Extensiones al Ambiente de Aprendizaje Automatizado WEKA

Autores: Héctor Matías González
Liana Isabel Araujo Pérez

Tutores: Dr. Carlos Morell Pérez
MSc. Yanet Rodríguez Sarabia

Curso 2005 - 2006

Dedicatoria

A quien merece más que yo la satisfacción de este trabajo,
A quien ha hecho posible este y muchos otros sueños,
A quien le debo más que un título,
A quien le debo mi vida,
Es para ti,
mamá

Quiero que vaya un pedacito de lo mejor del trabajo a toda esa gente que se que sintieron esta tesis en especial a mis familiares mas allegados.

Héctor

Dedicatoria

A mis padres por brindarme su amor sin límites, mostrándome siempre el camino correcto

A mis abuelos por su cariño y protección, en especial a mi abuelito Mayito, por ser la estrella que me ilumina día tras día

A mi tía por ser mi segunda mamá, mi guía en todo momento, por su ternura

A mi hermanito, por completar mi felicidad.

Liana Isabel

Agradecimientos

Por sobre todo a nuestros tutores Yanet y Morell por sentir esta tesis como si fuera la de ellos
A nuestros padres por estar siempre en el lugar correcto
A nuestros tíos por su cariño
A nuestros amigos por brindarnos tantos momentos gratos

de Hector:
A alguien muy especial que se ha sacrificado casi como nosotros en la etapa final a quien es mas que un primo, a socio

de Liana:
A mis abuelos, por consentirme tanto, a Argelio y William por tanto apoyo

El secreto de la sabiduría, del poder y del conocimiento es la humildad

Ernest Hemingway

RESUMEN

Este trabajo presenta extensiones realizadas a la herramienta de Aprendizaje Automatizado Weka. Se incorporan los algoritmos que utiliza un nuevo modelo híbrido de Razonamiento Basado en Casos. Específicamente se añade un nuevo tipo de dato para manejar un atributo numérico como variable lingüística y un nuevo filtro con esta finalidad, y se implementan dos algoritmos de clasificación. El diseño de Weka hace que la incorporación de nuevos modelos no sea una tarea tan compleja, sin embargo en el trabajo queda definida una metodología para adicionar nuevos filtros y clasificadores que facilita aún más esta tarea. Finalmente, se validan los algoritmos implementados utilizando conjuntos de datos internacionales, mostrando la factibilidad de utilizar esta herramienta y los nuevos algoritmos en el campo de la Inteligencia Artificial.

ABSTRACT

This work presents the extensions made to the “Weka” Machine Learning tool in order to incorporate the algorithms used in a new Case-Based Reasoning hybrid model. Specifically, a new data type is added to handle a numerical attribute as a linguistic variable. A new filter and two classification algorithms are purposefully implemented by using Weka. Although its environment was designed with the goal of reducing the complexity when adding new models, additionally a methodology for incorporating new filters and classification algorithms is outlined. Finally, the implemented algorithms are validated by means of well-known international datasets, therefore concluding the feasibility of using this tool and the new algorithms in the Artificial intelligence field.

INTRODUCCIÓN 1

CAPÍTULO 1. EXTENDIENDO WEKA, UNA NECESIDAD DEL GRUPO DE INTELIGENCIA ARTIFICIAL 4

1.1 Sobre el Aprendizaje Automatizado 6

1.2 Familiarización con el ambiente de Aprendizaje Automatizado Weka 6
1.2.1 Entrada de datos 8
1.2.2 Algoritmos para el Aprendizaje Automatizado 10
1.2.3 Validación con un conjunto de datos 13
1.2.4 Interioridades de Weka 14

1.3 Dos algoritmos de aprendizaje basados en similaridad 18
1.3.1 Modelación de los atributos numéricos 20
1.3.2 El modelo Fuzzy-SIAC 22
1.3.3 El modelo ConFuCiuS 24

CAPÍTULO 2. UNA METODOLOGÍA PARA EXTENDER WEKA CON NUEVOS FILTROS Y CLASIFICADORES 26

2.1 Incluyendo un nuevo filtro a Weka 27
2.1.1 Implementando un nuevo filtro 28
2.1.2 Utilizando un filtro 34
2.1.3 Ejemplo con un discretizador 34

2.2 Incluyendo un nuevo clasificador a Weka 39
2.2.1 Implementando un nuevo clasificador 39
2.2.2 Utilizando un clasificador 42
2.2.3 Ejemplo con una red neuronal simple 43

CAPÍTULO 3. IMPLEMENTACIÓN Y VALIDACIÓN DE LOS NUEVOS ALGORITMOS UTILIZANDO WEKA 49

3.1 Conjuntos borrosos en Weka 50

3.2 Implementación de un filtro para trabajar con atributos borrosos 52

3.3 Usando la metodología propuesta para añadir los nuevos modelos 53
3.3.1 Fuzzy-SIAC 55
3.3.2 ConFuCiuS 57

3.4 Validación de los algoritmos implementados 61
3.4.1 Validación de Fuzzy-SIAC 62
3.4.2 Validación de ConFuCiuS 65

3.5 Desarrollando aplicaciones a la medida del usuario 70

3.6 Factibilidad de la implementación de nuevos modelos en Weka 71

CONCLUSIONES 74

RECOMENDACIONES 77

REFERENCIAS BIBLIOGRAFICAS 78

ANEXOS 82

INTRODUCCIÓN

Desde los inicios de la computación se han tratado de ingeniar métodos con el objetivo de lograr que la computadora aprenda por sí misma. El interés es lograr programarla para que mejore automáticamente con su propia experiencia. En particular, el campo de la Inteligencia Artificial (IA) que estudia los métodos de solución de problemas de aprendizaje por las computadoras se le denomina Aprendizaje Automatizado (Machine Learning). Aún no se ha podido lograr que las computadoras aprendan casi tan bien como el hombre aprende, sin embargo se han creado algoritmos que son eficaces para ciertos tipos de tareas de este ámbito, emergiendo así una comprensión teórica de aprendizaje.

El Aprendizaje Automatizado se ha convertido en un campo multidisciplinario, mostrando resultados de inteligencia artificial, probabilidades y estadística, filosofía, teoría de cómputo, teoría de control, teoría de información, psicología, neurobiología, ciencia cognoscitiva, complejidad computacional y otros campos. La creación de diversos algoritmos de aprendizaje automatizado ha traído consigo la difícil tarea de seleccionar alguno de ellos a la hora que se requiera su utilización. La herramienta WEKA (Waikato Environment for Knowledge Analysis) es un ambiente de trabajo para la prueba y validación de algoritmos de la IA, que ha tenido gran difusión entre los investigadores y docentes de esta área. Weka tiene implementado un gran número de algoritmos conocidos, varias formas para preprocesar los archivos de datos a utilizar por tales algoritmos; así como facilidades para validar los mismos.

El grupo de IA de nuestra Universidad trabaja en el desarrollo de nuevos modelos en esta área. Es importante la validación de estos modelos con diferentes tipos de conjuntos de datos, así como la comparación de los resultados de su desempeño con otros modelos similares ya existentes reportados en la bibliografía, utilizando estos juegos de datos internacionalmente reconocidos. En particular, resulta de interés la validación de dos extensiones al modelo híbrido de Razonamiento Basado en Casos propuesto en (García y Bello, 1996) e implementado en el SISI (Sistema Inteligente de Selección de Información). Los nuevos modelos manejan los rasgos numéricos como conjuntos borrosos, y utilizan los

enfoques conexionista y basado en casos; permitiendo el desarrollo de aplicaciones que aprovechan las ventajas de la computación blanda o softcomputing (Pal et al., 2001). Una de estas extensiones al modelo original, de manera similar a este, utiliza como resolvedor de problemas una red neuronal artificial (RNA) de tipo asociativa. La otra variante emplea el razonamiento basado en casos (RBC) como resolvedor de problemas, en lugar de la RNA, pero utilizando la información almacenada en los pesos de la RNA para definir el criterio de similitud a emplear.

Una de las características más interesantes de Weka es la posibilidad de modificar su código y obtener versiones adaptadas con otras funcionalidades ampliando así sus posibilidades de uso. Al ser una herramienta orientada a la extensibilidad posee una estructura factible para ello. Atendiendo a lo anteriormente explicado el presente trabajo tiene como:

Objetivo general

Extender las funcionalidades del Ambiente de Aprendizaje Automatizado Weka a través de la implementación de dos algoritmos de clasificación que se utilizan en la extensión de un modelo híbrido de Razonamiento Basado en Casos.

Objetivos específicos

- Asimilar la herramienta de Aprendizaje Automatizado Weka
- Incorporar a Weka un nuevo tipo de dato para manejar un atributo numérico como variable lingüística
- Proponer una metodología para adicionar nuevos filtros y clasificadores a Weka
- Implementar los algoritmos Fuzzy-SIAC, ConFuCiuS y el filtro EqualWidthFuzzyfier, aplicando la metodología propuesta
- Validar el desempeño de los algoritmos incluidos mostrando los resultados obtenidos con juegos de datos internacionalmente reconocidos
- Mostrar la factibilidad de desarrollar un software a la medida utilizando algoritmos ya implementados en la herramienta

El capítulo 1 aborda conceptos generales relacionados con el campo del Aprendizaje Automatizado, familiariza al lector con las principales opciones que brinda Weka, los diferentes tipos de algoritmos de aprendizaje que están ya implementados en la versión 3.5.2 de esta herramienta. Se profundiza en las interioridades de su implementación analizando su estructura. Finalmente se describen nuevas extensiones a un modelo híbrido existente que combina RNA y RBC, donde se utilizan los algoritmos de clasificación a implementar en esta herramienta como objetivo del presente trabajo. El capítulo 2 propone una metodología para la implementación de nuevos filtros y clasificadores en Weka, que constituye un resultado importante de este trabajo para realizar extensiones futuras a la herramienta. Además se incluye, a modo de ejemplo, un algoritmo de clasificación y uno de preprocesamiento de datos, siguiendo los pasos de la metodología. Por último, en el capítulo 3 se detallan cada una de las implementaciones realizadas con la finalidad de validar y utilizar los nuevos algoritmos utilizando esta herramienta.

CAPÍTULO 1. EXTENDIENDO WEKA, UNA NECESIDAD DEL GRUPO DE INTELIGENCIA ARTIFICIAL

En el año 1992 Ian Witten, profesor del Departamento de Ciencia de la Computación de la Universidad de Waikato, Nueva Zelanda, creó una aplicación que más tarde, en 1993, recibiría el nombre de Weka, creándose con ello una interfaz e infraestructura para la misma. Su nombre: Weka (*Gallirallus australis*) se debe a un ave endémica de este país, de aspecto pardo y tamaño similar a una gallina, se encuentra en peligro de extinción y es famosa por su curiosidad y agresividad.

En 1994 fue terminada la primera versión de Weka, aunque no publicada, montada en una interfaz de usuario con varios algoritmos de aprendizaje escritos en lenguaje C. Muchas versiones de Weka fueron construyéndose hasta que en octubre de 1996 es publicada la primera versión (2.1).

En julio de 1997 se publica la versión 2.2 con la inclusión de nuevos algoritmos de aprendizaje y con la facilidad de configurar la corrida de una gran escala de experimentos. Cerca de esta fecha es tomada la decisión de rescribir Weka en lenguaje Java. A mediados de 1999 es liberada la versión 3 de Weka con todo su código implementado en Java. Después de la versión 3 se han publicado varias versiones, cada una de ellas ofreciendo esencialmente como mejora la adición de nuevos algoritmos.

La aplicación comenzada por Ian Witten en 1992 aún continúa ampliándose. El desarrollo de mejores extensiones al sistema ha dejado de concentrarse en el lugar donde fue creado, para dispersarse por todo el mundo, donde cantidad de científicos incluyen y validan sus propios modelos.

Este ambiente de Aprendizaje Automatizado está contenido por una extensa colección de algoritmos de la IA, útiles para ser aplicados sobre datos mediante las interfaces gráficas de usuario (GUI: Graphical User Interface) que ofrece o para usarlos dentro de cualquier aplicación. Contiene las herramientas necesarias para realizar transformaciones sobre los datos, tareas de clasificación, regresión, agrupamiento, asociación y visualización.

1.1 Sobre el Aprendizaje Automatizado

El aprendizaje automatizado es la disciplina que estudia cómo construir sistemas computacionales que mejoren automáticamente mediante la experiencia. En otras palabras, se dice que un programa ha "aprendido" a desarrollar la tarea *T* si después de proporcionarle la experiencia *E* el sistema es capaz de desempeñarse razonablemente bien cuando nuevas situaciones de la tarea se presentan. Aquí *E* es generalmente un conjunto de ejemplos de *T*, y el desempeño es medido usando una métrica de calidad *P*. Por lo tanto, un problema de aprendizaje bien definido requiere que *T*, *E* y *P* estén bien especificados (Mitchell, 1997).

El Aprendizaje Automatizado pretende generar expresiones de clasificación lo suficientemente simples como para ser fácilmente entendidas por los humanos. Como en los métodos estadísticos, el conocimiento que ya se posea puede ser utilizado en el proceso, pero la generación de estas expresiones ocurre sin la intervención de los humanos. Tiene una amplia gama de aplicaciones, incluyendo motores de búsqueda, diagnósticos médicos, detección de fraude en el uso de tarjetas de crédito, análisis del mercado de valores, clasificación de secuencias de ADN, reconocimiento del habla y del lenguaje escrito, juegos y robótica.

A pesar que se desconoce cómo lograr que las computadoras aprendan tan bien como las personas, ciertos algoritmos propuestos en el campo han resultado efectivos en varias tareas de aprendizaje.

1.2 Familiarización con el ambiente de Aprendizaje Automatizado Weka

Weka es un sistema multiplataforma y de amplio uso probado bajo sistemas operativos Linux, Windows y Macintosh. Puede ser usado desde la perspectiva de usuario mediante las seis interfaces que brinda, a través de la línea de comando desde donde se pueden invocar cada uno de los algoritmos incluidos en la herramienta como programas individuales y mediante la creación de un programa Java que llame a las funciones que se desee.

Weka (versión 3.5.2) dispone de seis interfaces de usuario diferentes que pueden ser accedidas mediante la ventana de selección de interfaces (GUI Chooser), que constituye la interfaz de usuario gráfica (GUI: Grafic User Interface) como se muestra en la Figura 1:

- **Interfaz para línea de comando** (Simple CLI: Command Line Interface): permite invocar desde la línea de comandos cada uno de los algoritmos incluidos en Weka como programas individuales. Los resultados se muestran únicamente en modo texto. A pesar de ser en apariencia muy simple es extremadamente potente porque permite realizar cualquier operación soportada por Weka de forma directa; no obstante, es muy complicada de manejar ya que es necesario un conocimiento completo de la aplicación. Su utilidad es pequeña desde que se fue recubriendo Weka con interfaces. Actualmente ya prácticamente sólo es útil como una herramienta de ayuda a la fase de pruebas. Es muy beneficiosa principalmente para los sistemas operativos que no proporcionan su propia interfaz para línea de comandos.

- **Explorador** (Explorer): interfaz de usuario gráfica para acceder a los algoritmos implementados en la herramienta para realizar el aprendizaje automatizado. Es el modo más usado y descriptivo. Permite realizar operaciones sobre un sólo archivo de datos.

- **Experimentador** (Experimenter): facilita la realización de experimentos en lotes, incluso con diferentes algoritmos y varios conjuntos de datos a la vez.

- **Flujo de conocimiento** (KnowledgeFlow): proporciona una interfaz netamente gráfica para el trabajo con los algoritmos centrales de Weka. Esencialmente tiene las mismas funciones del Explorador aunque algunas de ellas aún no están disponibles. El usuario puede seleccionar los componentes de Weka de una barra de herramientas, y conectarlos juntos para formar un "flujo del conocimiento" que permitirá procesar y analizar datos.

- **Visualizador de Arff** (ArffViewer): interfaz para la edición de ficheros con extensión arff.

- **Log:** muestra la traza de la máquina virtual de acuerdo a la ejecución del programa

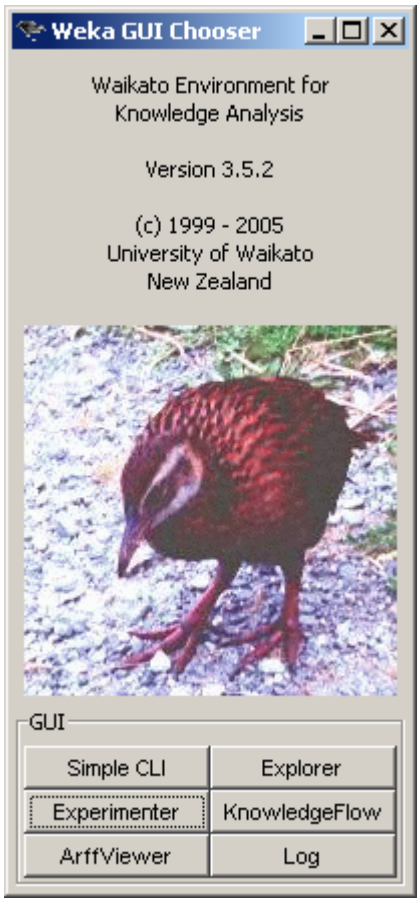


Figura 1 Ventana de selección de interfaces (GUI Chooser)

1.2.1 Entrada de datos

Weka denomina a cada uno de los casos proporcionados en el conjunto de datos de entrada instancias, cada una de las cuales posee propiedades o rasgos que las definen. Los rasgos presentes en cada conjunto de datos son llamados atributos.

El formato de archivos con el que trabaja Weka es denominado *arff*, acrónimo de **A**ttribute-**R**elation **F**ile **F**ormat. Este formato está compuesto por una estructura claramente diferenciada en tres partes:

Sección de encabezamiento: se define el nombre de la relación

Sección de declaración de atributos: se declaran los atributos a utilizar especificando su tipo. Los tipos aceptados por la herramienta son:

a) Numérico (*Numeric*): expresa números reales[1]

b) Entero (*Integer*): expresa números enteros

c) Fecha (*Date*): expresa fechas

d) Cadena (*String*): expresa cadenas de texto, con las restricciones del tipo String[2]

e) Enumerado: expresa entre llaves y separados por comas los posibles valores (caracteres o cadenas de caracteres) que puede tomar el atributo.

Sección de datos: se declaran los datos que componen la relación

A continuación se especifica la sintaxis a utilizar para declarar cada una de estas secciones que forman el fichero con extensión *arff*.

El formato del encabezamiento es el siguiente:

@relation <nombre-de-la-relación>, donde: <nombre-de-la-relación> es de tipo cadena. Si dicho nombre contiene algún espacio será necesario expresarlo entrecomillado.

La sintaxis para declarar los atributos es:

@attribute <nombre-del-atributo> <tipo>, donde: <nombre-del-atributo> es de tipo cadena, y al igual que el nombre de la relación, si hay algún espacio, es necesario poner comillas.

La sección de datos se encabeza con @data <conjunto-de-datos>, donde en <conjunto-de-datos> se especifican todas las instancias; separando los valores de los atributos para una misma instancia entre comas y las instancias (relaciones entre los atributos) con saltos de línea.

[1] debido a que weka es un programa anglosajón la separación de la parte decimal y entera de los números reales se realiza mediante un punto en vez de una coma.

[2] entendiendo como tipo string el ofrecido por java.

En el caso de que algún dato sea desconocido se expresará con un símbolo "?". Es posible añadir comentarios con el símbolo " %", que indicará que desde ese símbolo hasta el final de la línea es todo un comentario. Los comentarios pueden situarse en cualquier lugar del fichero.

El formato por defecto de los ficheros que usa Weka es el *arff,* mas eso no significa que sea el único que admita. Esta herramienta tiene intérpretes de otros formatos como CSV, que son archivos separados por comas o tabuladores (la primera línea contiene los atributos) y C4.5 que son archivos codificados según el formato C4.5, donde los datos se agrupan de tal manera que en un fichero *.names* estarán los nombres de los atributos y en un fichero .data los datos en sí. Al leer Weka ficheros codificados según el formato C4.5 asume que ambos ficheros (el de definición de atributos y el de datos) están en el mismo directorio, por lo que sólo es necesario especificar uno de los dos. Además, las instancias pueden leerse también de un URL (Uniform Resource Locator) o de una base de datos en SQL usando JDBC.

1.2.2 Algoritmos para el Aprendizaje Automatizado

Los algoritmos de Aprendizaje Automatizado implementados en Weka permiten realizar tareas tales como: clasificación, agrupamiento, búsqueda de asociaciones y selección de atributos. El "Explorer", que es el ambiente gráfico básico de Weka, organiza estos algoritmos de acuerdo a las tareas mencionadas utilizando para su identificación las pestañas: "Classify", "Cluster", "Associate" y "Select attributes" respectivamente.

La pestaña "Preprocess" también está presente dentro del modo Explorador. Es la única que se encuentra activa cuando se selecciona este modo. Esta permite cargar un archivo de datos, y a partir de esta acción: se muestran los atributos, algunos estadísticos sobre ellos, relativos a la muestra contenida en el archivo de datos, así como de forma gráfica la relación que existe entre los atributos que se consideran y la clase.

A través de la interfaz de preprocesamiento es posible aplicar cualquiera de los algoritmos de preprocesamiento implementados en Weka, denominados filtros. Estos algoritmos transforman de alguna manera el conjunto de datos de entrada, modificando sus atributos o sus instancias. Son un objeto de estudio dentro del campo del Aprendizaje Automatizado.

Algunos algoritmos de preprocesamiento implementados en Weka son:

Discretize: discretiza un rango de atributos numéricos del conjunto de datos, transformándolos en atributos nominales.

Normalize: normaliza todos los valores numéricos del conjunto de datos dados.

ReplaceMissingValues: remplaza todos los valores perdidos del conjunto de datos. En el caso de atributos nominales lo remplaza por la moda y en los numéricos por la media.

NominalToBinary: convierte todos los atributos nominales en atributos numéricos binarios.

Una vez cargado un archivo, quedan habilitadas las demás opciones que permiten aplicar los diversos algoritmos de aprendizaje presentes para clasificación, búsqueda de asociaciones y selección de atributos.

La pestaña "Classify" pone a disposición del usuario los algoritmos de clasificación implementados, los cuales se agrupan en paquetes atendiendo a la técnica que empleen. Weka tiene implementado clasificadores basados en redes bayesianas, análisis de regresión, redes neuronales, árboles de decisión, los de tipo perezoso, reglas de producción, y meta-clasificadores. Una vez seleccionado uno de ellos, será aplicado sobre las instancias del conjunto de datos activo.

Algunos de los algoritmos para resolver problemas de clasificación ya implementados son:

IBk (Aha y Kibler, 1991): perezoso

VotedPerceptron (Freund y Schapire, 1998): redes neuronales

NaiveBayesSimple (Duda y Hart, 1973): redes bayesianas

Id3 (Quinlan, 1986), J48 (Quinlan, 1993): árboles de decisión

Bagging (Breiman, 1996), Dagging (Ting y Witten, 1997) y AdditiveRegression (Friedman, 1999): meta-clasificadores

OneR (Holte, 1993), DecisionTable (Kohavi, 1995a), M5Rules (Hall, 1999): basados en reglas

La interfaz de agrupamiento (Cluster) es muy similar a la de los clasificadores. En ella se encuentran varios algoritmos de agrupamiento que pueden ser aplicados al conjunto de instancias. Una opción propia de este apartado es la posibilidad de observar en una forma gráfica la asignación de las muestras en grupos. Algunos de los algoritmos para resolver problemas de agrupamiento disponibles en esta versión de Weka son: DBScan (Kriegel et al., 1996) y OPTICS (Ankerst et al., 1999).

La pestaña "Associate" permite aplicar métodos orientados a la búsqueda de asociaciones entre datos. Éste es sin duda el apartado más sencillo y más simple de manejar, carente de apenas opciones. Es importante reseñar que estos métodos sólo funcionan con datos nominales. Algunos algoritmos de búsqueda de asociaciones implementados en Weka son: A priori (Agrawal y Srikant, 1994), Tertius (Flach y Lachiche, 1999), PriorEstimation (Scheffer, 2001).

Los algoritmos de selección de atributos, mostrados en la pestaña "Select attributes" tienen como objetivo identificar, mediante un conjunto de datos que posee ciertos atributos, aquellos atributos que tienen más peso a la hora de determinar si los datos son de una clase o de otra. Ejemplos de algoritmos de selección de atributos implementados en Weka son: CfsSubsetEval (Hall, 1998), GeneticSearch (Goldberg, 1989), RaceSearch (Moore y Lee, 1994).

La última de las pestañas que se encuentran dentro del modo Explorador es la pestaña "Visualize". Muestra gráficamente la distribución de todos los atributos mediante gráficas en dos dimensiones, en las que va representando a través de los ejes todos los posibles pares de combinaciones de los atributos. Permite observar correlaciones y asociaciones entre los atributos de forma gráfica.

1.2.3 Validación con un conjunto de datos

Muchos de los algoritmos disponibles en Weka antes de ser usados necesitan especificar qué datos serán usados como conjunto de entrenamiento y como conjunto de prueba. Estos son conjuntos disjuntos, el primero de ellos utilizado por los algoritmos de Aprendizaje Automatizado para realizar el proceso de aprendizaje fijando sus parámetros (ejemplo, los pesos de una RNA), mientras que los ejemplos del conjunto de prueba se utilizan para medir el desempeño del algoritmo.

Weka implementa cuatro variantes para la validación:

- Usar los mismos datos como conjunto de entrenamiento y de prueba, conformado con todas las instancias del archivo de datos (*Use training set*).
- Usar todas las instancias del conjunto de datos solamente como conjunto de entrenamiento, y proporcionar como conjunto de prueba un nuevo archivo de datos (*Supplied test set*).
- Realizar una validación cruzada estratificada con un número de particiones dado (*Folds*) (*Cross-validation*). La validación cruzada (Kohavi, 1995b) parte de dividir los datos en *n* partes. Luego, se forman *n* muestras diferentes tomando como conjunto de prueba una de ellas, y como conjunto de entrenamiento las n−1 partes restantes. Se dice estratificada porque cada una de las partes conserva las propiedades de la muestra original (porcentaje de elementos de cada clase).
- Tomar un porcentaje del conjunto de datos como muestra de entrenamiento y lo restante como muestra de prueba (*Percentage split*).

Estas variantes están disponibles cuando a través del Explorador, explicado en el epígrafe anterior, se definen esquemas individuales para aplicar algoritmos de clasificación, agrupamiento o selección de rasgos. También algunas de ellas se usan desde el modo Experimentador.

El modo Experimentador (Experimenter) permite crear, correr, modificar y analizar experimentos de una forma más conveniente que analizando los esquemas individualmente. Implementa tres variantes de validación: validación-cruzada estratificada y entrenamiento con un porcentaje de la población, tomando ese porcentaje de forma aleatoria o de forma ordenada. De esta manera se pueden aplicar uno o varios de los algoritmos presentes en Weka para las tareas antes mencionadas, utilizando además diferentes conjuntos de datos.

Los resultados obtenidos en los experimentos pueden ser archivados de tres formas distintas: en un fichero *arff*, en un fichero CSV y en una base de datos. Esto brinda la posibilidad de al mismo tiempo tener muchos resultados, y luego poder realizar contrastes estadísticos entre ellos.

La posibilidad de realizar experimentos es una de las ventajas de Weka, principalmente por el ahorro de tiempo. La corrida de esquemas con un gran número de instancias puede tardar mucho tiempo, sin embargo el modo Experimentador permite realizar las configuraciones necesarias para ejecutar un experimento de forma paralela en varias terminales mediante Java RMI. El modo usado para esto es el cliente-servidor.

1.2.4 Interioridades de Weka

Weka se implementa en Java, lenguaje de alto nivel ampliamente difundido y multiplataforma, lo que posibilita que sistemas escritos en este lenguaje como Weka puedan ejecutarse en cualquier plataforma sobre la que haya una máquina virtual Java disponible. Esta herramienta sigue los preceptos del código abierto (open source), por lo su código fuente está totalmente disponible[3], permitiendo la modificación del mismo. Solo es necesario recompilarlo para posteriormente agregar extensiones al sistema. Además es un software de distribución gratuita lo que posibilita su uso, copia, estudio, modificación y redistribución sin restricciones de licencias.

Las clases de Weka están organizadas en paquetes (Ver anexo #1). Un paquete es la agrupación de clases e interfaces donde lo habitual es que las clases que lo formen estén

[3] El código fuente de Weka (versión 3.5.2) puede encontrarse en http://www.cs.waikato.ac.nz/ml/weka

relacionadas y se ubiquen en un mismo directorio. Esta organización de la estructura de Weka hace que añadir, eliminar o modificar elementos no sea una tarea compleja. Está formado por 10 paquetes globales, y dentro de ellos se agrupan otros paquetes que aunque su contenido se ajusta al paquete padre ayudan a organizar aun mejor la estructura de clases e interfaces.

Los paquetes globales son:

“associations”: contiene las clases que implementan los algoritmos de asociación

“attributeSelection”: contiene las clases que implementan técnicas de selección de atributos.

“classifiers”: agrupa todas las clases que implementan algoritmos de clasificación y estas a su vez se organizan en subpaquetes de acuerdo al tipo de clasificador.

“clusterers”: contiene las clases que implementan algoritmos de agrupamiento.

“core”: paquete central que contiene las clases controladoras del sistema

“datagenerators”: paquete que contiene clases útiles en la generación de conjuntos de datos atendiendo al tipo de algoritmo que será usado.

“estimators”: clases que realizan estimaciones (generalmente probabilísticas) sobre los datos

“experiment”: contiene las clases controladoras relacionadas con el modo Experimentador

“filters”: está constituido por las clases que implementan algoritmos de preprocesamiento

“gui”: contiene todas las clases que implementan los paneles de interacción con el usuario, agrupadas en subpaquetes correspondientes a cada una de las interfaces.

El paquete “core” constituye el centro del sistema Weka. Es usado en la mayoría de las clases existentes. Las clases principales del paquete “core” son: *Attribute*, *Instante*, e *Instances*.

Mediante un objeto de la clase *Attribute* podremos representar un atributo. Su contenido es el nombre, el tipo y en el caso de que sea un atributo nominal, los posibles valores.

Los métodos más usados de la clase *Attribute* son:

- *enumerateValues*: retorna una enumeración de todos los valores de un atributo si es de tipo nominal, cadena, o una relación de atributos, o valor nulo en otro caso.
- *index*: retorna el índice de un atributo

Los métodos *isNominal*, *isNumeric*, *isRelationValued*, *isString*, *isDate* retornan verdadero si el atributo es del tipo especificado en el nombre del método.

- *name*: retorna el nombre del atributo
- *numValues*: retorna el número de valores del atributo. Si este no es nominal, cadena o relación de valores retorna cero.
- *toString*: retorna la descripción de un atributo de la misma manera en que puede ser declarado en un archivo *.arff*
- *value***:** retorna el valor de los atributos nominales o de cadena

La clase *Instance* se utiliza para manejar una instancia. Un objeto de esta clase contiene el valor del atributo de la instancia en particular. Todos los valores (numérico, nominal o cadena) son internamente almacenados como números en punto flotante. Si un atributo es nominal (o cadena), se almacena el valor del índice del correspondiente valor nominal (o cadena), en la definición del atributo. Se escoge esta representación a favor de una elegante programación orientada a objetos, incrementando la velocidad de procesamiento y reduciendo el consumo de memoria. Esta clase es *serializable* por lo que los objetos pueden ser volcados directamente sobre un fichero y también cargados de uno. En Java un objeto es serializable cuando su contenido (datos) y su estructura se transforman en una secuencia de bytes al ser almacenado. Esto hace que los objetos puedan ser enviados por algún flujo de datos con comodidad.

Un objeto de la clase *Instances* contiene un conjunto ordenado de instancias, lo que conforma un conjunto de datos. Las instancias son almacenadas, al igual que la clase Instante, como números reales, incluso las nominales, que como se explicó anteriormente, utilizando como índice la declaración del atributo e indexándolas según este orden.

Los métodos más útiles de la clase *Instance* son:

- *classAttribute*: devuelve la clase de la que procede el atributo
- *classValue*: devuelve el valor de la clase del atributo
- *value*: devuelve el valor del atributo que ocupa una posición determinada
- *enumerateAttributes*: devuelve una enumeración de los atributos que contiene esa instancia
- *weight*: devuelve el peso de una instancia en concreto

Los métodos más útiles de la clase *Instances*:

- *numInstances*: devuelve el número de instancias que contiene el conjunto de datos
- *instance*: devuelve la instancia que ocupa la posición i
- *enumerateInstances*: devuelve una enumeración de las instancias que posee el conjunto de datos
- *attribute*: existen dos métodos con este nombre, la diferencia es que uno recibe como parámetro el índice del atributo, y el otro el nombre, ambos retornan el atributo.
- *enumerateAttributes*: retorna una enumeración de todos los atributos
- *numAttributes*: retorna el número de atributos como un entero
- *attributeStats*: calcula estadísticos de los valores un atributo especificado

Las implementaciones de los esquemas reales de aprendizaje son el recurso más valioso que Weka proporciona. El paquete "classifiers" contiene la implementación de la mayoría de los algoritmos de clasificación y predicción numérica. Contiene la clase *Classifier* que define la estructura general de un esquema para clasificación o predicción numérica. Además incluye la clase *Evaluate* útil para la validación de los algoritmos del paquete.

El paquete "filters" contiene las clases que implementan los algoritmos de preprocesamiento de datos presentes en Weka. Agupados en subpaquetes según su tipo.

1.3 Dos algoritmos de aprendizaje basados en similaridad

Las Redes Neuronales Artificiales (RNA) y el Razonamiento Basado en Casos (RBC) son dos enfoques de la IA que usan la noción de similaridad extensivamente (Lopez y Alipio, 2000). El RBC es la esencia de cómo razonan los humanos, mientras que las RNA son métodos masivamente paralelos preferidos con respecto al primer enfoque por su costo computacional. Por otro lado, una RNA es una caja negra que no le permite chequear al usuario si una solución dada a un problema es factible.

Un modelo que combina los enfoques antes mencionados se presenta en (García y Bello, 1996) como variante del modelo VDM (Value Different Metric) propuesto por Stanfill y Walts (Stanfill y Waltz, 1986). Este modelo híbrido que se implementa en SISI, que en lo adelante será referenciado como *modelo original*, permite desarrollar Sistemas Basados en el Conocimiento. Esta combinación toma las ventajas del aprendizaje que usan las RNA, a la vez que evade mediante el uso del RBC su imposibilidad de explicar una solución. Dado un nuevo problema a resolver, la RNA sugiere un valor para el rasgo objetivo; mientras que la explicación basada en casos consiste en justificar esta solución dada presentando los casos más similares de la base de casos al problema resuelto. El módulo basado en casos utiliza una función de similitud, que se define considerando los pesos de la RNA.

Ambos enfoques aprenden a partir de ejemplos del dominio de aplicación, que se almacenan en un conjunto de entrenamiento o base de casos, como usualmente es denominado en estos enfoques respectivamente. Sea *CB* una base de casos y $A=\{a_1, a_2,..,$

$a_m\}$ el conjunto de *m* atributos que describen un caso *e* de *CB*. Sea Da_i el dominio del *i-ésimo* atributo *i*= 1…m. Cada ejemplo e_k de *CB* se representa por $e_k = (a_{1k}, a_{2k},.., a_{mk})$ y pertenece al universo finito $Da_1 \text{x } Da_2$ x.. x Da_m (espacio de atributos). Cuando el índice no es importante, un ejemplo será representado por $e= (a_1, a_2,.., a_m)$. Cuando se resuelven problemas prácticos a menudo los ejemplos de *CB* tienen valores perdidos o ausentes (*missing values*). Por ejemplo, $e_2= (?, a_2,.., a_m)$ representa que para el segundo ejemplo de la base de casos no se conoce el valor del primer rasgo a_1.

El *modelo original* usa una variante simplificada del modelo de RNA Activación Interactiva y Competencia propuesto por Rumelhart en (McClelland y Rumelhart, 1989), el cual se referencia como SIAC. En la topología de este tipo de RNA, llamadas asociativas, las neuronas se organizan por grupos, donde cada grupo se corresponde con un rasgo. Es decir, en el grupo correspondiente al atributo a_i se hace corresponder una neurona a cada elemento de Da_i. Existe un arco dirigido entre todo par de nodos situados en grupos diferentes, el cual tiene asociado un valor numérico representando el peso de ese enlace. Este valor, que es una medida de la fortaleza con que dos valores de rasgos diferentes se relacionan, se calcula a partir de los ejemplos de *CB* utilizando un mecanismo de aprendizaje Hebbiano (Sima, 1995). Específicamente el modelo SIAC considera la cantidad de casos de la BC que tienen a la vez los valores representados por estas neuronas, relativo a la cantidad de casos que tiene uno de ellos.

Una vez que se presenta a la RNA el patrón correspondiente a una solicitud *q*, se calcula el grado de activación de las neuronas correspondientes al grupo asociado al rasgo objetivo *t*. Esta se define como una suma pesada de las activaciones recibidas desde las neuronas que representan los rasgos predictores. El valor representativo T_j, asociado a la neurona activada con una mayor valor, será el valor inferido para el rasgo *t*.

Es frecuente en aplicaciones reales utilizar atributos numéricos para describir un caso. Luego, muchos valores diferentes podrían aparecer en la BC. Para resolver este problema con el *modelo original*, se requiere tomar algunos valores que pertenezcan al dominio de ese rasgo y que representaran un grupo de valores cercanos. Estos valores son denominados valores representativos. Por ejemplo, en el área del diagnóstico médico, para el atributo

"edad de la persona" se pudieran considerar los valores 15, 25, 35, 40, 50, 60, 75, 90 y 100 como valores representativos para este rasgo, siguiendo el criterio de los expertos humanos. Otra variante para seleccionar estos valores sería aplicando métodos de discretización (Kurgan et al, 2004), y luego seleccionando un valor representativo por cada intervalo obtenido. Ambas variantes se centran en la manipulación de números o de símbolos; y se sigue entonces un enfoque duro (crisp set), donde un valor del dominio es o no representado por un valor representativo de ese atributo.

Sin embargo, cuando la modelación computacional utiliza palabras (ejemplo: pequeño, alto, viejo), pueden ser usadas proposiciones del lenguaje natural así como la capacidad crucial del cerebro para manipular percepciones. La combinación de las RNA y la Lógica Borrosa es una de las hibridaciones más conocidas dentro de las computación blanda (softcomputing), y que captura el mérito de la teoría de las RNA y la lógica borrosa. Esta integración permite desarrollar sistemas más inteligentes para manejar conceptos de la vida real con una interpretación imprecisa o ambigua (Jang, 1997), (Pal, 2001).

Las extensiones de este modelo que se explican a continuación, tienen en común la modelación de los atributos numéricos utilizando conjuntos borrosos. Esto permite discretizar un atributo numérico con mayor naturalidad para definir la topología de la RNA utilizada, a la vez que se tiene una medida más precisa de cuán cercanos están un valor del dominio de un atributo de este tipo y un valor representativo correspondiente.

1.3.1 Modelación de los atributos numéricos

Cuando un atributo numérico $p \in P$ se modela como variable lingüística, el conjunto de valores que este toma en la BC se considera el universo de la variable a modelar. Luego, el conjunto de términos lingüísticos $Rp = \{P_1, P_2, \ldots, P_{/Rp/}\}$ es el conjunto de valores representativos del atributo p, donde P_i representa el j-ésimo conjunto borroso. Retomando el ejemplo anterior del atributo "edad de una persona", se podrían definir los términos lingüísticos "joven", "adulto" y "viejo". Luego, para cada uno de estos se define un conjunto borroso especificando una función de pertenencia (FP), con todas las ventajas que esto puede representar (García, 2000).

El procedimiento que se define a continuación para obtener los conjuntos borrosos de una variable lingüística consta de dos etapas. En la primera etapa se definen los términos lingüísticos, particionando el universo de la variable lingüística mediante el discretizador no supervisado Equal Width. La segunda etapa permite estimar los parámetros de una FP de tipo Trapezoidal para cada uno de los términos previamente definidos.

Sea p un atributo predictor de tipo numérico. Su dominio D_p (conjunto de valores e_p para todo e de CB) se define como el universo de la variable lingüística asociada a este atributo. Después de aplicar la discretización, sean L_j y U_j los extremos izquierdo y derecho respectivamente del intervalo j-ésimo obtenido.

Una FP Trapezoidal se especifica por cuatro parámetros $\{a, b, c, d\}$, los cuáles se definen para el intervalo j-ésimo, con $1<j<n$ como sigue:

$$a_j = c_{j-1} \tag{1}$$

$$b_j = L_j + \frac{L_j + U_j}{4} \tag{2}$$

$$c_j = U_j - \frac{L_j + U_j}{4} \tag{3}$$

$$d_j = b_{j+1} \tag{4}$$

Cuando j=1 (FP definida para los valores más pequeños del atributo), entonces $a_1= b_1= c_1= L_1$. Si por el contrario $j=n$, entonces $b_n= c_n= d_n= U_n$.

A continuación se describen los nuevos algoritmos a implementar, los cuales consideran para los atributos numéricos el preprocesamiento previamente definido. El epígrafe que sigue presenta la variante borrosa de la RNA utilizada en el *modelo original*; mientras que el epígrafe 1.3.3 trata un nuevo enfoque de estos modelos, donde el problema se resuelve por la componente basada en casos en lugar de por la RNA.

1.3.2 El modelo Fuzzy-SIAC

El modelo Fuzzy-SIAC (en inglés, Simple implementation of the Interactive Activation and Competition), es una implementación borrosa de la variante simplificada del modelo de RNA Activación Interactiva y Competencia utilizada en el *modelo original*. Esta es la RNA utilizada en la variante del *modelo original* propuesta en (Rodríguez et al., 2006) modelando los rasgos predictores utilizando conjuntos borrosos.

En el grupo correspondiente a un rasgo de tipo numérico, se ubica una neurona por cada uno de los términos lingüísticos definidos. Cuando se presenta una solicitud q, la neurona de procesamiento N_{Pi} correspondiente al i-ésimo valor representativo Pi para el atributo p es o no activada en el *modelo original*. Sin embargo, cuando este atributo se modela usando conjuntos borrosos, se requiere de un procedimiento para obtener el valor p_{iq} como entrada a esta neurona N_{Pi}. Esto se logra modificando la topología utilizada en el *modelo original* mediante la adición de una capa de preprocesamiento. Esta nueva capa, que tiene una neurona de preprocesamiento para cada valor representativo, se define como una neurona borrosa de tipo I acorde a lo planteado en (Len y Lee, 1999). Este tipo de neurona recibe entradas no borrosas (valor del atributo en la solicitud) y tiene asociado un grado de membresía $\mu_{Pi}(p_q)$ como peso. La salida de la neurona de preprocesamiento correspondiente con el i-ésimo valor representativo P_i es el resultado de la función f_{Pi}: $D_p \rightarrow [0, 1]$. Si el rasgo es numérico se utiliza la expresión (5), y en otro caso la expresión (6).

$$f_{P_i}(p_q) = \mu_{P_i}(p_q) \tag{5}$$

$$f_{P_i}(q_p) = \begin{cases} 1 \text{ if } q_p = P_i \\ 0 \text{ en otro caso} \end{cases} \tag{6}$$

donde:

p_q : representa el valor dado al atributo p en el patrón de entrada,

R_p : conjunto de valores representativos del atributo p,

$\mu_{Pi}(p_q)$: denota el grado de pertenencia de p_q al *i*-ésimo término lingüístico P_i.

Cuando el valor de p_q no se conoce, se puede interpretar siguiendo la teoría de la posibilidad propuesta por Zadeh en (Zadeh, 1978) que un evento perfectamente posible ha ocurrido. Este evento es aquel que no contradiga ninguna situación que se pueda dar, que en el contexto que nos ocupa se puede representar como que $\mu_{Pi}(p_q)=1$ para todos los valores representativos definidos para el rasgo *p*. Esta variante, o simplemente no hacer ningún tratamiento a estos valores, se consideran en los experimentos a realizar con conjuntos de datos que tienen esta característica.

Nótese que si se sigue un enfoque duro para modelar un atributo numérico *p*, se estaría utilizando el *modelo original* donde:

- El conjunto de valores representativos $R_p \subseteq D_p$,

- Las neuronas de preprocesamiento correspondientes tendrían asociado un peso $\mu_{Pi}(p_q)=1$ para todo $P_i \in R_p$, y se usarían las mismas expresiones definidas con anterioridad.

Se aplica la misma idea del modelo SIAC para el cálculo de los pesos. Los pesos se modifican teniendo en cuenta las activaciones de las neuronas (excitación) después que se presentan ciertos estímulos (información de entrada a la red neuronal), sin considerar si era deseada o no esta activación obtenida. En el nuevo modelo, donde los valores representativos para los rasgos numéricos son términos lingüísticos, se utilizarían entonces las activaciones que reciben las neuronas de procesamiento como salida de las neuronas de preprocesamiento explicadas anteriormente.

Sean *a*, *b* dos atributos de *A*. El valor etiquetado por $w_{Ai,Bj}$ representa el peso del arco asociado entre las neuronas de procesamiento N_{Ai} and N_{Bj} correspondientes a los valores representativos A_i y B_j respectivamente. Se emplea una medida basada en la frecuencia relativa y por tanto se obtiene una matriz de pesos no simétrica. Su extensión a un enfoque borroso se basa en utilizar la t-norma "producto" en lugar de la clásica intersección, y la cardinalidad mediante el uso de una extensión simple $\sum$-*count* (Zadeh, 1983).

$$w_{A_i,B_j} = \frac{\sum_{k=1}^{n} f_{A_i}(a_{ik}) f_{B_j}(b_{jk})}{\sum_{k=1}^{n} f_{A_i}(a_{ik})} \quad (7)$$

Después de entrenada la RNA, cuando se presenta una solicitud, se emplea la misma idea del *modelo original* para completar el patrón correspondiente.

1.3.3 El modelo ConFuCiuS

El modelo ConFuCiuS (en inglés, Conecctionist Fuzzy Case-based System), es un modelo conexionista borroso para desarrollar Sistemas Basados en Casos. Este modelo se define como una instancia del clasificador basado en los vecinos más cercanos (en inglés, *k*-NN or *k*- Nearest Neighbords), usando una función de distancia que utiliza los pesos del modelo Fuzzy-SIAC explicado anteriormente.

Después que la RNA es entrenada a partir de los ejemplos almacenados en la muestra de aprendizaje correspondiente, sus pesos y el grado de pertenencia a los conjuntos borrosos definidos, son considerados para definir el criterio de similaridad a emplear en el módulo de recuperación de este modelo. De esta manera, una función de distancia local y un esquema de pesado de atributos son automáticamente propuestos a partir de ejemplos. Luego, se reduce la ingeniería del conocimiento requerida con el uso de este modelo para desarrollar Sistemas Basados en Casos. El nuevo enfoque mejora el desempeño del *modelo original*.

La fuerza predictiva que tiene el valor dado al atributo *a* de un objeto *x* en el valor representativo *T* para el atributo objetivo *t* es una medida de la activación recibida por la neurona de procesamiento N_T solamente considerando ese valor.

$$S(x_a,T) = \frac{\sum_{A_i \in R_a} w_{Ai,T} f_{A_i}(x_a)}{\sum_{A_i \in R_a} f_{A_i}(x_a)} \quad (8)$$

Donde:

-$w_{Ai,T}$ representa el peso del arco existente entre los valores representativos A_i y T

-*f()* es la función definida en la sección anterior

Cuando el valor del rasgo *a* en el caso a comparar no se conoce, siempre se asume que en la expresión anterior *f()* es 1 para todos los valores representativos de este rasgo, por la misma razón explicada anteriormente; mientras que la influencia del tratamiento a los valores ausentes en el cálculo de $w_{Ai,T}$ dependerá de la variante que se asuma en el modelo de RNA. La validación de ConFuCius considera estas variantes.

La diferencia de los valores x_a y x_b para el atributo predictivo *a* en el contexto del atributo objetivo *t* se define como:

$$difference(x_a, y_a) = \sqrt{\sum_{T \in R_t} (S(x_a, T) - S(y_a, T))^2} \qquad (9)$$

donde R_t representa el conjunto de valores representativos (etiquetas lingüísticas) del atributo *t.*

La importancia del atributo predictor a para un objeto *x* en el contexto del atributo objetivo se define como:

$$I_t(x, x_a) = \sqrt{\sum_{T \in R_t} (S(x_a, T))^2} \qquad (10)$$

Y finalmente, la función de distancia definida por la siguiente expresión, donde se emplean los términos anteriormente definidos, es la utilizada en el módulo de recuperación de ConFuCiuS con *r*=2 (distancia Euclidiana).

$$d(e,q) = \left(\sum_{a=1}^{m-1} I_t(q, q_a) \cdot (difference(e_a, q_a))^r \right)^{\frac{1}{r}} \qquad (11)$$

CAPÍTULO 2. UNA METODOLOGÍA PARA EXTENDER WEKA CON NUEVOS FILTROS Y CLASIFICADORES

Una de las características más interesantes de Weka es la posibilidad de modificar su código y obtener versiones adaptadas con funcionalidades que no contengan las versiones oficiales, ampliando así sus posibilidades de uso. Desde el inicio se diseñó como una herramienta orientada a la extensibilidad, por lo que se creó una estructura factible para ello.

Como se aborda en el capítulo anterior, el diseño e implementación de las clases que componen el sistema se conciben para que la adición de un nuevo algoritmo sea una tarea relativamente fácil. Para realizarla no es necesario tener en cuenta detalles que se relacionan indirectamente con la implementación del algoritmo en cuestión, tales como: la lectura del fichero de datos, el cálculo de valores estadísticos, entre otros.

En los epígrafes 2.1 y 2.2 se desarrollarán metodologías para la adición de un nuevo algoritmo de preprocesamiento y de clasificación respectivamente en Weka. Esto se hará mostrando una serie de pasos a seguir para la adición, que serán aplicados para incluir un ejemplo sencillo, aplicando estos pasos.

2.1 Incluyendo un nuevo filtro a Weka

Las herramientas de preprocesamiento de datos en Weka son llamadas filtros. La herramienta agrupa todos los filtros en dos categorías: supervisados y no supervisados. Dentro de ambas se agrupan a la vez en filtros de atributos y de instancias.

Los filtros supervisados (supervised) transforman los atributos teniendo en cuenta la interdependencia entre el atributo clase y los valores de los demás atributos; mientras que los no supervisados (unsupervised) transforman sin considerar el atributo clase.

Los filtros de atributos realizan el preprocesamiento en dirección a los rasgos del conjunto de datos, significando que los mismos hacen cambios en el número o definición de los atributos. Por otro lado, los filtros de instancias realizan un preprocesamiento orientado a las mismas, por lo que no afectan los atributos del conjunto de datos. Permiten realizar acciones como adicionar, eliminar o modificar instancias.

Los algoritmos de filtrado implementados en Weka pueden necesitar o no de la acumulación de estadística del conjunto de datos completo antes de procesar cualquier instancia. De acuerdo a si lo necesiten o no será el comportamiento del filtro.

El funcionamiento de los filtros de forma general se basa en tomar las instancias en lotes, preprocesarlas y ubicarlas en una cola de salida de instancias ya filtradas, por lo que todos hacen uso de esta cola y de una variable bandera que indica la terminación o no de un lote de entrada. Un filtro no debe cambiar los datos de entrada, ni agregar instancias al conjunto de datos usado para definir el formato de entrada.

2.1.1 Implementando un nuevo filtro

Para implementar un nuevo filtro en Weka se deben seguir los siguientes pasos:

1- Crear la clase

2- Hacer uso obligatorio de los paquetes “weka.filters” y “weka.core”

3- Especificar la clase de la que se hereda

4- Especificar las interfaces a implementar

5- Definir la entrada de parámetros al filtro

6- Especificar los métodos a implementar de la clase *Filter*, de acuerdo a la funcionalidad del filtro

7- Crear los métodos necesarios

A continuación se explicarán cada uno de los pasos anteriormente mencionados:

1- Crear la clase

La primera acción ha llevar a cabo es crear una nueva clase con el nombre del filtro a adicionar y situarla en el paquete que le corresponde de acuerdo a las características del algoritmo (supervisado o no supervisado, atributo o instancia).

2- Hacer uso obligatorio de los paquetes “weka.filters” y “weka.core”

Estos paquetes son imprescindibles a la hora de crear un nuevo filtro. El primero de ellos contiene la clase *Filter* que define la estructura general de un esquema de filtrado. En el segundo existen clases importantes como *Instante* e *Instances*, las cuales se utilizan por todos los filtros para manipular los datos.

Si el filtro necesita parámetros de entrada también se debe importar “java.util” debido a que hay clases de este paquete que son usadas por el método que construye las opciones.

3- Especificar la clase de la que se hereda

Todos los filtros deben tener como superclase la clase abstracta *Filter*, porque es en ella donde se encuentran los métodos que deben ser redefinidos por el nuevo algoritmo.

4- Especificar las interfaces a implementar

De acuerdo a la funcionalidad del filtro deben escogerse las interfaces necesarias.

Las posibles interfaces dentro del paquete “filters” son:

- *SupervisedFilter*: para los filtros supervisados
- *UnsupervisedFilter*: para los filtros no supervisados
- *StreamableFilter*: debe ser implementada por los filtros que son persistentes

Las mencionadas son interfaces vacías que se usan solamente para indicar el comportamiento de los filtros, y son aplicables tanto al filtrado de atributos como al filtrado de instancias.

Las posibles interfaces dentro del paquete “weka.core” son:

- *OptionHandler*: si su filtro necesita opciones de entrada

- *WeightedInstancesHandler*: usa la información proporcionada por los pesos de las instancias. Es una interfaz vacía que solo se utiliza como un indicador.

- *AdditionalMeasureProducer*: interfaz que puede producir algunas medidas diferentes a la calculada por el módulo de evaluación.

- *Copyable*: Las clases que la implementan indican que se puede realizar una copia superficial a sus objetos a diferencia del *cloneable* de Java que permite una copia en profundidad de los objetos.

DistanceFunction: interfaz para clases que necesiten calcular y retornar distancias entre dos instancias.

- *Drawable*: interfaz que puede util a clases que pueden realizar algunos dibujos como gráficos.

- *Matchable*: interfaz para clases que puedan ser compatibles con algoritmos que se basen en los árboles

- *Randomizable:* interfaz para las clases que trabajan con números aleatorios

- *Summarizable:* interfaz que suministra un pequeño resumen de la clase (como opuesto a *toString* el cual es a menudo una descripción bastante completa)

- *Undoable:* Interfaz implementada por clases sostienen el deshacer

5- Definir la entrada de parámetros al filtro

Al adicionar un filtro en Weka, se debe analizar si su esquema requiere tomar opciones específicas introducidas por el usuario. En caso de necesitarlas Weka ya tiene implementada toda la interfaz visual para la entrada de datos, pero las opciones o campos que requiere el filtro deben ser especificadas a través de la interfaz *OptionHandler*. Por tanto es necesario implementar los tres métodos que posee dicha interfaz:

- *listOptions*: construye cada una de las opciones que queremos mostrar declarándolas como instancias de la clase Option y especificando su descripción. Retorna una enumeración de estas opciones – es por ello que si se usan estos métodos debe ser importado el paquete "java.util".

- *setOptions*: es el encargado de tomar las opciones que se le pasan al filtro. Es aquí donde se deben almacenar las mismas en las variables referentes a cada una para su uso posterior. A este método se le pasa una lista de opciones en forma de arreglo de *String*, por lo que se encarga también de separar estas opciones y hacer las conversiones necesarias para cada opción.

- *getOptions*: obtiene la configuración actual del filtro. Retorna un arreglo de *String* que puede ser pasado a *setOptions*.

Después de fijadas las opciones deben ser implementados tres métodos por cada uno de los parámetros. El primero es el encargado de almacenar en la variable creada para ello el valor entrado. Su nombre debe ser el mismo que se quiere mostrar en la interfaz, antepuesto por la palabra *set*. El segundo es el encargado de devolver el valor de esa variable, y su nombre es igual al primero, solo que cambiando *set* por *get*. El último de ellos es el encargado de describir el parámetro a manera de ayuda en la interfaz visual, este retorna una cadena de caracteres con la descripción del parámetro. Si se usa en modo línea de comando esta descripción ya fue puesta en el método *setOptions* al crear la misma con el constructor de la clase *Opcion*, el nombre de dicho método es el nombre de la opción terminada en *TipText.*

6- Especificar los métodos a implementar de la clase *Filter*, de acuerdo a la funcionalidad del filtro

Escribir un nuevo filtro en Weka esencialmente implica sobrescribir algunos de los métodos heredados por él de la clase *Filter*, conociendo como necesitan ser cambiados para tipos particulares de algoritmos de filtrado. Es imprescindible entonces analizar la superclase abstracta *Filter*, cuyos métodos serán automáticamente heredados por sus subclases.

Los métodos que se heredan de la clase *Filter* son:

setInputFormat: permite conocer el formato de entrada de una instancia. Recibe como parámetro un objeto de la clase *Instances* y utiliza la información de sus atributos para interpretar las futuras instancias de entrada; es decir, el contenido de las instancias es ignorado, solo se requiere su estructura. Cuando se sobrescriba este método primeramente se debe llamar al *setInputFormat* de la clase *Filter*, en el que se limpia el formato de salida y la cola de salida, y se actualiza la variable que indica un nuevo lote de datos. Si el filtro es capaz de determinar el formato de datos de salida antes de ver cualquiera de las instancias de entrada, entonces debe fijarlo en este método.

- *setOutputFormat*: permite fijar el formato de salida de las instancias. Las clases derivadas de *Filter* deben hacer uso de este método una vez que hayan podido determinar el formato de salida. Este método vacía la cola de salida.

- *input*: permite entrar una instancia para filtrarse. Ordinariamente la instancia es procesada y se hace disponible para la salida inmediatamente. Los filtros que requieren ver todos los casos antes de producir la salida deben verificar en este método que se hayan entrado todas las instancias, si esto no ocurre la instancia a procesar será añadida al buffer de entrada para ser tratada cuando se le informe al filtro que han sido entradas todas las instancias. Si la entrada marca el comienzo de un nuevo lote de datos, la cola de salida es limpiada.

- *batchFinished*: notifica al filtro cuando se han entrado todas las instancias del entrenamiento, este método informa que la estadística obtenida de la entrada de datos recolectada – datos de entrenamiento – no debe ser puesta al día cuando se reciben otros datos. Si hay instancias del entrenamiento que esperan salida, el *batchFinished* retorna verdadero, si no retorna falso. En todos los algoritmos de filtrado, una vez que se haya llamado al *batchFinished*, el formato de salida puede ser leído y las instancias de entrenamiento filtradas están listas para la salida. La primera vez que el *input* se llama después de *batchFinished*, la cola de salida es reajustada – todas las instancias de entrenamiento son removidas de ella.

- *getInputFormat*: permite obtener el formato de entrada

- *getOutputFormat*: permite obtener el formato de salida

Para los filtros que procesan instancias inmediatamente, el formato de salida es determinado tan pronto como se haya especificado el formato de entrada y es, al igual que este, almacenado como un objeto de la clase *Instances*, que es retornado por el *getOutputFormat*. Sin embargo, para los filtros que deban ver el conjunto de datos completo antes de procesar cualquier instancia individual, la situación depende del algoritmo de filtrado en particular. Consecuentemente el método *setInputFormat* retorna verdadero si el formato de salida se puede determinar tan pronto como se haya especificado el formato de entrada, y falso de otra manera.

- *isOutputFormatDefined*: permite comprobar si existe el formato de salida. Retorna verdadero si el formato está listo para ser recogido

Después de haber filtrado una instancia, esta fue enviada a la cola de salida. Luego:

- *output*: recupera las instancias filtradas removiéndola de la cola de salida. Retorna la instancia que ha sido filtrada más recientemente.

- *outputPeek*: obtiene la instancia de salida sin quitarla de la cola

- *numPendingOutput:* permite conocer el número de casos existentes en la cola

- *main*: se encarga de ejecutar el filtro de forma independiente sirviendo así para comprobar su funcionamiento. Es llamado el método estático *filterFile* de la clase *Filter*, pasándole como parámetro el filtro en cuestión y la lista de argumentos del mismo. Además se debe llamar al método *batchFilterFile* si el filtro es capaz de trabajar con múltiples lotes de datos.

7- Crear los métodos necesarios

- *globalInfo*: permite mostrar una descripción del filtro. Retorna una cadena de caracteres con el texto que se muestra. Este método puede ser o no redefinido, en caso de los filtros que no lo tengan simplemente no mostrarán ninguna ayuda o descripción del mismo.

2.1.2 Utilizando un filtro

Los filtros implementados en Weka pueden ser usados no solo por los usuarios de la aplicación, sino también por otras clases dentro del sistema. Por ejemplo, algunos algoritmos de clasificación necesitan preprocesar los datos para después comenzar su trabajo. Esta es la razón por la que algunos filtros implementan métodos que brindan facilidades para poder ser llamados desde otras clases.

Para usar un filtro en otra clase de Weka primero se debe hacer uso de la clase *Filter* que se encuentra en el paquete "filters" y de la clase del filtro a usar, por lo que se debe saber en que paquete se encuentra para importarlo. La clase abstracta *Filter* contiene un método estático llamado *useFilter* al que se le pasa el conjunto de datos a filtrar y el filtro que se quiera aplicar. Este método se encarga de ejecutar dicho filtro y devolver el nuevo conjunto de datos preprocesados.

Para lograr esto debe crearse un objeto del filtro a aplicar, luego llamar al método *setInputFormat* con el conjunto de datos a preprocesar para fijar el formato de entrada y posteriormente invocar al método *useFilter* con los datos y el objeto ya actualizado.

Los algoritmos de Weka también pueden ser utilizados en otras aplicaciones. En este caso solo se tendrían que verificar las clases que usan para importar esos paquetes. Algunas clases e interfaces de estos paquetes no son necesarias como los relacionados con la visualización de las opciones del filtro o la evaluación del mismo.

2.1.3 Ejemplo con un discretizador

El algoritmo de preprocesamiento EqualWidth divide el conjunto de datos en un número de intervalos, definido por el usuario, de igual anchura. No considera el atributo clase para realizar la transformación de los datos. Constituye uno de los algoritmos de

preprocesamiento no supervisados más conocido y sencillo. Necesita la entrada de dos parámetros por parte del usuario: número de intervalos en que van a ser divididos los datos y los atributos a los cuales se le quiere aplicar el algoritmo. Seguidamente se describirá la incorporación del filtro de igual anchura (Equal Width) en Weka, usando como guía los pasos descritos en el epígrafe 2.1.1.

Creación de la clase

Primeramente se crea una nueva clase con el nombre EqualWidth y se sitúa en el paquete "weka.filters.unsupervised.attribute". Se ubica dentro de este paquete porque el filtro no considera el atributo clase y en su desempeño le cambia el dominio a los atributos: los que eran de tipo numérico ahora pasarán a ser nominales tomando n valores posibles, siendo n el número de intervalos dados como entrada al filtro.

Paquetes importados

Deben ser importados los paquetes necesarios, primeramente los dos obligatorios para cualquier filtro: "weka.filters" y "weka.core". Además es necesario importar el paquete "java.util" debido a que el filtro necesita parámetros de entrada.

Herencia de clases e implementación de interfaces

Este filtro hereda de la clase Filter e implementa dos interfaces: *OptionHandler* y *UnsupervisedFilter*. La primera porque se necesitan parámetros de entrada y la segunda debido a que es un filtro no supervisado. Esta última se usa solamente como indicador, ya que es una interfaz vacía.

Parámetros de entrada al filtro

Las opciones de entrada que necesita el filtro son: índice de cada uno de los atributos a discretizar, y el número de intervalos. Los valores de cada una de las opciones son almacenados en la variables m_DiscretizeCols y *m_NumBins* respectivamente. La variable *m_DiscretizeCols* es de tipo *Range*, clase implementada en Weka que se utiliza para almacenar rangos, por defecto tiene valor first-last, brindando la posibilidad de poner las

constantes first y last significando que van a ser preprocesados todos los atributos. El valor por defecto del número de intervalos es 10.

Al utilizar la interfaz OptionHandler deben ser implementados los métodos que ella posee (*listOptions, setOptions, getOptions*).

En *listOptions* se construyen las dos opciones del filtro, creándolas como objetos de la clase Opcion que se encuentra en el paquete "weka.core" y está concebida para almacenar la información concerniente a una opción. Estos objetos se construyen y a su vez se almacenan en una variable de tipo Vector[4] y son retornados como una enumeración a través de la interfaz Enumeration, perteneciente al paquete java.util.

El método *setOptions* recibe como parámetro las opciones en forma de arreglo de cadena. Es el encargado de validar cada una de ellas y almacenarlas mediante los métodos *setBins* y *setAttributeIndices*.

Las opciones válidas son:

-B: especifica el número de intervalos

-R: especifica el índice de los atributos a discretizar

El método *getOptions* devuelve la configuración actual del filtro en un arreglo de cadena. Esta lista es retornada de forma tal que pueda ser pasada al método *setOptions*, con la estructura: -B, número de intervalos, -R, índices de los atributos a discretizar.

Por cada una de las opciones declaradas se definieron los métodos para colocar su valor en la variable creada para ello, devolver el valor de esa variable y mostrar información adicional de cada opción (*set*, *get* y *TipTexts*)

getBins: retorna el número de intervalos en la variable m_NumBins

setBins: le asigna a la variable m_NumBins el número de intervalos definido por el usuario

[4] Clase encontrada en el paquete java.util. Sirve para almacenar una lista de objetos.

binsTipText: retorna el texto "Number of bins."

getAttributeIndices: retorna una cadena con la lista de rangos separada por comas

setAttributeIndices: coloca los atributos que van a ser preprocesados. Pone los rangos desde una representación String.

attributeIndicesTipText: muesta un texto adicional informando como deben indicarse el índice de los atributos a preprocesar

setAttributeIndicesArray: coloca en el arreglo los índices de los atributos a transformar

EqualWidth redefine varios de los métodos heredados de Filter, e implementa otros nuevos con responsabilidades específicas del filtro.

La clase tiene dos constructores, ambos inicializan el índice de los atributos a discretizar. A uno no se le pasan parámetros y pondría los índices por defecto (first-last), y al otro se le pasan los índices de los atributos a discretizar.

Métodos redefinidos de la clase *Filter*

setInputFormat: mediante este método se fija el formato de entrada de los datos. Primeramente se llama al setInputFormat de la clase Filter y después se inicializa la variable *m_CutPoints* que almacena los puntos de corte de los intervalos con vacío porque aún no se han calculado. Además se inicializa la lista de índices de los atributos a discretizar (*m_DiscretizeCols*) con la cantidad de atributos del conjunto.

Input: es el encargado de clasificar una instancia. Si la variable *m_NewBatch* tiene valor Verdadero significa que la instancia es la primera de un nuevo lote de datos, entonces debe ser llamado el método *resetQueue* para vaciar la cola de salida y poner la variable *m_NewBatch* en Falso.

El EqualWidth necesita ver todo el conjunto instancias para poder calcular los valores máximos y mínimos de cada atributo y así formar los puntos de corte. Debido a esto en el *input* debe verificarse si los puntos de corte han sido calculados. Si se calcularon la

instancia será preprocesada mediante el método convertInstance poniéndola disponible en la cola de salida, de lo contrario solo pondrá la instancia en el buffer de entrada (*bufferInput*).

batchFinished: es llamado después que se ha terminado de entrar todo el lote de datos. Como los datos deben haber sido entrados completamente para poder comenzar a filtrar es aquí donde se realiza este proceso. Primero se calculan los puntos de corte para después recorrer todo el buffer de entrada filtrando cada una de las instancias con convertInstance. El *batchFinished* también limpia el buffer de entrada y *m_NewBatch* recibe Verdadero nuevamente porque se ha terminado el lote.

Métodos definidos por la clase

setOutputFormat: especifica el formato de salida de las instancias después de ser preprocesadas. Los atributos que anteriormente eran numéricos ahora pasarán a ser nominales y por tanto deben ser definidos en este método, especificando los posibles valores que pueden tomar cada uno de ellos. Los valores nominales – que serán la misma cantidad que intervalos especificados – tendrán el nombre que forman los puntos extremos del intervalo en cuestión. Por lo tanto este método define el nuevo formato de datos y llama al *setOutputFormat* de la clase Filter para que actualice dicho formato.

getCutPoints: Obtiene los puntos de corte para un atributo. Recibe como parámetro el índice del atributo. Retorna un arreglo contenido por los puntos de corte.

calculateCutPoints: Calcula los puntos de corte para cada atributo utilizando para ello el método *calculateCutPointsByEqualWidthBinning.*

calculateCutPointsByEqualWidthBinning: calcula los puntos de corte para un único atributo.

convertInstance: convierte los valores de los atributos numéricos de una instancia – si están entre los que se preprocesan – a uno de los posibles valores de este atributo ya convertido a nominal.

main: fue implementado de la misma manera en que deben hacerlo todos los filtros: llamando a los métodos *filterFile* y *batchFilterFile* de la clase *Filter*

globalInfo: se devuelve como cadena de caracteres una pequeña descripción del funcionamiento del algoritmo

2.2 Incluyendo un nuevo clasificador a Weka

Weka denomina clasificador a cualquier modelo con capacidad de predecir un valor nominal (clase discreta, clasificación) o un valor numérico (regresión). Los clasificadores son agrupados en diferentes paquetes de acuerdo a la técnica que empleen. Estos paquetes se encuentran dentro del paquete "weka.classifiers". Ellos son:

"bayes": algoritmos de redes bayesianas

"functions": algoritmos de redes neuronales y regresión numérica

"lazy": algoritmos perezosos (un aprendizaje que no es fuera de linea, se hace fuera del tiempo de ejecución)

"meta": algoritmos meta-clasificadores, es decir, que usan como entrada un clasificador base

"misc": algoritmos de clasificación que no están incluidos en ninguna otra categoría

"trees": los algoritmos de árboles de decisión

"rules": algoritmos que implementan reglas de asociación

2.2.1 Implementando un nuevo clasificador

Para crear un nuevo clasificador en Weka han de seguirse los siguientes pasos:

1- Crear la clase

2- Hacer uso obligatorio de los paquetes "weka.classifiers" y "weka.core"

3- Especificar que se hereda de la clase *Classifier*

4- Especificar las interfaces a implementar

5- Especificar la entrada de parámetros

6- Especificar los métodos a implementar de la clase *Classifier*, de acuerdo a la funcionalidad del clasificador

7- Nuevos métodos a crear

A continuación se explicarán cada uno de los pasos:

1- Crear la clase

Primeramente se crea una nueva clase con el nombre del clasificador a adicionar y se sitúa dentro de su paquete correspondiente de acuerdo al tipo de clasificador que se implementará.

2- Hacer uso obligatorio de los siguientes paquetes "weka.classifiers" y "weka.core"

Es imprescindible incluir los paquetes anteriores dentro nuevo clasificador. El primero contiene la mayoría de los algoritmos de clasificación y predicción numérica, además de contener la clase *Classifier* que constituye el esquema general de un algoritmo de clasificación. El segundo, al ser el paquete central del sistema, tiene clases que deben ser usadas por todos los clasificadores.

Si el clasificador necesita parámetros de entrada también se debe importar "Java.util" debido a que hay clases de este paquete que son usadas por el método que construye las opciones.

3- Especificar que se hereda de la clase *Classifier* o de una de sus subclases

4- Especificar las interfaces a implementar de acuerdo a la funcionalidad del clasificador

Las posibles interfaces dentro del paquete "weka.core" son las mismas especificadas anteriormente para añadir un filtro.

Las posibles interfaces dentro del paquete "weka.classifiers" son:

- *IntervalEstimator*: para clasificadores que usen intervalos confidenciales
- *IterativeClassifier:* para clasificadores que pueden inducir modelos de complejidad creciente
- *Sourcable*: para clasificadores que pueden ser convertidos a código Java
- *UpdateableClassifier*: interfaz para modelos de clasificadores incrementales que pueden aprender usando una instancia

5- Especificar la entrada de parámetros para el clasificador

La entrada de parámetros de los clasificadores funciona igual a la de los filtros: si el algoritmo de clasificación necesita parámetros de entrada, entonces es necesario utilizar la interfaz *OptionHandler*, implementando cada uno de sus métodos (listOptions, setOptions, getOptions).

6- Especificar los métodos a implementar de la clase *Classifier*, de acuerdo a la funcionalidad del clasificador

Los métodos principales a tener en cuenta a la hora de implementar un nuevo clasificador para Weka serán heredados de la clase abstracta *Classifier* y que deben ser redefinidos de acuerdo al objetivo que persiga el algoritmo de aprendizaje a implementar. Esta clase es la más importante del paquete classifiers y que constituye una superclase de todos los clasificadores existentes. Los métodos que deben ser redefinidos son:

- *buildClassifier:* se encarga de la construcción del modelo del clasificador tomando como parámetro las instancias de entrenamiento. Debe inicializar todas las variables que correspondan a las opciones específicas del esquema. Nunca debe modificar ningún valor

de las instancias. Después de terminada la ejecución de este método el clasificador debe ser capaz de predecir la clase de cualquier instancia nueva.

- *classifyInstance:* permite clasificar una instancia concreta. Devuelve la clase en la que se ha clasificado o un valor perdido si no se consigue clasificar. En el caso de los clasificadores de predicción numérica la función devuelve el número predicho.

- *distributionForInstance:* cumple la misma función que el método anterior, solo que el resultado de la clasificación se retorna de una forma diferente. Devuelve la distribución de una instancia en forma de vector. Si el clasificador no ha logrado clasificar la instancia dada devolverá un vector lleno de ceros. Si lo consigue y la clase es numérica devolverá un vector de un solo elemento que será el valor obtenido. En los demás casos devolverá un vector con el grado de pertenencia de la instancia dada a cada una de las clases.

7- Nuevos métodos a crear

- *globalInfo*: retorna una descripción del clasificador. Retorna una cadena de caracteres con el texto que se muestra. Este método puede ser o no redefinido. En caso de los clasificadores que no lo tengan simplemente no mostrarán ninguna ayuda o descripción del mismo.

- *main*: se encarga de ejecutar el filtro de forma independiente sirviendo así para comprobar su funcionamiento. En este método solo es llamada la función estática *evaluateModel* de la clase *Evaluation* pasándole como parámetro el propio filtro.

2.2.2 Utilizando un clasificador

Los clasificadores al igual que los filtros implementados en Weka pueden ser usados no solo por los usuarios de la aplicación sino también por otras clases dentro del sistema.

La reutilización de clasificadores o parte de ellos se manifiesta en Weka de varias formas. Muchos clasificadores heredan de otros clasificadores, en este caso solo se tendrían que redefinir algunos métodos, es por tanto conveniente que los métodos sean implementados de forma tal que puedan ser heredados por otras clases.

Otra variante es que algunos clasificadores utilizan otros como es el caso de algunos meta-clasificadores. En este caso primeramente el clasificador a utilizar debe ser entrenado para esto es utilizado el método *buildClassifier*, después se mandará a clasificar cada una de las instancias deseadas mediante uno de los dos métodos que se pueden encargar de esta tarea *distributionForInstance* o *classifyInstance*.

Los clasificadores también se pueden exportar de Weka, para esto solo tendrían que exportarse los paquetes o las clases que intervengan en el proceso de clasificación y que se usen en el algoritmo a implementar. Al menos deben ser importadas las clases *Instante*, *Instances* y la clase que contenga el clasificador a utilizar. La forma de utilizar el clasificador es muy similar a usarlo en Weka, primero lo entrenamos y después se usa para clasificar.

2.2.3 Ejemplo con una red neuronal simple

El Perceptron Simple como se explica en (Hilera y Martínez, 1995) fue el primer modelo de red neuronal artificial propuesto por Rosenblatt en 1958 (Rosenblatt, 1958). Está formado por varias neuronas lineales para recibir las entradas a la red y una neurona de salida. Es capaz de decidir cuando una entrada presentada pertenece una entrada presentada a la red. La única neurona de salida del Perceptron realiza la suma ponderada de las entradas y se le resta el umbral. El umbral es el desplazamiento de la función de activación en el eje de coordenadas debido a características internas de la propia neurona.

En el proceso de entrenamiento, el Perceptron se expone a un conjunto de patrones de entrada, y los pesos de la red son reajustados de forma que al final del entrenamiento se obtengan las salidas esperadas para cada uno de esos patrones de entrada.

En el algoritmo de entrenamiento primeramente se inicializan los pesos y el umbral, asignándole valores aleatorios a cada uno de los pesos (w_i) de las conexiones y el umbral (L) se le asigna valor w_0. Seguidamente se presenta un nuevo patrón de entrada $X_p = (X_1,X_2,..,X_n)$ junto con la salida esperada d(t). Con estos datos se calcula la salida actual mediante (12).

Donde *f* se define por: $f(x) = \begin{Bmatrix} 1 & \text{si} & X \geq 0 \\ 0 & \text{si} & X < 0 \end{Bmatrix}$

Seguidamente se adaptan los pesos mediante (13) donde d(t) representa la salida deseada, y será 1 si el patrón pertenece a la clase A, y -1 si es de la clase B. En estas ecuaciones, h es un factor de ganancia (0.0<h<1.0), que debe ser ajustado de forma que satisfaga tanto los requerimientos de aprendizaje rápido como la estabilidad de las estimaciones de los pesos. El proceso se repite hasta un número de iteraciones predeterminado por el usuario.

$$y(t) = f\left(\sum_{i=0}^{N-1} w_i(t) * x_i - L\right) \tag{12}$$

$$w_i(t+1) = w_i(t) + h*\left(d(t) - y(t)\right) * x_i(t) \qquad 0 \leq i \leq N-1 \tag{13}$$

En resumen, la técnica de aprendizaje del Perceptron es un algoritmo iterativo sobre un conjunto de ejemplos de entrenamiento mediante el cual se realiza una búsqueda en el espacio de todos los valores posibles para el vector de peso W.

Seguidamente se describirá la incorporación del Perceptron Simple a Weka, usando como guía la metodología para adicionar nuevos clasificadores a Weka, propuesta en el epígrafe 2.2.1.

Creación de la clase

Primeramente se crea una nueva clase llamada *SimplePerceptron* y se sitúa en el paquete "weka.classifiers.functions" debido a que este algoritmo está basado en redes neuronales y por tanto debe ser situado en el paquete referente a funciones.

Paquetes importados

Es imprescindible importar los paquetes "weka.classifiers" y "weka.core", que son los cualquier clasificador utilizará, solo que del primer paquete se importaron las clases *Classifier* y *Evaluation* que son las necesarias, *Classifier* porque representa el esquema

general de un clasificador y *Evaluation* debido a que es la clase utilizada para evaluar los clasificadores.

Se importaron paquetes incluidos dentro del paquete "weka.filters" debido a que es necesario el uso de los filtros como:

"weka.filters.Filter",

"weka.filters.unsupervised.attribute.NominalToBinary",

"weka.filters.unsupervised.attribute.ReplaceMissingValues",

Además se importó "java.util", debido a que el filtro necesita opciones de entrada.

Herencia de clases e implementación de interfaces

El *SimplePerceptron* hereda de la clase *Classifier* como todo clasificador e implementa solamente la interfaz *OptionHandler* debido a que necesita varios parámetros de entrada.

Entrada de parámetros

Los parámetros que necesita el clasificador para su funcionamiento, de acuerdo al nombre que aparece en la interfaz de la entrada de parámetros son:

bias: umbral, desplazamiento de la función de activación en el eje de coordenadas. Su valor es almacenado en la variable *m_Bias*.

numIterations: representa el número de iteraciones para entrenar el Perceptron. Se almacena en la variable *m_ NumIterations.*

step: constituye el factor de ganancia. Se almacena en la variable *m_Step*.

Para lograr incluir las opciones antes mencionadas se programaron los métodos *listOptions*, *setOptions* y *getOptions* pertenecientes a la Interfaz OptionHandler:

listOptions: es el encargado de construir las tres opciones del algoritmo de clasificación, creándolas como objetos de la clase *Opcion*. Esta clase se encuentra en el paquete "weka.core" y está concebida para almacenar la información concerniente a una opción. El método retorna la enumeración de las opciones.

setOptions: se encarga de validar y tomar cada una de las opciones de la cadena de entrada:

-B: umbral, por defecto tiene valor cero

-I: número de iteraciones, por defecto tiene valor uno

-E: factor de ganancia, por defecto tiene valor uno

getOptions: devuelve la configuración actual del filtro en forma de una lista de cadena de caracteres. Esta lista se devuelve de forma tal que pueda ser pasada al método setOptions, la lista sería de la forma: -B, factor de activación, -I, número de iteraciones, -E, factor de ganancia

De la misma forma que en la adición de un filtro se crearon los tres métodos que corresponden a cada opción:

toString: retorna una descripción textual del nombre del clasificador: "SimplePerceptron"

numIterationsTipText: retorna la información acerca del parámetro numero de iteraciones introducido por el usuario. El texto que devuelve es: "Number of iterations to be performed."

getNumIterations: retorna el valor del número de iteraciones que fue almacenado en la variable m_NumIterations

setNumIterations: almacena en la variable m_NumIterations el número de iteraciones entrado como parámetro

StepTipText: retorna el texto "Step for the polynomial kernel." acerca que describe el parámetro step entrado como parámetro

getStep: retorna el valor del parámetro step almacenado en la variable *m_Step*

setStep: le asigna el valor a la variable *m_Step*

getBias: retorna el valor del bias almacenado en la variable m_Bias

setBias: almacena el valor de la variable m_Bias entrado como parámetro

Métodos heredados de *Classifier*

buildClassifier: se encarga de construir el clasificador. Este método recibe como parámetro el conjunto de entrenamiento. En este caso de los algoritmos de redes neuronales, en este método es donde se debe construir y entrenar la red, para que una vez finalizado el método la red esté lista para clasificar una nueva instancia

El Perceptron es una red de una sola neurona que solo puede clasificar en clases simbólicas y que el número de las mismas no sea mayor de dos, por lo que primeramente verifica esto y levanta una excepción si el numero de clases es mayor de dos o la clase es un atributo numérico.

El clasificador debe tener en cuenta los valores perdidos (missing values), el algoritmo simplemente les asigna valor; a los numéricos la media de ese atributo y a los nominales la moda (valor que ocurre con mayor frecuencia). Para realizar esta acción se utilizó un filtro ya implementado en Weka llamado ReplaceMissingValues, por lo que fue necesario importar el paquete "weka.filters.unsupervised.attribute.ReplaceMissingValues". Se crea un objeto de la clase *ReplaceMissingValues y* se llama al método *setInputformat* con el conjunto de datos de entrenamiento para fijar el formato de salida. Par poder hacer uso de este filtro debemos llamar al método *useFilter* de la clase Filter (clase que también se importa).

Para garantizar la correcta entrada de los valores de las instancias a la neurona deben ser transformados los atributos nominales que puedan tomar mas de dos valores para esto se utilizó otro de los filtros ya implementados en Weka denominado *NominalToBinary*, este

convierte los atributos nominales especificados en binarios. Un atributo con k valores es transformado en k atributos binarios si la clase es nominal (restricción antes verificada).

Después de preprocesado el conjunto de datos de entrenamiento ya el Perceptron está listo para ser entrenado. El siguiente paso es el cálculo de los pesos de cada una de las entradas mediante el entrenamiento de la neurona realizando iteraciones sucesivas especificadas como una opción al algoritmo, donde en cada una de las iteraciones se actualizan los pesos siguiendo el algoritmo uno por cada atributo y se almacenan en la variable *m_Weights* declarada como un arreglo de números reales.

La forma en que se realiza la actualización de los pesos es determinando el valor de la clase que calcula la neurona y comparándolo con el valor real del atributo clase para esta instancia; si son iguales se pasa a la próxima instancia si son diferentes se actualizan los pesos y se continua el entrenamiento. Una vez finalizado el entrenamiento la red está lista para el proceso de clasificación.

distributionForInstance: se encarga de la clasificación de una instancia recibida como parámetro. Devuelve la distribución de dicha instancia para cada clase. En este al igual que *builClassifier* debe transformar los valores de la instancia dada para ser entrados a la neurona por lo que son aplicados los algoritmos de filtrado *ReplaceMissingValues* y *NominalToBinary* para reemplazar los valores perdidos y para convertir los atributos nominales a binarios. Luego de la conversión de valores de los atributos, calculamos el valor de la función de activación. Si el valor de la función resultó menor que cero clasificamos la instancia con el primer valor de la clase, de lo contrario con el valor dos, retornando el arreglo con estos valores. La forma de retornar esta información en el arreglo seria poniendo uno en la posición de la clase en que se clasificó y cero en la otra posición del arreglo.

makePrediction: calcula la función de activación de la neurona utilizando los pesos y las entradas que no son mas que los valores de la instancia dada para cada uno de los atributos.

main: se llama al método *evaluateModel* de la clase *Evaluation* para evaluar el algoritmo de clasificación.

CAPÍTULO 3. IMPLEMENTACIÓN Y VALIDACIÓN DE LOS NUEVOS ALGORITMOS UTILIZANDO WEKA

En este capítulo se describirá la implementación y adición a Weka de dos nuevos modelos: Fuzzy-SIAC y ConFuCiuS utilizando como guía la metodología descrita en el capítulo anterior. Estos nuevos modelos exigen un trabajo con datos borrosos lo que creó la necesidad del trabajo de Weka con una nueva estructura capaz de manejar datos de este tipo. Para ello se creó un nuevo tipo de dato en Weka, agregándolo a los que ya el sistema ofrecía para el trabajo con los atributos.

Para que los nuevos modelos obtengan las instancias de entradas de forma borrosa se adoptan dos variantes: que los datos sean entrados al sistema ya modelados borrosos o que el propio Weka sea el encargado de preprocesar los mismos mediante un algoritmo de preprocesamiento. Por lo que se construyó el nuevo filtro EqualWidthFuzzyfier para permitir las conversiones a este nuevo tipo de dato.

La inclusión de los modelos a permitido comparar su desempeño con otros algoritmos ya incluidos en Weka se mostrarán los resultados obtenidos de estas corridas.

El capítulo abordará los detalles de implementación de los tres algoritmos mencionados, los dos modelos y el filtro. Además de los detalles de la implementación del nuevo tipo de dato. Se mostrará una aplicación a la medida creada a partir del código fuente de Weka.

3.1 Conjuntos borrosos en Weka

Weka cuenta con tipos numéricos y simbólicos para manejar los atributos en los conjuntos de datos, pero no tiene ningún tipo que represente conjuntos borrosos (fuzzy), todas las variantes que ofrece Weka en el manejo de datos es para datos duros (crisp). Los modelos adicionados Fuzzy-SIAC y ConFuCiuS requieren los datos de entrada en forma borrosa.

Para lograr el tratamiento de datos borrosos fue necesario crear el tipo de dato denominado LINGUISTIC. Este nuevo tipo permite asociar a cada atributo numérico funciones de pertenencia. Por cada uno de los términos lingüísticos que modelan al atributo es declarada una función de pertenencia de las tres que se definieron: Gaussiana (14), Trapezoidal (15) y Triangular (16), las mismas son usadas a su vez para calcular el grado de membresía de un valor a cada uno de los términos lingüísticos.

$$\text{Trapezoidal}(x) = \begin{cases} \dfrac{x-a}{b-a} & a < x < b \\ 1 & b \le x \le c \\ \dfrac{d-x}{d-c} & c < x < d \\ 0 & \textit{en otro caso} \end{cases} \qquad (14)$$

$$\text{Triangular}(x) = \begin{cases} \dfrac{x-a}{b-a} & a < x < b \\ \dfrac{c-x}{c-b} & b \le x < c \\ 0 & \textit{en otro caso} \end{cases} \qquad (15)$$

$$\text{Gaussian}(x) = e^{-\left(\frac{1}{2}\right)\left(\frac{x-c}{s}\right)^2} \qquad (16)$$

Para implementar las funciones de membresía se creó la clase abstracta *MemberShipFunction* de la que deben heredar todas las clases que calculen funciones de pertenencia. Esta clase solo cuenta con el método abstracto *evaluate* encargado de evaluar un valor dado en la función de pertenencia que lo esté redefiniendo. Las clases creadas para implementar las tres funciones de pertenencia fueron: *Gaussian*, *Trapezoid* y *Triangle,* las que constituyen subclases de *MemberShipFunction* y redefinen el método *evaluate* para el cálculo del grado de pertenencia de un valor dado según la función que la define.

Se implementó una nueva clase de atributo llamada *LinguisticAttribute* que hereda de la clase *Attribute*. Los atributos que sean de tipo LINGUISTIC serán instanciados con la misma, a diferencia del resto de los tipos que maneja Weka en el que todos los atributos son creados a partir de la clase *Attribute* y el tipo es solo un campo de esta clase.

La nueva clase para manejar los atributos borrosos, implementa un constructor al que se le pasa como parámetro una lista de las funciones de membresía que serán usadas para

calcular la pertenencia de un valor a cada término lingüístico que definen al atributo. Además implementa la función *membership* la cual recibe un valor real y el índice de uno de los términos lingüísticos como parámetro, y devuelve el grado de pertenencia del valor al término lingüístico especificado. Posee un método llamado *termsCount* que devuelve la cantidad de términos lingüísticos especificados.

Implementadas las clases necesarias para tratar el nuevo tipo de rasgo se hace necesario definir la forma de declaración que tendrán los atributos de tipo LINGUISTIC en los ficheros de entrada *.arff.* Para lograr esto se validaron las declaraciones de atributos de este tipo y se almacenaron las definiciones de los mismos, esto se realizó en el método *parseAttribute* encargado de la validación de todos los token[5].

La nueva sintaxis para la declaración de los atributos borrosos es la siguiente:

@attribute <nombre-del-atributo> LINGUISTIC {<función-de-pertenencia> (<parámetros>),…,<función-de-pertenencia>(<parámetros>)}

Donde <función-de-pertenencia> puede tomar los valores, *Gaussian*, *Trapezoidal* o *Triangular* y los parámetros varían de acuerdo a cada función.

Ver ejemplo en el anexo #2

3.2 Implementación de un filtro para trabajar con atributos borrosos

Debido a la implementación de un nuevo tipo de dato en Weka fue necesario la creación de un filtro que fuese capaz de convertir los datos numéricos a términos lingüísticos. El nuevo algoritmo de preprocesamiento es llamado *EqualWidthFuzzyfier* y está ubicado dentro de los filtros no supervisados de atributos.

El filtro recibe como parámetro el número de términos lingüísticos en que se desea modelar los atributos numéricos y la función de membresía a utilizar para el cálculo de los valores de pertenencia. La frontera de los intervalos que se usarán para calcular los parámetros de las funciones de membresía es hallada utilizando el algoritmo de igual

[5] Token: término en ingles utilizado para definir cada una de las cadenas que se leen del fichero de entrada.

anchura (*Equal Width*).

EqualWidthFuzzyfier, al igual que el *Equal Width,* debe ver todas las instancias antes de comenzar a preprocesarlas. Esto obliga a fijar el formato de salida de los datos en el método *bachFinished*, instanciando en el los atributos mediante la clase *LinguisticAttribute,* creando primeramente la función de pertenencia que usará cada uno de ellos y calculando además los parámetros que necesita cada función.

En la nueva clase del filtro se implementaron los métodos *buildGaussianTermSet*, *buildTrapezoidalTermSet y buildTriangularTermSet*, respectivos a las funciones de pertenencia Gaussiana, Trapezoidal y Triangular, los cuales se encargan de construir un vector de objetos de funciones de pertenencia con sus respectivos parámetros. Este vector es usado para llamar al constructor de la clase *LinguisticAttribute.*

Los parámetros de las funciones Trapezoidal y Triangular son calculados mediante los métodos *calculateTrapezoidParameters* y *calculateTriangleParameters* respectivamente. Para el caso de la función Gaussiana los parámetros se calcularon en el propio método *buildGaussianTermSet*.

Las instancias del conjunto de datos, después de filtrarse, no sufren ningún cambio, colocándose en la cola de salida listas para ser recogidas.

3.3 Usando la metodología propuesta para añadir los nuevos modelos

La metodología propuesta en el capítulo 2 fue aplicada para la implementación e inclusión en Weka de dos nuevos algoritmos, Fuzzy-SIAC y ConFuCiuS. Estos manejan los datos en forma borrosa por lo que hay que garantizar que todos los atributos numéricos hayan sido modelados como rasgos lingüísticos. Para esto puede ser de gran ayuda el filtro implementado *EqualWidthFuzzyfier*.

La función utilizada fue una trapezoidal por tanto esta función calcula los valores A, B, C, D (Figura 2) de cada uno de los trapecios.

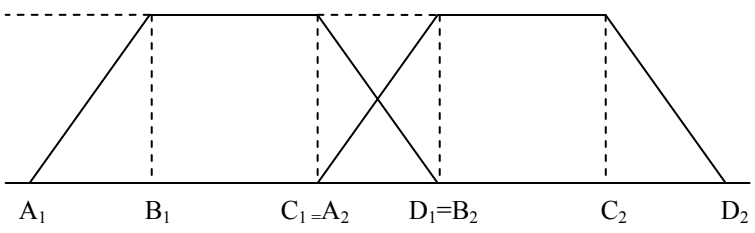


Figura 2 Funciones de membresía utilizadas para el cálculo de los valores de pertenencia

Los métodos *calculateMSDegree* y *calculateNumberColumns* son comunes en las implementaciones de los modelos Fuzzy-SIAC y ConFuCiuS.

El método *calculateMSDegree* preprocesa los datos calculando los grados de pertenencia para un conjunto de instancias. Construye una matriz que tiene tantas filas como instancias preprocesadas y tantas columnas como el total de términos lingüísticos calculados por el método *calculateNumberColumns*. Si el atributo es lingüístico utiliza el método *membership* de la clase *LinguisticAttribute* para determinar los valores de pertenencia de cada instancia por cada término lingüístico.

La matriz construida tiene tantas columnas correspondientes a un atributo como términos lingüísticos hayan sido fijados, este número puede ser obtenido mediante el método *termsCount*. Para los atributos nominales se crean tantas columnas en la matriz como valores estén en el dominio del rasgo. Al evaluarlos se pondrá uno en el valor que corresponda a la instancia dada y cero en las demás. Si algún valor de la instancia es perdido entonces en las columnas correspondientes al mismo se pondrá el valor entrado como opción al clasificador. La matriz preprocesada es el valor de retorno de la función *calculateMSDegree*. Esta función genera una excepción si existen rasgos que no son nominales o lingüísticos.

El método *calculateNumberColumns* calcula la cantidad de columnas que tendrá un atributo determinado en la matriz preprocesada.

3.3.1 Fuzzy-SIAC

El modelo Fuzzy-SIAC es una extensión borrosa de la variante simplificada del modelo de RNA Activación Interactiva y Competencia. Utilizando la metodología propuesta en el Capítulo 2 para la implementación en Weka de un nuevo algoritmo de clasificación, se implementó la red neuronal que actúa como clasificador en este modelo. A continuación se describirán los detalles de programación de este algoritmo de clasificación.

Primeramente se crea una clase llamada *FuzzySIAC,* encargada de implementar el algoritmo y se ubica dentro del paquete "weka.classifiers.functions", como el resto de los que utilizan un enfoque de redes neuronales. Esta tiene como superclase a *Classifier* e implementa la interfaz *OptionHandler*, debido a que el clasificador necesita un parámetro de entrada. Se importaron los paquetes imprescindibles para cada clasificador ("weka.core" y "weka.classifiers"), además de "java.util".

El parámetro de entrada necesitado, *missingSustitution*, es la constante con la que van a ser sustituidos todos los valores perdidos presentes en el conjunto de datos. Es almacenado en la variable *m_MissingValues* que por defecto tiene valor cero.

Fueron redefinidos los métodos *buildClassifier* y *classifyInstance* heredados de la superclase *Classifier,* el primero para construir y entrenar el clasificador y el segundo para clasificar una instancia.

El método *buildClassifier* primeramente chequea que los atributos sean de tipos nominales o lingüísticos y que el atributo clase sea nominal. En este método son inicializadas las matrices que almacenan los datos preprocesados y los pesos. La variable *m_MSDegree* creada como un arreglo bidimensional, es llenada con los valores de pertenencia de cada instancia a cada término lingüístico utilizando el método c*alculateMSDegree.*

Después de realizarse el preprocesamiento de los datos debe ser calculada la matriz de pesos, representada por la variable *m_WeightMatrix*, empleando una medida basada en la frecuencia relativa y por tanto obteniendo una matriz de pesos no simétrica. La dimensión

de la misma es de MxM, donde M es el total de términos lingüísticos y se puede calcular utilizando la función *getSumNumberColumns.*

classifyInstance: primeramente se verifica si la matriz de pesos pudo ser calculada anteriormente. En este método se calculan todos los valores de pertenencia de las instancias a cada término lingüístico, lo que corresponde a una posible fila de la matriz *m_MSDegree*. Estos valores de pertenencia se almacenan en un arreglo unidimensional que servirá como entrada a la red neuronal, calculado mediante la función *calculateRowMSDegree.* Además son calculadas las funciones de activación de las neuronas de salida (una por cada clase). El mayor valor devuelto por estas neuronas determina en que clase se clasifica la instancia dada. Se retorna este valor de la clase como el resultado de la clasificación de la instancia.

Además de los métodos redefinidos de la superclase se crearon otros nuevos, en su mayoría para preprocesar los datos.

calculateRowMSDegree: calcula los grados de pertenencia de cada uno de los atributos de una instancia dada. Si el atributo es lingüístico evalúa cada valor de la instancia en la función de membresía definida, el arreglo que es devuelto tiene tantas columnas como intervalos hayan sido fijados para este tipo de atributo. Para los atributos nominales se crean tantos escaques en el arreglo de retorno como valores estén en el dominio del rasgo y se pondrá uno en el valor que corresponda a la instancia dada y cero en las demás. Si algún valor de la instancia es perdido entonces en las columnas correspondientes al mismo se pondrá el valor entrado como opción al algoritmo. El valor de retorno de esta función es un arreglo equivalente a una de las filas de la matriz que devuelve c*alculateMSDegre.*

getSumNumberColumns: retorna el número total de términos lingüísticos usados, es decir, el número de columnas que tiene la matriz preprocesada. Esta función suma para cada atributo la cantidad de términos lingüísticos que lo representa. En el caso de los atributos lingüísticos utiliza el método de la clase *LinguisticAttribute*: *termsCount* y para los nominales sería el número de posibles valores que pueden tomar cada uno de ellos.

getClassIndex: como todos los atributos después de preprocesados están en una misma matriz y no todos tienen la misma cantidad de columnas, esta función devuelve a partir de que columna comienza el atributo clase.

getAttributeNumberColumns: devuelve el número de columnas que tiene un atributo en la matriz preprocesada. El índice del atributo en el conjunto de datos es recibido como parámetro.

main: se encarga de evaluar el modelo llamando a la función *evaluateModel* de la clase *Evaluation*, contenida en el paquete "weka.classifiers".

3.3.2 ConFuCiuS

El modelo ConFuCiuS combina dos componentes fundamentales: la RNA Fuzzy-SIAC explicada anteriormente y la regla de los vecinos más cercanos (*k*-NN o *k*- Nearest Neighbords). Su implementación no contempla, como en el modelo planteado en el capítulo 1, la importancia de cada rasgo, la misma es tomada con valor uno para cada atributo.

La implementación e incorporación de ConFuCiuS dentro de Weka se auxilia del clasificador IBk que implementa el algoritmo del *k*-NN estándar, ya presente en la herramienta. La clase IBk está contenida en el paquete "weka.classifiers.lazy". Este clasificador recibe como uno de sus parámetros de entrada el algoritmo de búsqueda de los k vecinos mas cercanos, Weka implementa dos: *LinearNN* y *KDTree*. El primero de ellos es el que viene por defecto, implementa un algoritmo de búsqueda de los vecinos más cercanos por fuerza bruta. *KDTree* implementa una estructura de árbol para el trabajo con las instancias, utiliza el método "divide y vencerás" y no es capaz de trabajar con conjuntos de datos que posean valores perdidos. Estos algoritmos de búsqueda se encuentran implementados en el paquete "weka.core".

Los algoritmos de búsqueda explicados reciben como uno de sus parámetros la función de distancia que se desea utilizar para la búsqueda de los vecinos más cercanos. Esta define como será calcula la similitud entre dos instancias. Weka ofrece solo una forma, los

vecinos mas cercanos a una instancia se calculan en términos de la distancia Euclideana estándar. La clase que implementa esta distancia esta contenida en el paquete "weka.core" y se denomina *EuclideanDistance*.

ConFuCiuS propone una nueva función de distancia para el modelo IBk y exige que los atributos del conjunto de datos sean nominales o lingüísticos. La adición del nuevo modelo se restringe a la creación de una clase que implemente una nueva función de distancia homóloga a *EuclideanDistance*.

La nueva clase que implementa la función de distancia fue implementada en la clase *FuzzyVDM*, la misma debe utilizar la interfaz *DistanceFunction* como el resto de las funciones de distancia. Se utiliza también la interfaz *OptionHandler*, que ya es implementada debido a que *DistanceFunction* hereda de esta interfaz y es necesaria ya que se necesita una opción para tomar el valor por el que deben ser sustituidos los valores perdidos. La interfaz *DistanceFunction* define métodos necesarios a implementar para el cálculo de la distancia entre dos instancias, ellos son:

setInstances: toma el conjunto de instancias que pueden ser usadas para inicializar las variables necesarias.

getInstances: obtiene el conjunto de instancias actual.

distance: este método esta sobrecargado pero todos realizan la misma función, la del calculo de la distancia entre dos instancias.

postProcessDistances: realiza un procesamiento posterior de las distancias ya calculadas si es necesario y retorna estos nuevos valores.

update: sirve para actualizar la función de distancia.

La nueva clase implementada para el cálculo de la similitud se denomina FuzzyVDM y se encuentra en el paquete "weka.core". Implementa las interfaces *DistanceFunction*, *Cloneable*, *Serializable*.

En esta clase no se hace necesario importar ningún paquete de Weka.

Para el cálculo de la distancia se necesitan dos opciones de entrada:

missingSustitution: valor por el que serán sustituidos los valores perdidos en el calculo de la matriz preprocesada que será usada en la matriz de pesos por defecto cero.

q: grado de la función de distancia por defecto debe ser dos pero se deja la opción para permitir cambiar el grado de la potencia y de la raíz que se calcula en la formula de diferencia entre un atributo de dos instancias diferentes.

Se implementan los métodos correspondientes a cada una de las opciones:

-M: valor que sustituye los valores perdidos

-Q: representa el grado de la función

Los métodos a implementar de forma obligatoria serían los definidos en la interfaz *DistanceFunction*. La implementación de cada uno de ellos es descrita a continuación:

setInstances: este método recibe como parámetro un conjunto de instancias por lo que es aquí donde debe ser calculada la matriz de pesos correspondiente a la red neuronal. La forma de calcular esta matriz se realiza de la misma forma que en el modelo Fuzzy-SIAC ya descrito. Primeramente debe ser calculada la matriz preprocesada y después mediante la formula utilizada basándose en la frecuencia relativa calcular los pesos.

distance: esta función es la encargada de devolver el valor de similitud entre dos instancias. Para calcular la similaridad utiliza otra función llamada *difference* que halla la diferencia entre dos instancia para un atributo, por lo que la misma es llamada para cada uno de los atributos de las instancias. La distancia que es el valor retornado seria la sumatoria de todas estas diferencias elevadas al cuadrado.

postProcessDistances: este método es llamado después que se han calculado todas las distancias, y se llama con las distancias de los k vecinos mas cercanos para ser procesadas. Como la forma de calcular la función de distancia esta definida con $r = 2$ (distancia

Euclidiana), descrita en el capítulo 1, el método calcula y retorna la raíz cuadrada de cada una de estas distancias.

update: actualiza la clase con la llegada de una nueva instancia. En este caso no es necesario actualizar ninguno de los valores utilizados ni de la matriz de pesos ni de la preprocesada.

getInstances: devuelve el conjunto de instancias pasado en *setInstances*.

Métodos implementados en la clase ConFuCiuS:

Al igual que el modelo Fuzzy-SIAC los datos son preprocesados en la clase y son construidas las matrices de pesos y preprocesada. Muchos de los métodos presentes en la clase *FuzzySIAC* son implementados aquí de la misma manera.

Los métodos *calculateNumberColumns*, *calculateMSDegree*, *getSumNumberColumns*, *getAttributeNumberColumns* fueron implementados igual a los de la clase FuzzySIAC.

calculateValueMSDegree: este método recibe como parámetro un valor y el índice de un atributo. Se encarga de calcular y devolver los valores de pertenencia de este valor a cada uno de los términos lingüísticos asociados al atributo pasado como parámetro.

calculateWeightMatrix: este método se encarga de calcular la matriz de pesos –en la clase FuzzySIAC esto se hace en el método *builClassifier*– lo hace de la misma manera que el modelo Fuzzy-SIAC.

getAttIndex: devuelve el índice de un atributo dado como parámetro en la matriz preprocesada.

difference: calcula la diferencia entre los valores de un atributo determinado de dos instancias dadas como parámetro también. Aquí son usadas las matrices de peso y preprocesadas esta diferencia es calculada por la ecuación (6) del epígrafe 1.3.3.

VDMPrueba: constructor de la clase aquí son calculadas las matrices de pesos y preprocesadas con el conjunto de datos pasado como parámetro.

3.4 Validación de los algoritmos implementados

Una vez implementados en Weka los nuevos algoritmos: Fuzzy-SIAC y ConFuCiuS, estos van a ser evaluados con los conjuntos de datos internacionalmente reconocidos de la UCIMLR (Murphi y Aha) cuyo listado especificando sus características aparecen en el anexo # 3. Nótese que es una muestra representativa de conjuntos de casos a considerar, pues estos difieren en características tales como: ausencia de información en los datos, cantidad de ejemplos, cantidad de atributos y naturaleza de los mismos, así como cantidad de clases. La entrada de datos solo requirió definir un nuevo fichero de entrada (con extensión *arff*) para aquellos conjuntos de datos donde se utilizan atributos numéricos, especificando ahora que estos atributos son de tipo LINGUISTIC. Esto constituye otra ventaja para la validación una vez que se tienen los algoritmos implementados en Weka, pues la entrada de datos ya está definida y además se disponen de los ficheros en el formato que lo requiere la herramienta para la mayoría de los archivos que se utilizan. Específicamente, en los experimentos cuyos resultados se muestran a continuación, se utilizaron funciones de pertenencia trapezoidales.

Se le aplica a cada archivo de datos una validación cruzada estratificada con 10 particiones. Se toma como desempeño para un algoritmo con un conjunto de datos el valor que reporta esta herramienta referido al porciento de instancias correctamente clasificadas con este conjunto de datos. Nótese que este procedimiento, imprescindible en la validación de un algoritmo, ya está implementado en la herramienta y sólo fue necesario definir la variante utilizar de las opciones que se disponen.

Posteriormente, se realiza un análisis estadístico de estos resultados, para hacer conclusiones con la muestra de archivos de datos utilizados sobre la factibilidad de los nuevos algoritmos que se utilizan en la extensión al *modelo original*. Este problema, desde el punto de vista estadístico con la finalidad explicada, se puede modelar como una comparación de poblaciones de muestras apareadas, donde se compara un variable continua: desempeño del algoritmo, considerando dos momentos diferentes en correspondencia con los algoritmos a comparar: el que se está validando y el que se toma como referencia. Desde el punto de vista estadístico se formula la siguiente hipótesis:

H_o: Existe homogeneidad de los rangos medios

Si la significación de la prueba realizada es menor 0.05 (valor para la significación que se toma en este caso), se rechaza H_o con un 5% de error, es decir se considera que los rangos medios de la variable a comparar muestra diferencias significativas. En caso contrario, no se tienen elementos para rechazar H_o, y se concluye que no hay diferencias significativas.

Para probar esta hipótesis nos auxiliaremos del paquete estadístico *SPSS* (versión 9.0). Las pruebas a realizar son no paramétricas considerando el tamaño de la muestra, y en particular se utiliza la prueba de rangos de "*Wilcoxon"*.

3.4.1 Validación de Fuzzy-SIAC

Atendiendo a que es posible que en los datos aparezca ausencia de información, primeramente se procedió a determinar si el desempeño del nuevo algoritmo de RNA es sensible al tratamiento de la información ausente como se explicó anteriormente, o es posible no especificar ningún tratamiento al respecto dada la ventaja de las RNA de ser resistentes a los fallos. La tabla siguiente, que se refiere solo a los conjuntos de datos con valores ausentes, muestra los resultados de Fuzzy-SIAC con estas variantes. La primera columna se refiere a no haber considerado ningún tratamiento para los valores ausentes.

Nombres de los conjuntos de datos	*Fuzzy-SIAC(0)*	*Fuzzy-SIAC(1)*
23- credit-a	73.04%	**73.33%**
24-hepatitis	79.35%	79.35%
25- anneal	76.17%	76.17%
26- labor	64.91%	64.91%
27- sick	93.89%	93.89%
28- sick-euthyroid	90.74%	90.74%
29- colic	66.03%	**66.30%**
30- allbp	95.25%	95.25%
31- allhyper	97.25%	97.25%
32- allhypo	92.14%	92.14%
33- allrep	96.89%	96.89%
Promedio	84.15%	84.20%

Tabla 1: Resultados del desempeño de Fuzzy-SIAC con y sin tratamiento de los valores ausentes

El análisis de la comparación de la variable desempeño para estas dos variantes muestra que no hay diferencias significativas (significación: 0.18) en el desempeño de la RNA tratando los valores ausentes de la manera especificada o no tratándolos. Esto denota que las RNA son resistentes a los ruidos, y por tanto la RNA para su entrenamiento solo considerará la información disponible en ese ejemplo, sin asumir ningún valor para completar la información ausente ni siquiera después del preprocesamiento.

La tabla siguiente muestra los resultados obtenidos por el nuevo algoritmo (Fuzzy-SIAC), y la RNA utilizada del *modelo original* (SIAC). Nótese que solamente se consideraron archivos de datos con rasgos numéricos, ya que la extensión está dada por preprocesar este tipo de rasgo de una manera diferente, lo cual debe influir también en su desempeño. Tener este algoritmo implementado en Weka propició que los resultados del *modelo original* se tuviesen a partir de realizar un procesamiento diferente utilizando el filtro que implementa el "Equal Width" implementado ejemplificando la metodología, con el mismo algoritmo Fuzzy-SIAC.

Nombres de conjuntos de datos	*SIAC*	*Fuzzy-SIAC(0)*
1- Iris	**94.00%**	92.00%
2- pima-diabetes	**66.15%**	65.10%
3- glass	**64.02%**	54.67%
4- liverDBupa	**59.13%**	58.55%
5- vehicle	**54.73%**	54.26%
6- wine	97.19%	**97.75%**
7- wbc	**96.49%**	95.61%
8- ionosphere	74.64%	**78.92%**
9- sonar	**72.12%**	67.31%
10- vowel	61.11%	**64.14%**
11- segment	**80.30%**	78.27%
22- zoo	61.39%	61.39%
23- credit-a	71.30%	**73.04%**
24-hepatitis	79.35%	79.35%
25- anneal	76.17%	76.17%
26- labor	**91.23%**	64.91%
27- sick	93.89%	93.89%
28- sick-euthyroid	90.74%	90.74%
29- colic	66.30%	66.03%
30- allbp	95.25%	95.25%
31- allhyper	97.25%	97.25%
32- allhypo	92.14%	92.14%
33- allrep	96.89%	96.89%
Promedio	**79.64%**	77.98%

Tabla 2: Resultados del desempeño de SIAC y Fuzzy-SIAC, con archivos de datos que contienen rasgos numéricos

Aunque el modelo SIAC muestra un promedio del desempeño más alto, el análisis estadístico de la comparación de estos dos modelos muestra que no hay diferencias significativas (significación: 0.17). Esto permite concluir que el desempeño de la RNA del

modelo original (SIAC) y el de su implementación borrosa (Fuzzy-SIAC), para todos los archivos de datos en general utilizados es similar.

3.4.2 Validación de ConFuCiuS

Considerando que el criterio de similitud local que se emplea en este modelo utiliza los pesos de la RNA validada anteriormente, el primer experimento está orientado a validar el efecto que traería en el desempeño de este modelo las dos variantes explicadas para el tratamiento de los valores ausentes a la hora de entrenar la RNA, a partir de los ejemplos del conjunto de entrenamiento. En todos los casos sí se adopta este criterio para manejar los valores ausentes en el módulo basado en casos, que se resume en asumir 1 como valor de pertenencia del valor ausente a los valores definidos para representar ese atributo en la RNA (valores representativos). La primera columna de la siguiente tabla se refiere al desempeño de ConFuCiuS sin tratar los valores ausentes a la hora de entrenar la RNA.

Nombres de conjuntos de datos	*ConFuCiuS(0)*	*ConFuCiuS (1)*
19- audiology	**75.22%**	73.89%
20- soybean	93.12%	**93.85%**
21- lung-cancer	46.88%	**50.00%**
23- credit-a	83.33%	**83.77%**
24-hepatitis	**81.29%**	79.35%
25- anneal	**98.44%**	97.33%
26- labor	**78.95%**	77.19%
27- sick	96.36%	**96.43%**
28- sick-euthyroid	94.72%	**95.00%**
29- colic	**76.36%**	75%
30- allbp	95.93%	**96.14%**
31- allhyper	97.86%	**97.93%**
32- allhypo	**96.32%**	93.46%
33- allrep	**97.93%**	97.68%
Promedio	**86.62%**	86.22%

Tabla 3: Resultados del desempeño de ConFuCiuS sin o con tratamiento de valores ausentes, con archivos de datos con información ausente

La comparación de estos resultados muestra que no hay diferencias significativas en el desempeño de ConFuCiuS, considerando las dos variantes explicadas para el tratamiento de los valores ausentes antes de obtener los pesos de la RNA, que este modelo utiliza en la definición del criterio para comparar dos valores de un rasgo.

La tabla que se muestra a continuación refleja los desempeños obtenidos aplicando el resolvedor de problemas que emplea el *modelo original* (SIAC) y ConFuCiuS, para todos los archivos de datos considerados.

Nombres de conjuntos de datos	*SIAC*	*ConFuCiuS*
1- Iris	94.00%	**96.00%**
2- pima-diabetes	66.15%	**75.53%**
3- glass	64.02%	**69.16%**
4- liverDBupa	59.13%	**60.87%**
5- vehicle	54.73%	**66.78%**
6- wine	**97.19%**	95.51%
7- wbc	96.49%	**96.71%**
8- ionosphere	74.64%	**90.60%**
9- sonar	72.12%	**85.10%**
10- vowel	61.11%	**95.56%**
11- segment	80.30%	**95.93%**
12- kr-vs-kp	61.17%	**96.59%**
13- hayes-roth	75.00%	**78.79%**
14- contact-lenses	62.50%	**75.00%**
15- monks-1	75.00%	**80.65%**
16- monks-2	**62.13%**	56.80%
17- monks-3	**93.44%**	90.98%
18- tic-tac-toe	65.34%	**87.47%**
19- audiology	25.22%	**73.89%**
20- soybean	54.03%	**93.85%**
21- lung-cancer	25.22%	**50.00%**
22- zoo	61.39%	**93.07%**
23- credit-a	71.30%	**83.77%**
24-hepatitis	79.35%	79.35%
25- anneal	76.17%	**97.33%**

26- labor	**91.23%**	77.19%
27- sick	93.89%	**96.43%**
28- sick-euthyroid	90.74%	**95.00%**
29- colic	66.30%	**75%**
30- allbp	95.25%	**96.14%**
31- allhyper	97.25%	**97.93%**
32- allhypo	92.14%	**93.46%**
33- allrep	96.89%	**97.68%**
Promedio	73.66%	**84.67%**

Tabla 4: Resultados del desempeño de ConFuCiuS y del clasificador del *modelo original* con todos los archivos de datos

Los resultados estadísticos de la comparación (significación 0.00) permite concluir que hay diferencias altamente significativas entre el desempeño de ConFuCiuS, incluso considerando todos los atributos con igual importancia, y el de SIAC. Luego, atendiendo al análisis del epígrafe anterior, se tendría un resultado similar de la comparación de los desempeños de ConFuCiuS y de Fuzzy-SIAC. Nótese que el desempeño promedio es mayor con ConFuCiuS, que tiene mejor en 28 de 32 conjuntos de datos en que se diferencian. Esto muestra la superioridad del nuevo algoritmo de clasificación, donde el CBR actúa como resolvedor de problemas y no una RNA como en el *modelo original*.

Además resulta muy conveniente comparar este nuevo enfoque con otros similares ya reportados en la bibliografía, y en este sentido tener ConFuCiuS implementado en Weka facilita este fin. La herramienta tiene incorporados muchos algoritmos, y por tanto sin esfuerzos adicionales de programación se obtuvo el desempeño del *k*-NN estándar con estos mismos conjuntos de datos, mostrando los resultados en la tabla siguiente.

Nombres de conjuntos de datos	*ConFuCiuS*	*k-NN*
1- Iris	**96.00%**	95.33%
2- pima-diabetes	**75.53%**	72.66%
3- glass	69.16%	**71.96%**
4- liverDBupa	60.87%	**61.74%**
5- vehicle	66.78%	**71.51%**
6- wine	**95.51%**	94.94%
7- wbc	**96.71%**	96.93%
8- ionosphere	**90.60%**	86.61%
9- sonar	85.10%	**86.06%**
10- vowel	95.56%	**97.07%**
11- segment	95.93%	**96.02%**
12- kr-vs-kp	**96.59%**	96.50%
13- hayes-roth	**78.79%**	60.61%
14- contact-lenses	75.00%	**79.17%**
15- monks-1	**80.65%**	78.23%
16- monks-2	**56.80%**	53.25%
17- monks-3	**90.98%**	82.79%
18- tic-tac-toe	87.47%	**98.75%**
19- audiology	**73.89%**	68.58%
20- soybean	**93.85%**	91.36%
21- lung-cancer	**50.00%**	43.75%
22- zoo	**93.07%**	92.08%
23- credit-a	**83.77%**	84.64%
24-hepatitis	79.35%	**81.29%**
25- anneal	**97.33%**	93.43%
26- labor	77.19%	**91.23%**
27- sick	**96.43%**	95.64%
28- sick-euthyroid	**95.00%**	92.38%
29- colic	**75.00%**	64.13%
30- allbp	96.14%	**96.29%**
31- allhyper	97.93%	97.93%
32- allhypo	**93.46%**	92.93%
33- allrep	**97.68%**	97.07%
Promedio	**84.67%**	83.72%

Tabla 5: Resultados del desempeño de ConFuCiuS y del *k*-NN estándar con todos los archivos de datos

El análisis estadístico de los resultados muestra que no hay diferencias significativas entre el desempeño de ConFuCiuS y del *k*-NN estándar (significación 0.228), aunque el desempeño promedio es mayor con ConFuCiuS, que además resulta mejor con el 66% de los archivos de datos utilizados (21 de 32). Estos resultados muestran la factibilidad del enfoque propuesto, que además utiliza un criterio de comparación a nivel de atributo general sin importar la naturaleza simbólica o numérica de éste; y que se define automáticamente a partir de ejemplos, minimizando así la ingeniería del conocimiento requerida en estos sistemas. Adicionalmente, un análisis más detallado por tipo de conjuntos de datos se muestra a continuación:

- Conjunto de datos Tipo A: los que tienen atributos numéricos sin valores ausentes (filas 1-11 tabla 5). No arroja diferencias significativas (significación 0.676). Además, ambos algoritmos tienen un comportamiento muy similar: el desempeño con ConFuCiuS resulta mejor con el 45% de los archivos de datos utilizados (5 de 11).

- Conjunto de datos Tipo B: los que tienen atributos simbólicos sin valores ausentes (filas 12-18 tabla 5). No arroja diferencias significativas (significación 0.446). No obstante, el desempeño promedio es mayor con ConFuCiuS (81%) que con k-NN estándar (78%); obteniendo un mejor desempeño ConFuCiuS en el 71% de los archivos de datos utilizados (5 de 7).

- Conjunto de datos Tipo C: los que tienen atributos simbólicos y numéricos (mezclados) y/o valores ausentes (filas 19-33 tabla 5). Arroja diferencias medianamente significativas (significación 0.182). El desempeño promedio es mayor con ConFuCiuS (86.6%) que con k-NN estándar (85.5%); obteniendo un mejor desempeño ConFuCiuS en el 73% de los archivos de datos utilizados (11 de 15).

3.5 Desarrollando aplicaciones a la medida del usuario

Al ser Weka un sistema de código abierto, una de sus grandes ventajas es facilitar el desarrollo de sistemas reutilizando clases ya implementadas en esta herramienta. Luego que para un juego de datos se han probado los diferentes algoritmos y variantes de preprocesamiento de los datos disponibles, con la configuración seleccionada se puede desarrollar un software a la medida del usuario. Esto significa que reutilizando el código implementado se desarrolla un software, que independiza al usuario final de esta herramienta. En este epígrafe se ejemplifica lo anterior mediante un sistema para clasificar flores, que se entrena a partir del archivo de datos *Iris.data* y utiliza los nuevos algoritmos que se añadieron a Weka como resultado del presente trabajo.

Este sistema permite clasificar una flor en tres clases: Iris Setosa, Iris Versicolor e Iris Virginica. La entrada al mismo son los datos que se miden en el conjunto de datos original: ancho del pétalo (petal width), largo del pétalo (petal length), ancho del sépalo (sepal width) y largo del sépalo (sepal length), todos rasgos numéricos.

El sistema lo constituye una única ventana con cuatro cuadros de texto para recoger los datos de entrada y un botón para la acción de clasificación. Se imprime en la ventana la clasificación de la flor entrada en una de las tres clases según el criterio de los dos algoritmos utilizados.

El desarrollo de esta aplicación es una tarea sencilla, solo debe ser implementada la interfaz visual a utilizar, para la captura de datos y mostrar los resultados.

La implementación de la aplicación consiste en una sola clase llamada *IrisGUI* que construye la interfaz gráfica que recibe el mismo nombre de la clase e invoca los métodos necesarios para la clasificación. El software se apoya en las bibliotecas de clases implementadas en Weka, usando solamente las clases necesarias y tomando las facilidades para la lectura de ficheros o los cálculos estadísticos que se realizan.

Primeramente el clasificador es entrenado en el constructor de la clase, aquí se construye una instancia de la clase *Instances,* necesitando solamente pasar el fichero donde se

encuentre el conjunto de datos. El constructor de *Instances* se encarga de leer los datos del fichero y almacenarlos en el objeto *Instances*. Seguidamente se utiliza este objeto para entrenar los dos algoritmos a utilizar, Fuzzy-SIAC y ConFuCiuS, llamando al método *builClassifier* y pasándole el conjunto de entrenamiento ya construido. En el caso de ConFuCiuS lo que se hace es pasar al clasificador *IBk* la función de distancia *FuzzyVDM* para que no use *EuclidianDistance* y después se llama al *buildClassifier* de la clase *IBk*.

Después del entrenamiento el clasificador está listo para clasificar la nueva instancia. Se toman los valores entrados mediante la interfaz visual y se construye un objeto de la clase *Instances* pasándole estos valores. El objeto construido es pasado al método *classifyInstances* de las clases *FuzzySIAC* e *IBk*, que retorna el valor de la clase en que se clasifica la instancia.

La ventaja que ofrece Weka para el desarrollo de este tipo de aplicaciones es fundamentalmente la rapidez con que pueden ser implementadas. Las desventajas del clasificador de flores Iris creado son las desventajas propias del lenguaje Java, ya que la reutilización de código Weka implica que la implementación de sistemas de este tipo deba ser implementada en este lenguaje.

3.6 Factibilidad de la implementación de nuevos modelos en Weka

La extensibilidad de Weka hace que sea más factible adicionar un nuevo algoritmo a esta herramienta, que no implementarlo como un programa independiente. Algunas razones son:

- La implementación de un algoritmo se simplifica utilizando Weka. Implementar un nuevo modelo consiste en redefinir métodos ya existentes en Weka o crear otros nuevos con las funcionalidades específicas del algoritmo a adicionar.

- No se requiere programar la interfaz gráfica de usuario para utilizar el nuevo algoritmo.

- No se necesita programar lo referente a la entrada de los datos. Se reutilizaría la manera de hacerlo en Weka, que está independiente a la implementación de los algoritmos.

- Se facilita la validación del algoritmo implementado. No es necesario preocuparse por implementar las variantes de validación, ni las medidas de desempeño a emplear; se reutilizan las existentes en la herramienta.

- Se propicia y facilita la comparación del nuevo algoritmo con otros ya reportados en la literatura e implementados en la herramienta, facilitando el análisis de la factibilidad de este último, algo que sería más costoso en tiempo si se hubiera implementado como un modelo aislado.

- Se facilita la generalización de un nuevo algoritmo. Por ejemplo, un nuevo criterio para calcular distancia pudiera ser fácilmente validado con los algoritmos implementados.

- Facilita probar un conjunto de datos con todos los algoritmos disponibles en la herramienta a los efectos de seleccionar el adecuado.

- El tiempo de desarrollo del prototipo de un software a la medida utilizando un algoritmo implementado en Weka disminuye a partir de reutilizar su código.

- Es posible hacer corridas en lotes y en varias terminales, sin esfuerzos adicionales de programación. Utilizando el modo Experimentador que Weka tiene implementado se pueden realizar experimentos con varios modelos y conjuntos de datos a la vez, utilizando varias terminales; lo cual es muy recomendable para algoritmos que consuman cantidad de tiempo de corrida. Los resultados se almacenan en ficheros, para su posterior análisis estadístico.

- Se propicia el uso y divulgación de los nuevos modelos implementados. El hecho de queden incorporados a Weka los hace disponibles para la comunidad de científicos y usuarios de este campo.

- Facilita el preprocesamiento de los datos. Un algoritmo implementado en Weka pudiera requerir previamente una transformación a los datos originales almacenados en el fichero con extensión *arff*. La herramienta cuenta con una serie de algoritmos de preprocesamiento (filtros) implementados a tales efectos.

Nótese que el hecho de que los filtros estén separados de los algoritmos que usan los datos es una ventaja. De esta manera, se facilita la implementación de un nuevo modelo, si este requiere realizar un tipo de preprocesamiento ya implementado entre los tantos filtros disponibles en la herramienta.

El procesamiento de los datos utilizando filtros se puede combinar con un algoritmo de aprendizaje que los use dentro de Weka de dos maneras diferentes. La primera de ellas sería aplicar el filtro sobre todo el conjunto de datos y luego aplicar el algoritmo (ejemplo, discretizar los atributos numéricos antes de construir un árbol de decisión utilizando el Id3). La otra alternativa es utilizar los filtros dentro de la implementación de un algoritmo obedeciendo a exigencias de este (ejemplo: reemplazar valores ausentes por una constante).

CONCLUSIONES

Con la realización de este trabajo se incorporaron nuevas posibilidades de uso y funcionalidades al ambiente de Aprendizaje Automatizado Weka, como resultado de la implementación a este ambiente de dos nuevos algoritmos que se utilizan en la extensión de un modelo híbrido de Razonamiento Basado en Casos que se trabaja en el grupo de Inteligencia Artificial de la UCLV, lográndose así dar cumplimiento al objetivo planteado a partir de que:

- Se realizó un estudio de las facilidades que brinda esta herramienta a nivel de usuario y de la implementación en Java de su código fuente, comprobando que Weka brinda: un gran número de algoritmos conocidos, varias formas para preprocesar los archivos de datos a utilizar por tales algoritmos, así como facilidades para validar los mismos. Además, el diseño e implementación de las clases que componen el sistema se conciben para que su extensión no sea una tarea compleja.

- Se incorporó un nuevo tipo de dato que maneja un atributo numérico como variable lingüística; realizando las implementaciones y modificaciones requeridas para definir un atributo de este tipo en el fichero con extensión *arff* utilizando tres funciones de pertenencia: Triangular, Trapezoidal y Gaussiana. Esto propicia un pre-procesamiento de los datos siguiendo el mismo diseño que establece Weka con esta finalidad.

- Se propuso una metodología para la implementación de nuevos filtros y clasificadores en Weka, mostrando las facilidades de extender esta herramienta. Esta metodología se ejemplificó mediante la implementación de un filtro para discretizar un rasgo en una cantidad especificada de intervalos de igual amplitud y un algoritmo de clasificación basado en un Perceptron Simple.

- Se implementaron en Weka, aplicando la metodología propuesta, un nuevo filtro: *EqualWidthFuzzifier* y dos nuevos algoritmos de clasificación: Fuzzy-SIAC y

ConFuCiuS; con las ventajas que trae consigo tener implementados los algoritmos en esta herramienta. El filtro permite manejar un atributo numérico como variable lingüística, transformando el valor que toma un rasgo de este tipo en una instancia a grados de pertenencia a los términos lingüísticos definidos; como requieren los algoritmos que usan conjuntos borrosos. Por otro lado, la incorporación al Weka de estos algoritmos facilita la validación de las extensiones propuestas al modelo híbrido de Razonamiento Basado en Casos referido.

- Se validó el desempeño de los dos nuevos algoritmos de clasificación implementados con juegos de datos internacionalmente reconocidos, mostrando las facilidades que brinda Weka con esta finalidad. El análisis estadístico de los resultados muestra que Fuzzy-SIAC y SIAC no tienen diferencias significativas en el desempeño; mientras que Fuzzy-SIAC y ConFuCiuS sí se diferencian significativamente en su desempeño a favor de este último modelo, incluso considerando todos los atributos con igual importancia. Además el uso de Weka facilitó la comparación con otros modelos ya existentes implementados en la herramienta, como por ejemplo con el *k*-NN estándar, mostrando que: no hay diferencias significativas entre el desempeño de ConFuCiuS y del *k*-NN estándar; aunque el desempeño promedio es mayor con ConFuCiuS, que además tiene mejor desempeño con los archivos de datos que manejan atributos solamente simbólicos, o de naturaleza diferente (atributos mezclados) con ausencia de información.

- Se desarrolló un software para la clasificación de tipos de flores reutilizando el código de los dos modelos implementados en Weka. Se mostró que el esfuerzo requerido, una vez que se tiene el algoritmo implementado en la herramienta, para obtener un software a la medida que independice al usuario final de Weka es mínimo.

Se muestra que Weka es una herramienta orientada a la extensibilidad; cuya estructura hace que añadir, eliminar o modificar elementos no sea una tarea compleja; y que brinda facilidades para la validación de nuevos algoritmos. Como resultado quedan a disposición de los científicos de este campo los nuevos modelos y funcionalidades incorporados al

ambiente, lo que contribuye además a la divulgación y aplicación de las nuevas extensiones al modelo híbrido de Razonamiento Basado en Casos implementado en SISI.

RECOMENDACIONES

- Cambiar la visualización que ofrece Weka para mostrar la relación entre atributos en el panel “Preprocess” del modo Explorador para el tipo de rasgo lingüístico.

- Adicionar a la implementación del modelo ConFuCiuS el método de pesaje de rasgos.

- Agregar un nuevo algoritmo de filtrado supervisado para convertir los rasgos numéricos a lingüísticos.

REFERENCIAS BIBLIOGRAFICAS

Agrawal R. y R. Srikant, (1994). "Fast algorithms for mining association rules in large databases". Proc International Conference on Very Large Databases, pp. 478-499. Santiage, Chile: Morgan Kaufmann, Los Altos, CA.

Aha, D. y D. Kibler, (1991). "Instance-based learning algorithms", Machine Learning, 6, pp. 37-66.

Breiman, L., (1996). "Bagging predictors", Machine Learning, 24, pp 123-140.

Duda, R y P. Hart, (1973). *Pattern Classification and Scene Analysis*. Wiley, New York.

Flach, P. A. y N. Lachiche, (1999). "Confirmation-Guided Discovery of first-order rules with Tertius". Machine Learning, 42, pp. 61-95.

Freund, Y. y R. E. Schapire, (1998). "Large margin classification using the perceptron algorithm". Proc. 11th Annu. Conf. on Comput. Learning Theory, pp. 209-217, ACM Press, New York, NY.

Friedman, J.H., (1999). "Stochastic Gradient Boosting". Technical Report Stanford University, disponible en: http://www-stat.stanford.edu/~jhf/ftp/stobst.ps.

García, M. M. y R. E. Bello, (1996). "A model and its different applications to case-based reasoning" en *Knowledge-based systems.* 9, pp 465-473.

21 García, M. M., Rodríguez, Y. y R. Bello, (2000). "Usando conjuntos borrosos para implementar un modelo para sistemas basados en casos interpretativos". In Proceedings of IBERAMIA-SBIA. Eds por M. C. Monard y J.S. Sichman, Sao Paulo, Brasil

Hall, M. A., (1998). "Correlation-based Feature Subset Selection for Machine Learning". Thesis submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy at the University of Waikato.

Hilera, J. R. y V. J. Martínez, (1995). *Redes Neuronales Artificiales. Fundamentos, modelos y aplicaciones*, Madrid, España, Addison-Wesley.

Holte, R.C., (1993). "Very simple classification rules perform well on most commonly used datasets". Machine Learning, Vol. 11, pp. 63-91.

Jang, J.S.R., C.T. Sun y E. Mizutani, (1997). *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice-Hall

Kohavi, R. (1995a). "The Power of Decision Tables". In Proc European Conference on Machine Learning.

Kohavi, R., (1995b). "A study of cross-validation and bootstrap for accuracy estimation and model selection," International Joint Conference on Artificial Intelligence (IJCAI).

Kurgan L.; et. al., (2004). "CAIM Discretization Algorithm". IEEE Transactions on Knowledge and Data Engineering. Vol 16. No. 2.

Len, C.T. y C.S, Lee, (1999). *Neural Fuzzy Systems: A Neuro-Fuzzy Synergism to Intelligent System*. Prentice Hall

Lopes, A.A. y J.Alipio, (2000). *Integrating Rules and Cases in Learning via Case*

Explanation and Paradigm Shift. Springer-Verlag

McClelland, D. y E. Rumelhart, (1989). *Explorations in parallel distributed processing*. MIT Press

Mitchell, T., (1997). *Machine Learning*. McGraw-Hill Science, Engineering, Math. ISBN 0070428077.

Murphi, P.M. y D.W. Aha. UCI Repository of Machine-Learning Database, disponible en: http://www.ics.uci.edu/~mlearn/mlrepository.html

Pal, S.K., Dillon, T.S. y D.S Yeung, (2001). *Soft Computing in Case Based Reasoning*. Springer Verlag London

Quinlan, J.R., (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufman, San Mateo, CA.

Quinlan, J.R., (1986). *Induction of Decision Trees*. Machine Learning. Vol. 1, No.1, pp 81-106.

Rodríguez Y., M. M. García, B.D. Baets, R., Bello, C. Morell, (2006). Extending a Hybrid CBR-ANN Model by Modeling Predictive Attributes using Fuzzy Sets. Congreso Iberoamericano de Inteligencia Artificial (IBERAMIA)

Rosenblatt, F. (1958). "The Perceptron: A probabilistic model for information storage and organization in the brain" en *Psychological Review*, 65, pp 386-408.

Scheffer, T., (2001). Finding Association Rules That Trade Support Optimally against Confidence. Proc of the 5th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD'01), pp. 424-435. Freiburg, Germany: Springer-Verlag.

Sima, J., (1995). Neural Expert System. Neural Networks. Vol 8. No. 2.

Stanfill, C. y D. Waltz, (1986). Toward memory-based reasoning. Comm. of ACM, 29 ,pp1213-1228

Ting, K.M. y I.H. Witten, (1997). "Stacking Bagged and Dagged Models" Proceedings of the Fourteenth international Conference on Machine Learning (July 08 - 12, 1997). D. H. Fisher, Ed. Morgan Kaufmann Publishers, San Francisco, CA, 367-375.

Zadeh, L.A., (1983). A Computational Approach to Fuzzy Quantifiers in Natural Language. Computing and Mathematics with Applications, 9(1), pp 149–184

Zadeh, L.A., (1978). Fuzzy sets as a basis for a theory of possibility, Fuzzy Sets and Systems, 1,pp. 3-28

ANEXOS

Anexo 1. Estructura de paquetes de Weka

- associations (contiene las clases que implementan los métodos para realizar asociaciones.)
  - tertius (contiene las clases para implementar el método tertius)
- attributeSelection
- classifiers (contiene la implementación de los clasificadores existentes en Weka)
  - bayes (clases para implementar clasificadores *bayes*)
    - net (clases en desarrollo para aplicar el clasificador bayes en búsquedas web)
  - evaluation (clases que evalúa los resultados de un clasificador)
  - functions (contiene métodos de clasificación basados en funciones)
    - neural (contiene clases útiles a los calsificadores basados en redes neuronales)
    - pace (métodos numéricos sobre datos -manipulación de matrices, funciones estadísticas-)
    - supportVector (métodos de clasificación basados en máquinas de vectores soporte)
  - lazy (clasificadores perezosos)
    - kstar (métodos necesarios para implementar *Kstar*)
  - meta (metaclasificadores)
  - misc (clasificadores que se encuentren identificados en otras categorías)
  - rules (contiene clases para implementar reglas de decisión)
    - part (clases para implementar reglas sobre estructuras parciales de árboles)
  - tree (contiene métodos para implementar árboles)
    - adtree
    - j48
    - lmt
    - m5
  - xml
- clusterers (Contiene las clases donde se implementa los algoritmos de agrupamiento)
  - forOPTICSAndDBScan
- core (paquete central)
  - converters (clases para convertir entre tipos de archivos y para cargar archivos de forma secuencial)
  - matriz
  - stemmers
  - xml
- datagenerators (clases abstractas para generadores de datos y clases para representar tests.

- Classifiers
  - classification
  - regresión
- clusterers

- estimators (estimadores de probabilidades condicionadas y de probabilidades simples de diferentes distribuciones (Poisson, Normal . . .))
- experiment (clases que implementan la entidad experimento. Así mismo también están las que modelan los experimentos remotos)
  - xml
- filtres (contiene filtros para preprocesar los datos)
  - supervised (filtros supervisados)
    - attribute
    - instante
  - unsupervised (filtros no supervisados)
    - attribute
    - instante
- gui (contiene las clases que implementan las interfaces gráficas)
  - arffviewer
  - beans (clases básicas de Weka para el tratamiento de datos y creación de entidades principales del programa referentes a instancias (DataSource, TestsetListener, TetsetProducer, TrainingSet))
  - boundaryvisualizer (métodos para crear instancias a partir de otras)
  - experiment (interfaz del modo experimentador)
  - explorer (interfaz del modo explorador)
  - graphvisualizer (Clases para dibujar y mostrar gráficas)
  - images (imágenes utilizadas por la aplicacion)
  - sql ()
  - streams (clases para realizar operaciones sobre flujos de datos, que habitualmente será sobre un objeto de clase *Instance*)
  - treevisualizer (clases para elaborar árboles y mostrarlos de una forma gráfica)
  - visualize (contiene clases que implementan los métodos para poder seleccionar atributos en las gráficas y las clases principales para dibujar gráficas (Plot2d, VisualizePanel etc.))

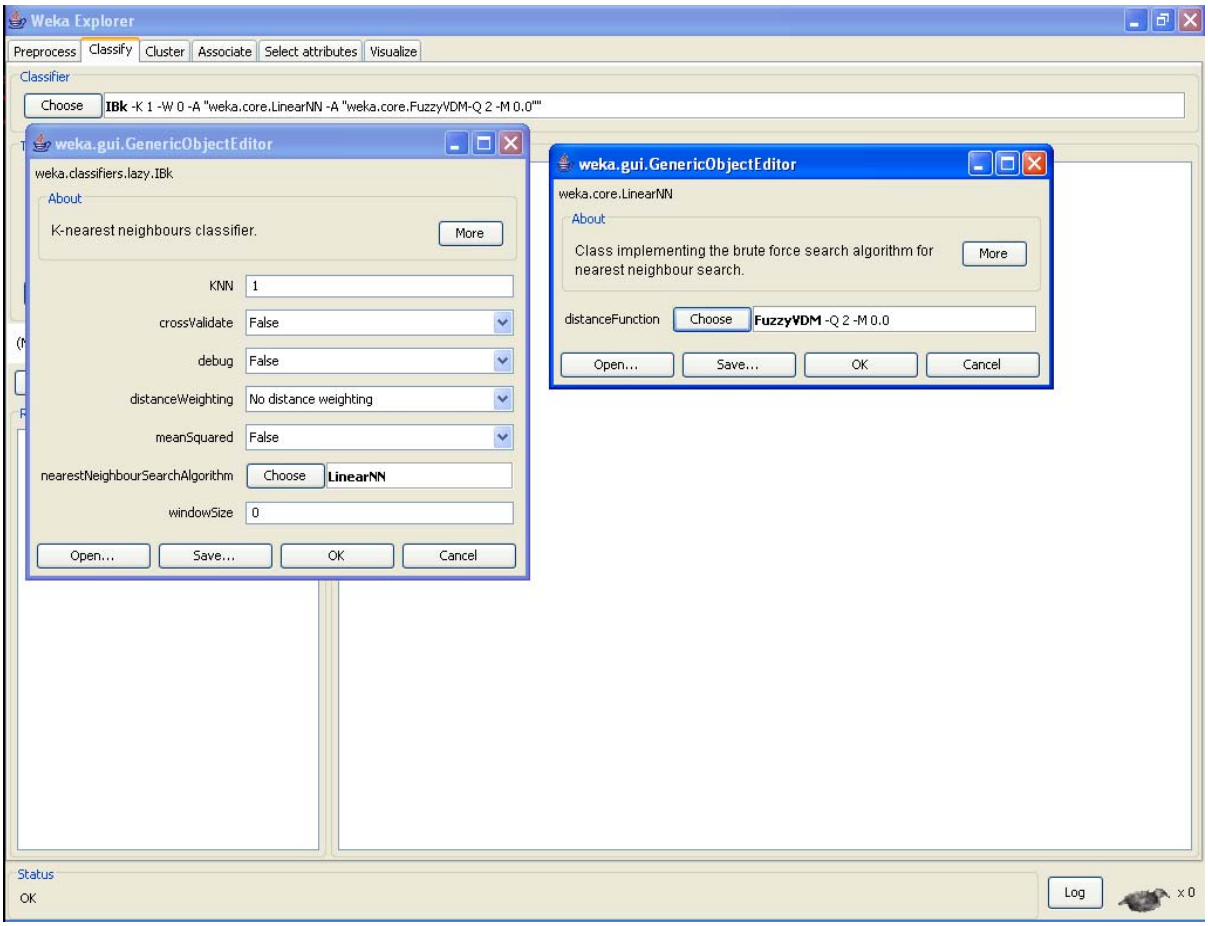
Anexo 2. Ejemplo de un fichero con extensión *arff* que declara atributos lingüísticos

```
@RELATION iris

@ATTRIBUTE sepallength  LINGUISTIC {Gaussian(5,1.6), Gaussian(6.6,1.6),
Gaussian(7.8,1) }
@ATTRIBUTE sepallength  LINGUISTIC {Gaussian(2.6,1.2), Gaussian(3.5,1) }
@ATTRIBUTE sepallength  LINGUISTIC {Gaussian(2,2), Gaussian(5,1.5),
Gaussian(6.3,1) }
@ATTRIBUTE sepallength  LINGUISTIC {Gaussian(0.7,0.6), Gaussian(1.8,0.5),
Gaussian(2,0.8) }
@ATTRIBUTE class  {Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
5.4,3.7,1.5,0.2,Iris-setosa
4.8,3.4,1.6,0.2,Iris-setosa
4.8,3.0,1.4,0.1,Iris-setosa
4.3,3.0,1.1,0.1,Iris-setosa
5.8,4.0,1.2,0.2,Iris-setosa
5.7,4.4,1.5,0.4,Iris-setosa
5.4,3.9,1.3,0.4,Iris-setosa
5.1,3.5,1.4,0.3,Iris-setosa
5.7,3.8,1.7,0.3,Iris-setosa
5.1,3.8,1.5,0.3,Iris-setosa
5.4,3.4,1.7,0.2,Iris-setosa
5.1,3.7,1.5,0.4,Iris-setosa
4.6,3.6,1.0,0.2,Iris-setosa
5.1,3.3,1.7,0.5,Iris-setosa
4.8,3.4,1.9,0.2,Iris-setosa
5.0,3.0,1.6,0.2,Iris-setosa
5.0,3.4,1.6,0.4,Iris-setosa
5.2,3.5,1.5,0.2,Iris-setosa
5.2,3.4,1.4,0.2,Iris-setosa
4.7,3.2,1.6,0.2,Iris-setosa
4.8,3.1,1.6,0.2,Iris-setosa
5.4,3.4,1.5,0.4,Iris-setosa
5.2,4.1,1.5,0.1,Iris-setosa
5.5,4.2,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
....
```

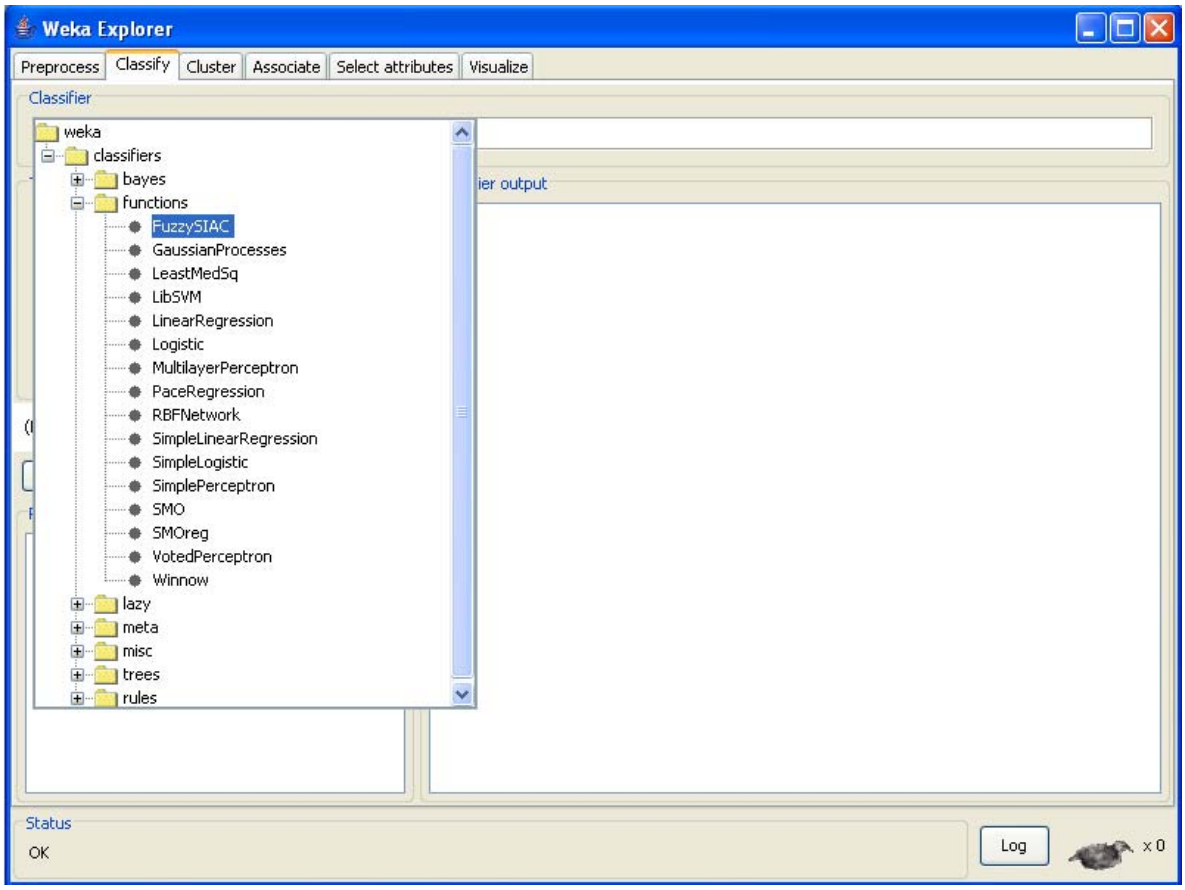
Anexo 3. Características de los ficheros usados en la validación de los algoritmos

Nombres de conjuntos de datos	*Características del conjunto de datos*					
	M.V.	*# Instancias*	*# Atributos*	*# atributos numéricos*	*# atributos simbólicos*	*# clases*
1- Iris	No	150	4	4	0	3
2- diabetes	No	768	8	8	0	2
3- glass	No	214	9	9	0	7
4- liverDBupa	No	345	6	6	0	2
5- vehicle	No	846	18	18	0	4
6- wine	No	178	13	13	0	2
7- wbc	No	683	9	9	0	2
8- inosphere	No	351	34	34	0	2
9- sonar	No	208	60	60	0	2
10- vowel	No	990	13	13	0	11
11- segment	No	2310	19	19	0	7
12- kr-vs-kp	No	3196	36	0	36	2
13- hayes-roth	No	132	4	0	4	3
14- contac-lenses	No	24	4	0	4	3
15- monks-1	No	124	6	0	6	2
16- monks-2	No	169	6	0	6	2
17- monks-3	No	122	6	0	6	2
18- tic-tac-toe	No	958	9	0	9	2
19- audiology	Yes	226	69	0	69	24
20- soybean	Yes	683	35	0	35	19
21-lung-cancer	Yes	32	56	0	56	3
22- zoo	Yes	101	17	1	17	7
23- credit-a	Yes	690	15	6	9	2
24- hepatitis	Yes	155	19	6	13	2
25- anneal	Yes	898	38	6	32	6
26- labor	Yes	57	16	8	8	2
27- sick	Yes	2800	29	7	22	2
28- sick-euthyroid	Yes	3163	25	7	18	2
29- colic	Yes	368	27	7	20	2
30-allbp	Yes	2800	29	7	22	3
31- allhyper	Yes	2800	29	7	22	5
32- allhypo	Yes	2800	29	7	22	5
33- allrep	Yes	2800	29	7	22	4

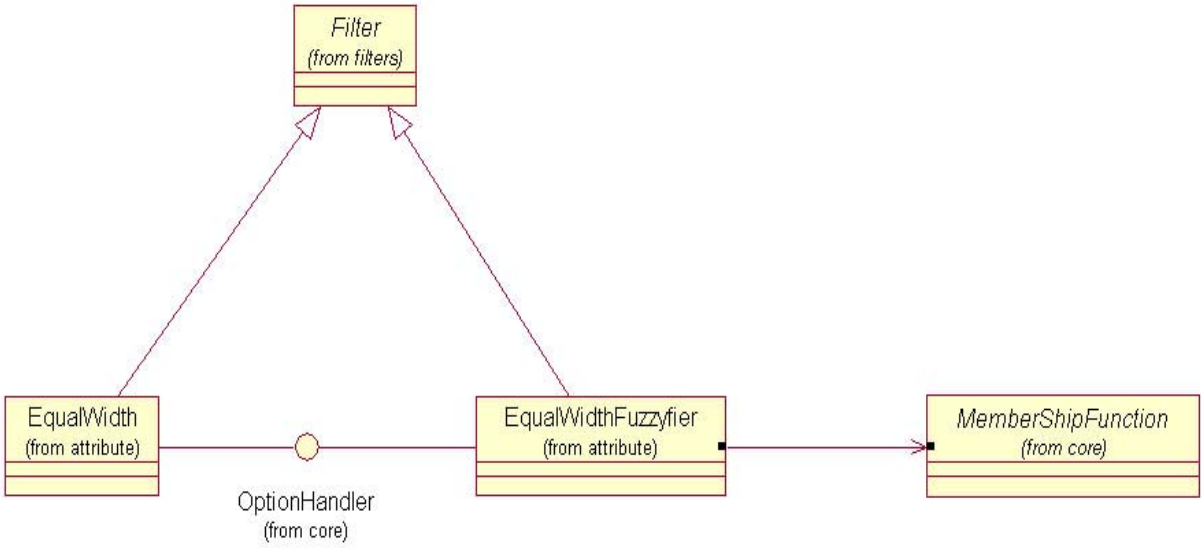
Anexo 4. Ventanas de opciones de la función de distancia FuzzyVDM implementada



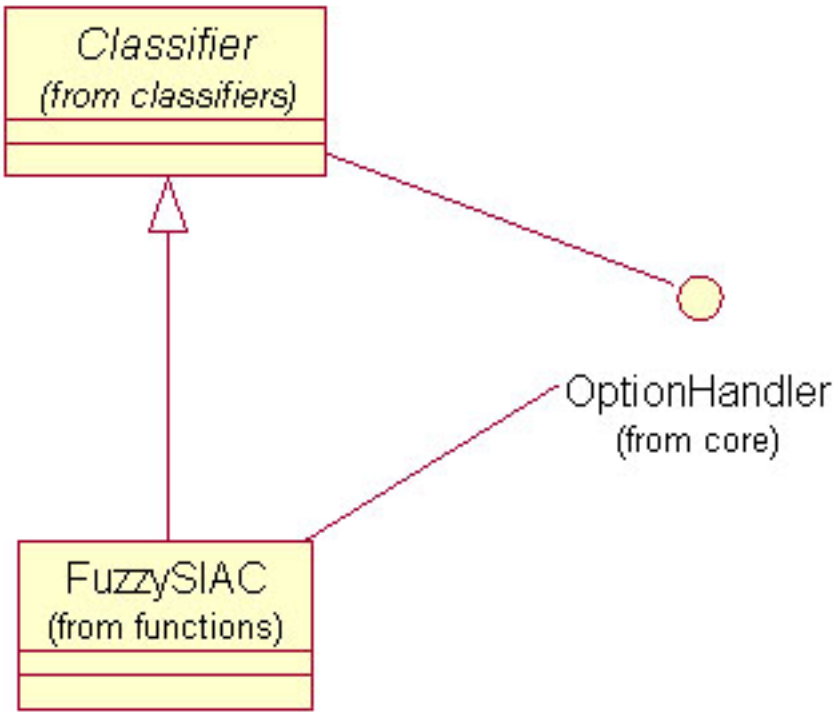
Anexo 5. Ventanas de opciones del clasificador Fuzzy-SIAC



Anexo 6. Diseño de las clases EqualWidth y EqualWidthFuzzyfier



Anexo 7. Diseño de la clase FuzzySIAC



Anexo 8. Diseño de la clase FuzzyVDM