones de selección de planes aplicables. Un ejemplo son las funciones que usan utilidades esperadas anotadas en las etiquetas de los planes para elegir entre los planes alternativos.

Los eventos para manejar fallas en los planes están disponibles en Jason (sección 3.2), aunque todavía no están formalizados en la semántica de AgentSpeak. Si una acción falla o no existe un plan aplicable para una submeta `!g` (del plan que se está ejecutando), entonces el plan entero se remueve del tope de la intención (implementada como una pila) y un evento interno para `-!g` (asociado con la misma intención) se genera. Si el programador proporcionó un plan que tiene un evento disparador `-!g` y es aplicable, este plan será colocado en el tope de la intención. Por lo tanto, el programador puede especificar en el cuerpo del plan cómo será manejada esa falla particular. Si ese plan no está disponible, se descarta la intención y se imprime una advertencia en la consola.

Las *acciones internas* pueden ser usadas en el contexto y en el cuerpo de los planes. Cualquier símbolo de acción que empiece con '`.`', o tenga un '`.`' en cualquier parte, denota una acción interna. Éstas son acciones definidas por el usuario que son ejecutadas internamente por el agente. En Jason, las acciones internas se codifican en Java, o en otros lenguajes de programación mediante el uso de JNI (Interfaz Nativa de Java), y pueden ser

organizadas en bibliotecas de acciones (la cadena a la izquierda del '.' se refiere al nombre de la biblioteca; las acciones internas estándar que proporciona Jason no tienen un nombre de biblioteca).

Varias acciones internas están disponibles en Jason, aunque aquí no se mencionan todas (en Bordini et al. [55] puede consultar una lista completa). Una de estas acciones es `.send`, que tiene como propósito implementar la comunicación entre los agentes. Su uso es: `.send(+receiver, +illocutionary_force, +prop_content)`, donde sus parámetros se explican en seguida. `receiver` se refiere a un nombre de agente (a quien va destinado el mensaje). `illocutionary_force` son fuerzas elocutivas: `tell`, `untell`, `achieve`, `unachieve`, `tellHow`, `untellHow`, `askIf`, `askOne`, `askAll` y `askHow`. Los efectos de recibir un mensaje con cada uno de estos tipos de actos elocutivos se explica en Bordini et al. [55]. Por último, `+prop_content` representa una literal (véase la definición de `literal` en la gramática que se presentó anteriormente).

2.5.2. Semántica

Como ya se mencionó, en esta sección se presenta la semántica de AgentSpeak(L). Como primer punto se introduce la semántica informal y posteriormente se aborda la semántica formal.

Semántica Informal

El intérprete de AgentSpeak además de disponer de una base de creencias y una biblioteca de planes, maneja un conjunto de *eventos* y un conjunto de *intenciones*, y para funcionar requiere tres *funciones de selección*. $\mathcal{S}_\mathcal{E}$, la función de selección de eventos escoge un evento del conjunto de eventos; $\mathcal{S}_\mathcal{O}$, la función de selección de opciones elige una "opción" (por ejemplo, un plan aplicable) del conjunto de planes aplicables; y $\mathcal{S}_\mathcal{I}$, la función de selección de intenciones selecciona una intención particular del conjunto de intenciones. Las funciones de selección se suponen específicas de un agente, en el sentido que hacen

elecciones con base en las características de un agente. Por lo tanto, estas funciones no se definirán aquí, y las opciones que elijan se considerarán no determinísticas.

Las *intenciones* son cursos particulares de acción a los que el agente se ha comprometido para manejar ciertos eventos. Cada intención se representa como una pila de planes parcialmente instanciados. Los *eventos* pueden iniciar la ejecución de planes que tienen eventos disparadores relevantes. Estos evento pueden ser *externos*, cuando se originan desde la percepción del ambiente (por ejemplo, la adición y eliminación de creencias que surgen a partir de la percepción son eventos externos); o *internos*, cuando se generan desde la ejecución de un plan del agente (por ejemplo, una submeta en un plan genera un evento de tipo "adición de una meta de logro"). En este último caso, el evento se acompaña con la intención que lo generó, debido a que el plan que se escoja para manejar dicho evento será colocado en el tope de esta intención. Los eventos externos crean nuevas intenciones, que representan focos de atención separados para que el agente actúe en el ambiente.

A continuación se describe más ampliamente el funcionamiento de un intérprete de AgentSpeak, que se muestra en la figura 2.15 (adaptada de Bordini et al. [56]). En ésta los conjuntos (de creencias, eventos, planes e intenciones) se representan como rectángulos. Las funciones de selección (de eventos, opciones e intenciones) se representan como diamantes. Y el procesamiento requerido en la interpretación de programas AgentSpeak se representa mediante círculos.

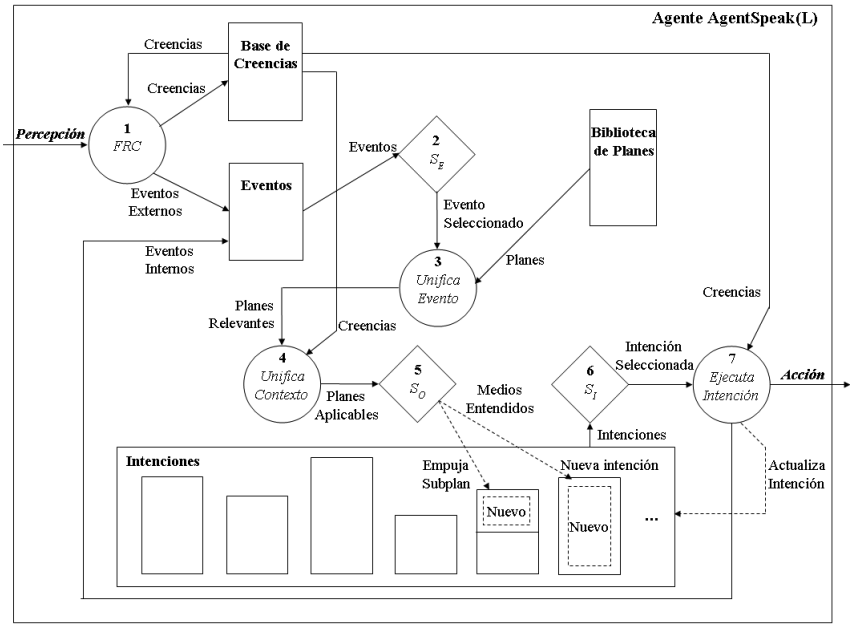

Figura 2.15: Ciclo de Interpretación de un Programa AgentSpeak.

En cada ciclo de interpretación de un programa de agente, el intérprete actualiza una lista de eventos, que se generan desde la percepción del ambiente, o a partir de la ejecución de intenciones (cuando se especifican submetas en el cuerpo de los planes). Se asume que las creencias se actualizan a través de la percepción y cuando existen cambios en las creencias del agente, lo que implica la inserción de un evento en el conjunto de eventos. Esta función de revisión de creencias (FRC) no es parte del intérprete AgentSpeak, sin embargo es un componente necesario de la arquitectura del agente.

Después que $\mathcal{S}_\mathcal{E}$ ha seleccionado un evento, el intérprete tiene que unificar este evento con los eventos disparadores de las cabezas de los planes. Esto genera un conjunto de *planes relevantes*. AgentSpeak(L) determina el conjunto de *planes aplicables* (aquellos que pueden ser usados en ese momento para manejar el evento que se eligió) al elegir los planes relevantes cuyo contexto unifica con las creencias del agente. Después $\mathcal{S}_\mathcal{O}$ escoge un plan aplicable (que llega a ser el *medio entendido* para tratar el evento seleccionado) desde este conjunto de planes aplicables, y lo coloca en el tope de una intención existente (si el evento

es interno), o en una nueva intención que crea en el conjunto de intenciones (si el evento es externo, por ejemplo, generado desde la percepción del ambiente).

Lo que resta es seleccionar una intención para que se ejecute en este ciclo. La función $\mathcal{S}_{\mathcal{I}}$ escoge una de las intenciones (una de las pilas de planes parcialmente instanciados dentro del conjunto de intenciones) del agente. Sobre el tope de esta intención se encuentra un plan, por lo tanto, la fórmula que se halla en el inicio del cuerpo de este plan se ejecuta. Esto implica que el agente aplique una acción básica sobre su ambiente, genere un evento interno (en caso que la fórmula seleccionada sea una meta de logro), o revise una meta de prueba (con el conjunto de creencias).

Si la intención consiste en ejecutar una acción básica o una meta de prueba, entonces el conjunto de intenciones necesita actualizarse. En el caso de una meta de prueba, se buscará en la base de creencias un átomo de creencia que unifique con el predicado de esta meta de prueba. Si la búsqueda tiene éxito, se llevará a cabo una instanciación de variables en el plan parcialmente instanciado que contenía la meta de prueba (y ésta se remueve de la intención que fue tomada). Cuando se selecciona una acción básica, el conjunto de intenciones se actualiza al remover esta acción desde la intención (el intérprete informa al componente de la arquitectura responsable de los efectores del agente qué acción se requiere). Cuando todas las fórmulas en el cuerpo del plan han sido removidas (el plan ha sido ejecutado), éste se elimina de la intención, y también la meta de logro que lo generó (si ese fue el caso). Esto termina un ciclo de ejecución, y otra vez se repite todo, inicialmente revisando el estado del ambiente después que los agentes han actuado sobre él, y luego generando los eventos relevantes, y así sucesivamente.

Semántica Formal

Una especificación de agente AgentSpeak *ag* está dada por la siguiente gramática:

$$
\begin{array}{lll}
ag & ::= & bs \quad ps \\
bs & ::= & b_1 \ldots b_n \qquad (n \geq 0) \\
ps & ::= & p_1 \ldots p_n \qquad (n \geq 1)
\end{array}
$$

$$
\begin{array}{lll}
p & ::= & te : ct \leftarrow h \\
te & ::= & +at \mid -at \mid +g \mid -g \\
ct & ::= & ct_1 \mid \mathtt{T} \\
ct_1 & ::= & at \mid \neg at \mid ct_1 \wedge ct_1 \\
h & ::= & h_1 \,;\, \mathtt{T} \mid \mathtt{T} \\
h_1 & ::= & a \mid g \mid u \mid h_1 ; h_1 \\
at & ::= & \mathtt{P}(t_1, \ldots, t_n) \qquad (n \geq 0) \\
 & \mid & \mathtt{P}(t_1, \ldots, t_n)[s_1, \ldots, s_m] \qquad (n \geq 0,\, m > 0) \\
s & ::= & \mathtt{percept} \mid \mathtt{self} \mid id \\
a & ::= & \mathtt{A}(t_1, \ldots, t_n) \qquad (n \geq 0) \\
g & ::= & !at \mid ?at \\
u & ::= & +at \mid -at
\end{array}
$$

La semántica de AgentSpeak se define al emplear la semántica operacional, un método que se usa ampliamente para proporcionar semántica a los lenguajes de programación y estudiar sus propiedades. La semántica operacional está dada por un conjunto de reglas que definen una relación de transición entre configuraciones $\langle ag, C, M, T, s \rangle$ donde:

- Un programa de agente ag está formado como se definió anteriormente, de un conjunto de creencias y un conjunto de planes.
- Una circunstancia C de un agente es una tupla $\langle I, E, A \rangle$ donde:
  - I es un conjunto de *intenciones* $\{i, i', \ldots\}$. Cada intención i es una pila de planes parcialmente instanciados.
  - E es un conjunto de *eventos* $\{(te, i), (te', i'), \ldots\}$. Cada evento está conformado por un par (te, i), donde te es un evento disparador e i es una intención (una pila de planes si se trata de un evento interno o `T` si es un evento externo).
  - A es un conjunto de *acciones* que se ejecutarán en el ambiente.
- M es una tupla $\langle In, Out, SI \rangle$ cuyos componentes registran los siguientes aspectos de comunicación de los agentes:

- *In* es la bandeja de entrada donde se almacenan los mensajes dirigidos a este agente. Los elementos de este conjunto tiene la forma $\langle mid, id, ilf, cnt \rangle$ donde *mid* es un identificador del mensaje, *id* identifica al remitente del mensaje, *ilf* es la fuerza elocutiva del mensaje, y *cnt* su contenido (que puede ser un predicado AgentSpeak, un conjunto de predicados AgentSpeak, o un conjunto de planes AgentSpeak, lo que dependerá de la fuerza elocutiva del mensaje).
- *Out* es donde el agente coloca los mensajes que desea enviar a los otros agentes. Los mensajes de este conjunto tienen el mismo formato que se definió anteriormente para los mensajes de entrada, excepto que ahora *id* especifica el agente destinatario del mensaje.
- *SI* se usa para no perder de vista las intenciones que fueron suspendidas debido al procesamiento de mensajes de comunicación. Las intenciones que se asocian a fuerzas elocutivas que requieren una respuesta del interlocutor se suspenden hasta que se recibe la contestación.

- T es la tupla $\langle R, Ap, \imath, \varepsilon, \rho \rangle$ que almacena información temporal que se emplea en etapas subsecuentes dentro de un ciclo de razonamiento. Los componentes de esta tupla son:
  - R el conjunto de *planes relevantes* (para el evento que se está manejando).
  - Ap el conjunto de *planes aplicables* (planes relevantes cuyo contexto es verdadero).
  - $\imath$, ε y ρ registran una intención, un evento y un plan aplicable particular, respectivamente, a través de la ejecución de un agente.
- El paso actual dentro del ciclo de razonamiento de un agente se denota simbólicamente por $s \in \{$`ProcMsg, SelEv, RelPl, ApplPl, SelAppl, AddIM, SelInt, ExecInt, ClrInt`$\}$, donde: `ProcMsg` se refiere al procesamiento de un mensaje de la bandeja de entrada, `SelEv` selecciona un evento del conjunto de eventos, `RelPl` recu-

pera los planes relevantes, `ApplPl` revisa entre los planes relevantes cuales se pueden ejecutar (planes aplicables), `SelAppl` selecciona un plan aplicable (medios entendidos), `AddIM` agrega el plan aplicable al conjunto de intenciones, `SelInt` selecciona una intención, `ExecInt` ejecuta la intención que se seleccionó, y `ClrInt` remueve una intención o un plan aplicable que pudieron haber finalizado en el paso anterior.

Las funciones de selección ($\mathcal{S}_{\mathcal{E}}$, $\mathcal{S}_{\mathcal{AP}}$, $\mathcal{S}_{\mathcal{I}}$) que usa un agente también son parte de su configuración. Sin embargo, no se incluyen en ésta debido a que se consideran fijas, es decir, definidas por el diseñador del agente.

Para mantener las reglas semánticas claras, se adoptan las siguientes notaciones:

- Si C es una circunstancia de un agente AgentSpeak, C_E se refiere al componente E de C (es decir, al conjunto de eventos). Lo mismo aplica para el resto de los elementos de una configuración.
- $T_i = \_$ indica que no se está ejecutanto una intención del agente. De la misma manera se emplea para T_ρ y T_ε.
- *i[p]* denota que una intención i tiene en su tope el plan p.

Si p es un plan de la forma $te : ct \leftarrow h$, entonces `TrEv`$(p) = te$ y `Ctxt`$(p) = ct$ recuperan el evento disparador y el contexto del plan, respectivamente. A continuación se definen algunas funciones auxiliares que serán útiles en la presentación de las reglas semánticas.

Def 1 (Unificador) *Sean α y β términos. Una sustitución θ tal que α y β sean idénticos ($\alpha\theta = \beta\theta$) recibe el nombre de unificador de α y β.*

Def 2 (Generalidad entre sustituciones) *Una sustitución θ se dice más general que una sustitución σ si, y solo si, existe una substitución γ tal que $\sigma = \theta\gamma$.*

Def 3 (MGU) *Un unificador θ se dice el unificador más general (MGU) de dos términos si, y solo si, θ es más general que cualquier otro unificador de dos términos.*

Def 4 (Consecuencia lógica) *Una fórmula atómica at con anotaciones* $s_{11}, \ldots, s_{1n}$ *es una consecuencia lógica de un conjunto de fórmulas atómicas aterrizadas bs,* $bs \models at_1[s_{11}, \ldots, s_{1n}]$ *si, y solo si, existe* $at_2[s_{21}, \ldots, s_{2m}] \in bs$ *tal que (i)* $at_1\theta = at_2$*, para algún unificador más general* θ*, y (ii)* $\{s_{11}, \ldots, s_{1n}\} \subseteq \{s_{21}, \ldots, s_{2m}\}$.

Def 5 (Planes relevantes) *Dados los planes ps de un agente y un evento disparador te, el conjunto* ***RelPlans****(ps, te) de planes relevantes es:*

$$\textit{RelPlans}(ps, te) = \{(p,\, \theta) \,|\, p \,\in\, ps \,\wedge\, \theta = MGU(te, \textit{TrEv}(p))\}$$

Def 6 (Planes aplicables) *Dado un conjunto de planes relevantes R y las creencias bs de un agente, el conjunto de planes aplicables* ***AppPlans****(bs, R) se define como:*

$$\textit{AppPlans}(bs, R) = \{(p,\, \theta' \,o\, \theta) \,|\, (p, \theta) \,\in\, R \,\wedge\, \theta' = MGU(bs, \textit{Ctxt}(p)\theta)\}$$

Def 7 (Prueba) *Dado un conjunto de fórmulas bs y una fórmula at, el conjunto de substituciones* ***Test****(bs, at) que se producen al probar at contra bs se define como:*

$$\textit{Test}(bs, at) = \{\theta \,|\, bs \models at\theta\}$$

Ahora se presentan las reglas que definen la semántica operacional del ciclo de decisión de AgentSpeak. En el caso general, la configuración inicial de un agente es $\langle ag, C, M, T, \texttt{ProcMsg}\rangle$, donde ag es el programa de agente, y los componentes C, M y T están vacíos. Un ciclo de razonamiento inicia con el procesamiento de los mensajes que ha recibido el agente, debido a esto s ha tomado el valor de `ProcMsg`. Posteriormente, se selecciona un evento (`SelEv`), que se explica con la semántica que se da en seguida.

- **Selección de un Evento:** La regla $\mathbf{SelEv_1}$ asume que existe una función de selección $\mathcal{S}_\mathcal{E}$ que selecciona eventos de un conjunto de eventos E. El evento seleccionado se elimina de E y se asigna al elemento ε de la circunstancia. La regla $\mathbf{SelEv_2}$ pasa a la parte del ciclo de ejecución de una intención, en caso que E este vacío.

$$\textbf{SelEv}_\textbf{1} \quad \frac{\mathcal{S}_\mathcal{E}(C_E) = \langle te, i\rangle}{\langle ag, C, M, T, \texttt{SelEv}\rangle \longrightarrow \langle ag, C', M, T', \texttt{RelPl}\rangle}$$

$$donde: \quad \begin{aligned} C'_E &= C_E \backslash \langle te, i\rangle \\ T'_\varepsilon &= \langle te, i\rangle \end{aligned}$$

$$\textbf{SelEv}_\textbf{2} \quad \frac{(C_E) = \{\}}{\langle ag, C, M, T, \texttt{SelEv}\rangle \longrightarrow \langle ag, C, M, T, \texttt{SelInt}\rangle}$$

- **Planes Relevantes:** La regla $\textbf{Rel}_\textbf{1}$ asigna al componente R el conjunto de planes relevantes. La regla $\textbf{Rel}_\textbf{2}$ se aplica cuando no hay planes relevantes para un evento. En este último caso, se descarta el evento y la intención asociada a él.

$$\textbf{Rel}_\textbf{1} \quad \frac{T_\varepsilon = \langle te, i\rangle \qquad \texttt{RelPlans}(ag_{ps}, te) \neq \{\}}{\langle ag, C, M, T, \texttt{RelPl}\rangle \longrightarrow \langle ag, C, M, T', \texttt{ApplPl}\rangle}$$

$$donde: \quad T'_R = \texttt{RelPlans}(ag_{ps}, te)$$

$$\textbf{Rel}_\textbf{2} \quad \frac{T_\varepsilon = \langle te, i\rangle \qquad \texttt{RelPlans}(ag_{ps}, te) = \{\}}{\langle ag, C, M, T, \texttt{RelPl}\rangle \longrightarrow \langle ag, C, M, T, \texttt{SelEv}\rangle}$$

Un enfoque alternativo para cuando no existen planes relevantes se describe en Ancona et al. [15].

- **Planes Aplicables:** La regla $\textbf{Appl}_\textbf{1}$ inicializa el componente T_{Ap} con el conjunto de planes aplicables, mientras que $\textbf{Appl}_\textbf{2}$ descarta el evento si no existen planes aplicables para dicho evento.

$$\textbf{Appl}_1 \quad \frac{\texttt{AppPlans}(ag_{bs}, T_R) \neq \{\}}{\langle ag, C, M, T, \texttt{ApplPl}\rangle \longrightarrow \langle ag, C, M, T', \texttt{SelAppl}\rangle}$$

$$donde: \quad T'_{Ap} \;=\; \texttt{AppPlans}(ag_{bs}, T_R)$$

$$\textbf{Appl}_2 \quad \frac{\texttt{AppPlans}(ag_{bs}, T_R) = \{\}}{\langle ag, C, M, T, \texttt{ApplPl}\rangle \longrightarrow \langle ag, C, M, T, \texttt{SelInt}\rangle}$$

- **Selección de un Plan Aplicable:** Esta regla asume que existe una función $\mathcal{S}_{\mathcal{AP}}$ que selecciona un plan del conjunto de planes aplicables T_{Ap}. El plan que se selecciona se asigna al componente T_ρ de la configuración.

$$\textbf{SelAppl} \quad \frac{\mathcal{S}_{\mathcal{AP}}(T_{Ap}) = (p, \theta)}{\langle ag, C, M, T, \texttt{SelAppl}\rangle \longrightarrow \langle ag, C, M, T', \texttt{AddIM}\rangle}$$

$$donde: \quad T'_\rho \;=\; (p, \theta)$$

- **Actualizando el Conjunto de Intenciones:** La regla **ExtEv** se aplica cuando el evento en ε es externo (es decir, generado mediante la percepción del agente). En este caso, se crea una intención y se le asigna el plan p que se encuentra en el componente ρ. Por otro lado, si el evento es interno (creado por la ejecución de un plan), la regla **IntEv** coloca el plan p en el tope de la intención asociada con este evento.

$$\textbf{ExtEv} \quad \frac{T_\varepsilon = \langle te, \texttt{T}\rangle \qquad T_\rho = (p, \theta)}{\langle ag, C, M, T, \texttt{AddIM}\rangle \longrightarrow \langle ag, C', M, T, \texttt{SelInt}\rangle}$$

$$donde: \quad C'_I \;=\; C_I \cup \{\, [p\theta] \,\}$$

$$\textbf{IntEv} \quad \frac{T_\varepsilon = \langle te, i \rangle \qquad T_\rho = (p, \theta)}{\langle ag, C, M, T, \texttt{AddIM} \rangle \longrightarrow \langle ag, C', M, T, \texttt{SelInt} \rangle}$$

$$donde: \quad C'_I \;=\; C_I \cup \{\, (i[p])\theta \,\}$$

La regla **IntEv** inserta la intención i (con p en el tope) que generó el evento interno al final de C_I. Es decir, suspende la intención i.

- **Selección de una Intención:** La regla $\textbf{IntSel}_\textbf{1}$ selecciona una intención (una pila de planes) mediante la función $\mathcal{S}_\mathcal{I}$, mientras que $\textbf{IntSel}_\textbf{2}$ reinicia el ciclo de razonamiento del agente debido a que el conjunto de intenciones está vacío.

$$\textbf{IntSel}_\textbf{1} \quad \frac{C_I \neq \{\} \qquad \mathcal{S}_\mathcal{I}(C_I) = i}{\langle ag, C, M, T, \texttt{SelInt} \rangle \longrightarrow \langle ag, C, M, T', \texttt{ExecInt} \rangle}$$

$$donde: \quad T'_i \;=\; i$$

$$\textbf{IntSel}_\textbf{2} \quad \frac{C_I = \{\}}{\langle ag, C, M, T, \texttt{SelInt} \rangle \longrightarrow \langle ag, C, M, T, \texttt{ProcMsg} \rangle}$$

- **Ejecutando el Cuerpo de un Plan:** Las reglas en este grupo ejecutan el cuerpo de un plan. El plan que se ejecuta es el que se encuentra en el tope de la intención que se seleccionó previamente. Todas estas reglas descartan la intención i; después otra intención puede ser seleccionada.

  Acciones: La acción a que está sobre el cuerpo del plan se elimina y se agrega al conjunto de acciones A. La intención asociada a este plan se actualiza para que refleje la eliminación de la acción.

$$\textbf{Action} \quad \frac{T_i = i[head \leftarrow a; h]}{\langle ag, C, M, T, \texttt{ExecInt} \rangle \longrightarrow \langle ag, C', M, T, s \rangle}$$

$$
\begin{aligned}
\textit{donde}: \quad C'_A &= C_A \cup \{a\} \\
C'_I &= (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \\
s &= \begin{cases} \texttt{ClrInt} & \text{if } h = \texttt{T} \\ \texttt{ProcMsg} & \text{en cualquier otro caso} \end{cases}
\end{aligned}
$$

Metas de Logro: Este regla añade un evento interno al conjunto de eventos E. Este evento después puede ser eventualmente seleccionado.

$$
\textbf{Achieve} \quad \frac{T_i = i[head \leftarrow !at; h]}{\langle ag, C, M, T, \texttt{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, \texttt{ProcMsg}\rangle}
$$

$$
\begin{aligned}
\textit{donde}: \quad C'_E &= C_E \cup \{\langle +!at, i[head \leftarrow h]\rangle\} \\
C'_I &= C_I \setminus \{T_i\}
\end{aligned}
$$

Metas de Prueba: Estas reglas se emplean cuando una meta de prueba *?at* debe ser ejecutada. La regla $\textbf{Test}_\textbf{1}$ se usa cuando existe un conjunto de sustituciones que pueden hacer a *at* una consecuencia lógica de las creencias del agente. Estas sustituciones se aplican en el plan entero, al que pertenece *at*. Por otro lado, cuando la meta de prueba se usa como un evento disparador de un plan, se emplea la regla $\textbf{Test}_\textbf{2}$.

$$
\textbf{Test}_\textbf{1} \quad \frac{T_i = i[head \leftarrow ?at; h] \qquad \texttt{Test}(ag_{bs}, at) \neq \{\}}{\langle ag, C, M, T, \texttt{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, s\rangle}
$$

$$
\begin{aligned}
\textit{donde}: \quad C'_I &= (C_I \setminus \{T_i\}) \cup \{(i[head \leftarrow h])\theta\} \\
&\quad \theta \in \texttt{Test}(ag_{bs}, at) \\
s &= \begin{cases} \texttt{ClrInt} & \text{if } h = \texttt{T} \\ \texttt{ProcMsg} & \text{en cualquier otro caso} \end{cases}
\end{aligned}
$$

$$
\textbf{Test}_\textbf{2} \quad \frac{T_i = i[head \leftarrow ?at; h] \qquad \texttt{Test}(ag_{bs}, at) = \{\}}{\langle ag, C, M, T, \texttt{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, s\rangle}
$$

$$
\begin{aligned}
\textit{donde}: \quad C'_E &= C_E \cup \{\langle +?at, i[head \leftarrow h]\rangle\} \\
C'_I &= C_I \setminus \{T_i\} \\
s &= \begin{cases} \texttt{ClrInt} & \text{if } h = \texttt{T} \\ \texttt{ProcMsg} & \text{en cualquier otro caso} \end{cases}
\end{aligned}
$$

Actualizando Creencias: La regla **AddBel** agrega un evento al conjunto de eventos E. La fórmula $+b$ se remueve del cuerpo del plan y el conjunto de intenciones se actualiza adecuadamente. La regla **DelBel** trabaja de forma similar. Con ambas reglas, el conjunto de creencias del agente se modifica, ya sea que el predicado b se incluya (regla **AddBel**) o se remueva (regla **DelBel**) de él.

$$
\textbf{AddBel} \quad \frac{T_i = i[head \leftarrow +b; h]}{\langle ag, C, M, T, \texttt{ExecInt}\rangle \longrightarrow \langle ag', C', M, T, s\rangle}
$$

$$
\begin{aligned}
\textit{donde}: \quad ag'_{bs} &= ag_{bs} + b[\texttt{self}] \\
C'_E &= C_E \cup \{\langle +b[\texttt{self}], \texttt{T}\rangle\} \\
C'_I &= (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \\
s &= \begin{cases} \texttt{ClrInt} & \text{if } h = \texttt{T} \\ \texttt{ProcMsg} & \text{en cualquier otro caso} \end{cases}
\end{aligned}
$$

$$
\textbf{DelBel} \quad \frac{T_i = i[head \leftarrow -b; h]}{\langle ag, C, M, T, \texttt{ExecInt}\rangle \longrightarrow \langle ag', C', M, T, s\rangle}
$$

$$
\begin{aligned}
\textit{donde}: \quad ag'_{bs} &= ag_{bs} - b[\texttt{self}] \\
C'_E &= C_E \cup \{\langle -b[\texttt{self}], \texttt{T}\rangle\} \\
C'_I &= (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \\
s &= \begin{cases} \texttt{ClrInt} & \text{if } h = \texttt{T} \\ \texttt{ProcMsg} & \text{en cualquier otro caso} \end{cases}
\end{aligned}
$$

- **Removiendo las Intenciones que han finalizado:** Las siguientes reglas eliminan las intenciones que han finalizado desde el conjunto de intenciones. La regla $\mathbf{ClrInt_1}$ remueve una intención que ha quedado vacía (sin metas o acciones). Por otra parte, la regla $\mathbf{ClrInt_2}$ elimina de una intención lo que ha sido dejado por un plan que había sido colocado en el tope de esta intención.

$$\mathbf{ClrInt_1} \quad \frac{j = [head \leftarrow \mathtt{T}],\ para\, alguna\, j \in C_I}{\langle ag, C, M, T, \mathtt{ClrInt}\rangle \longrightarrow \langle ag, C', M, T, \mathtt{ProcMsg}\rangle}$$

$$donde: \quad C'_I \ = \ C_I \setminus \{j\}$$

$$\mathbf{ClrInt_2} \quad \frac{j = i[head \leftarrow \mathtt{T}],\ para\, alguna\, j \in C_I}{\langle ag, C, M, T, \mathtt{ClrInt}\rangle \longrightarrow \langle ag, C', M, T, \mathtt{ProcMsg}\rangle}$$

$$donde: \quad C'_I \ = \ (C_I \setminus \{j\}) \cup \{i\}$$

2.6. Sistemas Multi-Agentes (MAS)

La investigación en MAS se ocupa del estudio, comportamiento y construcción de una colección de agentes autónomos que interactúan [12] entre ellos y con su ambiente. El estudio de estos sistemas también toma en cuenta aspectos sociales, que no se consideran en los agentes individuales. Un MAS se puede definir como una red de solucionadores de problemas que interactúan para resolver problemas que superan las capacidades individuales o conocimiento de cada uno de ellos. Estos solucionadores de problemas, llamados agentes, son autónomos y pueden ser heterogéneos [35]. La heterogeneidad se refiere a que estos agentes son diferentes en cuanto a: los dominios de discurso, los esquemas de representación del conocimiento, los paradigmas de resolución de problemas, las suposiciones acerca de su mundo y el de los otros agentes, etc. [49].

Las características de un MAS son: (1) cada agente tiene un punto de vista limitado, debido a que dispone de información o capacidades incompletas para solucionar el problema; (2) no existe un control global del sistema; (3) los datos están descentralizados; y (4) el procesamiento es asíncrono. Las habilidades de los MAS motivan el interés creciente en la investigación de éstos. Entre estas habilidades se encuentran [35]:

- Resolver problemas demasiado grandes que un agente centralizado no podría solucionar, debido a limitaciones o centralización de sus recursos, que pudiera generar un cuello de botella o fallar en tiempos críticos.

- Permitir la interconexión e interoperación de múltiples sistemas heterogéneos existentes, al construir un agente *"wrapper"* alrededor de ellos. Un agente wrapper es un tipo especial de agente, que actúa como intermediario entre agentes y otras entidades existentes que no necesariamente son agentes, con el fin de que la comunicación se lleve a cabo entre ellos [26].

- Proporcionar soluciones a problemas que naturalmente pueden ser considerados como una sociedad de agentes interactuantes autónomos. Un ejemplo, son las subastas electrónicas sobre el internet, donde se compran y se venden mercancías.

- Proveer soluciones que usan eficientemente fuentes de información que están espacialmente distribuidas. Por ejemplo, recoger información desde internet.

- Aportar soluciones en situaciones donde la pericia está distribuida. Por ejemplo, en ingeniería concurrente, cuidado de la salud y manufactura. La ingeniería concurrente es un enfoque para la manufactura que permite realizar simultáneamente el diseño y desarrollo de productos, procesos, y actividades de apoyo [57].

Uno de los múltiples usos de los MAS ha sido llevar a cabo la tarea de búsqueda de patrones a través del proceso KDD y la minería de datos (presentados en la sección 2.1). Algunos de los MAS más representativos en este dominio son:

- BODHI [27]. Es un sistema de descubrimiento de conocimiento distribuido basado en agentes que ofrece un sistema de intercambio de mensajes y que es capaz de manejar agentes móviles. Proporciona modelos de datos correctos con baja carga en la comunicación de la red. El proceso de minería es distribuido a los agentes móviles que se transladan sobre la red, llevando su estado, datos y conocimiento. Un agente que actúa como coordinador es responsable de iniciar y coordinar las tareas de minería de datos, además de comunicar y controlar a los agentes.

- PADMA [28]. Es una arquitectura basada en agentes para Minería de Datos Paralela y Distribuida (PDDM), que opera en sitios de datos homogéneos. Primero, los agentes que se encuentran situados en lugares diferentes aprenden modelos de *clusters* de datos parciales. Después, todos estos modelos se concentran en un sitio central que ejecuta un algoritmo de *clustering* de segundo nivel, que genera un modelo de cluster global. Los agentes individuales ejecutan *clustering* jerárquico en clasificación de documentos de texto, pero el sistema puede ser aplicado en diferentes dominios.

- JAM [8]. Es un MAS escrito en Java diseñado para PDDM. Diferentes algoritmos de aprendizaje tal como Ripper, CART, ID3, C4.5, Bayes, y WEPBLS pueden ser ejecutados sobre bases de datos (relacionales) heterogéneas por algún agente de aprendizaje. JAM también proporciona agentes de meta-aprendizaje que combinan los modelos aprendidos en los diferentes sitios dentro de un meta-clasificador que mejora la precisión predictiva general.

- Papyrus [62]. Es un sistema escrito en Java para PDDM que opera en clusters, meta-clusters y super-clusters (sistemas de cómputo que son creados al ensamblar un gran número de estaciones de trabajo o servidores dentro de una entidad de cómputo única) que contienen datos heterogéneos. Soporta diferentes tareas predictivas incluyendo C4.5. Con el fin de ejecutar todo el cómputo localmente y reducir la carga de la red, los agentes móviles transladan datos, resultados intermedios y modelos entre clusters; o desde sitios locales a una raíz central para producir el resultado final. La

coordinación de la tarea de aprendizaje general es hecha por un sitio raíz central o distribuida entre los clusters.

Una vez abordados los fundamentos de esta tesis se introducen en el siguiente capítulo la metodología de diseño Prometheus y las herramientas PDT, Jason y WEKA, que nos auxiliaron en el diseño y la implementación del MAS.

Capítulo 3

Métodos

En este capítulo se presentan la metodología de diseño y las herramientas que se utilizaron para llevar a cabo el diseño y el desarrollo del MAS. En la sección 3.1 se exponen la *metodología Prometheus* que guió el diseño del MAS y la *herramienta PDT* que da soporte a esta metodología. Para la implementación del MAS se empleó el intérprete de AgentSpeak(L) (presentado en la sección 2.5), *Jason*. Este intérprete se aborda en la sección 3.2. Mediante las acciones internas de los agentes del MAS es posible heredar código, lo cual es una gran ventaja debido a que permite ahorrar tiempo en el desarrollo del sistema. De esta forma, las acciones de los agentes encapsulan algunas clases en Java (que implementan principalmente algoritmos de preprocesamiento, minería de datos y evaluación de patrones), que proporcionó la herramienta de KDD, *WEKA*. Esta herramienta se introduce en la sección 3.3.

3.1. Prometheus y su herramienta de diseño (PDT)

Prometheus [38] es una metodología para desarrollar agentes inteligentes. Esta metodología abarca todas las fases de desarrollo de un sistema - especificación, diseño, implementación y prueba/depuración. Prometheus se aplica de manera iterativa, de forma que cuando se realizan cambios en algunas partes del diseño frecuentemente otras partes se afectan y, por tanto, se necesitan actualizar.

La herramienta PDT [37] da soporte a la metodología Prometheus. Esta herramienta proporciona: una interfaz gráfica que permite desarrollar las definiciones (diagramas) que se obtienen mediante la metodología Prometheus y asiste en el mantenimiento de un

diseño consistente debido a que proporciona verificación entre los diagramas; propagación automática de elementos de diseño cuando es posible y apropiado; y asistencia en la búsqueda de nombres. La versión actual de PDT se basa en el lenguaje de programación Java y se puede descargar libremente desde http://www.cs.rmit.edu.au/agents/pdt.

PDT soporta solamente las etapas de especificación y diseño del sistema. Sin embargo, se está trabajando en la implementación de una aplicación para Eclipse (plataforma de desarrollo abierta, que puede ser descargada desde http://www.eclipse.org/), que usará PDT para soportar el diseño de un sistema de agentes y a partir de este diseño será capaz de generar automáticamente código en un lenguaje de programación de agentes como JACK [51].

En la sección 3.1.1 se proporciona un repaso breve de la metodología Prometheus. La herramienta PDT se describe en la sección 3.1.2. Por último, la sección 3.1.3 presenta un escenario que muestra uno de los diagramas que se genera con la herramienta PDT.

3.1.1. Un repaso de la Metodología Prometheus

La metodología Prometheus consiste de tres fases principales: (i) Especificación del Sistema, (ii) Diseño de la Arquitectura, y (iii) Diseño Detallado. Cada una de éstas da como resultado definiciones de diseño específicos. A continuación se describe cada una de estas etapas, haciendo énfasis en las definiciones que producen.

Especificación del Sistema

La especificación del sistema consiste de los siguientes pasos, que se iteran hasta que la especificación se considera completa:

- Identificación de *actores* y sus interacciones con el sistema, en forma de *percepciones* y *acciones*;
- Desarrollo de *escenarios* que ilustran la operación del sistema;

- Identificación de las *metas* y submetas del sistema;
- Identificación de *datos externos*;
- Agrupamiento de metas y otros elementos para formar los *roles* básicos del sistema.

Los actores son personas o roles que interactuarán con el sistema. Estos actores también pueden ser otros sistemas de software. Para cada actor se identifican escenarios de interacción. Las entradas del sistema (que se definen como *percepciones*) son proporcionadas por los actores, mientras que las salidas del sistema (que se identifican como *acciones*) se dirigen a ellos.

Cada escenario se desarrolla con un número de pasos detallados, donde cada paso es una *meta*, *escenario*, *acción* o *percepción*. Las metas iniciales se identifican al examinar la descripción del sistema. El resto de las metas después se obtienen a través de un proceso de abstracción y refinamiento. Para cada meta, se plantean las preguntas *¿cómo?* y *¿por qué?*, por lo tanto se identifican nuevas metas, con las que se forma una jerarquía de metas.

El refinamiento de las metas puede ser de dos tipos: *refinamiento-AND* y *refinamiento-OR*. Si una meta se refina mediante la primera opción, las submetas (o respuestas a la pregunta *¿cómo?*) son pasos para lograr la meta general, y cada paso se debe realizar. En cambio, si la meta se refina mediante la segunda opción, entonces las submetas son caminos alternativos para alcanzar la meta, y por lo tanto es suficiente hacer cualquiera de ellas.

Después de que las metas y los escenarios se han desarrollado suficientemente, las metas similares se agrupan dentro de roles. Las acciones y percepciones también se asignan a los roles. Por último, a cada paso de los escenarios se le asigna el rol al que pertenece y se identifican los requerimientos de datos para cada escenario.

Diseño de la Arquitectura

Esta fase usa las definiciones que se produjeron en la Especificación del Sistema para determinar qué *tipos de agentes* serán incluidos en el sistema y la interacción que se llevará a cabo entre ellos. Los pasos en esta fase son:

- Determinar los *tipos de agentes*;
- Desarrollar los *protocolos de interacción*;
- Desarrollar el *diagrama general del sistema*.

Varios mecanismos se pueden usar para determinar los tipos de agentes al analizar agrupamientos de roles. Estos mecanismos incluyen *diagramas de acoplamiento de datos* y *diagramas de conocimiento de agentes*, los cuales ayudan al usuario en la identificación de los tipos de agentes que no se acoplan fuertemente.

Los escenarios obtenidos en la fase de Especificación del Sistema se pueden usar como una guía para desarrollar los *diagramas de interacción* que a su vez se emplean para derivar los *protocolos de interacción* que definen completamente las interacciones permitibles entre los agentes. La estructura del sistema se captura a través de un diagrama general del sistema, que muestra los tipos de agentes, los protocolos que especifican las interacciones entre los tipos de agentes, la interfaz con el ambiente en términos de percepciones, acciones, y datos externos. Los almacenamientos de datos que son compartidos entre los agentes también están presentes en este diagrama.

Diseño Detallado

Esta fase usa las definiciones que se elaboraron en la etapa de Diseño de la Arquitectura para definir los detalles internos de cada agente y especificar cómo los agentes realizarán sus tareas. Cada agente se refina en términos de sus *capacidades*, *eventos internos*, *planes*, y *estructuras de datos*. Cada capacidad tiene un diagrama general de capacidad que captura la estructura de los planes dentro de esta capacidad y los eventos que están asociados con estos planes. El comportamiento dinámico se describe mediante los *diagramas de procesos* que se basan en los protocolos de interacción que se identificaron en la fase anterior.

3.1.2. Herramienta PDT

La figura 3.1 muestra la herramienta PDT, que da soporte a la metodología Prometheus revisada previamente. Esta herramienta se empleó para elaborar el diseño del MAS que nos ocupa en este trabajo. Como puede ser apreciado, la vista *Diagrams* sobre la columna izquierda en PDT proporciona una variedad de diagramas que están organizados en las tres fases principales de la metodología: especificación del sistema, diseño de la arquitectura y diseño detallado.

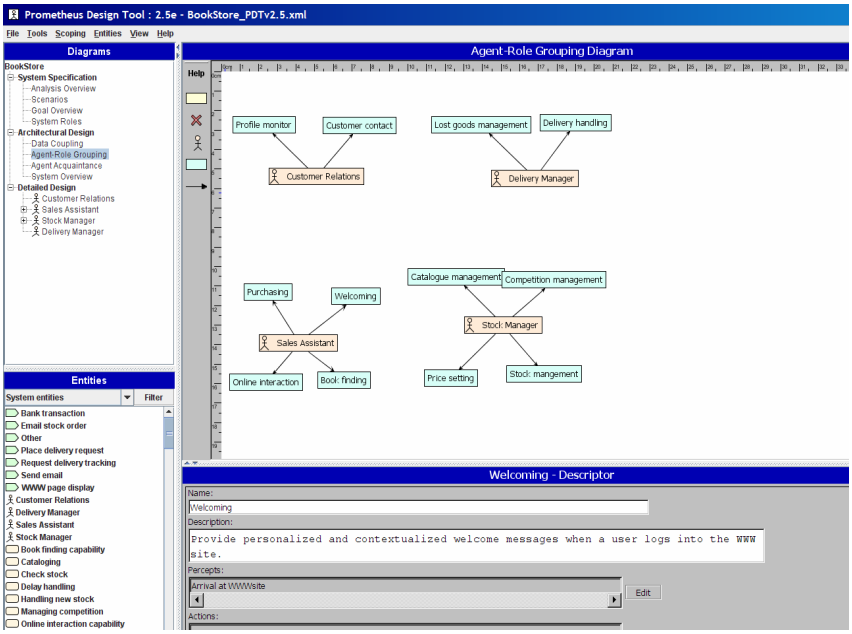

Figura 3.1: Herramienta PDT

Debajo de los diagramas de diseño sobre la columna izquierda en PDT se encuentra una lista desplazable de todas las entidades individuales del sistema organizadas por tipo de entidad. Esta lista puede ser filtrada por tipo de entidad, por ejemplo, para visualizar solamente las metas del sistema. Cuando una entidad se selecciona desde esta lista, o desde alguno de los diagramas, su descriptor es mostrado en la parte inferior de la columna derecha de PDT. El descriptor especifica los atributos de una cierta entidad. Por ejemplo,

si se tratará de un *rol* entonces el descriptor tendría los siguientes atributos: nombre del rol, una descripción breve de éste, las percepciones a las que responde, las acciones que produce, la información que usa, la información que produce y las metas con las que se asocia.

Las siguientes secciones describen algunas de las características de PDT y cómo éstas dan soporte a la metodología Prometheus.

Soporte de los Diagramas de Diseño

PDT proporciona una interfaz gráfica para introducir y editar diagramas junto con los descriptores para cada entidad.

En la etapa de *especificación del sistema* el diagrama *general de análisis* permite agregar actores, escenarios, percepciones, acciones, y sus asociaciones. Este diagrama forma la vista de alto nivel del sistema. Los escenarios después pueden ser desarrollados en el diagrama de *escenarios*. El diagrama *general de metas* muestra las metas del sistema y sus submetas. El diagrama de *roles* permite agrupar metas, percepciones y acciones dentro de roles.

El soporte principal para la etapa de diseño de la arquitectura está en el diagrama *general del sistema*. PDT mantiene la consistencia (ver el siguiente punto) de este diagrama y las especificaciones de protocolos con respecto a las interacciones entre los agentes.

El diseño detallado construye los detalles internos de cada agente en términos de sus capacidades, planes, eventos, etc., los cuales una vez completos, pueden ser mapeados dentro de un lenguaje de programación de agentes como AgentSpeak(L) (sección 2.5).

Verificación de la Consistencia

La verificación de la consistencia dentro de PDT se basa en dos aspectos. El primero de éstos está continuamente activo y se refiere a la interfaz de usuario que previene que ciertos errores ocurran. Algunos de los errores que se pueden prevenir son:

- Definición: no es posible tener referencias a entidades no existentes. Si se crea una

referencia a una entidad entonces la entidad también se genera si no existe, y cuando se borra una entidad todas sus referencias se eliminan.

- Nombramiento: no es posible asignar el mismo nombre a dos entidades, por ejemplo una meta y un plan llamados *DeterminarExistencias*, tampoco tener dos metas con el mismo nombre (entradas duplicadas del mismo tipo).

- Errores de tipo simple: solamente se permiten conexiones válidas entre entidades, por lo tanto, no es posible conectar dos acciones.

- Inconsistencia entre los diferentes niveles de detalle: por ejemplo, si un agente se especifica para que lea un conjunto de creencias, entonces no puede contener un plan que escriba en ese conjunto de creencias.

El otro aspecto de la verificación de la consistencia se ejecuta sobre demanda, a través del menú *Tools* con la opción *Crosscheck*. Esto genera una lista de errores y advertencias que pueden ser revisadas por el desarrollador.

Propagación de Entidades

Cuando es posible y apropiado, la información se propaga desde una parte del diseño hacia otra. Por ejemplo, si una meta se asocia con un rol, y el rol a su vez se asocia con un agente, entonces la meta automáticamente queda asociada con el agente. También se propagan representaciones gráficas de entidades. Por ejemplo, si una meta nueva se crea en el diagrama de roles, entonces un icono para esta meta automáticamente se agrega al diagrama general del sistema.

Vistas Jerárquicas

La herramienta permite que cada agente se desarrolle con tantas capas de abstracción como sea necesario con el fin de mantener cada una de estas capas manejable. Esto se logra al hacer uso de las capacidades y los diagramas generales de capacidades.

Búsqueda de Nombres

Las entidades pueden ser agregadas dentro de los diagramas al crearlas o seleccionarlas de una lista desplazable de entidades existentes que se presenta al usuario. Esto asegura que no ocurrirán errores de nombramiento. Esto también es útil cuando se desarrollan sistemas grandes, donde este tipo de errores que se presentan frecuentemente son difíciles de depurar en etapas posteriores.

Generación de Reportes y Diagramas

PDT tiene la habilidad para generar un documento de diseño en formato HTML. Este documento contiene figuras e información textual. La información textual se obtiene de los descriptores de cada entidad. El documento también contiene un índice ("Diccionario") con todas las entidades de diseño. PDT permite que cada diagrama se guarde como una imagen en formato PNG que después puede ser impresa, o incluida dentro de documentos.

3.1.3. Un escenario para ilustrar la herramienta PDT

El escenario incluye dos robots que están recogiendo basura en el planeta Marte, que se simula mediante la rejilla que se muestra en la figura 3.2.

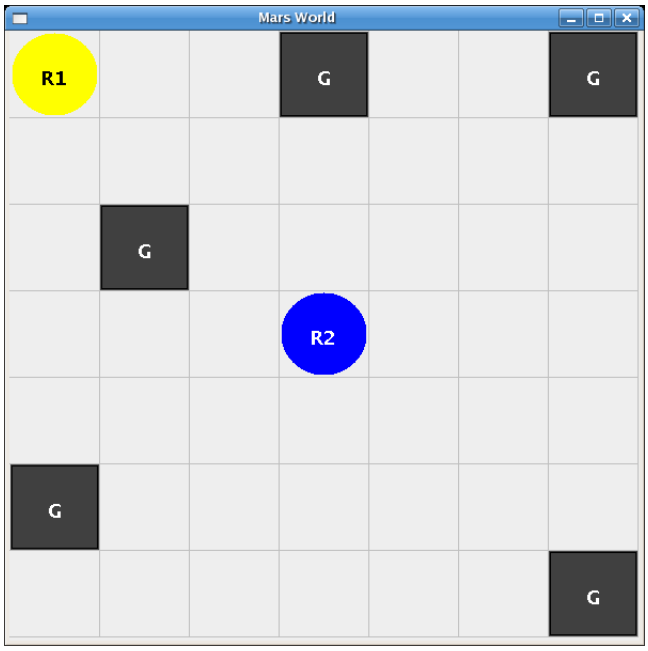

Figura 3.2: Simulación del ambiente de Marte.

El robot *r1* busca basura y cuando encuentra la recoge, va a donde se localiza el robot *r2*, tira la basura allí, y regresa al lugar donde encontró esta basura para continuar con su búsqueda desde esa posición. El robot *r2* está ubicado cerca de un incinerador. Cuando *r1* depósita basura en la ubicación de *r2*, éste la coloca en el incinerador. Varias piezas de basura (cuadros gris oscuro con la letra "G" en medio) son esparcidas aleatoriamente sobre la rejilla. Existe otra fuente de indeterminismo en este mundo, y ésta se refiere a una cierta imprecisión en el brazo del robot (*r1*) que recoge la basura. Sin embargo, el mecanismo es suficientemente bueno que nunca falla más de dos veces, es decir, en el peor de los casos *r1* intentará tomar la basura tres veces, pero para entonces ya la habrá recogido.

La figura 3.3 presenta el *diagrama general del sistema* que se obtiene para este escenario a través de la metodología Prometheus (sección 3.1.1) con la herramienta PDT (sección 3.1.2).

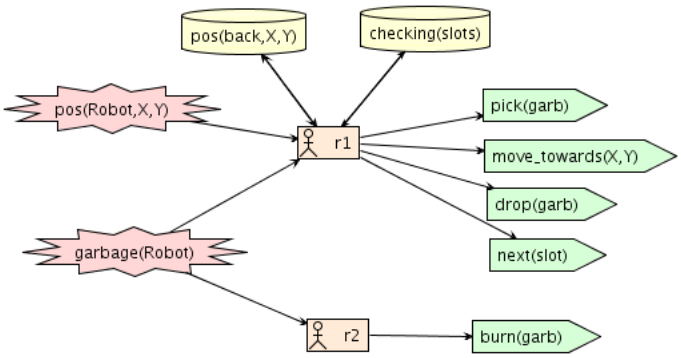

Figura 3.3: Diagrama general del sistema.

En este diagrama podemos apreciar a los robots *r1* y *r2*, sus percepciones (estrellas), creencias (repositorios) y acciones básicas (flechas). Aquí vemos que tanto el robot *r1* como *r2* perciben cuando hay basura en su ubicación (*garbage(Robot)*). Cuando *r1* se da cuenta de esto, lo primero que hace es guardar en su base de creencias la posición donde encontró la basura (*pos(back,X,Y)*), para que posteriormente pueda continuar con su labor de revisión a partir de esta posición. En seguida, *r1* toma la basura (*pick(garb)*), se dirige hacia donde se encuentra *r2* (*move_towards(X,Y)*), y ya ahí tira esta basura (*drop(garb)*). Después regresa a su ubicación original para continuar buscando más trozos de basura (*next(slot)*) hasta que ha limpiado toda la rejilla. Por su parte, cuando *r2* percibe basura en su ubicación, lo único que hace es quemarla (*burn(garb)*).

3.2. Jason

Jason es un intérprete, para una versión extendida del lenguaje de programación de agentes AgentSpeak(L) (ver sección 2.5), mediante el cual se llevó a cabo el desarrollo del MAS. Este intérprete está escrito en Java e implementa la semántica operacional de AgentSpeak(L) [5, 53] y sus extensiones [6, 15, 60]. Jason se distribuye libremente y puede ser descargado desde http://jason.sourceforge.net.

El ambiente de desarrollo integrado de Jason proporciona una interfaz gráfica, que

permite editar el archivo de configuración de un MAS y el código de los agentes escrito en AgentSpeak. A través de este ambiente es posible controlar la ejecución del MAS y distribuir a los agentes sobre una red de forma sencilla. Otra herramienta que acompaña a este ambiente y que es muy útil para depurar el MAS es el "inspector de mentes", que se muestra en la figura 3.4. Este inspector permite que el usuario observe el estado interno de los agentes en tiempo de ejecución.

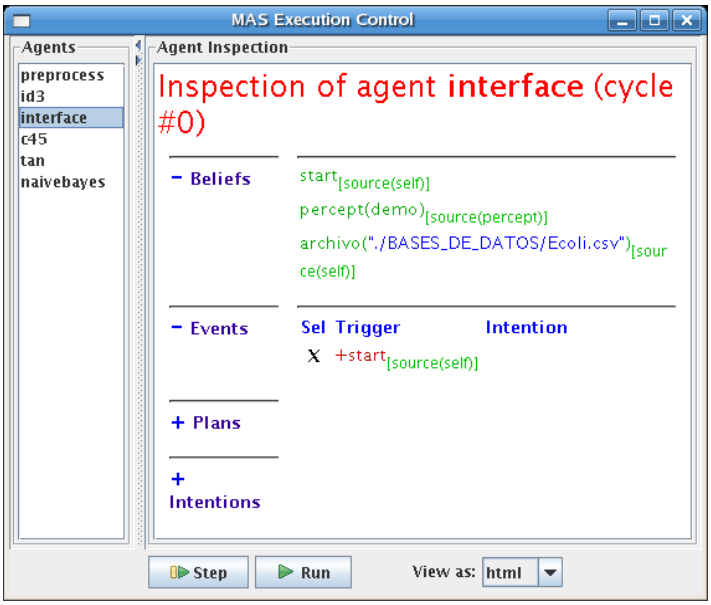

Figura 3.4: Inspector de Mentes de Jason.

Jason incluye: actos de habla basados en KQML para la comunicación entre los agentes; anotaciones de los planes para emplear funciones de selección basadas en la teoría de decisión; la posibilidad de ejecutar un MAS sobre una red de computadoras; funciones de selección configurables en Java; mecanismos de extensión y uso de código heredado, por medio de las "acciones internas" definidas por el usuario; y facilidades para la programación (en Java) del ambiente del MAS. Algunas de estas características se detallan en las siguientes secciones.

3.2.1. Configuración del MAS

La configuración de un MAS se especifica en un archivo de texto simple. La figura 3.5 muestra un ejemplo de este archivo de configuración para el escenario de los robots limpiadores que se presentó en la sección 3.1.3. La clase *marsEnv* implementa el ambiente del MAS; y el sistema tiene dos tipos de agentes: *r1* y *r2*.

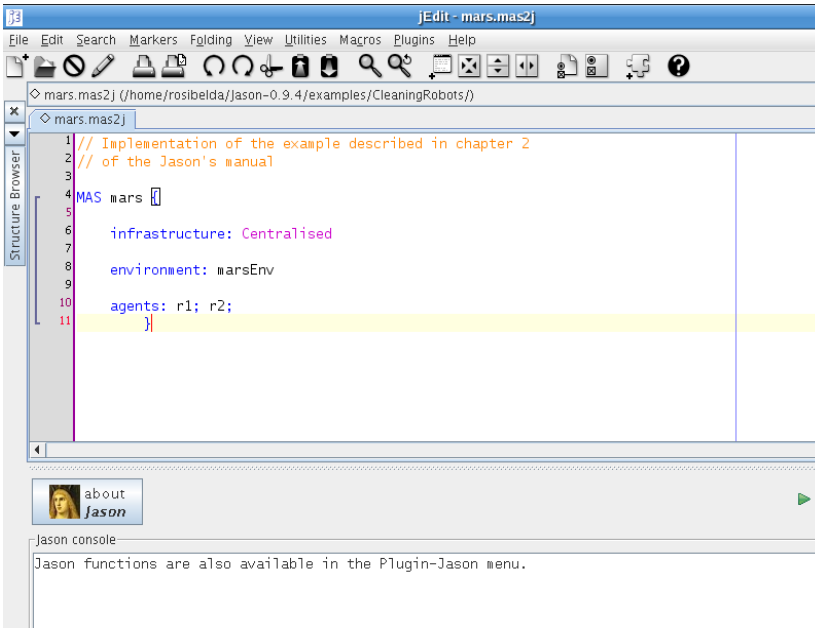

Figura 3.5: Configuración de un MAS

La gramática que se presenta a continuación proporciona la sintaxis que se emplea en el archivo de configuración. En esta gramática, ⟨`NUMBER`⟩ se usa para números enteros, ⟨`ASID`⟩ son identificadores AgentSpeak (nombres de los agentes), que deben empezar con una letra minúscula, ⟨`ID`⟩ es cualquier identificador, y ⟨`PATH`⟩ define la ruta donde se encuentra un archivo, de la misma forma como en los sistemas operativos.

mas → "MAS" ⟨ID⟩ "{"
["architecture" ":" ⟨ID⟩]
environment

```
                    agents
                "}"
environment     → "environment" ":" ⟨ID⟩ [ "at" ⟨ID⟩ ]
agents          → "agents" ":" ( agent )+
agent           → ⟨ASID⟩
                  [ filename ]
                  [ options ]
                  [ "agentArchClass" ⟨ID⟩ ]
                  [ "agentClass" ⟨ID⟩ ]
                  [ "#" ⟨NUMBER⟩ ]
                  [ "at" ⟨ID⟩ ]
                  ";"
filename        → [ ⟨PATH⟩ ] ⟨ID⟩
options         → "[" option ( "," option )* "]"
option          → "events" "=" ( "discard" | "requeue" | "retrieve" )
                | "intBels" "=" ( "sameFocus" | "newFocus" )
                | "verbose" "=" ⟨NUMBER⟩
```

El ⟨`ID`⟩ que aparece después de la palabra clave `MAS` es el nombre del MAS. La palabra clave `architecture` se emplea para especificar cuál de las dos arquitecturas de agentes disponibles en Jason será usada. Las opciones son: "`Centralised`" o "`Saci`"; la última de éstas permite distribuir el MAS sobre una red.

En seguida se define un ambiente (`environment`), mediante el nombre de la clase en Java que lo implementa. Se puede especificar el nombre de un servidor donde correrá este ambiente, siempre y cuando se eliga la opción SACI para la arquitectura del sistema.

La palabra `agents` se usa para especificar el conjunto de agentes que existirán en el MAS. Un agente se define por su nombre simbólico que se proporciona como un término en AgentSpeak (por ejemplo, como un identificador que comienza con una letra minúscula); este es el nombre que utilizarán los agentes para comunicarse entre ellos en el MAS. Después, opcionalmente se escribe el nombre del archivo que implementa en AgentSpeak al agente; por default Jason asume que este archivo se llama ⟨`nombre`⟩`.asl`, donde ⟨`nombre`⟩ es el

nombre simbólico del agente. También existe una lista opcional de configuraciones para el intérprete de AgentSpeak, que serán explicadas más adelante. Es posible crear un número de instancias (clones) de agentes del mismo tipo, que se indica con un número precedido por el símbolo `#`. Si este es el caso, el nombre del agente se formará con el nombre simbólico más un índice que indicará el número de instancia (comenzando a partir del 1). Si se requiere que el agente se ejecute en una red, se especifica con el nombre del servidor (después de la palabra "`at`" al final de la definición del agente) donde correrá este agente.

Las siguientes configuraciones están disponibles para el intérprete de AgentSpeak (éstas van seguidas de un signo '`=`' y después de una de las palabras claves que tienen asociadas, donde aquella que aparece subrayada es la opción default):

- `events:` las opciones son `discard`, `requeue` o `retrieve`; `discard` significa que se descartan los eventos externos para los que no existen planes aplicables, mientras que `requeue` se emplea cuando los eventos deben ser insertados al final de la lista de eventos del agente. Cuando `retrieve` se selecciona, la función `selectOption` definida por el usuario es llamada incluso si no hay planes relevantes/aplicables. Esto se puede usar, por ejemplo, para que los agentes soliciten planes a otros agentes quienes pueden tener el conocimiento necesario del que carecen actualmente los primeros de éstos.

- `intBels:` las opciones son `sameFocus` o `newFocus`; cuando las creencias internas son agregadas o removidas explícitamente dentro del cuerpo de un plan, el evento asociado es un evento disparador, los medios entendidos que resultan del plan aplicable que se seleccionó para ese evento pueden ser colocados en el tope de la pila de la intención que generó el evento (si `sameFocus` se selecciona). Si `newFocus` se escoge, el evento se maneja como externo (ejemplo, como la adición o eliminación de creencias a partir de la percepción del ambiente), por lo que se crea un nuevo foco de atención.

- `verbose:` un número entre `0` y `6` se debe especificar. Entre más grande es el número, más información acerca de ese agente (o agentes si el número de instancias es mayor a 1) se imprime en la consola donde el sistema se corre. El valor por default es `1`, que

imprime solamente las acciones que los agentes ejecutan en el ambiente y los mensajes que se intercambian.

Las arquitecturas de agentes definidas por el usuario y otras funciones también definidas por el usuario para cada agente pueden ser especificadas con las palabras claves `agentArchClass` y `agentClass`.

3.2.2. Creación de Ambientes

Los agentes pueden estar situados en ambientes reales o simulados. En el primer caso, el usuario tendría que personalizar la "arquitectura de agente general", como explica Bordini et al. en [56]; mientras que en el último caso, debe proporcionar una implementación del ambiente simulado a través de una clase en Java, que extiende la clase `Environment`. Como ejemplo, a continuación se presenta el ambiente simulado para el escenario de los robots limpiadores, que se introdujo en la sección 3.1.3:

```
1  public class MarsEnv extends Environment {
2
3      public static final int GSize = 7; //Dimensión del grid
4      public static final int GARB  = 16; //Código de la basura en el modelo del grid
5
6      public static final Term ns = DefaultTerm.parse("next(slot)");
7      public static final Term pg = DefaultTerm.parse("pick(garb)");
8      public static final Term dg = DefaultTerm.parse("drop(garb)");
9      public static final Term bg = DefaultTerm.parse("burn(garb)");
10     public static final Literal g1 = Literal.parseLiteral("garbage(r1)");
11     public static final Literal g2 = Literal.parseLiteral("garbage(r2)");
12
13     static Logger logger = Logger.getLogger(MarsEnv.class.getName());
14
15     MarsModel model;
16     MarsView  view;
17
18     @Override
19     public void init(String[] args) {
20         model = new MarsModel();
21         view = new MarsView(model);
```

```
22          model.setView(view);
23          updatePercepts();
24      }
25
26      @Override
27      public boolean executeAction(String ag, Structure action) {
28          try {
29              if (action.equals(ns)) {
30                  model.nextSlot();
31              } else if (action.getFunctor().equals("move_towards")) {
32                  int x = Integer.parseInt(action.getTerm(0).toString());
33                  int y = Integer.parseInt(action.getTerm(1).toString());
34                  model.moveTowards(x,y);
35              } else if (action.equals(pg)) {
36                  model.pickGarb();
37              } else if (action.equals(dg)) {
38                  model.dropGarb();
39              } else if (action.equals(bg)) {
40                  model.burnGarb();
41              } else {
42                  return false;
43              }
44          } catch (Exception e) {
45              e.printStackTrace();
46          }
47
48          updatePercepts();
49
50          try {
51              Thread.sleep(200);
52          } catch (Exception e) {}
53          return true;
54      }
55
56      /* Crea la percepción del agente con base a MarsModel */
57      void updatePercepts() {
58          clearPercepts();
59
60          Location r1Loc = model.getAgPos(0);
61          Location r2Loc = model.getAgPos(1);
62
63          Literal pos1 = Literal.parseLiteral("pos(r1," + r1Loc.x + "," + r1Loc.y + ")");
```

```
64          Literal pos2 = Literal.parseLiteral("pos(r2," + r2Loc.x + "," + r2Loc.y + ")");
65
66          addPercept(pos1);
67          addPercept(pos2);
68
69          if (model.hasObject(GARB, r1Loc)) {
70              addPercept(g1);
71          }
72          if (model.hasObject(GARB, r2Loc)) {
73              addPercept(g2);
74          }
75      }
76
77      class MarsModel extends GridWorldModel {
78
79          /* Clase que extiende la clase de Jason, GridWorldModel, para
80          implementar las acciones básicas (next(slot), move_towards(X,Y),
81          pick(garb), drop(garb) y burn(garb)) de los agentes */
82
83          ... }
84
85      class MarsView extends GridWorldView {
86
87          /* Clase que extiende la clase de Jason, GridWorldView, para
88          dibujar en el modelo del mundo de los agentes (rejilla), los dos
89          tipos de agentes (r1 y r2) que tiene el sistema y la basura que
90          se encuentra dispersa en este modelo */
91
92          ... }
93  }
```

Siempre que un agente intente ejecutar una acción básica (aquella que cambia el estado del ambiente), el nombre de este agente y un `Structure` que representa la acción elegida se enviarán como parámetros al método `executeAction` (línea 27).

En el ejemplo, se puede apreciar que `executeAction` ejecuta las acciones básicas: avanzar un paso (`model.nextSlot()`), moverse hacia donde se encuentra el otro agente (`model.moveTowards(x,y)`), recoger la basura (`model.pickGarb()`), tirar la basura (líneas 37-38) y quemar la basura (`model.burnGarb()`). Por último, este método actualiza las percepciones de los agentes (`updatePercepts()`), al indicarles su nueva localización (líneas

66-67), y si ésta tiene basura (`addPercept(g1), addPercept(g2)`).

Las dos clases que aparencen al final del ambiente, `MarsModel` (líneas 77-85) y `MarsView` (líneas 87-96), se encargan de implementar las acciones básicas de los agentes y de recrear la animación de este ambiente, respectivamente.

3.2.3. Definición de Nuevas Acciones Internas

Las acciones internas permiten que los agentes AgentSpeak permanezcan en un nivel de abstracción alto, además proporcionan una forma fácil de extender el código y de usar código heredado. Las acciones internas que son distribuidas con Jason (como parte de la biblioteca estándar de acciones internas) comienzan con un '.'. Las acciones internas que son implementadas por los usuarios deben ser organizadas en bibliotecas específicas. En el programa AgentSpeak la acción se accede por el nombre de la biblioteca, seguido de un '.' y luego por el nombre de la acción. Las bibliotecas se definen como paquetes en Java y cada acción en la biblioteca debe ser una clase en Java. El paquete y la clase se nombrarán de la misma forma como la biblioteca y la acción serán referidas en los programas AgentSpeak.

Por ejemplo, una de las acciones internas que se encuentra en la biblioteca estándar de Jason es *stopMAS*, que se encarga de abortar la ejecución de todos los agentes en el MAS, además, de cualquier ambiente simulado que existiese. El resto de las acciones internas de Jason las puede consultar desde el archivo doc/api/index.html de la distribución de Jason (carpeta donde se instala Jason).

3.3. Ambiente de Waikato para Análisis de Conocimiento (WEKA)

WEKA (figura 3.6) es la herramienta que proveyó las clases en Java que implementan los algoritmos de preprocesamiento (para reemplazar los valores faltantes y discretizar la base de datos), minería de datos (algoritmos ID3, C4.5, NB y TAN) y evaluación de

patrones (con el método de validación cruzada estratificada), que ejecutan los agentes del MAS. Esta herramienta proporciona soporte para llevar a cabo el proceso KDD, que entre otras cosas incluye: preparar los datos de entrada, aplicar algoritmos de minería de datos, evaluar estadísticamente los modelos aprendidos, y visualizar los datos de entrada y los resultados del aprendizaje. WEKA está escrita en Java y se puede descargar libremente desde http://www.cs.waikato.ac.nz/ml/weka/.

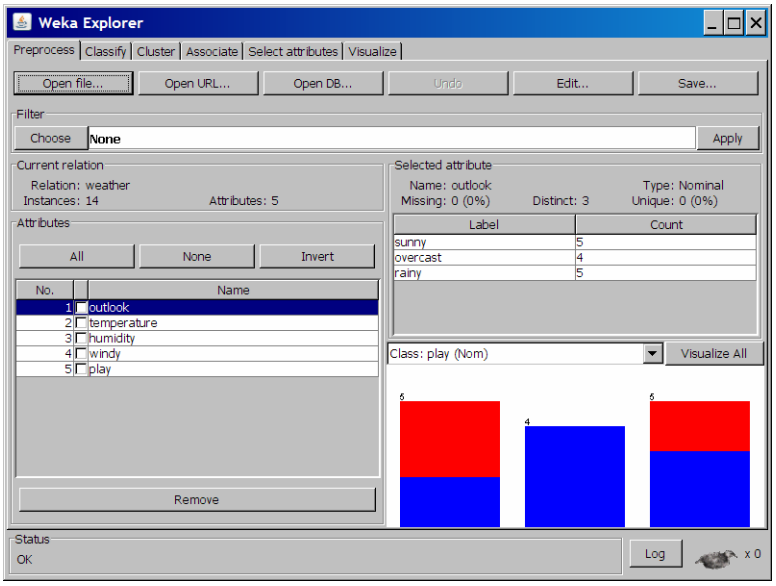

Figura 3.6: Herramienta WEKA.

WEKA contiene una variedad de algoritmos para realizar cada una de las tareas de minería de datos: clasificación, regresión, agrupamiento, aprendizaje de reglas de asociación y selección de atributos. Además, también dispone de facilidades para visualizar y preprocesar los datos [20]. La entrada de todos los algoritmos toma la forma de una tabla relacional, que puede ser leída desde un archivo o generada a través de una consulta de base de datos. El archivo puede estar en formato *.csv* o *.arff*, que es el formato nativo de WEKA. Una vez que los datos han sido leídos, varias herramientas de preprocesamiento de datos, llamados "filtros" se pueden aplicar a estos datos. Por ejemplo, es posible discretizar los datos numéricos.

Dentro de los algoritmos de clasificación se encuentran: los métodos Bayesianos (NB, NB complemento, NB multinomial, redes Bayesianas y AODE), los árboles de decisión (ID3, J4.8, árboles generados por poda de error reducido, árboles aleatorios, entre otros), las reglas (OneR, JRip, PART, tablas de decisión, Prism, etc.), los enfoques de separación del hiperplano (máquinas de vectores de soporte, regresión logística, perceptrones elegidos, Winnow y un perceptrón multi-capa) y los métodos de aprendizaje perezoso (IB1, IBk, reglas Bayesianas perezosas, KStar y aprendizaje ponderado localmente).

Además de los métodos de clasificación básicos, los esquemas de "meta-aprendizaje" permiten a los usuarios combinar uno o más algoritmos básicos en varias formas: "bagging", "boosting" (que incluye las variantes AdaboostM1 y LogitBoost) y "stacking". La clasificación puede ser sensible al coste, multi-clase o de clase nominal.

Existen implementaciones de varios algoritmos de regresión. Éstos incluyen regresión simple, regresión lineal múltiple, un perceptrón multi-capa, regresión de un vector de soporte, aprendizaje ponderado localmente, M5, reglas extraídas a partir de M5, entre otros. IB1 y IBk también pueden ser aplicados a problemas de regresión. Asimismo, existen esquemas de meta-aprendizaje como regresión aditiva y regresión por discretización.

Dentro de los algoritmos de agrupamiento se encuentran: KMeans, EM para modelos NB, agrupamiento primero el más lejos, Cobweb y agrupamiento basado en densidad. En el futuro cercano se espera incluir más algoritmos dentro de esta categoría.

Para el aprendizaje de reglas de asociación se dispone de Apriori, Tertius (que puede extraer reglas de primer orden) y Apriori Predictivo, que combina las estadísticas de confianza y soporte dentro de una medida única.

La selección de atributos se puede llevar a cabo con los enfoques "wrapper" y "filter". Entre los criterios de filtrado se encuentran la selección de características basada en correlación, la estadística chi-cuadrada, proporción de la ganancia, ganancia de información, incertidumbre simétrica y un criterio basado en máquina de vectores de soporte. También hay una diversidad de métodos de búsqueda: selección hacia delante y hacia atrás, búsqueda primero el mejor, búsqueda genética y búsqueda aleatoria.

Los filtros son procesos que pueden transformar un ejemplo de datos o conjuntos de ejemplos de datos. Estos filtros se clasifican de acuerdo a su aplicación, es decir, solo para el contexto de predicción ("supervisados") o para cualquier contexto ("no supervisados"). Los filtros se dividen aún más en "filtros de atributos", que trabajan sobre uno o más atributos de un ejemplo, y "filtros de ejemplos", que manipulan conjuntos de ejemplos.

Entre los filtros de atributos no supervisados se encuentran: agregar un nuevo atributo, añadir un indicador de agrupamiento, agregar ruido, copiar un atributo, discretizar un atributo numérico, normalizar o estandarizar un atributo numérico, mezclar valores de atributos, transformar valores nominales a binarios, remover atributos, reemplazar valores faltantes, calcular proyecciones aleatorias y procesamiento de series de tiempo. Los filtros de ejemplos no supervisados transforman ejemplos de datos escasos en ejemplos no escasos y viceversa, seleccionan al azar y re-muestrean conjuntos de ejemplos, y remueven ejemplos de acuerdo a un cierto criterio.

Los filtros de atributos supervisados incluyen soporte para la selección de atributos, discretización, transformación nominal a binaria y reordenamiento de los valores de la clase. Los filtros de ejemplos supervisados re-muestrean y sub-muestrean conjuntos de ejemplos para generar diferentes distribuciones de clases, por ejemplo, estratificada, uniforme y extensiones específicas de usuario.

En el siguiente capítulo se presenta el resultado de las exploraciones realizadas para dar soporte al proceso KDD mediante un MAS BDI. Este resultado se ve reflejado en el diseño y la implementación del proceso KDD como un MAS BDI.

Capítulo 4

Exploraciones sobre el soporte Multi-Agente BDI en el proceso KDD

En este capítulo se exponen el diseño y la implementación del MAS BDI, que se realizaron con la herramienta PDT (sección 3.1.2) y el intérprete de AgentSpeak(L) Jason (sección 3.2), respectivamente. Una de las ventajas que se encontró al utilizar estas herramientas es que la implementación del MAS se hizo de forma sencilla, debido a que los diagramas que se obtienen con PDT están expresados en el mismo lenguaje (creencias, metas, planes, eventos, etc.) que los agentes en AgentSpeak(L). La sección 4.1 presenta el diseño de este MAS, mientras que la sección 4.2 comenta algunos aspectos de su implementación.

4.1. Diseño del MAS

El diseño del MAS lo llevamos a cabo con la herramienta PDT (sección 3.1.2) versión 2.5f, que se basa en la metodología Prometheus (ver sección 3.1.1). Como recordaremos esta metodología consiste de tres fases principales: *Especificación del Sistema*, *Diseño de la Arquitectura* y *Diseño Detallado*, que nos guiarán para describir el diseño del MAS que presentamos a continuación.

4.1.1. Especificación del Sistema

El primer paso en esta etapa consistió en identificar las *metas* del MAS. Una vez hecho esto, tomamos cada una de ellas y nos preguntamos cómo sería cumplida, por lo tanto, generamos las *submetas*. Estas metas y submetas del sistema se capturaron en un *diagrama general de metas* dentro de PDT. Este diagrama se muestra en la figura 4.1.

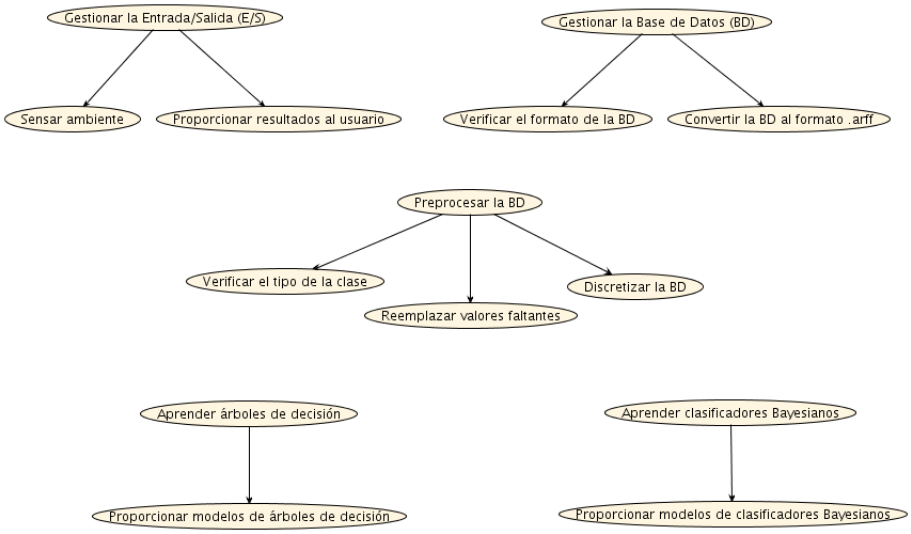

Figura 4.1: Diagrama general de metas.

En esta figura podemos apreciar que las metas y submetas del MAS son:

1. Gestionar la Entrada/Salida (E/S).
   1.1. Sensar el ambiente.
   1.2. Proporcionar resultados al usuario.
2. Gestionar la Base de Datos (BD).
   2.1. Verificar el formato de la BD.
   2.2. Convertir la BD al formato .arff.

3. Preprocesar la BD.

    3.1. Verificar de qué tipo es la clase.

    3.2. Reemplazar los valores faltantes.

    3.3. Discretizar la BD.

4. Aprender árboles de decisión.

    4.1. Proporcionar modelos de árboles de decisión.

5. Aprender clasificadores Bayesianos.

    5.1. Proporcionar modelos de clasificadores Bayesianos.

La meta 2 se creó para que la base de datos se convierta al formato .arff, pues éste es reconocido por todos los agentes del MAS. La submeta 3.1 se generó debido a que todos los algoritmos de aprendizaje (ID3, C4.5, NB y TAN), que son ejecutados por los agentes, solo pueden trabajar cuando la clase es nominal. Las submetas 3.2 y 3.3 se crearon con el fin de que el algoritmo ID3 puede aprender sus modelos sobre las bases de datos preprocesadas (sin valores faltantes ni numéricos). Finalmente, las metas 4 y 5 se concibieron para que los patrones extraídos por los agentes se representen mediante árboles de decisión y clasificadores Bayesianos.

Después de crear el diagrama general de metas desarrollamos los *escenarios*, que ilustran la operación del MAS. Como ya hemos mencionado, los escenarios constan de *percepciones*, *metas*, *escenarios* y *acciones*. Las percepciones y las acciones representan las entradas y salidas del sistema, respectivamente, mientras que las metas corresponden a las que definimos previamente. Los escenarios se capturaron en un *diagrama de escenarios* dentro de PDT. Este diagrama se presenta en la figura 4.2.

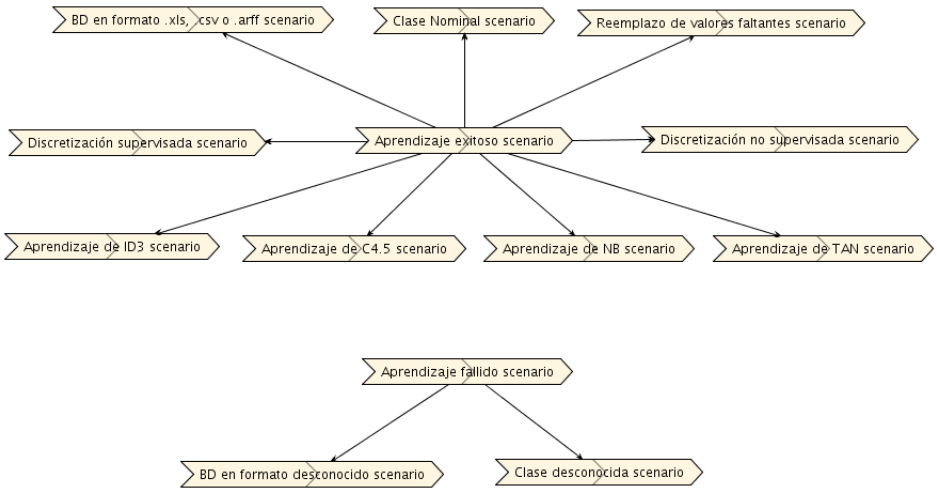

Figura 4.2: Diagrama de escenarios.

En esta figura podemos observar que los escenarios principales que pueden ocurrir en el MAS son:

1. Aprendizaje exitoso.

2. Aprendizaje fallido.

El escenario de "Aprendizaje exitoso" se lleva a cabo cuando se selecciona un modelo ganador a partir de los cuatro modelos aprendidos por los agentes, que ejecutan los algoritmos ID3, C4.5, NB y TAN. Mientras que el "Aprendizaje fallido" se presenta cuando la base de datos se ha proporcionado en un formato que los agentes desconocen (diferente de .xls, .csv o .arff), o cuando la clase de esta base de datos no es de tipo nominal.

En la figura 4.2 también es posible ver que existen otros escenarios que dependen de los escenarios comentados anteriormente. Por ejemplo, del "Aprendizaje exitoso" se originan:

1. BD en formato .xls, .csv o .arff.

2. Clase nominal.

3. Reemplazo de valores faltantes.

4. Discretización supervisada.

5. Discretización no supervisada.

6. Aprendizaje de ID3.

7. Aprendizaje de C4.5.

8. Aprendizaje de NB.

9. Aprendizaje de TAN.

y del "Aprendizaje fallido":

1. BD en formato desconocido.

2. Clase desconocida.

A continuación se presenta cada uno de estos escenarios en detalle. El escenario de "Aprendizaje exitoso" se muestra en la figura 4.3, donde la segunda columna define de qué tipo son los componentes del escenario. P se refiere a percepciones, G a metas, S a escenarios y A a acciones. La tercera columna lista los pasos del escenario. Cada uno de estos pasos se asocia a uno o varios roles, lo que se indica en la cuarta columna. Los roles se crearán en el siguiente paso, sin embargo, aquí mostramos los escenarios elaborados completamente. Por último, la columna final identifica los *datos* que serán leídos o escritos por el escenario, si ese fuera el caso.

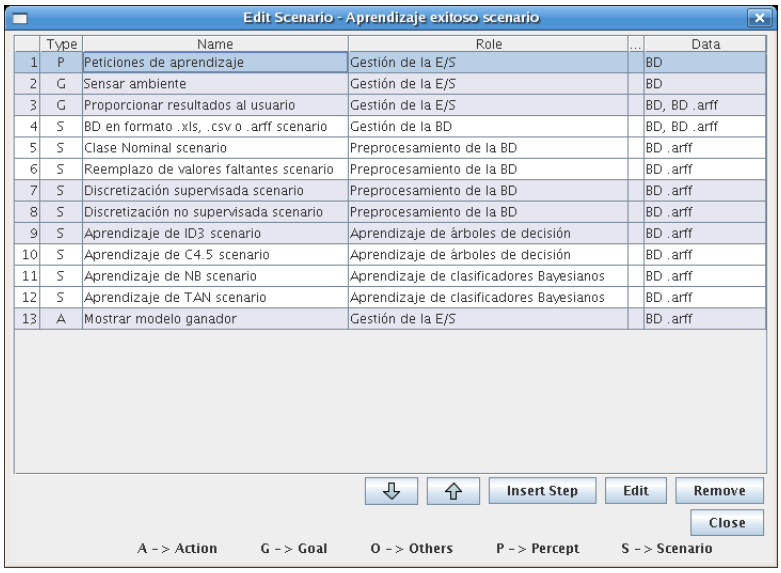

Figura 4.3: Escenario de Aprendizaje Exitoso.

Como se aprecia en esta figura, el escenario de "Aprendizaje exitoso" tiene como metas *Sensar el ambiente* y *Proporcionar resultados al usuario*. Inicia cuando percibe peticiones de aprendizaje. Luego se desarrollan sus escenarios (listados anteriormente), que detallaremos en seguida. Y finalmente, termina con la ejecución de la acción que muestra el modelo ganador.

El primer escenario del "Aprendizaje exitoso" es *BD en formato .xls, .csv o .arff*, que se presenta cuando la base de datos se proporciona en formato .xls, .csv o .arff. Este escenario se muestra en la figura 4.4.

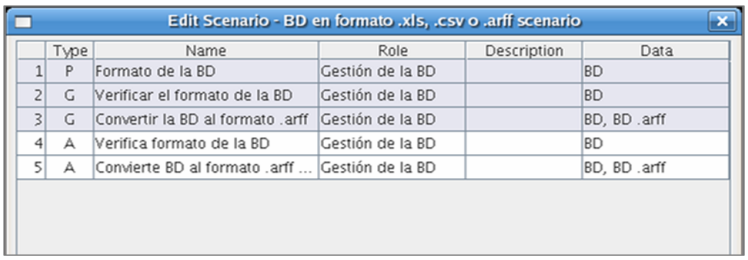

Figura 4.4: Escenario que se produce cuando la base de datos está en el formato .xls, .csv o .arff.

En esta figura podemos ver que el escenario percibe el formato de la base de datos y tiene como metas *Verificar el formato de la BD* y *Convertir la BD al formato .arff*. Para satisfacer estas metas realiza las acciones de verificar el formato de la base de datos y convertir esta base de datos a .arff (si se encuentra en .xls o .csv), respectivamente.

El escenario que se ejecuta después es "Clase Nominal" y se presenta en la figura 4.5.

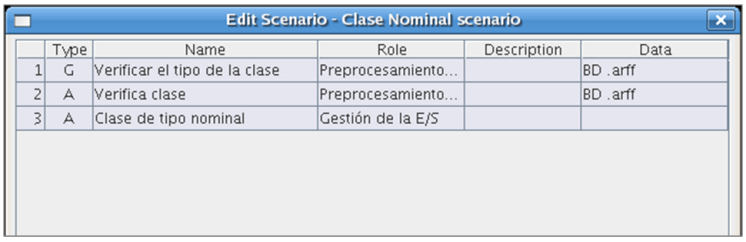

Figura 4.5: Escenario que se presenta cuando la clase es nominal.

Como podemos ver, este escenario tiene como meta *Verificar el tipo de la clase*, y para alcanzarla ejecuta la acción *Verifica clase*. Posteriormente, imprime *Clase de tipo nominal*.

El escenario de "Reemplazo de valores faltantes" se proporciona en la figura 4.6. Este escenario tiene como meta *Reemplazar valores faltantes*, y para esto lleva a cabo la acción *Reemplaza valores faltantes*.

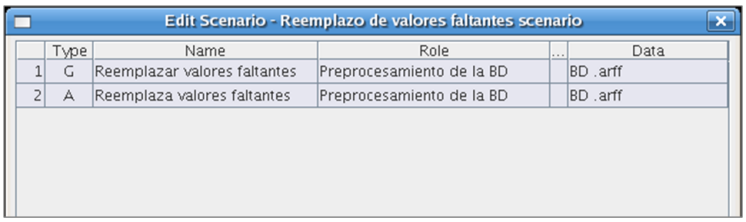

Figura 4.6: Escenario para el reemplazo de valores faltantes.

La figura 4.7 muestra el escenario de "Discretización supervisada", que tiene como meta *Discretizar la BD*. La única acción dentro de este escenario es *Discretiza la BD supervisadamente*. Es decir, en el proceso de discretización se toma en cuenta el valor de la clase para generar las diferentes categorías de datos. Cabe mencionar, que la discretización se basa en el método de Longitud de Descripción Mínima (MDL - Minimum Description Length) de Fayyad e Irani [70].

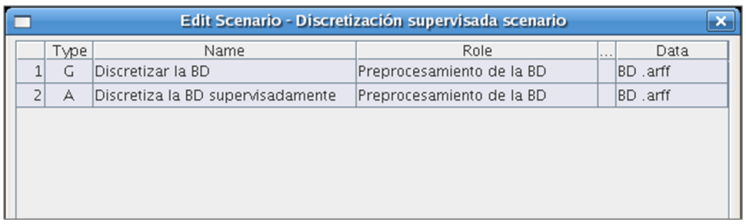

Figura 4.7: Escenario para discretizar la base de datos supervisadamente.

Por otra parte, la discretización no supervisada hace lo contrario que la discretización supervisada. Es decir, ignora la clase cuando crea las particiones de datos. La figura 4.8 presenta el escenario para la discretización no supervisada.

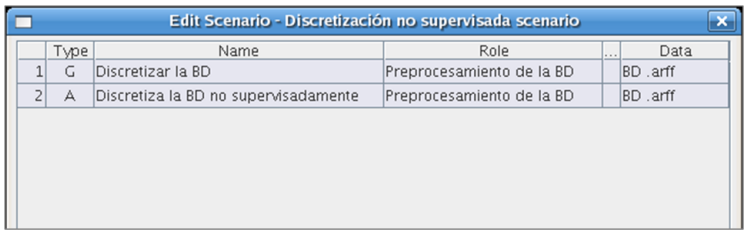

Figura 4.8: Escenario para discretizar la base de datos no supervisadamente.

Los escenarios "Aprendizaje de ID3" y "Aprendizaje de C4.5" comparten las mismas metas, que son *Aprender árboles de decisión* y *Proporcionar modelos de árboles de decisión*. Sin embargo, se diferencian en las acciones que ejecutan para lograr estas metas. Es importante notar que ambos escenarios aprenden los modelos con los datos discretizados de acuerdo a las dos formas (supervisada y no supervisadamente) comentadas anteriormente. Estos escenarios se muestran en las figuras 4.9 y 4.10, respectivamente.

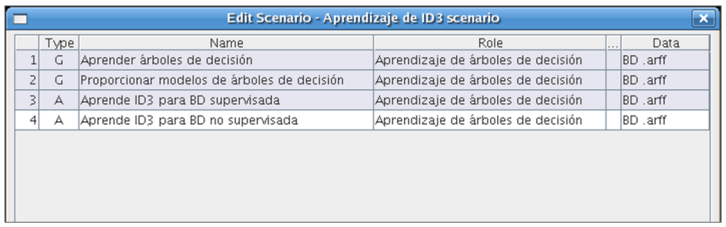

Figura 4.9: Escenario para aprender árboles de decisión con ID3.

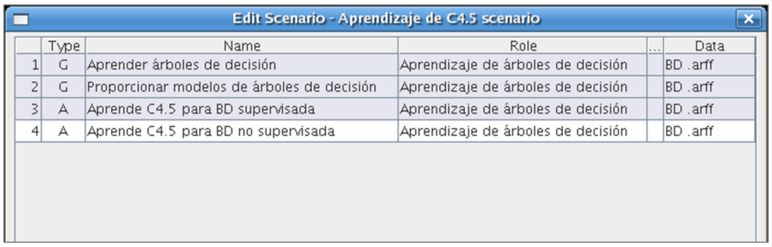

Figura 4.10: Escenario para aprender árboles de decisión con C4.5.

Los escenarios que construyen clasificadores Bayesianos, “Aprendizaje de NB” y “Aprendizaje de TAN”, también tienen las mismas metas, que son *Aprender clasificadores Bayesianos* y *Proporcionar modelos de clasificadores Bayesianos*. Sin embargo, el primero de estos escenarios aprende modelos NB, mientras que el segundo se enfoca en los clasificadores TAN. Las figuras 4.11 y 4.12 presentan estos escenarios, respectivamente.

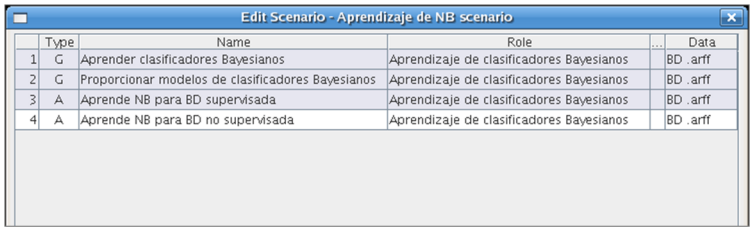

Figura 4.11: Escenario para aprender clasificadores Bayesianos con NB.

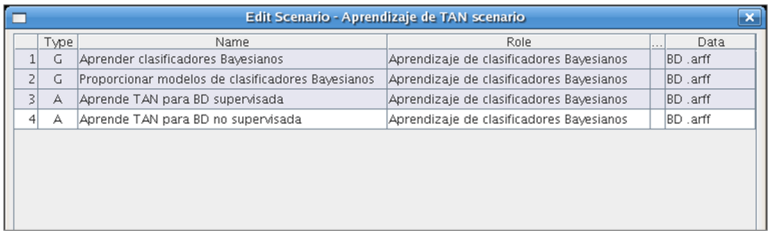

Figura 4.12: Escenario para aprender clasificadores Bayesianos con TAN.

El escenario “Aprendizaje fallido”, que ya se comentó anteriormente, se muestra en la figura 4.13. Éste difiere del “Aprendizaje exitoso” en los escenarios que contiene, *BD en formato desconocido* y *Clase desconocida*. Las figuras 4.14 y 4.15 muestran estos escenarios, respectivamente.

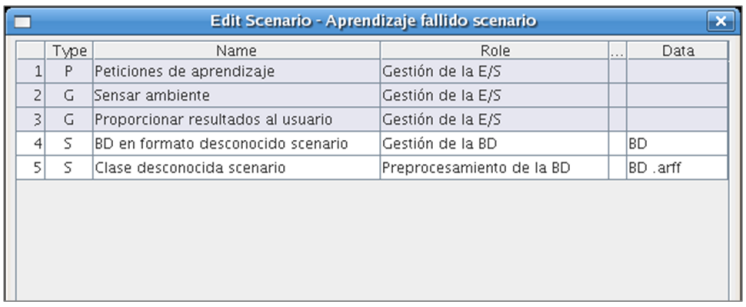

Figura 4.13: Escenario de Aprendizaje Fallido.

El escenario "BD en formato desconocido" lleva a cabo la acción *Verifica formato de la BD*, y cuando percibe que la base de datos no se encuentra en un formato aceptado (.xls, .csv o .arff) por los agentes ejecuta la acción *BD debe estar en formato .xls, .csv o .arff.*

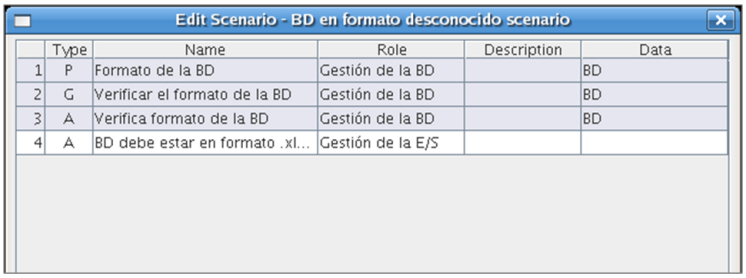

Figura 4.14: Escenario para la base de datos en formato desconocido.

El escenario "Clase desconocida" tiene como meta *Verificar el tipo de la clase*, por lo tanto, realiza la acción *Verifica clase*, y una vez que percibe que la clase no es nominal ejecuta *Clase debe ser nominal.*

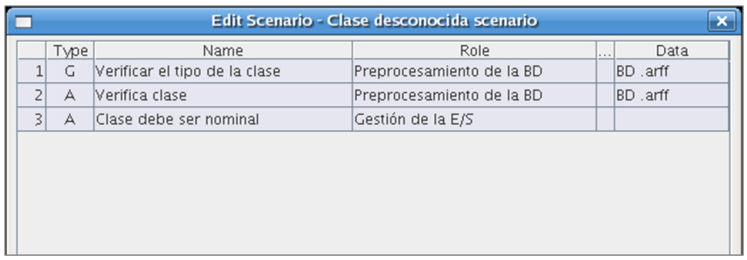

Figura 4.15: Escenario que se desarrolla cuando la clase no es nominal.

Una vez que las metas y los escenarios del sistema se habían desarrollado completamente, el siguiente paso fue crear los *roles* del sistema. Éstos se obtuvieron al agrupar las metas, percepciones y acciones en un *diagrama de roles* dentro de PDT. Este diagrama se muestra en la figura 4.16.

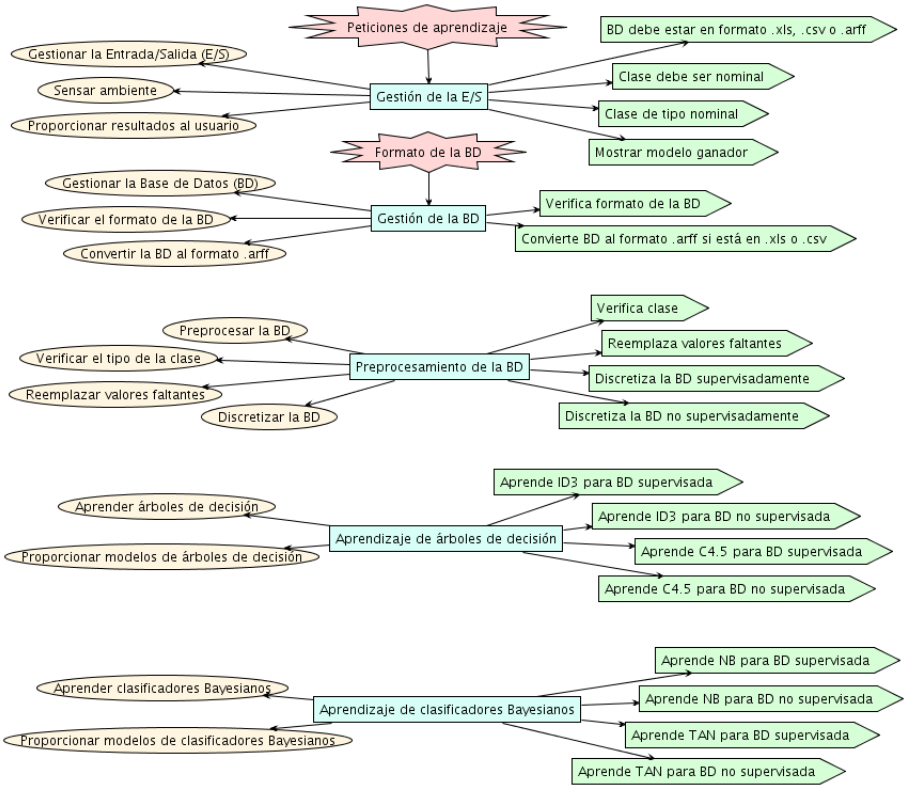

Figura 4.16: Roles del sistema.

En esta figura podemos ver que en el MAS se llevan a cabo cinco roles:

1. Gestión de la E/S.
2. Gestión de la BD.
3. Preprocesamiento de la BD.
4. Aprendizaje de árboles de decisión.
5. Aprendizaje de clasificadores Bayesianos.

El primero de éstos tiene como metas *Gestionar la Entrada/Salida (E/S)*, *Sensar ambiente* y *Proporcionar resultados al usuario*. Es el único de los roles que percibe peticiones de aprendizaje. Y comunica al usuario cuatro posibles acciones: la base de datos debe estar en formato .xls, .csv o .arff; la clase debe ser nominal; la clase es nominal o el modelo ganador.

Por su parte, "Gestión de la BD" percibe el formato de la base de datos, agrupa las metas *Gestionar la Base de Datos (BD)*, *Verificar el formato de la BD* y *Convertir la BD al formato .arff*. Para lograr estas metas lleva a cabo las acciones *Verifica formato de la BD* y *Convierte BD al formato .arff si está en .xls o .csv*.

"Preprocesamiento de la BD" se encarga de preprocesar la base de datos mediante la verificación de la clase, el reemplazo de valores faltantes y la discretización supervisada y no supervisadamente de la base de datos.

"Aprendizaje de árboles de decisión" y "Aprendizaje de clasificadores Bayesianos" construyen modelos para los dos tipos de discretización, donde el primero de estos roles emplea los algoritmos ID3 y C4.5, mientras que el segundo usa NB y TAN.

4.1.2. Diseño de la Arquitectura

Brevemente, esta fase tiene como objetivos identificar: los tipos de agentes que conformarán el MAS; las interacciones entre ellos; y la interfaz entre éstos y el ambiente en

forma de percepciones, acciones y datos externos. Para lograr esto, en PDT se genera un diagrama conocido como *diagrama general del sistema*, que se apoya en los diagramas de *acoplamiento de datos*, *agrupamiento de roles-agentes* y *conocimiento de agentes*. A continuación describimos cómo llevamos a cabo esta fase dentro de PDT, para el MAS que nos ocupa en este trabajo.

Dentro del *diseño de la arquitectura* lo primero que hicimos fue determinar qué roles leerían y/o escribirían los datos identificados en los escenarios. Por lo tanto, creamos el *diagrama de acoplamiento de datos* que se muestra en la figura 4.17.

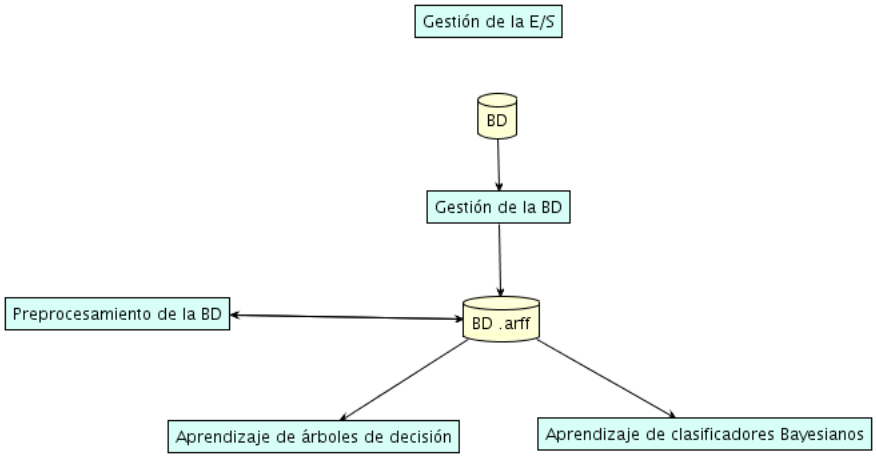

Figura 4.17: Diagrama de acoplamiento de datos.

En este diagrama vemos dos bases de datos, *BD* y *BD .arff*, donde la primera se refiere a la BD original que proporciona el usuario para llevar a cabo el aprendizaje. Mientras que *BD .arff* es la bases de datos que se obtiene al convertir *BD* (que está en formato .xls o .csv) con el rol *Gestión de la BD*. El rol *Preprocesamiento de la BD* recibe como entrada la base de datos en formato .arff, y sobre ésta realiza el reemplazo de valores faltantes y la discretización de datos. Tanto el *Aprendizaje de árboles de decisión* como el *Aprendizaje de clasificadores Bayesianos* construyen sus modelos con la base de datos preprocesada.

El siguiente paso en esta fase fue identificar con base en los roles creados los tipos de agentes requeridos en el sistema. Esto lo hicimos en un *diagrama de agrupamiento de*

roles-agentes en PDT. Este diagrama se puede ver en la figura 4.18.

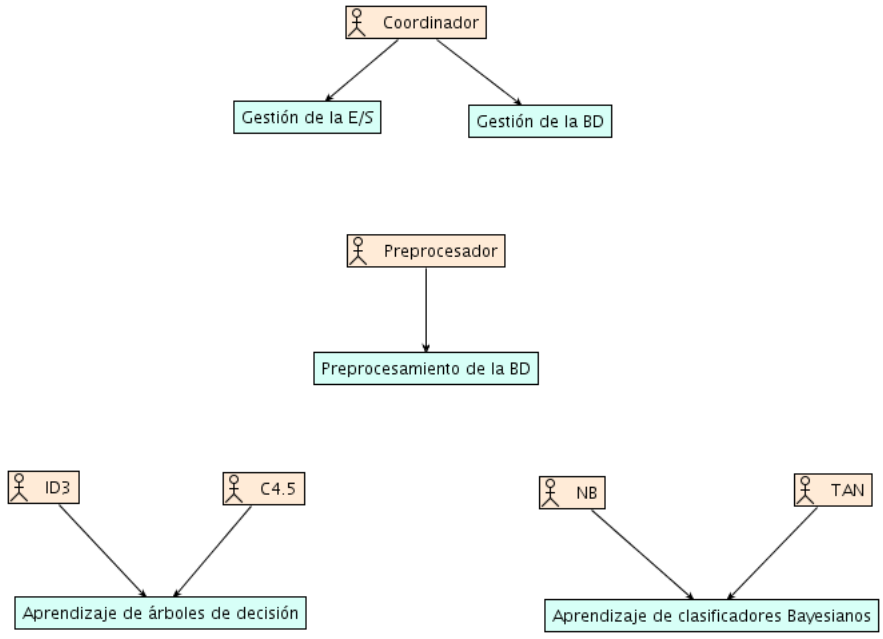

Figura 4.18: Diagrama de agrupamiento roles-agentes.

En esta figura se aprecia que el MAS está conformado de seis agentes: *Coordinador*, *Preprocesador*, *ID3*, *C4.5*, *NB* y *TAN*. El agente Coordinador se ocupa de gestionar la entrada/salida y la base de datos. Preprocesador se encarga de preprocesar la base de datos, es decir, suple los valores faltantes en esta base de datos y la discretiza de forma supervisada y no supervisada. Los agentes ID3 y C4.5 construyen árboles de decisión. Por último, NB y TAN aprenden clasificadores Bayesianos.

Una vez elaborados los dos diagramas que explicamos anteriormente, el siguiente paso fue crear el *diagrama general del sistema* que se muestra en la figura 4.19.

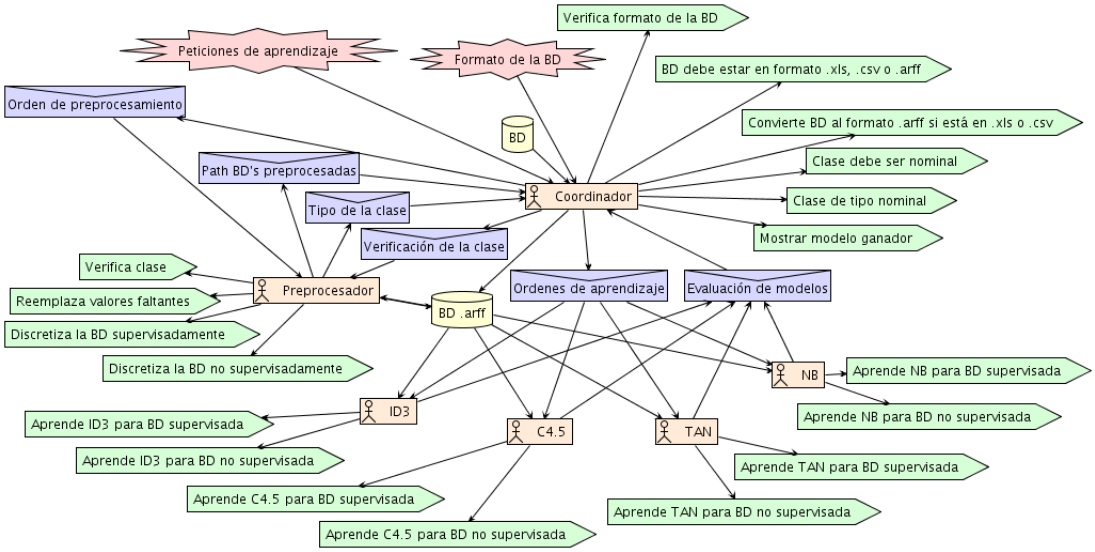

Figura 4.19: Diagrama general del sistema.

En este diagrama vemos que el agente Coordinador percibe peticiones de aprendizaje y el formato de la base de datos, y recibe ésta. Si la base de datos se encuentra en formato .xls o .csv la convierte a .arff, en otro caso, sino está en ninguno de estos formatos informa al usuario que debe proporcionarla en .xls, .csv o .arff. Después, este mismo agente indica a Preprocesador que revise de qué tipo es la clase. Si ésta no es nominal, Coordinador le dice al usuario que la clase de la base de datos debe ser de este tipo, en otro caso, imprime *Clase de tipo nominal* y envía una orden de preprocesamiento a Preprocesador, para que reemplace los valores faltantes y discretice la base de datos. Una vez que Preprocesador indica a Coordinador que la base de datos ha sido preprocesada, éste último emite ordenes de aprendizaje a ID3, C4.5, NB y TAN. Éstos aprenden sus respectivos modelos para los dos tipos de discretización de datos e informan a Coordinador el porcentaje de ejemplos clasificados correctamente para cada modelo. Finalmente, Coordinador selecciona y reporta el modelo ganador.

Un diagrama que se genera automáticamente mientras se elabora el diagrama anterior es el de *conocimiento de agentes*. Este diagrama muestra de qué tipo es la comunicación

entre los agentes que conforman el MAS. En la figura 4.20 se proporciona tal diagrama.

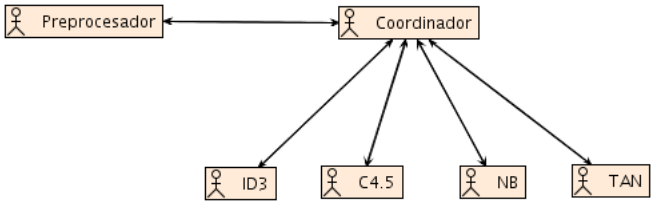

Figura 4.20: Diagrama de conocimiento de agentes.

En esta figura apreciamos que la comunicación entre Coordinador y Preprocesador es bidireccional, dado que el primero de éstos envía al segundo ordenes para verificar la clase de la base de datos y para preprocesar ésta, y éste (Preprocesador) a su vez regresa una respuesta a Coordinador. De igual forma, la comunicación entre Coordinador y los agentes aprendices (ID3, C4.5, NB y TAN) es bidireccional, pues después que Coordinador les indica que construyan sus modelos, ellos le envían la evaluación de éstos.

4.1.3. Diseño Detallado

En esta fase se especifica cómo los agentes llevarán a cabo sus tareas. Esto se logra al definir los detalles internos de cada agente, con base en sus percepciones, *planes*, acciones, datos que accesa y *mensajes* que envía a otros agentes del MAS.

La figura 4.21 muestra el *diagrama general del agente Coordinador* donde vemos que éste dispone de seis planes para realizar sus roles. El plan *Inicio* percibe las peticiones de aprendizaje, recibe la base de datos y lleva a cabo la acción que verifica el formato de la base de datos. Después, el plan *Verifica formato BD* percibe el formato de la base de datos (que se obtuvo con el plan anterior), si éste no es .xls, .csv o .arff informa al usuario que debe proporcionar la base de datos en uno de estos formatos, por otro lado, si el formato es .xls o .csv ejecuta el plan *Convierte la BD* que transforma la base de datos a .arff, en otro caso, envía una orden (a Preprocesador) para que se verifique de qué tipo es la clase. Cuando *Imprime tipo de la clase* percibe el tipo de la clase, imprime *Clase debe ser nominal*

o *Clase nominal*, dependiendo cual sea el caso. Si la clase fue nominal, emite una orden de preprocesador para que se reemplacen los valores faltantes y se discretice la base de datos. Una vez que *Construye modelos* percibe que la base de datos ha sido preprocesada envía ordenes de aprendizaje a los agentes ID3, C4.5, NB y TAN. Y cuando *Ganador* recibe las evaluaciones de los modelos aprendidos, selecciona y reporta el modelo ganador.

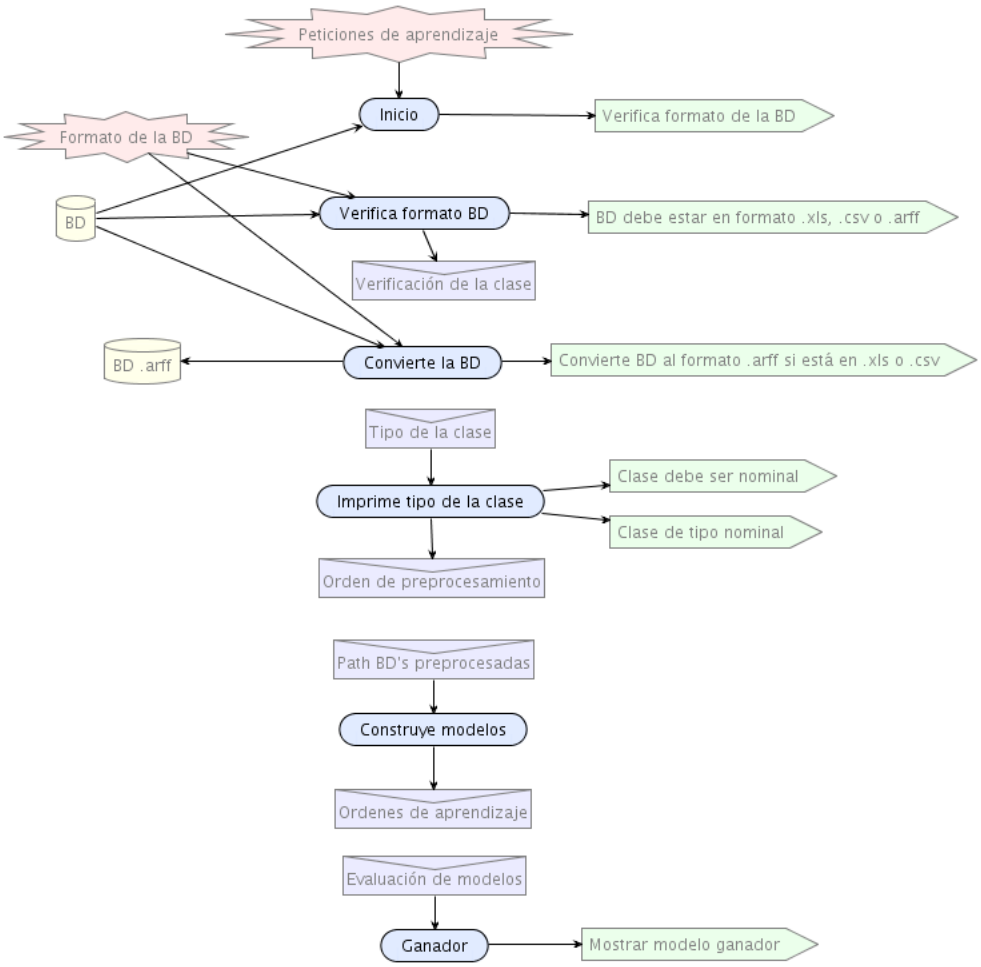

Figura 4.21: Diagrama general del agente Coordinador.

El *diagrama general del agente Preprocesador* aparece en la figura 4.22. Mediante este diagrama nos damos cuenta que Preprocesador tiene dos planes. *Verifica la clase* se ejecuta cuando recibe la orden para verificar la clase y el "path" de la base de datos en formato .arff,

después realiza la acción *Verifica clase* y por último envía el resultado (clase nominal o clase no nominal) de esta acción. Cuando *Preprocesa la BD* percibe la orden de preprocesamiento y el "path" de la base de datos en formato .arff, reemplaza los valores faltantes (con la moda cuando se trata de atributos nominales y con la media cuando los atributos son numéricos), discretiza (supervisada y no supervisadamente) esta base de datos y luego envía los "path's" donde se encuentran las bases de datos preprocesadas.

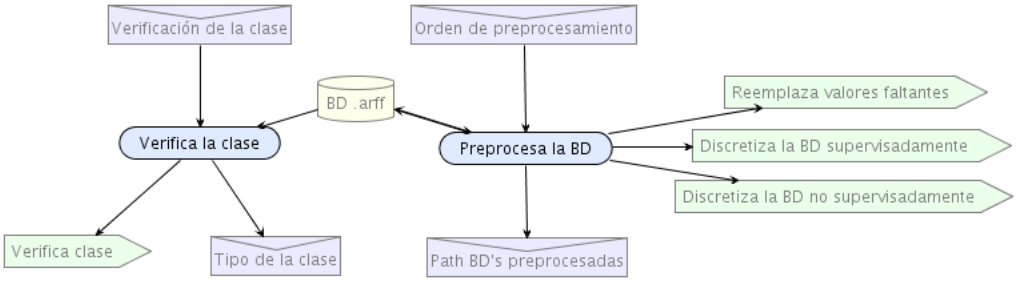

Figura 4.22: Diagrama general del agente Preprocesador.

Las figuras 4.23, 4.24, 4.25 y 4.26 muestran el *diagrama general del agente ID3*, *diagrama general del agente C4.5*, *diagrama general del agente NB* y *diagrama general del agente TAN*, respectivamente. Estos agentes cuentan con un solo plan, *Aprende modelos*. Este plan se ejecuta cuando percibe la orden de aprendizaje y el "path" de la base de datos preprocesada, posteriormente, lleva a cabo las acciones respectivas para aprender los modelos y finalmente envía la evaluación de éstos.

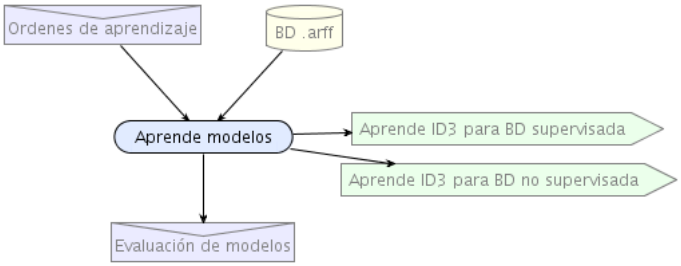

Figura 4.23: Diagrama general del agente ID3.

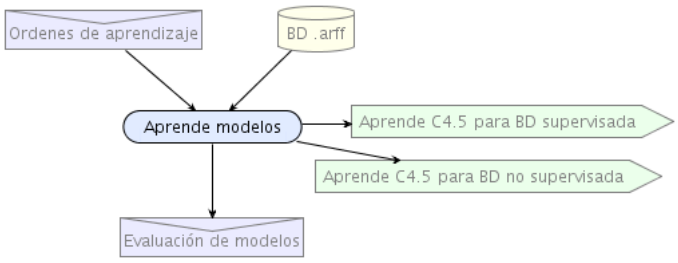

Figura 4.24: Diagrama general del agente C4.5.

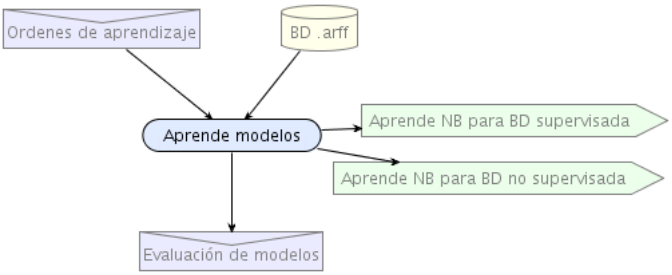

Figura 4.25: Diagrama general del agente NB.

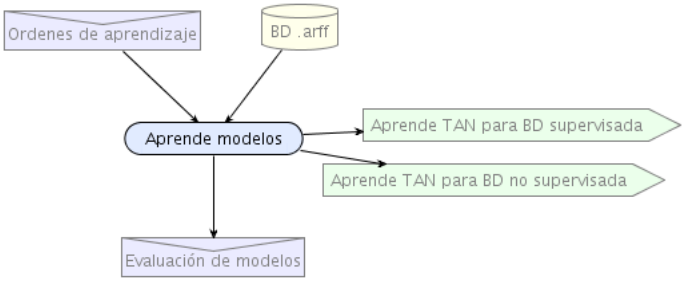

Figura 4.26: Diagrama general del agente TAN.

4.2. Implementación del MAS

El MAS fue desarrollado en el lenguaje de programación de agentes BDI, *AgentSpeak(L)* (sección 2.5), a través del intérprete *Jason* (sección 3.2), versión 0.9.4. Sin embargo, pos-

teriormente el MAS fue modificado para que corriera en las últimas versiones de este intérprete, 0.9.5b, 0.9.7 y 1.0. Este desarrollo se llevó a cabo en una portátil Inspiron 600m con las siguientes características: sistema operativo Fedora Core 5, procesador Intel(R) Pentium(R) M 1.70 GHz, 1 Gb en RAM, 80 Gb en disco duro.

De acuerdo al diseño que se elaboró para el MAS, éste consiste de seis agentes: *coordinador*, *preprocesador*, *id3*, *c45*, *nb* y *tan*. Es importante mencionar que estos agentes encapsulan dentro de sus acciones internas algunas clases en Java (que implementan algoritmos de preprocesamiento, minería de datos, evaluación de patrones, entre otros), que proporcionó la herramienta de KDD, WEKA (sección 3.3), versión 3.4.10. Para ejecutar Jason, WEKA y la herramienta de diseño PDT (sección 3.1.2) requerimos Java 1.5.0_07.

Las tablas 4.1 y 4.2 muestran los planes, acciones internas y clases de WEKA, que implementan a los agentes coordinador y preprocesador, respectivamente. Es importante notar que todas las acciones internas que se presentan en las tablas de esta sección, se refieren a las acciones que hemos definido, es decir, éstas no son parte de las acciones estándar que acompañan a Jason. En el apéndice A puede encontrar el código que implementa a los agentes del MAS, que se comenta en esta sección.

Tabla 4.1: Agente Coodinador.

Planes	**Acciones Internas**	**Clases WEKA**
@pi	verificaFormatoBD	
@pVFBD1		
@pVFBD2		
@pCBD1		
@pCBD2	convierteBD	Instances CSVLoader ArffSaver
@piTC1		
@piTC2		
@pcM		
@pg1		
@pg2		
@pg3		
@pg4		
@pg5		
@pg6		
@pg7		
@pg8		
@pg9		

En esta tabla vemos que el plan *@pCBD2* dispone de la acción interna *convierteBD*, que emplea las clases de WEKA: *Instances*, que manipula un conjunto de ejemplos ordenados, por ejemplo, el método "toString()" de esta clase devuelve estos ejemplos como una cadena en formato .arff; *CSVLoader*, que lee un archivo de texto que tiene sus columnas separadas por comas o por tabuladores; y *ArffSaver*, que escribe a un destino (archivo o salida estándar) en formato .arff. El objetivo de este plan es convertir la base de datos del formato .xls o .csv a .arff.

Tabla 4.2: Agente Preprocesador.

Planes	**Acciones Internas**	**Clases WEKA**
@pvC	verificaClase	ArffLoader Instances Atribute
@ppBD	reemplazaFaltantes	ArffLoader ArffSaver Instances Instance Filter ReplaceMissingValues
	discretizaS	ArffLoader ArffSaver Instances Instance Filter Discretize
	discretizaNS	ArffLoader ArffSaver Instances Instance Filter Discretize

Mediante esta tabla podemos ver que el plan *@ppBD* del agente preprocesador contiene las acciones internas *reemplazaFaltantes*, *discretizaS* y *discretizaNS*, que reemplazan los valores faltantes, discretizan la base de datos de forma supervisada y no supervisada, respectivamente. Las clases de WEKA utilizadas por estas acciones aparecen en la columna posterior a éstas.

En la tabla 4.3 se muestran los planes, acciones internas y clases de WEKA, que imple-

mentan a los agentes aprendices: id3, c45, nb y tan.

Tabla 4.3: Agentes Aprendices.

Agentes	**Planes**	**Acciones Internas**	**Clases WEKA**
id3	@paMID3	construyeID3	Instances Id3 Evaluation
c45	@paMC45	construyeC45	Instances J48 Evaluation
nb	@paMNB	construyeNB	Instances NaiveBayes Evaluation
tan	@paMTAN	construyeTAN	Instances Classifier Evaluation
id3, *c45*, *nb* y *tan*	@piTEyMDS		
	@piTEyMDNS		

En esta tabla es importante destacar que todos los agentes aprendices (id3, c45, nb y tan) disponen de los planes *@piTEyMDS* y *@piTEyMDNS*, que imprimen el tiempo que se tomó para construir el modelo con la base de datos discretizada supervisada y no supervisadamente, respectivamente. También vemos que estos planes no emplean acciones internas ni clases de WEKA, para llevar a cabo su tarea.

Además de las acciones internas que se presentaron en las tablas anteriores, los agentes hicieron uso de las acciones estándar: `.send`, `.print` y `.wait`. Como recordaremos `.send` (ver sección 2.5.1) implementa la comunicación entre los agentes del MAS. `.print` imprime cualquier tipo de mensajes en la consola donde se ejecuta el MAS. Estos mensajes pueden estar conformados de cadenas de texto y de términos AgentSpeak, que deben ser unificados (tener un valor) antes de imprimirse. Por último, `.wait` nos permite "dormir" o retrasar

la ejecución de un agente por un determinado número de milisegundos, con el objeto de sincronizar algunas acciones dentro del MAS.

Capítulo 5

Experimentación y Discusión

En este capítulo se presentan los resultados derivados de los experimentos llevados a cabo con el MAS y las contribuciones de este trabajo de tesis. La experimentación consistió en probar este MAS con distintas bases de datos, para comparar el desempeño (medido en términos de porcentajes de ejemplos clasificados correctamente y tiempo de construcción de los modelos) de los modelos aprendidos por los agentes. Las bases de datos se obtuvieron del repositorio de la Universidad de California [1]. La tabla 5.1 describe estas bases de datos, la segunda columna indica el número de ejemplos en la base de datos, la tercera columna proporciona el número de atributos que contiene ésta y la cuarta columna muestra cuantos valores tiene el atributo clase.

Tabla 5.1: Descripción de las Bases de Datos.

BD's	$\|\boldsymbol{Ejemplos}\|$	$\|\boldsymbol{Atributos}\|$	$\|\boldsymbol{Clases}\|$
Anneal	798	39	5
Balance_Scale	625	5	3
Credit_S	690	16	2
Diabetes	768	9	2
Ecoli	336	9	8
Hypothyroid	3163	26	2
Ionosphere	351	35	2
Iris	150	5	3
Lymphography	148	19	4
Segment	2310	20	7
Soybean	307	36	19
Vehicle	846	19	4
Zoo	101	18	7

El MAS se ejecutó diez veces con cada base de datos. Posteriormente, se promediaron los resultados (porcentajes de ejemplos clasificados correctamente y tiempos de construcción de los modelos) de las diez ejecuciones, para obtener las tablas que a continuación se presentan.

Las tablas 5.2 y 5.3 muestran los porcentajes de ejemplos clasificados correctamente de cada modelo aprendido por los agentes, con las bases de datos discretizadas supervisada y no supervisadamente, respectivamente.

Tabla 5.2: Porcentajes de ejemplos clasificados correctamente obtenidos con los datos discretizados supervisadamente.

BD's	**%ID3**	**%C4.5**	**%NB**	**%TAN**
Anneal	**93,23**	92.98	92.1	92.1
Balance_Scale	70.72	71.04	71.68	**72,32**
Credit_S	77.97	86.96	86.23	**87,25**
Diabetes	75.26	77.73	77.99	**78,12**
Ecoli	0	80.95	**85,42**	85.12
Hypothyroid	98.86	**99,24**	98.58	98.83
Ionosphere	89.17	88.89	90.6	**91,17**
Iris	**95,33**	93.33	94.67	94
Lymphography	79.05	77.7	**83,78**	82.43
Segment	94.85	**95,41**	91.77	92.21
Soybean	91.21	**92,51**	85.99	88.6
Vehicle	**71,63**	70.45	62.53	69.5
Zoo	1.98	94.06	**96,04**	**96,04**
$\boldsymbol{\mu}$	**72,25**	**86,25**	**85,95**	**86,74**

Tabla 5.3: Porcentajes de ejemplos clasificados correctamente logrados con los datos discretizados no supervisadamente.

BD's	**%ID3**	**%C4.5**	**%NB**	**%TAN**
Anneal	89.6	**89,97**	88.47	87.72
Balance_Scale	39.52	65.92	**90,88**	86.72
Credit_S	74.64	**85,8**	84.35	84.64
Diabetes	59.89	72.66	75.52	**76,95**
Ecoli	0	73.21	**83,03**	78.27
Hypothyroid	95.92	**97,53**	96.9	97.03
Ionosphere	85.75	88.6	**90,6**	**90,6**
Iris	91.33	**96**	95.33	91.33
Lymphography	76.35	75	**83,11**	79.05
Segment	91.13	**93,16**	88.96	89.52
Soybean	84.69	**89,25**	80.78	83.71
Vehicle	58.51	71.63	59.93	**72,46**
Zoo	1.98	92.08	90.1	**97,03**
$\boldsymbol{\mu}$	**65,33**	**83,91**	**85,23**	**85,77**

Al analizar los resultados de las dos tablas anteriores nos damos cuenta que en promedio, para los dos tipos de discretización de datos (supervisada y no supervisada), los modelos aprendidos con ID3 obtienen el peor desempeño, mientras que los construidos con TAN logran el porcentaje de clasificación correcta más alto. Aunque, en general los mejores clasificadores se consiguen cuando se aprenden a partir de datos discretizados supervisadamente.

Las tablas 5.4 y 5.5 proporcionan los tiempos de construcción de los modelos aprendidos por los agentes, con los datos discretizados supervisada y no supervisadamente, respectivamente.

Tabla 5.4: Tiempos de construcción de los modelos aprendidos con los datos discretizados supervisadamente.

BD's	**%ID3**	**%C4.5**	**%NB**	**%TAN**
Anneal	0.96	0.91	**0,22**	2.48
Balance_Scale	0.24	0.41	**0,11**	0.16
Credit_S	0.6	0.78	**0,15**	0.53
Diabetes	0.66	0.81	**0,2**	0.42
Ecoli	1.62	1.28	**0,3**	1.65
Hypothyroid	1.57	1.15	**0,33**	3.25
Ionosphere	0.63	0.65	**0,21**	1.47
Iris	0.31	0.17	**0,1**	0.26
Lymphography	0.48	0.46	**0,21**	0.37
Segment	1.04	0.57	**0,16**	2.72
Soybean	0.48	0.2	**0,05**	3.48
Vehicle	0.51	0.44	**0,04**	1.3
Zoo	0.12	0.05	**0,01**	0.53
$\boldsymbol{\mu}$	**0,71**	**0,61**	**0,16**	**1,43**

Tabla 5.5: Tiempos de construcción de los modelos aprendidos con los datos discretizados no supervisadamente.

BD's	**%ID3**	**%C4.5**	**%NB**	**%TAN**
Anneal	0.34	0.14	**0,02**	0.33
Balance_Scale	0.17	0.11	**0**	0.02
Credit_S	0.15	0.08	**0,01**	0.07
Diabetes	0.2	0.13	**0**	0.06
Ecoli	0.31	0.08	**0**	0.28
Hypothyroid	0.51	0.31	**0,06**	0.56
Ionosphere	0.21	0.06	**0,01**	0.14
Iris	0.06	0.13	**0**	0.04
Lymphography	0.12	0.06	**0**	0.16
Segment	0.4	0.18	**0,02**	0.37
Soybean	0.14	0.13	**0,01**	2.73
Vehicle	0.34	0.11	**0,02**	0.28
Zoo	0.13	0.06	**0,02**	0.29
$\boldsymbol{\mu}$	**0,24**	**0,12**	**0,01**	**0,41**

Los resultados de estas tablas nos indican que para los dos tipos de discretización de datos, NB siempre construye sus modelos en menor tiempo (debido a que solo aprende los parámetros del modelo, pues éste siempre es fijo) que el resto de los clasificadores, mientras que TAN es el que toma más tiempo en aprender sus modelos. También, podemos ver que cuando los datos se discretizan no supervisadamente, en general los modelos se construyen en menor tiempo.

Con respecto a las contribuciones de esta tesis podemos mencionar el diseño del proceso KDD que obtuvimos con la metodología Prometheus, que nos guía de manera clara para ejecutar una "instanciación" de este proceso, debido a que nos permite conocer quién (agente humano o de software) realizará cada una de las tareas, además, cómo (mediante los planes de los agentes) y cuándo se llevarán a cabo éstas. Esto es de gran ayuda, dado

que en la literatura no existe una metodología que nos oriente explícitamente sobre cómo realizar el proceso KDD. Por ejemplo, en la figura 5.1 apreciamos que el agente *Coordinador* verifica el formato de la base de datos, con la acción *Verifica formato de la BD* del plan *Inicio*, cuando percibe *Peticiones de aprendizaje* y recibe la *BD* (base de datos).

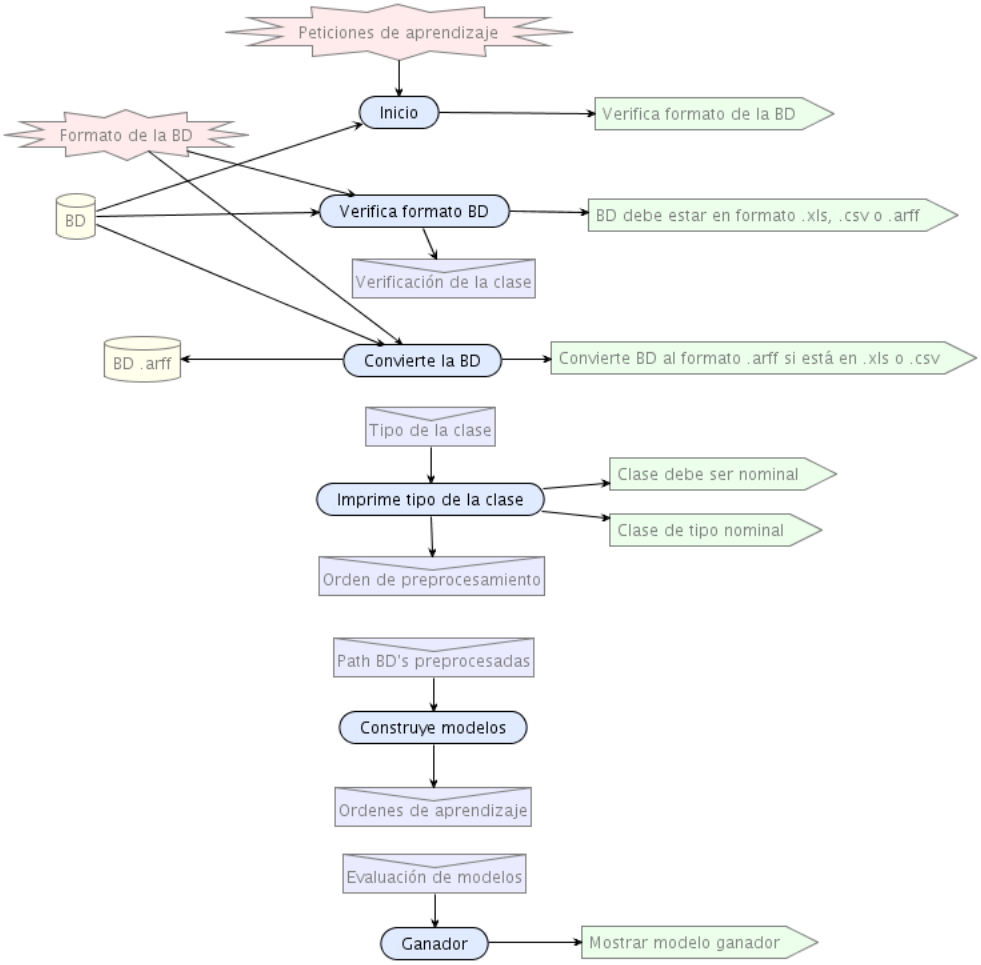

Figura 5.1: Diagrama general del agente Coordinador.

Otra de las aportaciones de nuestro trabajo es la integración que logramos entre el lenguaje de programación de agentes AgentSpeak(L), el intérprete Jason y la herramienta de KDD WEKA, lo cual dió lugar al MAS. La importancia de esto radica en el hecho de que abordamos un problema abierto, pues hasta ahora la literatura no nos indica cómo

realizar esta integración.

Capítulo 6

Conclusiones y Trabajo Futuro

En este capítulo se proporcionan las conclusiones y el trabajo futuro de la investigación que se desarrolló a lo largo de esta tesis. La sección 6.1 presenta las conclusiones de esta investigación, mientras que la sección 6.2 indica las tareas que continúan a partir de ella.

6.1. Conclusiones

En este trabajo se abordó el diseño e implementación de un MAS BDI, que resultó de explorar cómo dar soporte al proceso KDD, a través de la definición de *agentes BDI* especializados en algunas fases de este proceso (preprocesamiento, minería de datos y evaluación de patrones). Este MAS está conformado de seis agentes: *Coordinador*, que se encarga de gestionar la entrada y salida del sistema, y la base de datos; *Preprocesador*, que reemplaza los valores faltantes de la base de datos y la discretiza de manera supervisada y no supervisada; *ID3*, *C4.5*, *NB* y *TAN*, que a partir de la base de datos preprocesada — sin valores faltantes y discretizada — aprenden sus respectivos modelos de minería de datos, que envían al Coordinador para que éste seleccione y reporte al ganador.

El MAS realiza las tareas de minería de datos de *clasificación* y *modelado de dependencias*, que se implementan mediante los algoritmos ID3, C4.5, NB y TAN, que ejecutan los agentes aprendices (ID3, C4.5, NB y TAN) de este MAS. Los modelos aprendidos por estos algoritmos se representan mediante *árboles de decisión* y *clasificadores Bayesianos*. Para evaluar qué tan bien los modelos se ajustan a los datos, se utiliza el método de *validación cruzada estratificada con 10 particiones*.

Para elaborar el diseño del MAS se siguió la metodología *Prometheus* y se empleó la

herramienta *PDT* que da soporte a esta metodología. Algunas ventajas que se encontraron al utilizar Prometheus, para modelar el proceso KDD como un MAS BDI, son:

- Se obtuvo un entendimiento claro (representado por medio de los diagramas de diseño) del proceso KDD, por lo tanto, el diseño se podría comunicar de forma sencilla entre un equipo de trabajo.

- La transición entre el diseño y la implementación del MAS BDI se realizó de manera natural, debido a que Prometheus es una metodología dirigida al desarrollo de agentes inteligentes. Por consiguiente, los diagramas que se obtienen con Prometheus están expresados en los mismos términos (creencias, metas, planes, eventos, etc.) que los agentes BDI.

- Prometheus se puede usar para modelar cualquier proceso dentro de una organización, dado que sus diagramas ofrecen un nivel de abstracción alto, en el sentido que se pueden referir en un lenguaje muy cercano al nuestro (por ejemplo, mediante creencias, metas, escenarios, roles, planes, eventos, acciones, etc.).

La implementación del MAS se llevó a cabo en el lenguaje de programación de agentes *AgentSpeak(L)*, para lo cual se dispuso del intérprete *Jason*. El punto principal a favor de Jason, con respecto a otras plataformas de desarrollo de MAS (como JACK [51], JAM [42], JADEX [7], entre otras), es su fundamento teórico, que obtiene al implementar la semántica operacional de AgentSpeak(L).

Otra herramienta de gran utilidad para este trabajo fue *WEKA*, debido a que proveyó las clases (que implementan algoritmos de preprocesamiento, minería de datos, evaluación de patrones, entre otros) que permitieron extender de manera sencilla las acciones internas de los agentes. Esto fue posible gracias a que estas clases están implementadas en Java, que es uno de los lenguajes que se emplea en Jason para desarrollar las acciones de los agentes.

La experimentación llevada a cabo con el MAS consistió en ejecutar éste diez veces con 13 bases de datos (obtenidas del sitio de la Universidad de California [1]), para conocer el

desempeño (en términos de porcentajes de ejemplos clasificados correctamente y tiempos de construcción de los modelos) de los modelos aprendidos por los agentes. A partir del análisis de los resultados derivados de los experimentos, se encontró que para los dos tipos de discretización de datos (supervisada y no supervisada), TAN es el que proporciona los mejores modelos, aunque también es el que toma más tiempo en construirlos, mientras que ID3 es el que obtiene el peor desempeño. Además, nos dimos cuenta que no existe un algoritmo de aprendizaje único que pueda ser empleado en todos los dominios de aplicación. Por lo tanto, es necesario explorar los datos con diversos algoritmos, con el fin de encontrar el modelo que refleje mejor las características del fenómeno en cuestión.

Es importante destacar que el MAS es portable (es decir, puede ser ejecutado en diversos sistemas como Windows, Linux, Solaris, Mac OS X, etc.), gracias a que las herramientas (PDT, Jason y WEKA) que se utilizaron para su desarrollo tienen como plataforma el lenguaje Java. Además, el hecho de que estas herramientas se distribuyan libremente hace posible la construcción de sistemas sin costo alguno.

Finalmente, la exploración (sobre cómo dar soporte al proceso KDD a través de un MAS BDI) que llevamos a cabo en este trabajo nos dota de un MAS que resuelve en un contexto limitado problemas en el proceso KDD, pues aún quedan por implementar la iteración e interacción de este proceso, que se comentan en la siguiente sección.

6.2. Trabajo Futuro

El trabajo que se origina a partir de esta investigación consiste en:

- Realizar pruebas de hipótesis para comprobar si se sostienen los resultados que encontramos y que comentamos en el capítulo 5. Por ejemplo, que TAN es el mejor clasificador para ambos tipos de discretización (supervisada y no supervisada) o que los mejores modelos se consiguen cuando se construyen a partir de datos discretizados supervisadamente.

- Mejorar las competencias de los agentes para lograr la iteración del proceso KDD, por ejemplo, el agente TAN podría aprender varias estructuras de redes Bayesianas con una métrica de evaluación —Bayes, BDeu, MDL, Criterio de Información de Akaike o Entropía— diferente cada vez, y posteriormente, reportar al agente Coordinador la mejor de estas redes.

- Implementar la interacción entre el usuario y el MAS, mediante la integración de éste en WEKA. Por ejemplo, sería posible agregar un módulo en WEKA donde el usuario le indique al agente TAN que solo utilice la métrica MDL para construir redes Bayesianas, pues este usuario sabe que TAN no tiene buen desempeño con el resto de las métricas para los datos proporcionados.

Referencias

[1] A. Asuncion and D. Newman. *UCI Machine Learning Repository*. Irvine, CA: University of California, Department of Information and Computer Science. http://www.ics.uci.edu/ mlearn/MLRepository.html, 2007.

[2] A. Carreño. *Una plataforma para la definición de Sistemas Multiagentes y su aplicación en un problema de trabajo en grupo*. Tesis presentada para obtener el grado de Maestra en Inteligencia Artificial MIA-UV, Xalapa Veracruz, México, 2001.

[3] A. Dobra and J. Gehrke. *SECRET: A Scalable Linear Regression Tree Algorithm*. In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Edmonton, Alberta, Canada, July 2002.

[4] A. Guerra-Hernández, N. Cruz-Ramírez and R. Mondragón-Becerra. *Exploraciones sobre el soporte Multi-Agente en Minería de Datos*. Conferencia Internacional en Información, Comunicación y Diseño. Universidad Autónoma Metropolitana, Cuajimalpa, México, 2006. A imprimirse como capítulo de un libro.

[5] A. Moreira and R. Bordini. *An operational semantics for a BDI agent-oriented programming language*. In John-Jule Ch. Meyer and Michael J. Wooldridge, editors, Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), April 22-25, Toulouse, France, pages 45-59, 2002.

[6] A. Moreira, R. Vieira and R. Bordini. *Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication*. In Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia, 2003.

[7] A. Pokahr, L. Braubach and W. Lamersdorf. *Jadex: A BDI reasoning engine*. In R. Bordini, M. Dastani, A. Seghrouchni, and J. Dix, editors, Multi-Agent Programming. Kluwer, 2005.

[8] A. Prodromidis, P. Chan and S. Stolfo. *Meta-learning in Distributed Data Mining Systems: Issues and Approaches*. In Hillol Kargupta and Philip Chan, editors, Advances of Distributed Data Mining. MIT/AAAI Press, 2000.

[9] A. Rao. *AgentSpeak(L): BDI agents speak out in a logical computable language*. In W. Van de Velde and J. W. Perram, editors, Agents Breaking Away: Proceedings of the 7th 24 European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038), 42-55. Springer-Verlag, 1996. http://citeseer.ist.psu.edu/article/rao96agentspeakl.html

[10] A. Rao and M. Georgeff. *BDI Agents from Theory to Practice*. Technical Note 56, AAII, April 1995. http://citeseer.ist.psu.edu/rao95bdi.html

[11] A. Symeonidis and P. Mitkas. *Agent Intelligence Through Data Mining*. Springer-Verlag, 2005. ISBN 0387257578.

[12] C. Castelfranchi. *Modelling social action for AI agents*. Artificial Intelligence, 103(1998):157-182, 1998.

[13] C. Shannon. *A mathematical theory of communication*. Bell System Technical Journal, vol. 27, pp. 379-423 and 623-656, July and October, 1948.

[14] C. Shoemaker, M. Sao Pedro, S. A. Alvarez, and C. Ruiz. *Prediction vs. description: Two Data Mining Approaches to the Analysis of Genetic Data*. In Proc. of the Twelfth Genetic Analysis Workshop (GAW12), pages 449-453. Southwest Foundation for Biomedical Research, Oct. 2000.

[15] D. Ancona, V. Mascardi, J. Hübner and R. Bordini. *Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange*. In Nicholas R. Jennings, Carles Sierra, Liz So-

nenberg, and Milind Tambe, editors, Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, 19-23 July, pages 698-705, New York, NY, 2004. ACM Press.

[16] D. Dennett. *The Intentional Stance*. MIT Press, Cambridge, MA., USA, 1987.

[17] D. Geiger and D. Heckerman. *Knowledge representation and inference in similarity networks and Bayesian multinets*. In Artificial Intelligence 82 (pp. 45-74), 1996.

[18] D. Jensen, Y. Dong, B. Lerner, E. McCall, L. Osterweil, S. Sutton and A. Wise. *Coordinating Agent Activities in Knowledge Discovery Processes*. In Proceedings of Work Activities Coordination and Collaboration Conference (WACC), 1999.

[19] D. Michie, D. Spiegelhalter and C. TaylorD. *Machine learning of rules and trees*. In Machine Learning, Neural and Statistical Classification. Ellis Horwood, 1994.

[20] E. Frank, M. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. Witten and L. Trigg. *WEKA - A Machine Learning Workbench for Data Mining*. In O. Maimon and L. Rokach, editors, The Data Mining and Knowledge Discovery Handbook, pages 1305-1314. Springer, 2005.

[21] F. Cozman. *Generalizing variable-elimination in Bayesian Networks*. In Workshop on Prob. Reasoning in Bayesian Networks at SBIA/Iberamia, pages 21-26, 2000.

[22] F. Groen, M. Spaan and J. Kok. *Real world multiagent systems: information sharing, coordination and planning*. In Trilateral workshop on Military Applications of Agent Technology in ICT and Robotics, TNO FEL, The Hague, November 2004.

[23] G. Cooper and E. Herskovits. *A Bayesian method for the induction of probabilistic networks from data*. Machine Learning, 9:309–347, 1992.

[24] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999. ISBN 0262232030.

[25] H. Hamilton. *Overview of the KDD Process*. http://www2.cs.uregina.ca/~dbd/cs831/notes/kdd/1_kdd.html, consultada el 04 de diciembre de 2006.

[26] H. Huy, T. Kawamura, T. Hasegawa. *From Web Browsing to Web Service - Fertilizing Agent Environment*. In Proceedings of Workshop on Web-services and agent-based engineering (WSABE 2003), 2003.

[27] H. Kargupta, B. Park, D. Hershberger and E. Johnson. *Collective Data Mining: A New Perspective Towards Distributed Data Mining*. In Hillol Kargupta and Philip Chan, editors, Advances in Distributed and Parallel Knowledge Discovery, pages 133-184. MIT/AAAI Press, 2000.

[28] H. Kargupta, I. Hamzaoglu and B. Stafford. *Scalable, Distributed Data Mining Using An Agent Based Architecture*. In David Heckerman, Heikki Mannila, Daryl Pregibon, and Ramasamy Uthurusamy, editors, Proceedings of Knowledge Discovery And Data Mining, pages 211-214, Menlo Park, CA, 1997. AAAI Press.

[29] *Intérprete informático*. http://es.wikipedia.org/wiki/Int, consultada el 05 de diciembre de 2006.

[30] I. Rish. *An Empirical Study of the Naive Bayes Classifier*. In Proceedings of IJCAI-01 Workshop on Empirical Methods in Artificial Intelligence, 2001.

[31] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[32] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000. ISBN 1-55860-489-8.

[33] J. Quinlan. *Induction of decision trees*. Machine Learning, 1(1), 81-106, 1986.

[34] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[35] K. Sycara. *Multiagent Systems*. AI Magazine, 19(2):79-92, 1998.

[36] K. Thearling. *An Introduction to Data Mining: Discovering Hidden Value in Your Data* . White Paper. http://www.thearling.com/text/dmwhite/dmwhite.htm

[37] L. Padgham, J. Thangarajah and M. Winikoff. *Tool support for agent development using the prometheus methodology*. In Fifth International Conference on Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia, pages 383-388. IEEE Computer Society, 2005.

[38] L. Padgham and M. Winikoff. *Prometheus: A Methodology for Developing Intelligent Agents*. Proceedings of the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS 2002. July, 2002, Bologna, Italy.

[39] M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA., USA, and London, England, 1987.

[40] M. d'Inverno and M. Luck. *Engineering AgentSpeak(L): A formal computational model*. Journal of Logic and Computation, 8(3), 233-260, 1998. http://citeseer.ist.psu.edu/dinverno98engineering.html

[41] M. Dunham. *Data Mining: Introductory and Advanced Topics*. Prentice Hall PTR, 2002. ISBN 0130888923.

[42] M. Huber. *JAM: A BDI-Theoretic Mobile Agent Architecture*. In Proceedings of the Third Annual Conference on Autonomous Agents, pp. 236-243, O. Etzioni, J. P. Müller, J. Bradshaw, editors, ACM Press, 1999.

[43] M. Klusch, S. Lodi and G. Moro. *Agent-Based Distributed Data Mining: The KDEC Scheme*. In Intelligent Information Agents: The AgentLink Perspective, LNAI 2586, pages 104-122. Springer, July 2003.

[44] M. Wooldridge. *Intelligent Agents*. In G. Weiss, editor: Multiagent Systems (MIT Press), 1999.

[45] M. Wooldridge. *Reasoning About Rational Agents*. The MIT Press, 2000. ISBN 0-262-23213-8.

[46] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons Ltd, 2002. ISBN 978-0-471-49691-5.

[47] M. Wooldridge and N. Jennings. *Intelligent Agents: Theory and Practice*. In Knowledge Engineering Review 10(2), 1995.

[48] N. Friedman, D. Geiger and M. Goldszmidt. *Bayesian network classifiers*. Machine Learning, 29, 131-163, 1997.

[49] N. Jennings. *Agent software*. In Proc. UNICOM Seminar on Agent Software, pages 12-27, 1995.

[50] O. Etzioni and D. Weld. *Intelligent agents on the Internet: Fact, Fiction, and Forecast*. IEEE Expert, 10(4), August 1995.

[51] P. Busetta, R. Rönnquist, A. Hodgson and A. Lucas. *JACK Intelligent Agents - Components for Intelligent Agents in Java*. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998.

[52] R. Agrawal and R. Srikant. *Fast algorithms for mining association rules*. In Proceedings of the 20th International Conference on Very Large Databases, Santiago, September 1994.

[53] R. Bordini and A. Moreira. *Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L)*. Annuals of Mathematics and Artificial Intelligence, 42(1-3):197-226, September 2004. Special Issue on Computational Logic in Multi-Agent Systems.

[54] R. Bordini and J. Hübner. *BDI agent programming in agentspeak using Jason*. In Toni, F., and Torroni, P., eds., Proceedings of the Sixth International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VI), London, UK, 27-29 June, 2005, Revised Selected and Invited Papers, number 3900 in Lecture Notes in Computer Science, 143-164. Berlin: Springer-Verlag.

[55] R. Bordini, J. Hübner, et al. *Jason: A Java-based Interpreter for an Extended Version of AgentSpeak*, manual, release version 0.9 edn., July 2006. http://jason.sourceforge.net/.

[56] R. Bordini, M. Dastani, J. Dix and A. El Fallah Seghrouchni. *In Multi-Agent Programming: Languages, Platforms and Applications*. New York: Springer, 2005. ISBN: 0-387-24568-5.

[57] R. García. *Ingeniería concurrente y tecnologías de la información*. Ingenierías, Enero-Marzo 2004, Vol. VII, No. 22.

[58] R. Kohavi. *A study of cross-validation and bootstrap for accuracy estimation and model selection*. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence. San Mateo, CA: Morgan Kaufmann, 1995.

[59] R. Mihalcea and P. Tarau. *Multi-Document Summarization with Iterative Graph-based Algorithms*. In Proceedings of the First International Conference on Intelligent Analysis Methods and Tools (IA 2005), McLean, VA, May 2005.

[60] R. Vieira, A. Moreira, M. Wooldridge and R. Bordini. *On the formal semantics of speech-act based communication in an agent-oriented programming language*. Submitted article, to appear, 2005.

[61] R. Vilalta, C. Giraud-Carrier and P. Brazdil. *Meta-Learning: Concepts and Techniques*. Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers. Oded Maimon and Lior Rokach, Editors. Springer Publishers, 2005.

[62] S. Bailey, R. Grossman, H. Sivakumar and A. Turinsky. *Papyrus: A System for Data Mining Over Local and Wide Area Clusters and Super-clusters*. In Proceedings of the 1999 ACM/IEEE conference on Supercomputing, page 63, Portland, OR, 1999. ACM Press.

[63] SPSS. http://www.spss.com/clementine/.

[64] S. Russell and P. Norvig. *Inteligencia Artificial: Un enfoque moderno*. Prentice Hall, 1995.

[65] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[66] U. Fayyad, G. Piatetsky-Shapiro and P. Smyth. *From data mining to knowledge discovery in databases*. AI Magazine, 17(3):37–72, 1996.

[67] U. Fayyad, G. Piatetsky-Shapiero and P. Smyth. *Knowledge Discovery and Data Mining: Towards a Unifying Framework*. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, August 2-4, 1996.

[68] U. Fayyad, G. Piatetsky-Shapiro and P. Smyth. *The KDD process for extracting useful knowledge from volumes of data*. In Communications of the ACM – Data Mining and Knowledge Discovery in Databases, November 1996.

[69] U. Fayyad, G. Piatetsky-Shapiro and P. Smyth. *From Data Mining to Knowledge Discovery: An Overview*. In AKDDM, AAAI/MIT Press, 1996.

[70] U. Fayyad and K. Irani. *Multi-interval discretization of continuous-valued attributes for classification learning*. Proceedings of the 13th International Joint Conference on Artificial Intelligence, San Francisco, CA, Morgan Kaufmann (1993) 1022-1027.

[71] V. Mascardi, D. Demergasso and D. Ancona. *Languages for Programming BDI-style Agents: an Overview*. In F. Corradini, F. De Paoli, E. Merelli, and A. Omicini, editors, WOA 2005 - Workshop From Objects to Agents, pages 9-15, 2005.

[72] X. Zeng and T. Martinez. *Distribution-balanced stratified cross-validation for accuracy estimation*. Journal of Experimental & Theoretical Artificial Intelligence, Volume 12, Number 1, January 2000.

[73] Y. Shoham. *Agent-oriented programming*. Artificial Intelligence 60(1), pages 51-92, 1993.

Apéndice A

Código del MAS

En esta sección se presenta el código del MAS que se desarrolló como parte de este trabajo de tesis. Lo que se muestra en seguida es el archivo de configuración del MAS, donde la infraestructura de éste se definió como `Centralised`, es decir, el MAS se ejecuta en una sola máquina. Aquí también vemos que los agentes que conforman el MAS son: `coordinador`, `preprocesador`, `id3`, `c45`, `nb` y `tan`.

```
/* Jason Project */

MAS metaclasif {
    infrastructure: Centralised
    agents: coordinador;
            preprocesador;
            id3;
            c45;
            nb;
            tan;
}
```

El código del agente `coordinador` es el siguiente:

```
// coordinador in project metaclasif.mas2j

//CREENCIAS
inicio. //Inicia la ejecución del MAS
archivo("./BASES_DE_DATOS/Iris.csv").

//PLANES
@pi
+inicio: true
   <- .print("Agente Coordinador, encargado de dirigir el MAS...");
      ?archivo(PathBD);
```

```
      weka.verificaFormatoBD(PathBD,Formato);
      .print("BD proporcionada en formato ",Formato);
      !pVerificaFormatoBD(PathBD,Formato).

//Planes que toman acciones dependiendo de qué tipo es la base de datos
@pVFBD1
+!pVerificaFormatoBD(PathBD,Formato): not (Formato == ".xls" | Formato == ".csv" |
                                      Formato == ".arff")
   <- .print("BD debe ser proporcionada en formato .xls, .csv o .arff").

@pVFBD2
+!pVerificaFormatoBD(PathBD,Formato): Formato == ".xls" | Formato == ".csv" |
                                      Formato == ".arff"
   <- !pConvierteBD(PathBD,Formato);
      ?archivo(PathBDArff);
      .send(preprocesador,achieve,verificaClase(PathBDArff));
      .wait("+tipoClase(TipoClase)");
      !imprimeTClase.

//Planes que se encargan de convertir la base de datos al formato .arff
@pCBD1
+!pConvierteBD(PathBD,Formato): Formato == ".arff"
   <- true.

@pCBD2
+!pConvierteBD(PathBD,Formato): Formato == ".xls" | Formato == ".csv"
   <- .print("Convirtiendo base de datos al formato .arff...");
      weka.convierteBD(PathBD,Formato,[PathBDArff|BDArff]);
      -archivo(PathBD);
      +archivo(PathBDArff).

//Planes que entre otras cosas imprimen de qué tipo es la clase de la base de datos
@piTC1
+!imprimeTClase: tipoClase(TipoClase) & TipoClase == 0
   <- .print("Clase numérica no soportada por los agentes aprendices...").

@piTC2
+!imprimeTClase: tipoClase(TipoClase) & TipoClase == 1
   <- .print("La clase es de tipo nominal...");
      ?archivo(PathBDArff);
      .send(preprocesador,achieve,preprocesaBD(PathBDArff));
      -archivo(PathBDArff);
```

```
      .wait(10000);
      !construyeModelos.

//Plan que pide a los agentes aprendices que construyan sus modelos
@pcM
+!construyeModelos: archivoDS(PathBDDS) & archivoDNS(PathBDDNS)
   <- .send(id3, achieve, aprendeModelo(PathBDDS,PathBDDNS));
      .send(nb, achieve, aprendeModelo(PathBDDS,PathBDDNS));
      .send(c45, achieve, aprendeModelo(PathBDDS,PathBDDNS));
      .send(tan, achieve, aprendeModelo(PathBDDS,PathBDDNS));
      .wait(50000);
      !ganador.

//Planes que seleccionan el modelo ganador
@pg1
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PId3DS == PNbDS & PNbDS == PC45DS & PC45DS == PTANDS & PId3DNS
           == PNbDNS & PNbDNS == PC45DNS & PC45DNS == PTANDNS & PId3DN == PId3DNS
   <- .print("Hubo un empate; los modelos ID3, NB, C4.5 y TAN generados con datos
      discretizados supervisada y no supervisadamente obtuvieron ", PId3DS).

@pg2
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PId3DS > PNbDS & PId3DS > PC45DS & PId3DS > PTANDS & PId3DS >
           PId3DNS & PId3DS > PNbDNS & PId3DS > PC45DNS & PId3DS > PTANDNS
   <- .print("El ganador es ID3 generado con datos discretizados supervisadamente
      (DS), con ", PId3DS, "; mientras NB (DS) obtuvo ", PNbDS, ", C4.5 (DS) ",
      PC45DS, ", TAN (DS) ", PTANDS, ", ID3 con datos discretizados no supervisa-
      damente (DNS) ", PId3DNS, ", NB (DNS) ", PNbDNS, ", C4.5 (DNS) ", PC45DNS, "
      y TAN (DNS) ", PTANDNS).

@pg3
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PNbDS > PId3DS & PNbDS > PC45DS & PNbDS > PTANDS & PNbDS >
           PId3DNS & PNbDS > PNbDNS & PNbDS > PC45DNS & PNbDS > PTANDNS
   <- .print("El ganador es NB generado con datos discretizados supervisadamente
      (DS), con ", PNbDS, "; mientras ID3 (DS) obtuvo ", PId3DS, ", C4.5 (DS) ",
      PC45DS, ", TAN (DS) ", PTANDS, ", ID3 con datos discretizados no supervisa-
      damente (DNS) ", PId3DNS, ", NB (DNS) ", PNbDNS, ", C4.5 (DNS) ", PC45DNS, "
```

```
      y TAN (DNS) ", PTANDNS).

@pg4
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PC45DS > PId3DS & PC45DS > PNbDS & PC45DS > PTANDS & PC45DS >
           PId3DNS & PC45DS > PNbDNS & PC45DS > PC45DNS & PC45DS > PTANDNS
   <- .print("El ganador es C4.5 generado con datos discretizados supervisadamente
      (DS), con ", PC45DS, "; mientras ID3 (DS) obtuvo ", PId3DS, ", NB (DS) ",
      PNbDS, ", TAN (DS) ", PTANDS, ", ID3 con datos discretizados no supervisa-
      damente (DNS) ", PId3DNS, ", NB (DNS) ", PNbDNS, ", C4.5 (DNS) ", PC45DNS, "
      y TAN (DNS) ", PTANDNS).

@pg5
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PTANDS > PId3DS & PTANDS > PNbDS & PTANDS > PC45DS & PTANDS >
           PId3DNS & PTANDS > PNbDNS & PTANDS > PC45DNS & PTANDS > PTANDNS
   <- .print("El ganador es TAN generado con datos discretizados supervisadamente
      (DS), con ", PTANDS, "; mientras ID3 (DS) obtuvo ", PId3DS, ", NB (DS) ",
      PNbDS, ", C4.5 (DS) ", PC45DS, ", ID3 con datos discretizados no supervisa-
      damente (DNS) ", PId3DNS, ", NB (DNS) ", PNbDNS, ", C4.5 (DNS) ", PC45DNS, "
      y TAN (DNS) ", PTANDNS).

@pg6
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PId3DNS > PId3DS & PId3DNS > PNbDS & PId3DNS > PC45DS & PId3DNS >
           PTANDS & PId3DNS > PNbDNS & PId3DNS > PC45DNS & PId3DNS > PTANDNS
   <- .print("El ganador es ID3 generado con datos discretizados no supervisadamente
      (DNS), con ", PId3DNS, "; mientras NB (DNS) obtuvo ", PNbDNS, ", C4.5 (DNS) ",
      PC45DNS, ", TAN (DNS) ", PTANDNS, ", ID3 con datos discretizados supervisada-
      mente (DS) ", PId3DS, ", NB (DS) ", PNbDS, ", C4.5 (DS) ", PC45DS, " y TAN (DS)
      ", PTANDS).

@pg7
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PNbDNS > PId3DS & PNbDNS > PNbDS & PNbDNS > PC45DS & PNbDNS >
           PTANDS & PNbDNS > PId3DNS & PNbDNS > PC45DNS & PNbDNS > PTANDNS
   <- .print("El ganador es NB generado con datos discretizados no supervisadamente
      (DNS), con ", PNbDNS, "; mientras ID3 (DNS) obtuvo ", PId3DNS, ", C4.5 (DNS) ",
```

```
      PC45DNS, ", TAN (DNS) ", PTANDNS, ", ID3 con datos discretizados supervisada-
      mente (DS) ", PId3DS, ", NB (DS) ", PNbDS, ", C4.5 (DS) ", PC45DS, " y TAN (DS)
      ", PTANDS).

@pg8
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PC45DNS > PId3DS & PC45DNS > PNbDS & PC45DNS > PC45DS & PC45DNS
           > PTANDS & PC45DNS > PId3DNS & PC45DNS > PNbDNS & PC45DNS > PTANDNS
   <- .print("El ganador es C4.5 generado con datos discretizados no supervisadamente
      (DNS), con ", PC45DNS, "; mientras ID3 (DNS) obtuvo ", PId3DNS, ", NB (DNS) ",
      PNbDNS, ", TAN (DNS) ", PTANDNS, ", ID3 con datos discretizados supervisada-
      mente (DS) ", PId3DS, ", NB (DS) ", PNbDS, ", C4.5 (DS) ", PC45DS, " y TAN (DS)
      ", PTANDS).

@pg9
+!ganador: id3DS(PId3DS) & nbDS(PNbDS) & c45DS(PC45DS) & id3DNS(PId3DNS) &
           nbDNS(PNbDNS) & c45DNS(PC45DNS) & tanDS(PTANDS) & tanDNS(PTANDNS)
           & PTANDNS > PId3DS & PTANDNS > PNbDS & PTANDNS > PC45DS & PTANDNS >
           PTANDS & PTANDNS > PId3DNS & PTANDNS > PNbDNS & PTANDNS > PC45DNS
  <- .print("El ganador es TAN generado con datos discretizados no supervisadamente
     (DNS), con ", PTANDNS, "; mientras ID3 (DNS) obtuvo ", PId3DNS, ", NB (DNS) ",
     PNbDNS, ", C4.5 (DNS) ", PC45DNS, ", ID3 con datos discretizados supervisada-
     mente (DS) ", PId3DS, ", NB (DS) ", PNbDS, ", C4.5 (DS) ", PC45DS, " y TAN (DS)
     ", PTANDS).
```

El agente `preprocesador` se implementó con los dos planes que se muestran a continuación:

```
// preprocesador in project metaclasif.mas2j

//Plan que verifica de qué tipo es la clase
@pvC
+!verificaClase(PathBDArff): true
   <- .print("Verificando la clase de la BD...");
      weka.verificaClase(PathBDArff,TipoClase);
      .send(coordinador,tell,tipoClase(TipoClase)).

//Plan que reemplaza los valores faltantes y discretiza la base de datos
@ppBD
+!preprocesaBD(PathBDArff): true
   <- .print("Reemplazando valores faltantes...");
```

```
      weka.reemplazaFaltantes(PathBDArff);
      .print("Discretizando BD...");
      weka.discretizaS(PathBDArff,PathBDDS);
      weka.discretizaNS(PathBDArff,PathBDDNS);
      .send(coordinador,tell,archivoDS(PathBDDS));
      .send(coordinador,tell,archivoDNS(PathBDDNS)).
```

El código del agente `id3` es:

```
// id3 in project metaclasif.mas2j

//Plan que aprende ID3 para datos discretizados supervisada y no supervisadamente
@paMID3[atomic]
+!aprendeModelo(PathBDDS,PathBDDNS): true
   <- .print("Aprendiendo modelos ID3...");
      weka.construyeID3(PathBDDS,[PId3DS|MDS]);
      weka.construyeID3(PathBDDNS,[PId3DNS|MDNS]);
      !imprimeTEyMDS(MDS);
      !imprimeTEyMDNS(MDNS);
      .send(coordinador, tell, id3DS(PId3DS));
      .send(coordinador, tell, id3DNS(PId3DNS)).

//Planes que imprimen el tiempo que se tomó en la construcción de los modelos ID3
@piTEyMDS
+!imprimeTEyMDS([TE|MDS1]): true
   <- .print("Tiempo tomado en la construcción del modelo ID3 con datos discretiza-
      dos supervisadamente: ",TE).

@piTEyMDNS
+!imprimeTEyMDNS([TE|MDNS]): true
   <- .print("Tiempo tomado en la construcción del modelo ID3 con datos discretiza-
      dos no supervisadamente: ",TE).
```

El agente `c45` está dado por:

```
// Agent c45 in project metaclasif.mas2j

//Plan que aprende C4.5 para datos discretizados supervisada y no supervisadamente
@paMC45[atomic]
+!aprendeModelo(PathBDDS,PathBDDNS): true
   <- .print("Aprendiendo modelos C4.5...");
      weka.construyeC45(PathBDDS,[PC45DS|MDS]);
```

```
      weka.construyeC45(PathBDDNS,[PC45DNS|MDNS]);
      !imprimeTEyMDS(MDS);
      !imprimeTEyMDNS(MDNS);
      .send(coordinador, tell, c45DS(PC45DS));
      .send(coordinador, tell, c45DNS(PC45DNS)).

//Planes que imprimen el tiempo que se tomó en la construcción de los modelos C4.5
@piTEyMDS
+!imprimeTEyMDS([TE|MDS]): true
   <- .print("Tiempo tomado en la construcción del modelo C4.5 con datos discretiza-
      dos supervisadamente: ",TE).

@piTEyMDNS
+!imprimeTEyMDNS([TE|MDNS]): true
   <- .print("Tiempo tomado en la construcción del modelo C4.5 con datos discretiza-
      dos no supervisadamente: ",TE).
```

El código del agente `nb` es:

```
// nb in project metaclasif.mas2j

//Plan que aprende NB para datos discretizados supervisada y no supervisadamente
@paMNB[atomic]
+!aprendeModelo(PathBDDS,PathBDDNS): true
   <- .print("Aprendiendo modelos NB...");
      weka.construyeNB(PathBDDS,[PNbDS|MDS]);
      weka.construyeNB(PathBDDNS,[PNbDNS|MDNS]);
      !imprimeTEyMDS(MDS);
      !imprimeTEyMDNS(MDNS);
      .send(coordinador, tell, nbDS(PNbDS));
      .send(coordinador, tell, nbDNS(PNbDNS)).

//Planes que imprimen el tiempo que se tomó en la construcción de los modelos NB
@piTEyMDS
+!imprimeTEyMDS([TE|MDS]): true
   <- .print("Tiempo tomado en la construcción del modelo NB con datos discretiza-
      dos supervisadamente: ",TE).

@piTEyMDNS
+!imprimeTEyMDNS([TE|MDNS]): true
   <- .print("Tiempo tomado en la construcción del modelo NB con datos discretiza-
      dos no supervisadamente: ",TE).
```

El código que implementa el agente `tan` es:

```
// TAN in project metaclasif.mas2j

//Plan que aprende TAN para datos discretizados supervisada y no supervisadamente
@paMTAN[atomic]
+!aprendeModelo(PathBDDS,PathBDDNS): true
   <- .print("Aprendiendo modelos TAN...");
      weka.construyeTAN(PathBDDS,[PTANDS|MDS]);
      weka.construyeTAN(PathBDDNS,[PTANDNS|MDNS]);
      !imprimeTEyMDS(MDS);
      !imprimeTEyMDNS(MDNS);
      .send(coordinador, tell, tanDS(PTANDS));
      .send(coordinador, tell, tanDNS(PTANDNS)).

//Planes que imprimen el tiempo que se tomó en la construcción de los modelos TAN
@piTEyMDS
+!imprimeTEyMDS([TE|MDS]): true
   <- .print("Tiempo tomado en la construcción del modelo TAN con datos discretiza-
      dos supervisadamente: ",TE).

@piTEyMDNS
+!imprimeTEyMDNS([TE|MDNS]): true
   <- .print("Tiempo tomado en la construcción del modelo TAN con datos discretiza-
      dos no supervisadamente: ",TE).
```

A continuación se presenta el código que implementa las acciones internas, que ejecutan los agentes del MAS. La acción `verificaFormatoBD` es empleada por el agente `coordinador`, para conocer el formato de la base de datos. Por lo tanto, esta acción recibe el path a la base de datos, y devuelve el formato de ésta, por ejemplo: .csv.

```
1  // Internal action code for project metaclasif.mas2j
2
3  package weka;
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8
9  public class verificaFormatoBD extends DefaultInternalAction {
10
```

```
11      private Logger logger = Logger.getLogger("metaclasif.mas2j."+verificaFormatoBD.class.
12                              getName());
13
14      @Override
15      public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
16          StringTerm filename = (StringTerm)args[0];
17          String ext = filename.getString().substring(filename.getString().lastIndexOf('.'));
18          StringTermImpl formato = new StringTermImpl(ext);
19
20          return un.unifies(args[1],(Term)formato);
21      }
22  }
```

La acción `convierteBD` es llevada a cabo por el agente `coordinador`, con el fin de convertir la base de datos del formato .xls o .csv a .arff. Los parámetros de entrada de esta acción son: el path a la base de datos y el formato de ésta. Y su salida es: el path a la base de datos .arff y la base de datos.

```
1  // Internal action code for project Weka.mas2j
2
3  package weka;
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8  import java.io.File;
9  import weka.core.Instances;
10 import weka.core.converters.CSVLoader;
11 import weka.core.converters.ArffLoader;
12 import weka.core.converters.ArffSaver;
13 import weka.core.converters.SerializedInstancesLoader;
14
15 public class convierteBD extends DefaultInternalAction {
16
17      private Logger logger = Logger.getLogger("Weka.mas2j."+ convierteBD.class.getName());
18
19      @Override
20      public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
21          StringTerm filename = (StringTerm)args[0];
22          StringTerm ext = (StringTerm)args[1];
23          ListTerm lista = new ListTermImpl();
24          String nombreArch = filename.getString().substring(0,filename.getString().lastIndexOf('.'));
```

```
25          StringTerm stNombreArch = new StringTermImpl(nombreArch + ".arff");
26
27          lista.add(stNombreArch);
28          File fileArff = new File(nombreArch + ".arff");
29
30          if (!fileArff.exists())
31          {
32              try{
33                  if (ext.getString().equalsIgnoreCase(".csv"))
34                  {
35                      CSVLoader fileCSV = new CSVLoader();
36                      fileCSV.setSource(new File(filename.getString()));
37                      fileCSV.retrieveFile();
38                      Instances instances = fileCSV.getDataSet();
39                      ArffSaver archArff = new ArffSaver();
40                      archArff.setInstances(instances);
41                      archArff.setFile(fileArff);
42                      archArff.setDestination(fileArff);
43                      archArff.writeBatch();
44                      StringTerm inst = new StringTermImpl(instances.toString());
45                      lista.add(inst);
46                  }
47                  else
48                  {
49                      CSVLoader fileXLS = new CSVLoader();
50                      fileXLS.setSource(new File(filename.getString()));
51                      fileXLS.retrieveFile();
52                      Instances instances = fileXLS.getDataSet();
53                      StringTerm inst = new StringTermImpl(instances.toString());
54                      lista.add(inst);
55                  }
56              }
57              catch (Exception e) {System.err.println(e.getMessage());}
58          }
59          return un.unifies(args[2],(Term)lista);
60      }
61  }
```

La acción `verificaClase` es ejecutada por el agente `preprocesador`, para verificar de qué tipo es la clase de la base de datos. Recibe como parámetro de entrada el path a la base de datos .arff, y devuelve 0 o 1, dependiendo si la clase es numérica o nominal.

```java
// Internal action code for project metaclasif.mas2j

package weka;

import jason.asSemantics.*;
import jason.asSyntax.*;
import java.util.logging.*;
import java.io.File;
import weka.core.converters.ArffLoader;
import weka.core.Instances;
import weka.core.Attribute;

public class verificaClase extends DefaultInternalAction {

    private Logger logger = Logger.getLogger("metaclasif.mas2j."+verificaClase.class.getName());

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
        StringTerm filename = (StringTerm)args[0];
        ArffLoader fileArff = new ArffLoader();
        fileArff.setSource(new File(filename.getString()));
        fileArff.retrieveFile();
        Instances instances = fileArff.getDataSet();
        instances.setClassIndex(instances.numAttributes() - 1);
        Attribute clase = instances.classAttribute();

        //Obtiene el tipo de clase
        //0 Numeric
        //1 Nominal
        NumberTerm tipoClase = new NumberTermImpl(clase.type());

        return un.unifies(args[1],(Term)tipoClase);
    }
}
```

La acción `reemplazaFaltantes` es realizada por el agente `preprocesador`, para reemplazar los valores faltantes de la base de datos. Por lo tanto, esta acción recibe como entrada el path a la base de datos .arff.

```java
// Internal action code for project metaclasif.mas2j

package weka;
```

```
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8  import java.io.File;
9  import weka.core.converters.ArffLoader;
10 import weka.core.converters.ArffSaver;
11 import weka.core.Instances;
12 import weka.core.Instance;
13 import weka.filters.Filter;
14 import weka.filters.unsupervised.attribute.ReplaceMissingValues;
15
16 public class reemplazaFaltantes extends DefaultInternalAction {
17
18     private Logger logger = Logger.getLogger("metaclasif.mas2j."+reemplazaFaltantes.class.getName());
19
20     @Override
21     public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
22         StringTerm filename = (StringTerm)args[0];
23         ArffLoader fileArff = new ArffLoader();
24         fileArff.setSource(new File(filename.getString()));
25         fileArff.retrieveFile();
26         Instances instances = fileArff.getDataSet();
27
28         //Reemplaza valores faltantes
29         Filter filter = new ReplaceMissingValues();
30         filter.setInputFormat(instances);
31         for (int i = 0; i < instances.numInstances(); i++) {
32             filter.input(instances.instance(i));
33         }
34         filter.batchFinished();
35         Instances newData = filter.getOutputFormat();
36         Instance processed;
37         while ((processed = filter.output()) != null) {
38             newData.add(processed);
39         }
40
41         //Guarda instancias sin faltantes
42         ArffSaver archArff = new ArffSaver();
43         archArff.setInstances(newData);
44         archArff.setFile(new File(filename.getString()));
45         archArff.setDestination(new File(filename.getString()));
```

```
46          archArff.writeBatch();
47
48          return true;
49      }
50  }
```

La acción `discretizaS` es llevada a cabo por el agente `preprocesador`, para discretizar de forma supervisada los atributos numéricos de la base de datos. Este tipo de discretización toma en cuenta los valores del atributo clase para generar las diferentes particiones del atributo que se discretizará. Como podemos ver en el código que se presenta en seguida, el parámetro de entrada de esta acción es el path a la base de datos .arff, y la salida es el path a la base de datos discretizada.

```
1  // Internal action code for project metaclasif.mas2j
2
3  package weka;
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8  import java.io.*;
9  import weka.core.converters.ArffLoader;
10 import weka.core.converters.ArffSaver;
11 import weka.core.Instances;
12 import weka.core.Instance;
13 import weka.filters.Filter;
14 import weka.filters.supervised.attribute.Discretize;
15
16 public class discretizaS extends DefaultInternalAction {
17
18     private Logger logger = Logger.getLogger("metaclasif.mas2j."+discretizaS.class.getName());
19
20     @Override
21     public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
22         StringTerm filename = (StringTerm)args[0];
23         String nombreArch = filename.getString().substring(0,filename.getString().lastIndexOf('.'));
24         File fileDS = new File(nombreArch + "DS.arff");
25
26         if (!fileDS.exists())
27         {
28             try{
```

```
29                  File fichero = new File(filename.getString());
30                  ArffLoader file = new ArffLoader();
31                  file.setSource(fichero);
32                  file.retrieveFile();
33                  Instances instances = file.getDataSet();
34                  instances.setClassIndex(instances.numAttributes() - 1);
35
36                  //Discretiza BD
37                  Filter filter = new Discretize();
38                  filter.setInputFormat(instances);
39                  for (int i = 0; i < instances.numInstances(); i++) {
40                      filter.input(instances.instance(i));
41                  }
42                  filter.batchFinished();
43                  Instances newData = filter.getOutputFormat();
44                  Instance processed;
45                  while ((processed = filter.output()) != null) {
46                      newData.add(processed);
47                  }
48
49                  //Guarda instancias discretizadas
50                  ArffSaver archArff = new ArffSaver();
51                  archArff.setInstances(newData);
52                  archArff.setFile(new File(nombreArch + "DS.arff"));
53                  archArff.setDestination(new File(nombreArch + "DS.arff"));
54                  archArff.writeBatch();
55              }catch (Exception e) {System.err.println(e.getMessage());}
56          }
57          StringTerm pathBDDS = new StringTermImpl(nombreArch + "DS.arff");
58
59          return un.unifies(args[1],(Term)pathBDDS);
60      }
61  }
```

La acción `discretizaNS` también es ejecutada por el agente `preprocesador`, con el fin de discretizar de manera no supervisada los atributos numéricos de la base de datos. La discretización no supervisada no toma en cuenta los valores del atributo clase para generar las diferentes particiones del atributo que se discretizará. Los parámetros de entrada y de salida de esta acción son: el path a la base de datos .arff y el path a la base de datos discretizada, respectivamente.

```
1 // Internal action code for project metaclasif.mas2j
2
3 package weka;
4
5 import jason.asSemantics.*;
6 import jason.asSyntax.*;
7 import java.util.logging.*;
8 import java.io.*;
9 import weka.core.converters.ArffLoader;
10 import weka.core.converters.ArffSaver;
11 import weka.core.Instances;
12 import weka.core.Instance;
13 import weka.filters.Filter;
14 import weka.filters.unsupervised.attribute.Discretize;
15
16 public class discretizaNS extends DefaultInternalAction {
17
18     private Logger logger = Logger.getLogger("metaclasif.mas2j."+discretizaNS.class.getName());
19
20     @Override
21     public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
22         StringTerm filename = (StringTerm)args[0];
23         String nombreArch = filename.getString().substring(0,filename.getString().lastIndexOf('.'));
24         File fileDNS = new File(nombreArch + "DNS.arff");
25
26         if (!fileDNS.exists())
27         {
28             try{
29                 File fichero = new File(filename.getString());
30                 ArffLoader file = new ArffLoader();
31                 file.setSource(fichero);
32                 file.retrieveFile();
33                 Instances instances = file.getDataSet();
34                 instances.setClassIndex(instances.numAttributes() - 1);
35
36                 //Discretiza BD
37                 Filter filter = new Discretize();
38                 filter.setInputFormat(instances);
39                 for (int i = 0; i < instances.numInstances(); i++) {
40                     filter.input(instances.instance(i));
41                 }
42                 filter.batchFinished();
```

```
43                  Instances newData = filter.getOutputFormat();
44                  Instance processed;
45                  while ((processed = filter.output()) != null) {
46                      newData.add(processed);
47                  }
48
49                  //Guarda instancias discretizadas
50                  ArffSaver archArff = new ArffSaver();
51                  archArff.setInstances(newData);
52                  archArff.setFile(new File(nombreArch + "DNS.arff"));
53                  archArff.setDestination(new File(nombreArch + "DNS.arff"));
54                  archArff.writeBatch();
55              }catch (Exception e) {System.err.println(e.getMessage());}
56          }
57          StringTerm pathBDDNS = new StringTermImpl(nombreArch + "DNS.arff");
58
59          return un.unifies(args[1],(Term)pathBDDNS);
60      }
61  }
```

La acción `construyeID3` es llevada a cabo por el agente `id3`, para aprender un árbol de decisión con el algoritmo ID3. Por lo tanto, esta acción recibe como entrada el path a la base de datos, y como salida regresa el porcentaje de ejemplos clasificados correctamente por el modelo aprendido (árbol de decisión), el tiempo que se tomó en la construcción de este modelo y el modelo mismo.

```
1  // Internal action code for project Weka.mas2j
2
3  package weka;
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8  import java.io.BufferedReader;
9  import java.io.FileReader;
10 import weka.core.Instances;
11 import weka.classifiers.trees.Id3;
12 import weka.classifiers.Evaluation;
13
14 public class construyeID3 extends DefaultInternalAction {
15
```

```
16      private Logger logger = Logger.getLogger("Weka.mas2j."+construyeID3.class.getName());
17
18      @Override
19      public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
20
21          try{
22              Stopwatch stopwatch = new Stopwatch();//clase para medir tiempo de ejecución
23              StringTerm filename = (StringTerm)args[0];
24              Instances data = new Instances(new BufferedReader(new FileReader(filename.getString())));
25
26              //indica cuál es la clase
27              data.setClassIndex(data.numAttributes() - 1);
28
29              //construye el clasificador
30              Id3 tree = new Id3();
31              stopwatch.start(); //inicio del tiempo de ejecución
32              tree.buildClassifier(data);
33              stopwatch.stop();//parada del tiempo de ejecución
34
35              //evalúa el clasificador
36              Evaluation eval = new Evaluation(data);
37              int folds = 10;
38              eval.crossValidateModel(tree, data, folds, data.getRandomNumberGenerator(1));
39
40              //regresa porcentaje de ejemplos correctamente clasificados, tiempo de ejecución y el
41              //clasificador
42              ListTerm ide = new ListTermImpl();
43              StringTerm pctC = new StringTermImpl(Double.toString(eval.pctCorrect()));
44              ide.add(pctC);
45              StringTerm tiempoE = new StringTermImpl(stopwatch.toString());
46              ide.add(tiempoE);
47              StringTerm arbol = new StringTermImpl(tree.toString());
48              ide.add(arbol);
49
50              return un.unifies(args[1],(Term)ide);
51          }catch (Exception e) {System.err.println(e.getMessage());}
52
53          return true;
54      }
55  }
```

La acción `construyeC45` es ejecutada por el agente `c45`, para aprender un árbol de decisión con el algoritmo C4.5. Esta acción recibe como entrada el path a la base de datos, y como salida regresa el porcentaje de ejemplos clasificados correctamente por el modelo aprendido (árbol de decisión), el tiempo que se tomó en la construcción de este modelo y el modelo mismo.

```
1  // Internal action code for project Weka.mas2j
2
3  package weka;
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8  import java.io.BufferedReader;
9  import java.io.FileReader;
10 import weka.core.Instances;
11 import weka.classifiers.trees.J48;
12 import weka.classifiers.Evaluation;
13
14 public class construyeC45 extends DefaultInternalAction {
15
16     private Logger logger = Logger.getLogger("Weka.mas2j."+construyeC45.class.getName());
17
18     @Override
19     public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
20
21         try{
22             Stopwatch stopwatch = new Stopwatch();//clase para medir tiempo de ejecución
23             StringTerm filename = (StringTerm)args[0];
24             Instances data = new Instances(new BufferedReader(new FileReader(filename.getString())));
25
26             //indica cuál es la clase
27             data.setClassIndex(data.numAttributes() - 1);
28
29             //construye el clasificador
30             String[] options = new String[2];
31             options[0] = "-C 0.25";//factor de confianza utilizado en la poda
32             options[1] = "-M 2";//número mínimo de instancias por hoja
33             J48 clasif = new J48();
34             clasif.setOptions(options);
35             stopwatch.start();//inicio del tiempo de ejecución
```

```
36              clasif.buildClassifier(data);
37              stopwatch.stop();//parada del tiempo de ejecución
38
39              //evalúa el clasificador
40              Evaluation eval = new Evaluation(data);
41              int folds = 10;
42              eval.crossValidateModel(clasif, data, folds, data.getRandomNumberGenerator(1));
43
44              //regresa porcentaje de ejemplos correctamente clasificados, tiempo de ejecución y el
45              //clasificador
46              ListTerm c45 = new ListTermImpl();
47              StringTerm pctC = new StringTermImpl(Double.toString(eval.pctCorrect()));
48              c45.add(pctC);
49              StringTerm tiempoE = new StringTermImpl(stopwatch.toString());
50              c45.add(tiempoE);
51              StringTerm arbol = new StringTermImpl(clasif.toString());
52              c45.add(arbol);
53
54              return un.unifies(args[1],(Term)c45);
55          }catch (Exception e) {System.err.println(e.getMessage());}
56
57          return true;
58      }
59  }
```

La acción `construyeNB` es realizada por el agente `nb`, para aprender el clasificador NB. construyeNB recibe como entrada el path a la base de datos, y como salida regresa el porcentaje de ejemplos clasificados correctamente por el modelo NB, el tiempo que se tomó en la construcción de este modelo y el modelo mismo.

```
1  // Internal action code for project Weka.mas2j
2
3  package weka;
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8  import java.io.BufferedReader;
9  import java.io.FileReader;
10 import weka.core.Instances;
11 import weka.classifiers.bayes.NaiveBayes;
12 import weka.classifiers.Evaluation;
```

```
13
14 public class construyeNB extends DefaultInternalAction {
15
16     private Logger logger = Logger.getLogger("Weka.mas2j."+construyeNB.class.getName());
17
18     @Override
19     public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
20
21         try{
22             Stopwatch stopwatch = new Stopwatch();//clase para medir tiempo de ejecución
23             StringTerm filename = (StringTerm)args[0];
24             Instances data = new Instances(new BufferedReader(new FileReader(filename.getString())));
25
26             //indica cuál es la clase
27             data.setClassIndex(data.numAttributes() - 1);
28
29             //construye el clasificador
30             String[] options = new String[1];
31             options[0] = "-K";/*usa un estimador de kernel para atributos numéricos en lugar de una
32                                 distribución normal.*/
33             NaiveBayes clasif = new NaiveBayes();
34             clasif.setOptions(options);
35             stopwatch.start(); //inicio del tiempo de ejecución
36             clasif.buildClassifier(data);
37             stopwatch.stop();//parada del tiempo de ejecución
38
39             //evalúa el clasificador
40             Evaluation eval = new Evaluation(data);
41             int folds = 10;
42             eval.crossValidateModel(clasif, data, folds, data.getRandomNumberGenerator(1));
43
44             //regresa porcentaje de ejemplos correctamente clasificados, tiempo de ejecución y el
45             //clasificador
46             ListTerm nb = new ListTermImpl();
47             StringTerm pctC = new StringTermImpl(Double.toString(eval.pctCorrect()));
48             nb.add(pctC);
49             StringTerm tiempoE = new StringTermImpl(stopwatch.toString());
50             nb.add(tiempoE);
51             StringTerm redb = new StringTermImpl(clasif.toString());
52             nb.add(redb);
53
54             return un.unifies(args[1],(Term)nb);
```

```
55          }catch (Exception e) {System.err.println(e.getMessage());}
56
57          return true;
58      }
59  }
```

La acción `construyeTAN` es llevada a cabo por el agente `tan`, para aprender un clasificador Bayesiano con el algoritmo TAN. construyeTAN recibe como entrada el path a la base de datos, y como salida devuelve el porcentaje de ejemplos clasificados correctamente por el clasificador Bayesiano aprendido, el tiempo que se tomó en la construcción de este clasificador y el clasificador mismo.

```
1  // Internal action code for project Weka.mas2j
2
3  package weka;
4
5  import jason.asSemantics.*;
6  import jason.asSyntax.*;
7  import java.util.logging.*;
8  import java.io.BufferedReader;
9  import java.io.FileReader;
10 import weka.core.Instances;
11 import weka.classifiers.Classifier;
12 import weka.classifiers.Evaluation;
13
14 public class construyeTAN extends DefaultInternalAction {
15
16     private Logger logger = Logger.getLogger("Weka.mas2j."+construyeTAN.class.getName());
17
18     @Override
19     public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
20
21         try{
22             Stopwatch stopwatch = new Stopwatch();//clase para medir tiempo de ejecución
23             StringTerm filename = (StringTerm)args[0];
24             Instances data = new Instances(new BufferedReader(new FileReader(filename.getString())));
25
26             //indica cuál es la clase
27             data.setClassIndex(data.numAttributes() - 1);
28
29             //construye el clasificador
```

```
30              String[] opcionesClasificador = new
31              String[]{"-D","-Q","weka.classifiers.bayes.net.search.local.TAN","--","-S","MDL","-E","weka
32              .classifiers.bayes.net.estimate.SimpleEstimator","--","-A","0.5"};
33              Classifier clasif =
34              Classifier.forName("weka.classifiers.bayes.BayesNet",opcionesClasificador);
35              stopwatch.start();//inicio del tiempo de ejecución
36              clasif.buildClassifier(data);
37              stopwatch.stop();//parada del tiempo de ejecución
38
39              //evalúa el clasificador
40              Evaluation eval = new Evaluation(data);
41              int folds = 10;
42              eval.crossValidateModel(clasif, data, folds, data.getRandomNumberGenerator(1));
43
44              //regresa porcentaje de ejemplos correctamente clasificados, tiempo de ejecución y el
45              //clasificador
46              ListTerm tan = new ListTermImpl();
47              StringTerm pctC = new StringTermImpl(Double.toString(eval.pctCorrect()));
48              tan.add(pctC);
49              StringTerm tiempoE = new StringTermImpl(stopwatch.toString());
50              tan.add(tiempoE);
51              StringTerm arbol = new StringTermImpl(clasif.toString());
52              tan.add(arbol);
53
54              return un.unifies(args[1],(Term)tan);
55          }catch (Exception e) {System.err.println(e.getMessage());}
56
57          return true;
58      }
59  }
```