Práctica de Minería de Datos

Introducción al

Curso de Doctorado Extracción Automática de Conocimiento en Bases de Datos e Ingeniería del Software

Universitat Politècnica de València

José Hernández Orallo. **(**jorallo@dsic.upv.es**).** Marzo 2006
Cèsar Ferri Ramírez. **(**cferri@dsic.upv.es**).** Marzo 2006

Índice

En esta práctica se vamos a descubrir paso a paso las posibilidades del Weka utilizando ejemplos muy sencillos y sin realizar validaciones apropiadas de los modelos extraídos. El objetivo de este boletín es manejarse con cierta soltura en el Weka.

1. El entorno de trabajo del Weka

Weka[1] es un conjunto de librerias JAVA para la extracción de conocimientos desde bases de datos. Es un software ha sido desarrollado bajo licencia GPL lo cual ha impulsado que sea una de las suites más utilizadas en el área en los últimos años.

La versión 3.4.7 incluye las siguientes características:

- Diversas fuentes de datos (ASCII, JDBC).
- Interfaz visual basado en procesos/flujos de datos (rutas).
- Distintas herramientas de minería de datos: reglas de asociación (a priori, Tertius, ...), , agrupación/segmentación/conglomerado (Cobweb, EM y k-medias), clasificación (redes neuronales, reglas y árboles de decisión, aprendizaje Bayesiona) y regresión (Regresión lineal, SVM..).
- Manipulación de datos (pick & mix, muestreo, combinación y separación).
- Combinación de modelos (Bagging, Boosting ...)
- Visualización anterior (datos en múltiples gráficas) y posterior (árboles, curvas ROC, curvas de coste..).
- Entorno de experimentos, con la posibilidad de realizar pruebas estadísticas (t-test).

[1] http://www.cs.waikato.ac.nz/~ml/weka/index.html

Figura 1. Ventana inicial de Weka.

Como se puede ver en la parte inferior de la Figura 1, Weka define 4 entornes de trabajo

- *Simple CLI*: Entorno consola para invocar directamente con java a los paquetes de weka
- *Explorer*: Entorno visual que ofrece una interfaz gráfica para el uso de los paquetes
- *Experimenter*: Entorno centrado en la automatización de tareas de manera que se facilite la realización de experimentos a gran escala.
- *KnowledgeFlow*: Permite generar proyectos de minería de datos mediante la generación de flujos de información.

1.1 *Explorer*

En esta parte vamos a centrarnos en el entorno Explorer, ya que permite el acceso a la mayoría de las funcionalidades integradas en Weka de una manera sencilla.

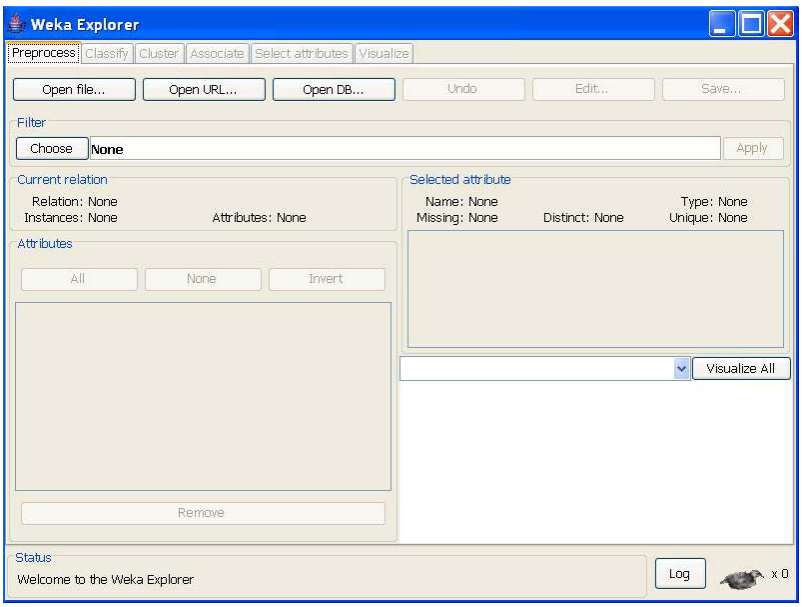

Figura 2. Ventana del Explorador

Como se puede observar existen 6 sub-entornos de ejecución:

- *Preprocess*: Incluye las herramientas y filtros para cargar y manipular los datos
- *Classification*: Acceso a las técnicas de clasificación y regresión
- *Cluster*: Integra varios métodos de agrupamiento
- *Associate*: Incluye una pocas técnicas de reglas de asociación
- *Select Attributes*: Permite aplicar diversas técnicas para la reducción del número de atributos
- *Visualize*: En este apartado podemos estudiar el comportamiento de los datos mediante técnicas de visualización.

2. Un primer ejemplo

Vamos a empezar con un ejemplo muy sencillo. En primer lugar, aparte del Weka, dispones de un fichero zip disponible en www.dsic.upv.es/~cferri/weka con todos los ficheros de datos necesarios para realizar este boletín. En particular, en este primer ejemplo, vamos a trabajar con los datos acerca de los días que se ha podido jugar al tenis, dependiendo de diversos aspectos meteorológicos. El objetivo es poder determinar (predecir) si hoy podremos jugar al tenis. Los datos de que disponemos están en el fichero: weather.arff" y son los siguientes:

```
Sky         Temperature  Humidity  Wind      PlayTennis
Sunny       Hot          High      Weak      No
Sunny       Hot          High      Strong    No
Overcast    Hot          High      Weak      Yes
Rain        Mild         High      Weak      Yes
Rain        Cool         Normal    Weak      Yes
Rain        Cool         Normal    Strong    No
Overcast    Cool         Normal    Strong    Yes
Sunny       Mild         High      Weak      No
Sunny       Cool         Normal    Weak      Yes
Rain        Mild         Normal    Weak      Yes
Sunny       Mild         Normal    Strong    Yes
Overcast    Mild         High      Strong    Yes
Overcast    Hot          Normal    Weak      Yes
Rain        Mild         High      Strong    No
```

Abrimos el Weka y lanzamos el Explorer. Lo primero que vamos a hacer es cargar los datos en el área de trabajo. Para ello, pincha en el botón "Open file" del entorno "preprocess". Seleccionamos el fichero "weather.arff" y si todo ha ido bien veremos la pantalla de la Figura 3. Weka utiliza un formato específico de datos, el formato arff. Un fichero con este formato, no sólo contiene los datos desde donde vamos a efectuar el aprendizaje, además incluye meta-información sobre los propios datos, como por ejemplo el nombre y tipo de cada atributo, así como una descripción textual del origen de los datos. Podemos convertir ficheros en texto conteniendo un registro por línea y con los atributos separados con comas (formato csv) a ficheror arff mediante el uso de un filtro convertidor.

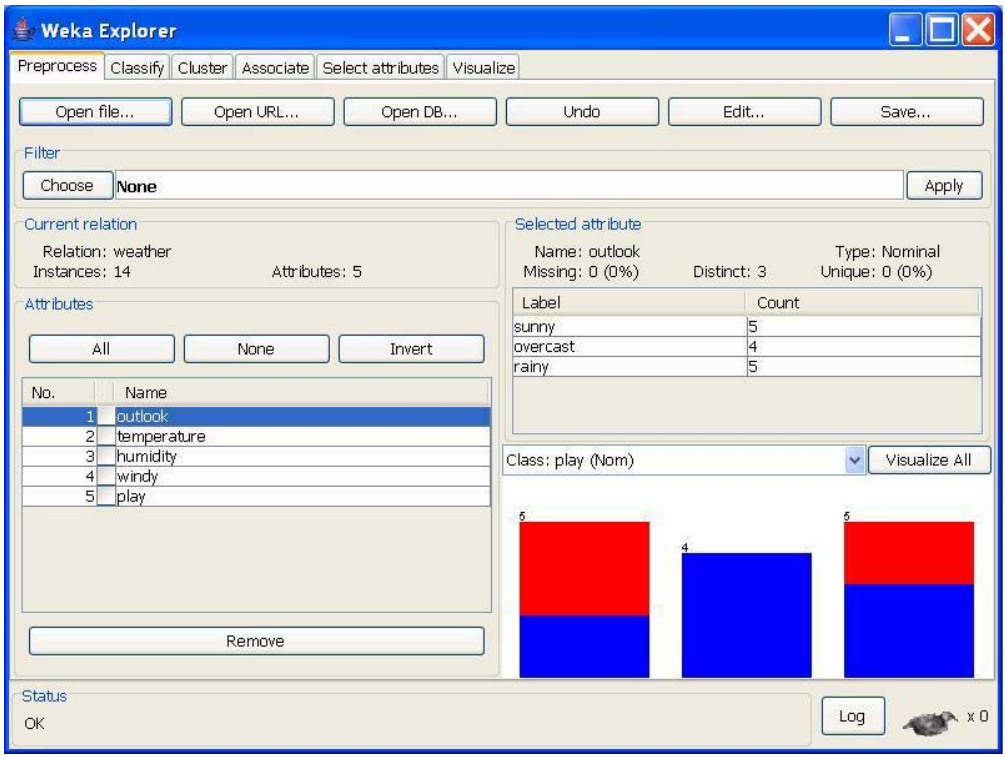

Figura 3. Cargando el dataset Weather

Desde esta ventana podemos conocer bastantes detalles del dataset que acabamos de cargar. Por ejemplo, el sistema nos indica que tenemos 14 registros con 5 atributos. Si seleccionamos cada uno de los atributos, conoceremos más información del atributo en cuestión: tipo(nominal o numérico), valores distintos, registros que no tienen información de ese atributo, el valor máximo y mínimo (sólo en atributos numéricos), y finalmente un histograma con información sobre la distribución de los ejemplos para ese atributo, reflejando con el uso de colores la distribución de clases de cada uno de los registros. Por ejemplo, en la Figura 3 podemos observar que el atributo Outlook tiene tres valores diferentes (*Sunny, Overcast y Rainy*) siendo la distribución de [5,4,5]. En el caso de los 5 registros donde el atributo *Outlook=sunny*, tenemos 3 con clase *no* y 2 con clase *yes*[2], cuando *Outlook=overcast* los 4 registros son *yes*, y finalmente cuando *Outlook=rainy* existen 3 con clase *yes*, y 2 con clase *no*.

Pulsando en el botón *Choose* en *Filter*, tendremos acceso a multitud de herramientas para el preprocesamiento de datos. Estas herramientas permiten (entre otras muchas funcionalidades):

- Realizar un filtrado de atributos
- Cambiar el tipo de los atributos (discretizar o numerizar)
- Realizar muestreos sobre los datos
- Normalizar atributos numéricos
- Unificar valores de un mismo atributo

Una vez cargado el fichero, ya estamos en disposición de aprender un modelo (en este caso un árbol de decisión). Para ello, seleccionamos en la pestaña *Classify*.

Como podemos observar, el entorno cambia bastante con respecto a la ventana anterior. Pulsando en el botón *choose* de C*lassifier* podemos configurar el método de clasificación o regresión que queramos utilizar. Estos métodos se han agrupado a grandes rasgos en las siguientes familias[3]:

[2] Podemos conocer el significado de cada color seleccionando el atributo clase, en este caso *play*.
[3] Esta clasificación no es exhaustiva, ya que hay métodos de aprendizaje que caerian en más de uno de estos grupos.

- Bayes. Métodos basados en el paradigma delaprendizaje de Bayes
- Funciones. Métodos “matemáticos”: Redes neuronales, regresiones, SVM…
- Lazy. Métodos que utilizan el paradigma de aprendizaje perezoso, es decir no construyen un modelo
- Meta. Métodos que permiten combinar diferentes métodos de aprendizaje
- Trees. Métodos que aprenden mediante la generación de árboles de decisión
- Rules. Métodos que aprenden modelos que se pueden expresar como reglas.

Además en esta ventana podemos establecer como queremos efectuar la validación del modelo aprendido

- Use training set. Con esta opción Weka entrenará el método con todos los datos disponibles y a posteriori realiza la evaluación sobre los mismos datos.
- Supplied test set. Con esta opción podemos cargar un conjunto de datos (normalmente diferentes a los de aprendizaje) con los cuales se realizará la evaluación
- Cross-validation. Se realiza la evaluación mediante la técnica de validación cruzada. En este caso podemos establecer el número de pliegues a utilizar.
- Percentage split. Se define un porcentaje con el que se aprende el modelo. La evaluación se realiza con los datos restantes.

Para este primer ejemplo vamos a utilizar el algoritmo clásico de aprendizaje de árboles de decisión C4.5 (J48 es el nombre que se le da en Weka), Para ello pulsamos *Choose*, seleccionamos *J48* en *Trees*. Si pulsáramos sobre la ventana que contiene el nombre del método podríamos modificar los parámetros específicos de este método. En este caso dejaremos los valores por defecto.

Por último seleccionamos como opción de evaluación (*test options*) la opción *Use training set*, y ya estamos listos para ejecutar el método de aprendizaje. Para ello pulsamos el botón *Start* que despierta al pájaro Weka de su letargo y realiza el aprendizaje del modelo predictivo, en este caso un árbol de decisión.

Si no ha habido problemas, el sistema nos muestra en la caja “*Classifier Output*” la siguiente información:

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     weather
Instances:    14
Attributes:   5
              outlook
              temperature
              humidity
              windy
              play
Test mode:    evaluate on training data


=== Classifier model (full training set) ===


J48 pruned tree
```

```
------------------

outlook = sunny
|   humidity <= 75: yes (2.0)
|   humidity > 75: no (3.0)
outlook = overcast: yes (4.0)
outlook = rainy
|   windy = TRUE: no (2.0)
|   windy = FALSE: yes (3.0)

Number of Leaves  :     5
Size of the tree :      8

Time taken to build model: 0.25 seconds

=== Evaluation on training set ===
=== Summary ===

Correctly Classified Instances          14               100      %
Incorrectly Classified Instances         0                0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances               14

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   Class
                 1         0          1         1         1        yes
                 1         0          1         1         1        no

=== Confusion Matrix ===

 a b   <-- classified as
 9 0 | a = yes
 0 5 | b = no
```

Como podemos observar weka nos informa en primer lugar de algunos parámetros del dataset. A continuación nos muestra de manera textual el modelo aprendido (en este caso el árbol de decisión).

Por ultimo nos incluye información sobre la evaluación del modelo. En este problema, el árbol aprendido tiene una precisión máxima (100%) por lo que todas las medidas de error son 0. Además podemos conocer más detalles de la distribución de los ejemplos por clase en la matriz de confusión).

Podemos visualizar el árbol de una manera más atractiva si pulsamos el botón derecho sobre el texto *trees.J48* de la caja *Result-list*. Seleccionamos la opción Visualize Tree, y obtendremos el árbol de decisión de la Figura 4.

Además, la Figura 4 nos muestra para cada hoja cuántos ejemplos de la evidencia son cubiertos. Con el botón derecho sobre el texto *trees.J48* de la caja *Result-list* tendremos acceso otras opciones más avanzadas para el análisis del modelo aprendido.

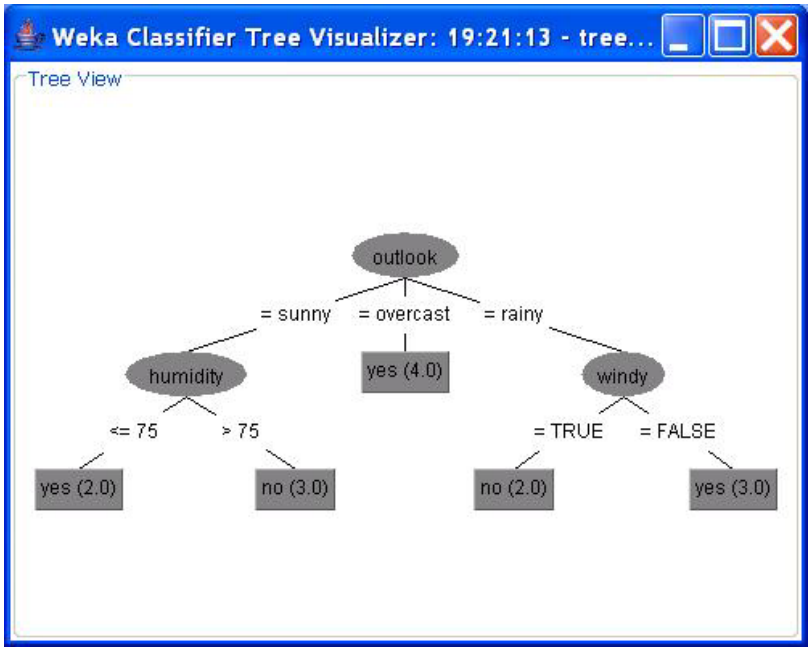

Figura 4. Árbol representado gráficamente.

3. Un problema de clasificación

Vamos a abordar problemas más complejos a partir de ahora, en particular en la parte de preparación de datos.

3.1 *Enunciado del problema. Selección de Fármaco*

En este caso se trata de predecir el tipo de fármaco (drug) que se debe administrar a un paciente afectado de rinitis alérgica según distintos parámetros/variables. Las variables que se recogen en los historiales clínicos de cada paciente son:

- Age: Edad
- Sex: Sexo
- BP (Blood Pressure): Tensión sanguínea.
- Cholesterol: nivel de colesterol.
- Na: Nivel de sodio en la sangre.
- K: Nivel de potasio en la sangre.

Hay cinco fármacos posibles: DrugA, DrugB, DrugC, DrugX, DrugY. Se han recogido los datos del medicamento idóneo para muchos pacientes en cuatro hospitales. Se pretende, para nuevos pacientes, determinar el mejor medicamento a probar.

3.2 *Resolución del problema*

En primer lugar vamos a cargar los datos del primer hospital fichero “drug1n.arff”, ya que al ser el de menor tamaño (200 registros), permite hacer más pruebas inicialmente.

La primera pregunta que nos podemos hacer es ver qué fármacos son más comunes en general, para ver si todos suelen ser igualmente efectivos en términos generales. Para ello seleccionamos el atributo *drug*, y de esta manera vemos al distribución por clases. Podemos concluir que el fármaco más efectivo es el Y, que se administra con éxito en casi la mitad de los pacientes. Una regla vulgar sería aplicar el fármaco Y, en el caso que falle, el fármaco X, y así sucesivamente siguiendo las frecuencias de uso con éxito.

Weka tiene un método que permite generar este modelo tan simple, es decir asignar a todos los ejemplos la clase mayoritaria, recibe el nombre de *ZeroR*[4] en la familia *rules*. Si vamos a la parte de Classify y ejecutamos este modelo evaluamos sobre todos los datos de entrenamiento, veremos que como era de esperar obtenemos un clasificador de precisión 45.5%.

Podemos utilizar métodos más complejos como el J48. Si probáis esta técnica veréis que se obtiene el árbol de la Figura 5 que obtiene una precisión del 97%.

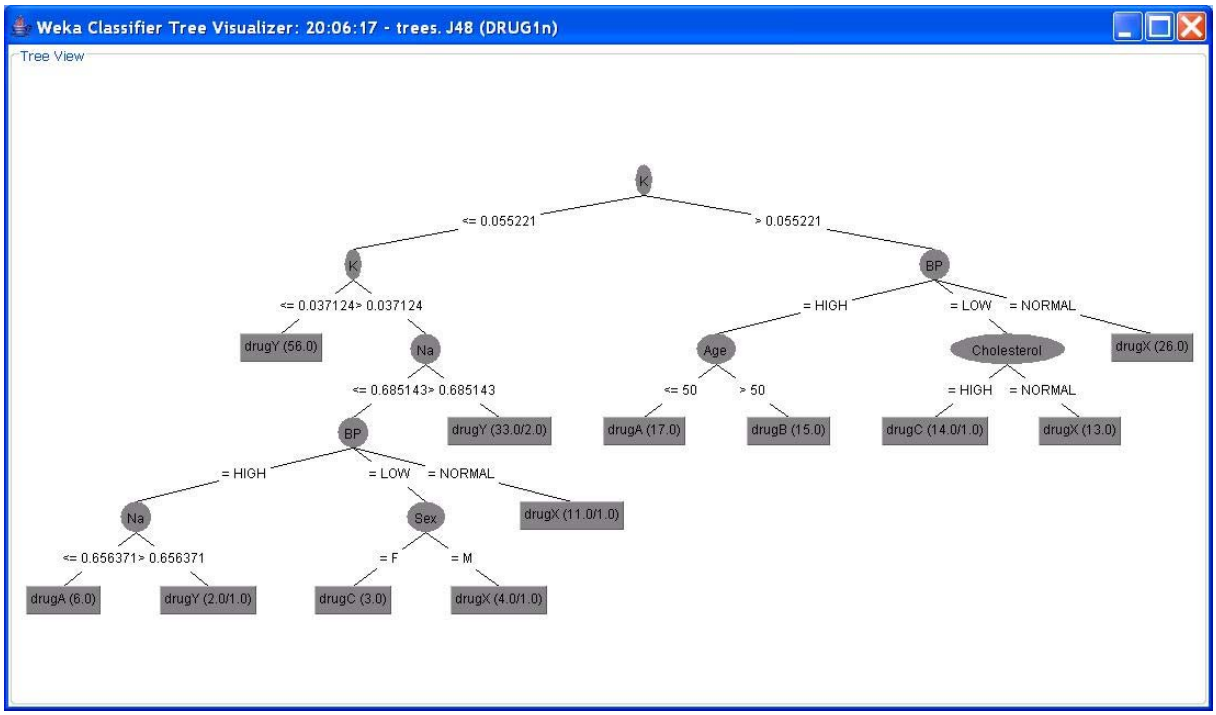

Figura 5. Árbol de decisión directamente sobre los datos

Como podemos observar, el árbol tiene bastantes reglas (en concreto 12). Podemos ver cuál es el acierto (también denominado precisión o *accuracy*) de este árbol respecto a los datos de entrenamiento. Sin embargo, en este caso tenemos un error de sólo el 3,5% sobre los datos de entrenamiento. Este modelo es muchísimo mejor que sí sólo nos guiamos por la distribución, que nos daría un error de más del 50% (el 54,5% de las veces el medicamento DRUGY no es el adecuado).

De todas maneras, es posible hacerlo mejor... ¿pero cómo? ¿con otro tipo de algoritmo de aprendizaje, una red neuronal, p.ej.?

Es posible que otros modelos (p.ej. las redes neuronales) dieran mejor resultado (ya lo probaremos), pero el asunto aquí es que posiblemente no hemos examinado suficientemente los datos de entrada.

[4] En el caso de regresión este método asigna la media aritmética a todos los casos predecir.

Vamos a analizar, con más detenimiento, los atributos de entrada del problema. Es posible que se puedan establecer mejores modelos si combinamos algunos atributos. Podemos analizar pares de atributos utilizando diferentes gráficos.

Para comparar la relación entre atributos en Weka debemos acudir al entorno *Visualize*, donde podemos realizar gráficas entre pares de atributos y ver si tienen alguna relación con las clases. De entre todas las combinaciones posibles, destaca la que utiliza los parámetros de los niveles de sodio y potasio (K y Na) ver

Pulsando en la parte inferior sobre las clases, podemos cambiar los colores asignados a las clases, mejorando la visualización del gráfico. Además podemos ampliar y ver con más detalle alguna zona del gráfico con *select instance* y *submit*.

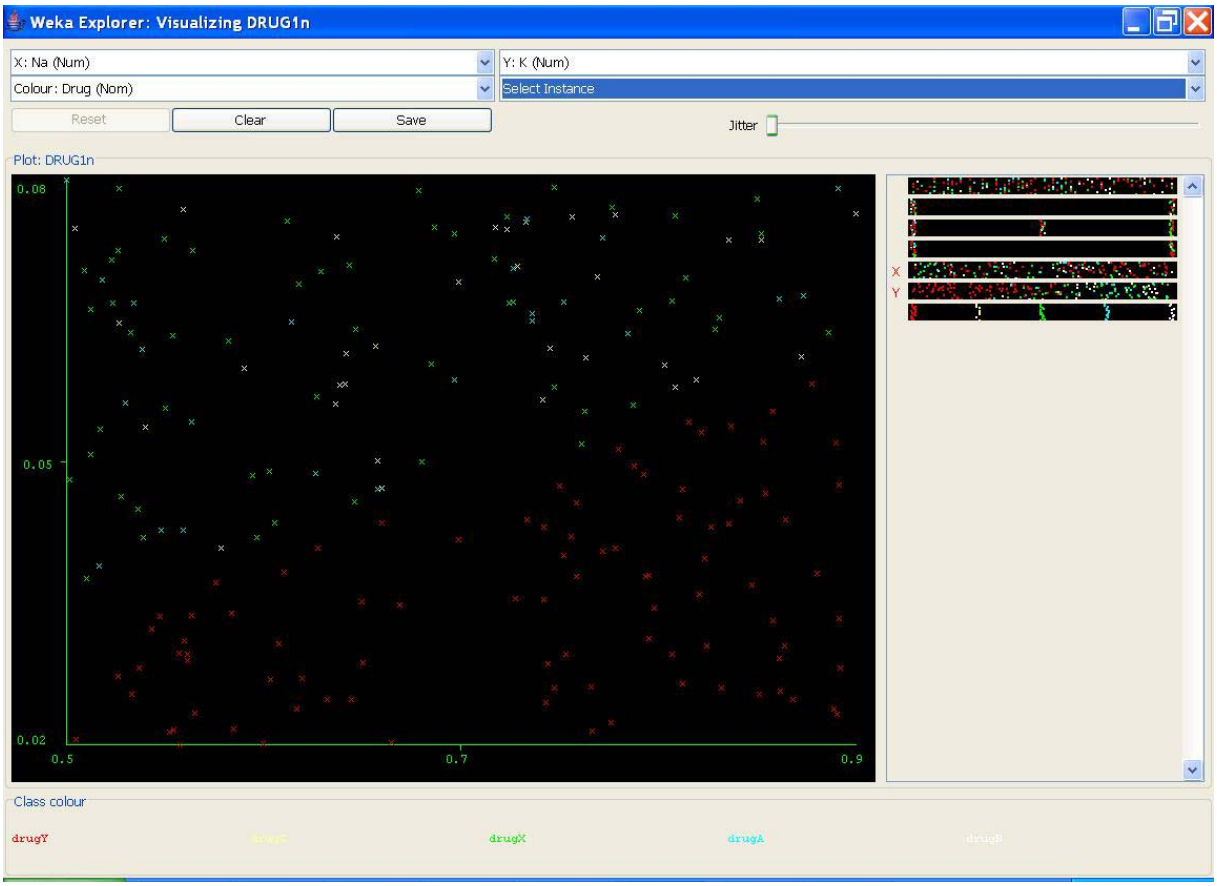

Figura 6. Resultado de un nodo Gráfico (Na x K x Drug)

En este gráfico sí que se ven algunas características muy significativas. Parece haber una clara separación *lineal* entre una relación K/Na alta y una relación K/Na baja. De hecho, para las concentraciones K/Na bajas, el fármaco Y es el más efectivo de una manera clara y parece mostrarse que por encima de un cierto cociente K/Na ese medicamento deja de ser efectivo y se debe recurrir a los otros cuatro.

Podemos utilizar este conocimiento que acabamos de extraer para mejorar nuestros modelos. **Hemos establecido que el medicamento a administrar depende en gran medida del cociente entre K/Na**. Por tanto, vamos a realizar un nuevo modelo que utilice este cociente. Para ello, vamos a crear un nuevo atributo derivado (también llamados atributos pick & mix) mediante el uso de los filtros del entorno "*preprocess*".

Seleccionamos esta parte de la aplicación, y pulsamos *Choose* en *Filter*. De entre todos los filtros posibles elegimos *Unsupervised.Attribute.Addexpression*. Este atributo nos permite la creación de nuevos atributos mediante la combinación de atributos ya existentes (Pick&Mix). Para configurar este filtro pulsamos sobre la propia ventana del texto, y nos aparecen las opciones posibles. Sólo tenemos que modificar la opción *expression*, asignandole el valor "a5/a6" (a5 corresponde a K y a6 corresponde a NA), y la opción *name*, donde colocamos el nombre del nuevo atributo "Na_to_Ka".

Aceptamos con OK, y una vez configurado el filtro lo empleamos con *apply*. Veremos en ese momento como aparece el nuevo atributo en la lista de atributos. Siempre podremos deshacer estos cambios pulsando en *undo*.

A continuación, vamos a ver como funciona el nuevo atributo, para ello volvemos a aprender un árbol de decisión J48 utilizando otra vez todos los datos para la evaluación. Atención que debemos indicarle que la clase es el atributo *drugs* !!! ya que por defecto toma la clase como el último atributo. Esto lo hacemos en la ventana de selección de la clase, en el entorno *Classify*.

Ahora, si examinamos el modelo aprendido tenemos lo siguiente:

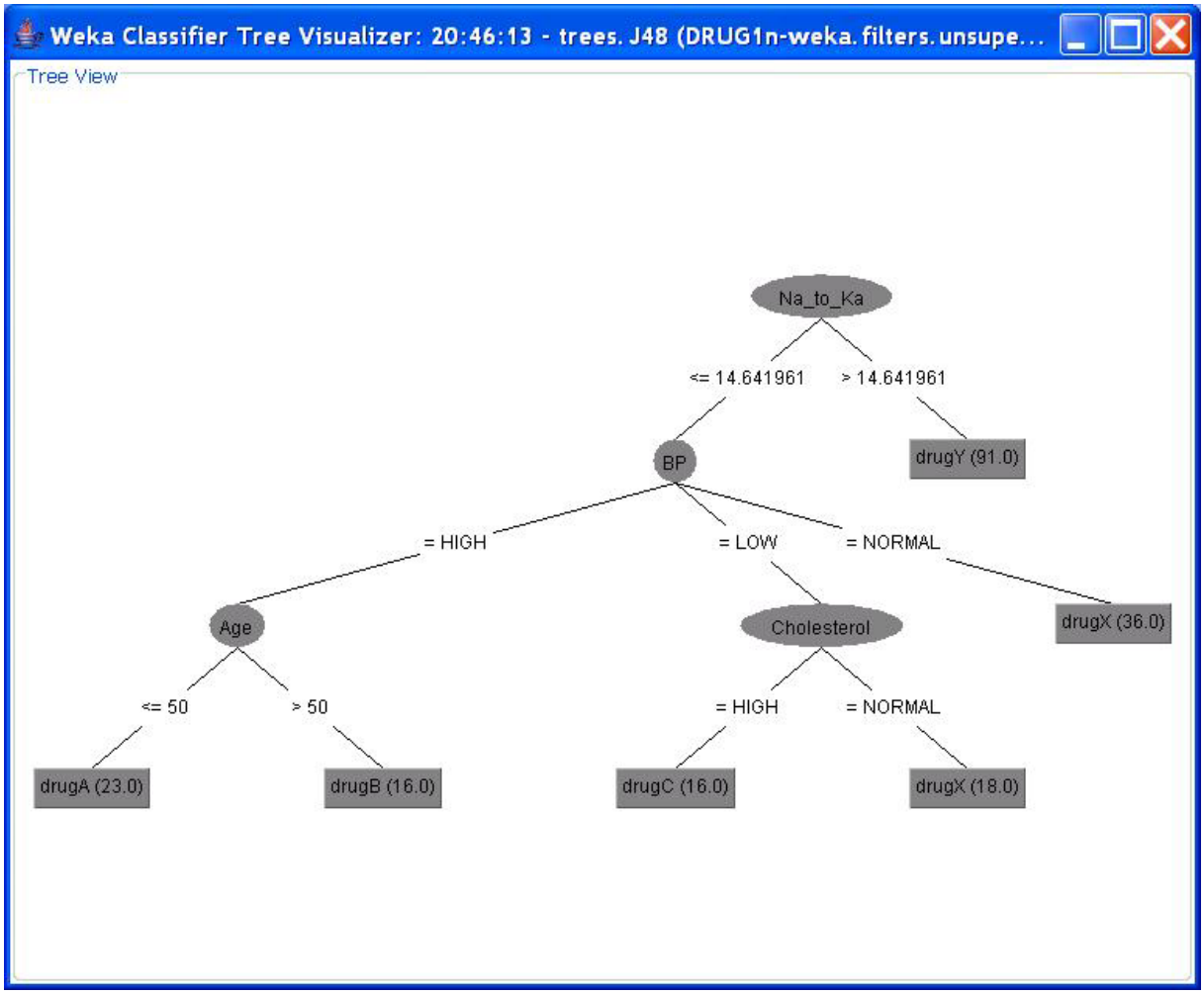

Figura 7. Segundo modelo que utiliza el atributo derivado Na_to_K

Modelo mucho más simple y corto que el anterior, en el que se ve la importancia del atributo que hemos creado Na_to_K. Además si analizamos su calidad vemos que el nuevo modelo un 100% de precisión, con lo que el modelo es mucho más fiable que antes.

No obstante, los árboles de decisión pueden tener poda y tener porcentajes que no son del 100% ni siquiera con los mismos datos que se han usado para el aprendizaje. De hecho si cambiamos el método de evaluación a validación cruzada con 10 pliegues, podremos observar que la precisión baja ligeramente (99%). Sin embargo, tal y como vimos en la parte de teoría, este método de evaluación es mucho más justo. Observad, que el modelo que aparece corresponde al aprendizaje al modelo aprendido con todos los datos. Weka muestra este modelo de manera sólo informativa, ya que no lo utiliza para la evaluación

3.3 *Filtrado de atributos*

En el árbol de decisión final se puede observar que no se utilizan todos los atributos para efectuar una clasificación, esto indica que hay atributos que no son significativos para la resolución del

problema. Existen métodos como los árboles de decisión, a los cuales no les afecta de manera grave la presencia de atributos no significativos, ya que en el propio mecanismo de aprendizaje realizan una selección de atributos por su relevancia. Sin embargo, otros métodos no realizan este proceso, por lo que si realizamos un filtrado de atributos previo al aprendizaje podremos mejorar de manera relevante su precisión, y al mismo tiempo simplificamos los modelos.

Un ejemplo de método de aprendizaje que reduce su *calidad* ante la presencia de atributos no relevantes, es el método *Naive Bayes*. Veamos que precisión obtiene este método, para ello seleccionamos en *Classify* el método *NaiveBayes* y lo evaluamos con evaluación cruzada (10 pliegues). Este método obtiene una precisión de 91%, por lo tanto una precisión mucha más baja que los árboles de decisión.

Bien, en este caso probablemente los atributos no relevantes están afectando a la calidad del método. Veamos como podemos efectuar un filtrado de atributos. Para ello vamos a la sección *Select Attributes*. Esta sección esta especializada en este proceso, y contiene una gran cantidad de técnicas diversas para realizar la selección de atributos. Si recordamos la parte teórica, sabremos que hay dos familias de técnicas: los *wrappers*, y los métodos de filtro. Dadas las características del problema en este caso podemos probar con una técnica *wrapper* realizando una búsqueda exhaustiva. Para ello, pulsamos *Choose* de *AttributeEvaluator* y seleccionamos el método *WrapperSubsetEval*. Para configurarlo pulsamos en la ventana de texto. Vamos a utilizar el propio *Naive Bayes* para el *wrapper*, por lo que seleccionaremos es método en *classifier*. Por otra parte en *SearchMethod*, indicamos que queremos una búsqueda exhaustiva eligiendo *ExhaustiveSearch*. De esta manera tendremos la herramienta configurada tal y como aparece en la Figura 8.

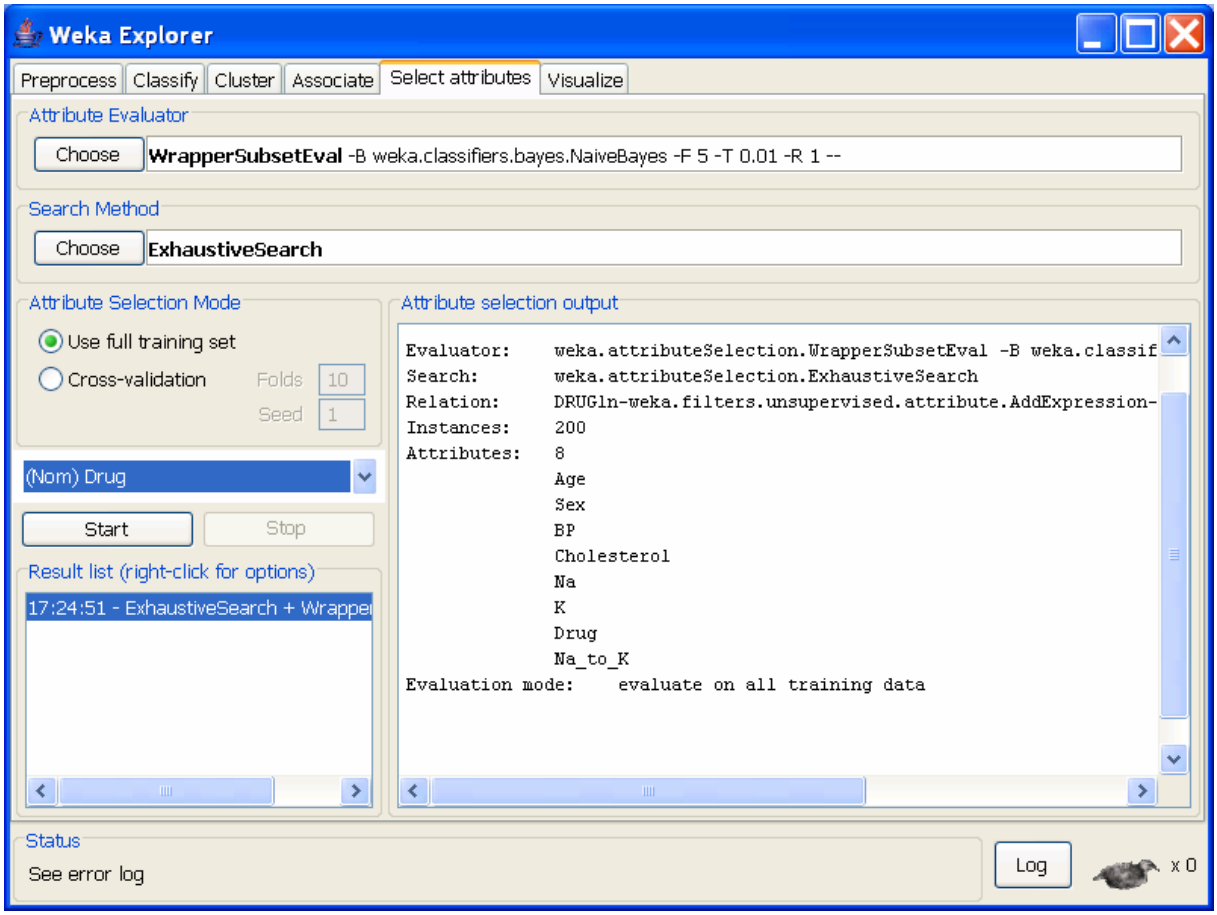

Figura 8. Filtrando atributos.

A continuación ejecutamos el método de filtrado pulsando *Start*. Tras esperar un poco veremos, que este método de filtraje nos recomienda usar los atributos: *Age*, *Sex*, *BP*. *Cholesterol* y *Na_to_K*. Veamos si tiene razón, para ello volvemos a la pantalla *preprocess*, y eliminamos los atributos

descartados (Na y K)[5], para ello los marcamos en la parte de *Attributes*, y pulsamos en *Remove*. En realidad, podemos aplicar los filtros directamente al conjunto de datos si accedemos a los métodos a través de *Filter* en la ventana *Preprocess*.

Finalmente, para conocer la precisión que obtiene *NaiveBayes* con este subconjunto de atributos, volvemos a la ventana *Classify*, y seleccionamos el método *NaiveBayes*. Con validación cruzada ahora se alcanzan 96.5 puntos de precisión.

Por ultimo, observad que también podemos utilizar los propios métodos de aprendizaje como métodos de filtrado. Por ejemplo, si seleccionamos los atributos que utiliza el árbol de decisión: *Age*, *BP*. *Cholesterol* y *Na_to_K*. obtenemos una precisión del 96.5% con *NaiveBayes*.

3.4 *Aprendizaje Sensible al Coste*

En muchos problemas, nos encontraremos situaciones donde dependiendo el tipo de error que cometamos tendremos un coste diferente asociado. Por ejemplo, suponed que en un entorno bancario tenemos un modelo que nos recomienda si conceder o no un crédito a un determinado cliente a partir de las características propias de ese cliente. Obviamente, y desde el punto de vista del banco, es mucho más costoso que el sistema se equivoque dando un crédito a una persona que no lo devuelve, que la situación contraria, denegar un crédito a un cliente que sí lo devolvería.

Para este tipo de problemas, la información sobre los costes de los errores viene expresada a través de una matriz de coste. En este tipo de matrices recoge el coste de cada una de las posibles combinaciones entre la case predicha por el modelo y la clase real.

Weka nos ofrece mecanismos para evaluar modelos con respecto al coste de clasificación, así como de métodos de aprendizaje basados en reducir el coste en vez de incrementar la precisión.

En este ejemplo vamos a utilizar el dataset “credit-g.arff”. Este conjunto de datos contiene 1.000 ejemplos que representan clientes de una entidad bancaria que demandaron un crédito. Existen siete atributos numéricos y 13 nominales. Los atributos de cada objeto indican información sobre el cliente en cuestión: estado civil, edad, parado, etc., así como información del crédito: propósito del crédito, cantidad solicitada, etc. La clase es binaria e indica si el cliente puede ser considerado como fiable para concederle el crédito o no.

Este dataset, trae información de los costes de clasificación errónea, en concreto la siguiente matriz de coste:

		Real	
		No	Sí
Predicho	No	0	1
	Sí	5	0

Esta tabla indica que es 5 veces más costoso si se otorga un crédito a una persona que no lo devuelve, que la situación contraria,

Bien, vamos a ver cómo podemos evaluar los modelos con respecto a la matriz de coste en Weka. Para ello vamos a la ventana *Classify*, y en *Test Options*, pulsamos *More Options*. Hablilitamos la opcion *Cost-Sensitive Evaluation*. Pulsamos *Set* para introducir la matriz. Indicamos el número de clases, 2, e introducimos la matriz de costes. Es importante que pulsemos *enter* tras escribir las cifras ya que es bastante quisquillosa esta ventana. Finalmente cerramos todas las ventanas.

Empezamos con la clasificación más simple, asignar siempre la clase mayoritaria, (*ZeroR*). Lanzamos la evaluación con validación cruzada de 10 pliegues. Y si todo ha ido bien, veremos que se obtiene una precisión del 70 %, y un coste de de 1500 unidades. Era de esperar pues nos equivocamos 300 veces pero en el peor de los casos. Es importante que confirméis que estáis trabajando con la matriz de coste configurada correctamente. Esto lo podéis asegurar observando la

[5] Por supuesto no debemos eliminar la clase.

matriz de coste que aparece en el texto *Classifier Output*, en la parte *Evaluation cost matrix*. Si no es la correcta, volved a configurarla, en ocasiones da mucha guerra esta ventana.

Probemos ahora con otro método, por ejemplo Naive Bayes. Para ello seleccionamos, *NaiveBayes* de *Bayes*. En esta ocasión obtenemos una precisión de 75.4% y un coste de 850 unidades. Mejoramos ambas medidas, sin embargo hemos de tener en cuenta que estos métodos están ignorando la matriz de costes, por lo que se construyen de manera que intentan reducir el número de errores. Weka, ofrece varias técnicas que permiten adaptar las técnicas a un contexto con costes. Dos de estas técnicas están integradas *CostSensitiveClassifier* en *Meta*. Este paquete ofrece dos técnicas bastante sencillas para que un método de aprendizaje sea sensible al contexto. La primera versión consiste en que los modelos aprendidos asignen las clases de manera que se minimicen los costes de clasificación errónea. Normalmente, esta asignación se realiza de manera que se reduzcan los errores en la clasificación. La segunda técnica consiste en modificar los pesos asignados a cada clase de manera que se da más importancia a los ejemplos que son susceptibles de cometer los errores más costosos. Para optar entre estos métodos utilizamos la opción *MinimizeExpectedCost*. Si fijamos esa opción como cierta, se utiliza la primera técnica (asignación de clase de coste mínimo), si la dejamos como falsa, se utiliza la segunda (compensar los costes mediante los pesos de los ejemplos).

Probemos como funcionan estos métodos. Seleccionad este paquete dejando *ZeroR* como clasificador base. Además hemos de indicarle al método la matriz de coste que ha de utilizar en el aprendizaje. Esto se hace, fijando *CostMatrixSource* como *Use Explicit Matriz Cost*. Y en *Cost Matrix* introducimos la misma matriz de coste que hemos utilizado para la evaluación. *MinimizeExpectedCost* lo dejamos como falso.

El esquema resultante sería:

```
weka.classifiers.meta.CostSensitiveClassifier -cost-matrix "[0.0 1.0; 5.0 0.0]" -S 1 -W
weka.classifiers.rules.ZeroR
```

Si lanzamos el método veremos que ahora en vez de tomar la clase mayoritaria para clasificar todos los ejemplos, toma la clase minoritaria, de esta manera, aunque la precisión es muy baja (30%), el coste total es menor 700.

Veamos que sucede si en vez de *ZeroR* utilzamos NaiveBayes y tomando la misma configuración de *CostSensitiveClassifier*. Obtenemos, como es de esperar un resultado bastante mejor: 69% de precisión y un coste de 530 unidades. Finalmente, si probamos el otro modo de convertir NaiveBayes en un clasificador sensible al contexto (es decir cambiamos *MinimizeExpectedCost* a True), todavía reducimos más el coste: Precisión 69.5% y coste 517 unidades.

3.5 *Combinación de Modelos*

Uno de los aspectos más destacados de Weka es la gran cantidad de métodos de combinación de modelos que posee. Estos métodos mejoran la calidad de las predicciones mediante combinando las predicciones de varios modelos. En realidad estos métodos implementan la estrategia de “cuatro ojos ven más que dos”, aunque también es cierto que para que esta afirmación se cumpla los ojos deben tener buena vista, y además no deben tener un comportamiento idéntico, ya que en ese caso no habría mejora.

Por el contrario, utilizar métodos de combinación de modelos también tiene desventajas. Principalmente, el modelo se complica, perdiendo comprensibilidad. Además, necesitamos más recursos (tiempo y memoria) para el aprendizaje y utilización de modelos combinados.

En este ejemplo vamos a utilizar de nuevo el dataset “credit-g.arff”. Iniciamos el ejercicio cargando el fichero “credit-g.arff”. A continuación, acudimos a la pantalla *Classify*. Vamos a ver que precisión obtiene el método J48 en su configuración por defecto utilizando validación cruzada con 10 pliegues. Para este problema, J48 obtiene una precisión de 70.5%, lo cual, aunque a priori no está mal, es realmente un resultado muy pobre. Fijaos que utilizando el criterio de aceptar todos los créditos tendríamos una precisión del 70%. Por lo tanto J48 casi no aporta nada a este problema.

Vamos a empezar con el método *Bagging*. Como sabeis esta técnica se fundamenta en construir un conjunto de *n* modelos mediante el aprendizaje desde *n* conjuntos de datos. Cada conjunto de dato se construye realizando un muestreo con repetición del conjunto de datos de entrenamiento.

Para seleccionar *Bagging*, pulsamos *Choose* y en *Meta*, tendremos el método *Bagging*. Vamos a Configurarlo. En este caso las opciones más importantes son *numIterations* donde marcamos el número de modelos base que queremos crear, y *Classifier*, donde seleccionamos el método base con el cual deseamos crear los modelos base. Fijamos *numIterations*=20 y *Classifier*= J48. De esta manera tendremos el método configurado de la siguiente manera:

```
weka.classifiers.meta.Bagging -P 100 -S 1 -I 20 -W weka.classifiers.trees.J48 -- -C 0.25 -M 2
```

Si lanzamos weka, observareis que obviamente le cuesta más (en realidad 20 veces más). Y ahora obtenemos una precisión de 75%, hemos mejorado 5 puntos!!!. Puede parecer poco, pero pensad el dinero que podría suponer una mejora de 5 puntos en el acierto en la predicción de créditos impagados de una entidad bancaria

Veamos que pasa con *Boosting*. Esta técnica es bastante parecida al *Bagging*, aunque utiliza una estrategia más ingeniosa ya que cada iteración intenta corregir los errores cometidos anteriormente dando más peso a los datos que se han clasificado erróneamente.

Seleccionamos *AdaBoostM1* en Meta. En este método las opciones más importantes son también *numIterations* donde marcamos el número de iteraciones máximas (es decir de modelos base), y *Classifier*, donde seleccionamos el método base con el cual deseamos crear los modelos base. Como en caso anterior, fijamos *numIterations*=20 y *Classifier*= J48. De esta manera tendremos el método configurado de la siguiente manera:

```
weka.classifiers.meta.AdaBoostM1 -P 100 -S 1 -I 20 -W weka.classifiers.trees.J48 -- -C 0.25 -M 2
```

En este caso la mejora es más modesta, ya que nos quedamos en 71.6. Probemos a incrementar el número máximo de iteraciones a 40. Como veréis, mejoramos un poco la predicción, ya que llegamos a 73.5, eso sí, a base de dedicar más tiempo al aprendizaje.

4. Un problema de agrupación

4.1 *Enunciado: Agrupación de Empleados*

La empresa de software para Internet “Memolum Web” quiere extraer tipologías de empleados, con el objetivo de hacer una política de personal más fundamentada y seleccionar a qué grupos incentivar. Las variables que se recogen de las fichas de los 15 empleados de la empresa son:

- Sueldo: sueldo anual en euros.
- Casado: si está casado o no.
- Coche: si viene en coche a trabajar (o al menos si lo aparca en el párking de la empresa).
- Hijos: si tiene hijos.
- Alq/Prop: si vive en una casa alquilada o propia.
- Sindic.: si pertenece al sindicato revolucionario de Internet
- Bajas/Año: media del nº de bajas por año
- Antigüedad: antigüedad en la empresa
- Sexo: H: hombre, M: mujer.

Los datos de los 15 empleados se encuentran en el fichero “empleados.arff”. Se intenta extraer grupos de entre estos quince empleados.

4.2 *Resolución del Problema*

Vamos a utilizar un algoritmo de clustering para obtener grupos sobre esta población. En primer lugar vamos a probar con tres grupos. Para ello acudimos a la ventana *Cluster*. Vamos a empezar a

trabajar con el algoritmo de Kmedias. Para ello en Clusterer, pulsamos Choose, y seleccionamos SimpleKmeans. Lo configuramos definiendo como 3 el *NumClusters*.

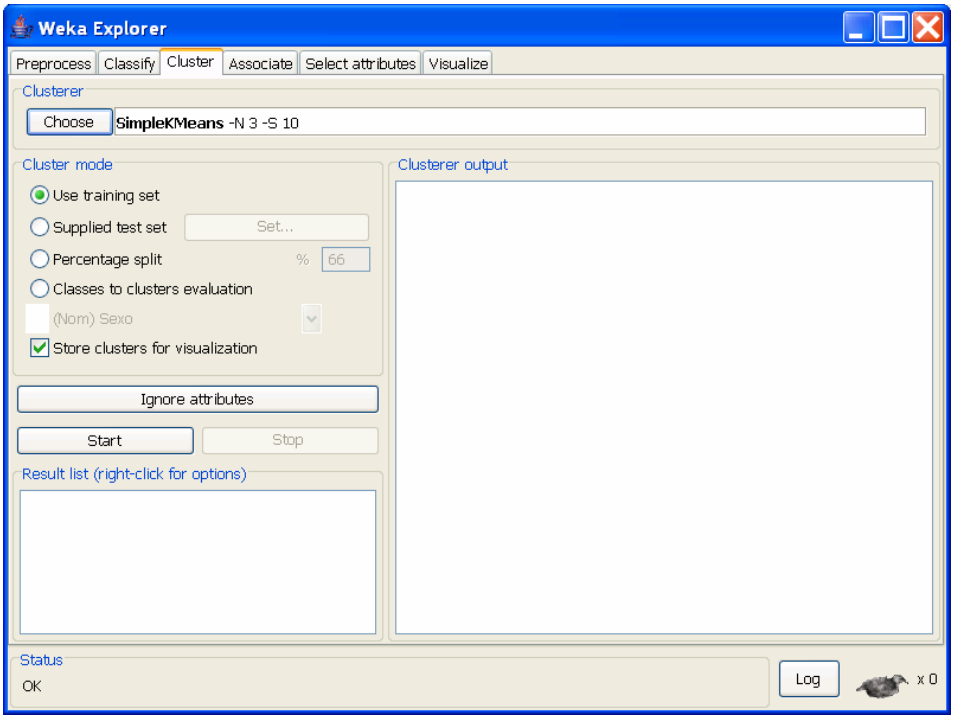

Figura 9. Determinando el número de clusters

Ahora podemos ejecutar el método pulsando *Start*.

Si examinamos el resultado conoceremos la suma media de errores cuadráticos de cada valor con respecto a su centro. Además vemos qué características tiene cada cluster/conglomerados. A continuación, se muestra de una manera más resumida a cómo lo muestra el Weka (que incluye desviaciones):

cluster 1	cluster 2	cluster 3
6 Ejemplos	5 examples	4 examples
Sueldo : 29166.6667 Casado : No Coche : No Hijos : 0.1667 Alq/Prop : Alquiler Sindic. : Sí Bajas/Año : 6.16 Antigüedad : 8.333 Sexo : M	Sueldo : 16600 Casado : Sí Coche : Sí Hijos : 1.8 Alq/Prop : Prop Sindic. : No Bajas/Año : 3.4 Antigüedad : 8.4 Sexo : H	Sueldo : 14500 Casado : Sí Coche : Sí Hijos : 0.25 Alq/Prop : Alquiler Sindic. : No Bajas/Año : 6.25 Antigüedad : 7.75 Sexo : H

Si pulsamos el botón derecho sobre SimpleKmeans de la parte de *Result List* Podremos ver a qué Grupo va a parar cada ejemplo. Pero además, podemos estudiar cómo se asignan los ejemplos a los clusters dependiendo de los atributos seleccionados para dibujar la gráfica.

5. Reglas de Asociación y Dependencias

Vamos a estudiar ahora los datos del hundimiento del Titanic. Los datos se encuentran en el fichero "titanic.arff" y corresponden a las características de los 2.201 pasajeros del Titanic. Estos datos son reales y se han obtenido de: *"Report on the Loss of the 'Titanic' (S.S.)" (1990), British Board of Trade Inquiry Report_ (reprint), Gloucester, UK: Allan Sutton Publishing*.

Para este ejemplo sólo se van a considerar cuatro variables:

- Clase (0 = tripulación, 1 = primera, 2 = segunda, 3 = tercera)
- Edad (1 = adulto, 0 = niño)
- Sexo (1 = hombre, 0 = mujer)
- Sobrevivió (1 = sí, 0 = no)

Para ello, vamos a ver que reglas de asociación interesantes podemos extraer de estos atributos. Para ejecutar los métodos en Weka de reglas de asociaciación, seleccionamos la ventana de *associate*.

Entre otros, este sistema de minería de datos provee el paquete "WEKA.associations.Apriori" que contiene la implementación del algoritmo de aprendizaje de reglas de asociación Apriori. Podemos configurar este algoritmo con varias opciones: con la opción "*UpperBoundMinSupport*" indicamos el límite superior de cobertura requerido para aceptar un conjunto de ítems. Si no se encuentran conjuntos de ítems suficientes para generar las reglas requeridas se va disminuyendo el límite hasta llegar al límite inferior (opción "*LowerBoundMinSupport*"). Con la opción "*minMetric*" indicamos la confianza mínima (u otras métricas dependiendo del criterio de ordenación) para mostrar una regla de asociación; y con la opción "*numRules*" indicamos el número de reglas que deseamos que aparezcan en pantalla. La ordenación de estas reglas en pantalla puede configurarse mediante la opción "*MetricType*", algunas opciones que se pueden utilizar son: confianza de la regla, *lift* (confianza divido por el número de ejemplos cubiertos por la parte derecha de la regla), y otras más elaboradas.

Veamos las reglas que extrae este algoritmo tomando los valores por defecto:

```
 1. Clase=0 885 ==> Edad=1 885    conf:(1)
 2. Clase=0 Sexo=1 862 ==> Edad=1 862    conf:(1)
 3. Sexo=1 Sobrevivió?=0 1364 ==> Edad=1 1329    conf:(0.97)
 4. Clase=0 885 ==> Edad=1 Sexo=1 862    conf:(0.97)
 5. Clase=0 Edad=1 885 ==> Sexo=1 862    conf:(0.97)
 6. Clase=0 885 ==> Sexo=1 862    conf:(0.97)
 7. Sobrevivió?=0 1490 ==> Edad=1 1438    conf:(0.97)
 8. Sexo=1 1731 ==> Edad=1 1667    conf:(0.96)
 9. Edad=1 Sobrevivió?=0 1438 ==> Sexo=1 1329    conf:(0.92)
10. Sobrevivió?=0 1490 ==> Sexo=1 1364    conf:(0.92)
```

En cada regla, tenemos la cobertura de la parte izquierda y de la regla, así como la confianza de la regla. Podemos conocer alguna regla interesante aunque otras los son menos. Por ejemplo, la regla 1 indica que, como era de esperar toda la tripulación es adulta. La regla 2 nos indica lo mismo, pero teniendo en cuenta a los varones. Parecidas conclusiones podemos sacar de las reglas 4, 5 y 6. La regla 3 nos indica que los varones que murieron fueron en su mayoría adultos (97%). La regla 7 destaca que la mayoría que murieron fueron adultos (97%). Y finalmente la 10 informa que la mayoría de los muertos fueron hombres (92%).

Cabe destacar que la calidad de las reglas de asociación que aprendamos muchas veces viene lastrada por la presencia de atributos que estén fuertemente descompensados. Por ejemplo, en este caso la escasa presencia de niños provoca que no aparezcan en las reglas de asociación, ya que las reglas con este ítemset poseen una baja cobertura y son filtradas. Podemos mitigar parcialmente este fenómeno si cambiamos el método de selección de reglas.