

Tutorial Weka 3.6.0

Ricardo Aler 2009

Contenidos:

- 0. Descarga**
- 1. Entrar al programa**
- 2. El Explorer:**
 - 2.0. El Explorer: Preprocesamiento (preprocess)**
 - 2.1. El Explorer, Clasificación (classify)**
 - 2.2. El Explorer, Selección de atributos (Attribute Selection, visualization)**
- 3. El Experimenter**

0. Descarga

Weka es una herramienta de dominio público escrita en Java que se puede descargar de esta dirección:

<http://www.cs.waikato.ac.nz/~ml/weka/>

Este tutorial se centra en la versión 3.6.0 y su objetivo es dar un conocimiento básico de Weka. Se puede obtener información más detallada bien en la URL anterior, bien en el libro de Weka “Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)” de Witten y Frank.

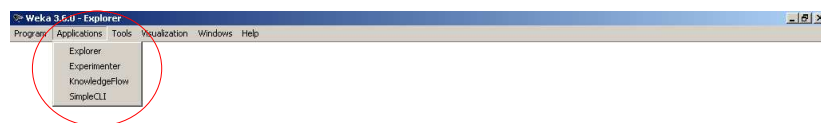
Aunque no va a ser necesario para este tutorial, en ocasiones es importante incrementar la cantidad de memoria que Weka puede utilizar. Esto se puede hacer creando un fichero .bat (en Windows) como este:

```
@echo off
javaw -Xmx1024m -classpath "C:\Program Files (x86)\Weka-3-6\weka.jar" weka.gui.Main
```

para ello habrá que localizar previamente el directorio donde está instalado el .jar de Weka. El script anterior le da a Weka 1 Giga de memoria (1024 MB). Pero no va a ser necesario hacer este cambio para el tutorial.

1. Entrar al programa

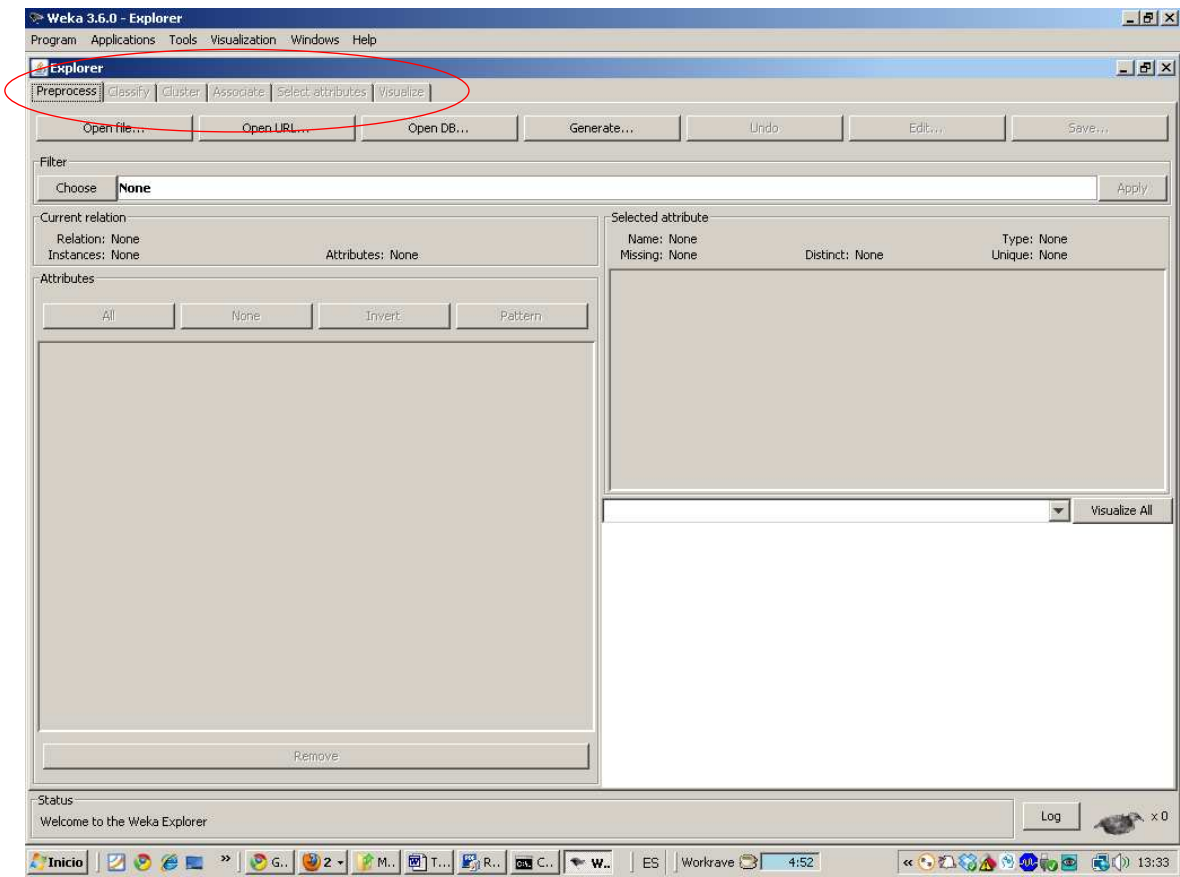
La primera pantalla de Weka muestra una serie de opciones en su parte superior. La más importante es **Applications**, donde se pueden ver las distintas subherramientas de Weka. Las más importantes son **Explorer** (para explorar los datos) y **Experimenter** (para realizar experimentos que comparen estadísticamente distintos algoritmos en distintos conjuntos de datos, de manera automatizada). De momento usaremos el Explorer.



2. 0. El Explorer: preprocesamiento (preprocess)

El Explorer permite visualizar y aplicar distintos algoritmos de aprendizaje a un conjunto de datos. Cada una de las tareas de minería de datos viene representada por una pestaña en la parte superior. Estas son:

- **Preprocess:** visualización y preprocesado de los datos (aplicación de filtros)
- **Classify:** Aplicación de algoritmos de clasificación y regresión
- **Cluster:** Agrupación
- **Associate:** Asociación
- **Select Attributes:** Selección de atributos
- **Visualize:** Visualización de los datos por parejas de atributos



A partir de aquí usaremos el fichero **iris.arff**, que ya viene en el formato de datos de Weka (formato arff). El objetivo de este dominio es construir un clasificador que permita clasificar plantas en tres clases: setosa, versicolor y virgínica, en términos de la longitud y anchura de pétalos y sépalos. Cargaremos el fichero con la opción **Open File** (arriba a la izquierda). Existen otras opciones para cargar datos desde una página web, desde una base de datos, o incluso para generar datos artificiales adaptados a clasificadores concretos, pero no utilizaremos esas posibilidades aquí. Podemos observar como en la parte izquierda se nos da información sobre el número de datos o instancias (150) y el número de atributos (5 = 4 mas la clase). También en la parte izquierda tenemos el nombre de los atributos. En la parte superior derecha aparece información estadística sobre los atributos (media, desviación típica, valores máximo y mínimo, **Unique** se refiere al número de valores que sólo aparecen una vez en ese atributo y **distinct** al número de valores distintos (o sea, excluyendo los valores repetidos). Un poco mas abajo, aparece el desglose de los valores del atributo por clase. En el eje X aparecen los valores del atributo seleccionado en la parte de la izquierda (ahora mismo, el atributo seleccionado es el **sepalength**, aunque podría ser cualquier otro). Recordemos que hay tres posibles clases, que en el gráfico aparecen representadas con los colores azul oscuro, rojo y azul claro.

Weka 3.6.0 - Explorer

Program Applications Tools Visualization Windows Help

Explorer

Preprocess Classify Cluster Associate Select attributes Visualize

Open file... Open URL... Open DB... Generate... Undo Edit... Save...

Filter: Choose **None** Selección de filtros para los datos Apply

Current relation: Relation: iris Instances: 150 Attributes: 5

Attributes: All None Invert Pattern

No.	Name
1	sepalength
2	sepalwidth
3	petallength
4	petalwidth
5	class

Nombres de atributos ↑

Selected attribute: Name: sepalength Type: Numeric Missing: 0 (0%) Distinct: 35 Unique: 9 (6%)

Statistic	Value
Minimum	4.3
Maximum	7.9
Mean	5.843
StdDev	0.828

Estadísticas de los datos

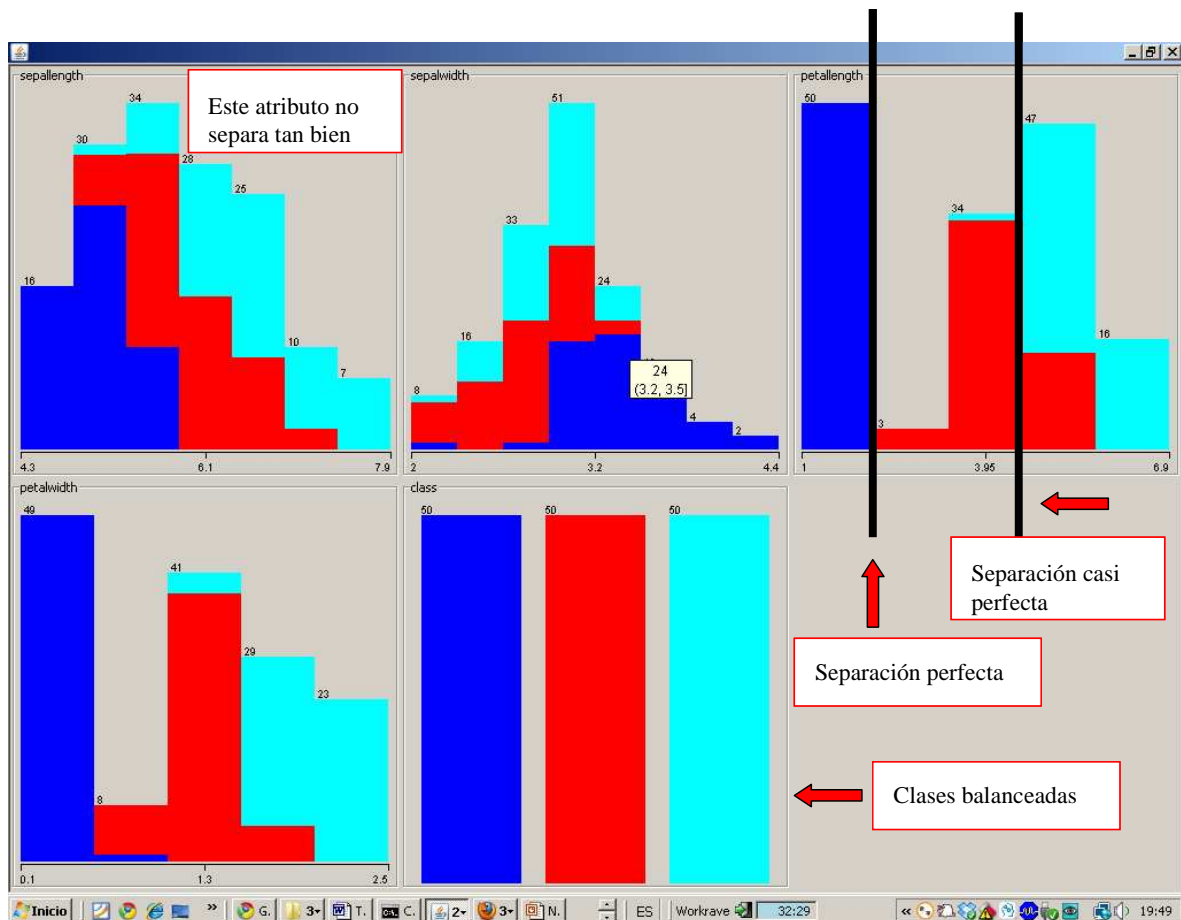
Class: class (Nom) Visualize All

Desglose del atributo seleccionado en la izquierda ↓

Histogram showing sepalength distribution by class (blue, red, cyan) with values: 16, 30, 34, 28, 25, 10, 7.

Status: OK Log

Si queremos podemos visualizar el desglose de todos los atributos a la vez. Si hay pocos atributos, es mas rápido esto que irlos visualizando uno a uno. Pulsemos el botón **Visualize All**, que está en la parte superior derecha del gráfico coloreado con el desglose. Esta visualización puede ser útil para comprobar cómo de efectivo es cada uno de los atributos, considerados por separado. Recordemos que el objetivo de este dominio es encontrar un buen clasificador, es decir, una frontera que separe los datos pertenecientes a distintas clases. ¿Qué atributos parecen buenos en este caso?. Por ejemplo, **petalength** separa perfectamente los 50 datos de la clase azul oscuro de las otras dos: si petalength vale menos de 2.18 es que el dato pertenece a la clase azul oscuro, y si el valor es mayor, pertenece a alguna de las otras dos (rojo o azul claro). También separa bastante bien la clase roja de la azul claro: si el valor es mayor de 3.36, es muy probable que el dato pertenezca a la clase azul claro, y en caso contrario, a la roja. Comprobar como el atributo **petalwidth** también es un buen separador y que **sepalength** y **sepalwidth** no lo son tanto. Es también interesante comprobar el desglose del último atributo, que es el de la clase (**class**). Observese que tenemos tres clases, y cada una de ellas tiene 50 datos. Eso quiere decir que estamos ante un problema balanceado. Si por ejemplo, hubiera 10 datos para la primera y segunda clase, y 140 para la tercera, estaríamos ante un problema desbalanceado.



En esta ventana también podemos seleccionar filtros para los datos y los atributos (selección de filtros en la parte superior izquierda) además de borrar algunos atributos. Los filtros permiten por ejemplo normalizar los datos, borrar algunos que cumplan algún criterio, crear nuevos atributos, etc. No utilizaremos estos filtros de momento.

Salvar datos filtrados

Selección de filtros para los datos

Ejecutar el filtro

Aquí se pueden borrar los atributos seleccionados

Remove

Statistic	Value
Minimum	4.3
Maximum	7.9
Mean	5.843
StdDev	0.828

No.	Name
1	sepalength
2	sepalwidth
3	petalength
4	petalwidth
5	class

Class: class (Nom) Visualize All

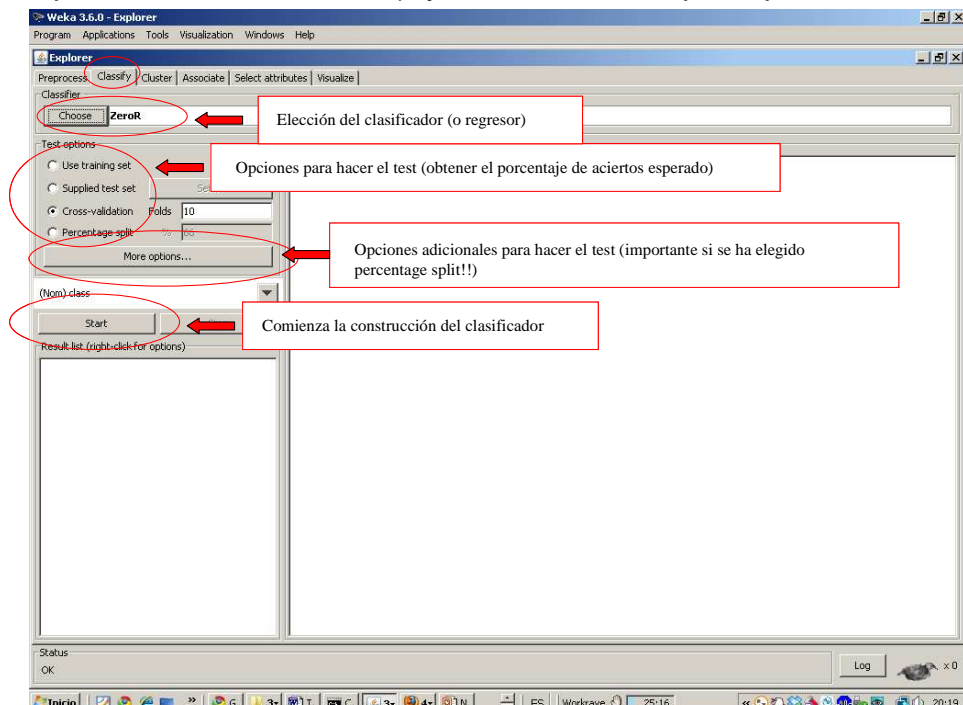
16 30 34 28 25 10 7

4.3 6.1 7.9

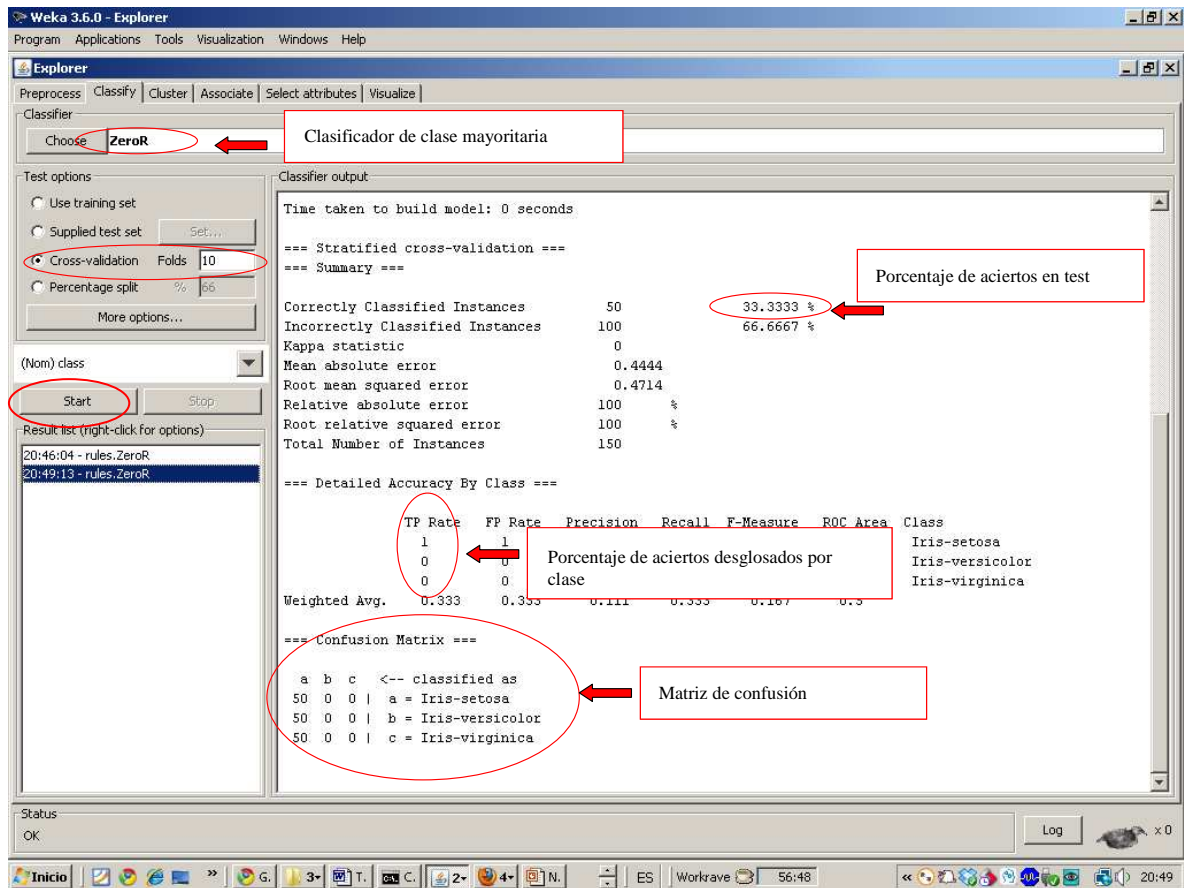
2.1. El Explorer, Clasificación (classify)

Vamos ahora a construir un primer clasificador para los datos. Pulsemos la pestaña **Classify**, que está arriba del todo a la izquierda en la ventana del Explorer. Primero tendremos que elegir el clasificador, en **Choose**, arriba a la izquierda (actualmente está seleccionado **ZeroR**). Debajo de **Choose**, podemos elegir las opciones de test, es decir, la manera de computar el porcentaje esperado de aciertos (en clasificación), o el error cuadrático medio (entre otros, en regresión). Estas opciones son:

- **Use training set:** En este caso usaremos para hacer el test el mismo conjunto que el de entrenamiento (el que se va a usar para construir el clasificador). Ya sabemos que esta opción nos dará un porcentaje demasiado optimista y no es conveniente usarlo.
- **Supplied test set:** Si tenemos un fichero con datos de test distintos a los de entrenamiento, aquí es donde podemos seleccionarlo
- **Crossvalidation:** Se calcula el porcentaje de aciertos esperado haciendo una validación cruzada de k hojas (podemos seleccionar el k, que por omisión es de 10=
- **Percentage split:** En este caso, se dividirá el conjunto de entrenamiento que ya habíamos seleccionado en la pestaña de **Preprocess** (el iris.arff) y se dividirá en dos partes: los primeros 66% de los datos para construir el clasificador y el 33% finales, para hacer el test. Podemos seleccionar el porcentaje para entrenamiento (por omisión, es de 66%). **Importante:** al utilizar esta opción, Weka desordena aleatoriamente el conjunto inicial (en este caso, el iris.arff) y después parte en 66% para entrenamiento y 33% para test. De esta manera, si construyéramos el clasificador dos veces, obtendríamos dos desordenaciones distintas, y por tanto dos porcentajes de aciertos en test ligeramente distintos. **SI NO QUEREMOS QUE SE DESORDENEN LOS DATOS, HAY QUE MARCAR LA OPCIÓN PRESERVE ORDER FOR PERCENTAGE SPLIT EN MORE OPTIONS.** Si marcamos esa opción, Weka siempre cogerá la misma primera parte del fichero para construir el clasificador (el 66%, a menos que se cambie este valor), y la misma última parte para hacer el test.



Vamos a construir el clasificador. En este momento, está seleccionado **ZeroR** (arriba a la izquierda). Este clasificador clasifica a todos los datos con la clase de la clase mayoritaria. Es decir, si el 90% de los datos son positivos y el 10% son negativos, clasificará a todos los datos como positivos. Es conveniente utilizar primero este clasificador, porque el porcentaje de aciertos que obtengamos con el, es el que habrá que superar con el resto de clasificadores. Antes de lanzarlo, seleccionemos la opción **Crossvalidation** (10) para hacer el test. Ahora pulsemos el botón **Start**. Observaremos que obtenemos un 33% de aciertos (lógico, puesto que las tres clases tienen 50 datos y ninguna es la mayoritaria). En el porcentaje de aciertos desglosado por clase (**TP rate**, o True Positive Rate) vemos que la primera clase la acierta al 100% (iris-setosa, TP rate = 1) y las otras dos las falla (TP rate = 0%). Esto es lógico, dada la manera en la que funciona ZeroR: sólo acierta la clase mayoritaria, o la primera, en caso de que ninguna sea mayoritaria. Mas abajo podemos ver la matriz de confusión, donde se muestra que todos los datos los clasifica como iris-setosa.



Ya sabemos que el porcentaje de aciertos a superar es el del 33%. Vamos a probar ahora con otro clasificador. Por ejemplo, el clasificador **PART**, que construye reglas.

The screenshot shows the Weka 3.6.0 Explorer interface. In the 'Classifier' pane on the left, the 'PART' classifier is selected and highlighted with a red oval. The main window displays the following performance metrics:

```
Time to build model: 0 seconds
Cross-validated cross-validation ===
====
Classified Instances      143      95.3333 %
Unclassified Instances    7       4.6667 %
Misclassified Instances    7       4.6667 %
Confusion Matrix
TP Rate  FP Rate  Precision  Recall  F-Measure  ROC Area  Class
0.98     0         1          0.98   0.99       0.99      Iris-setosa
0.94     0.04      0.922     0.94   0.931     0.954     Iris-versicolor
0.94     0.03      0.94      0.94   0.94       0.959     Iris-virginica
0.953    0.023    0.954     0.953  0.954     0.968
```

Below the metrics, the Confusion Matrix is displayed:

```
==== Confusion Matrix ====
a b c <-- classified as
49 1 0 | a = Iris-setosa
0 47 3 | b = Iris-versicolor
0 3 47 | c = Iris-virginica
```

The status bar at the bottom shows 'OK' and a 'Log' button.

Seleccionado **PART**, pulsemos **Start**. Vemos que ahora el porcentaje de aciertos es 94%, bastante bueno, comparado con el 33% del caso base (ZeroR). En el desglose de los aciertos por clase, vemos que todas se aciertan bastante bien, siendo iris-setosa la que tiene mas aciertos (98%) e iris-virginica la peor (90% de aciertos). Es por tanto bastante equilibrado en los aciertos de las distintas clases, lo que tiene sentido, puesto que partíamos de un conjunto de datos balanceado (si no hubiera estado balanceado, es muy posible que la clase mayoritaria se hubiera acertado mejor que las minoritarias). En la matriz de confusión, podemos ver como los datos acertados están en la diagonal, y los fallados fuera de ella.

The screenshot shows the Weka 3.6.0 Explorer interface. The classifier selected is 'PART -M 2 -C 0.25 -Q'. The classifier output displays the following data:

Classifier output

Time taken to build model: 0 seconds

=== Stratified cross-validation ===

=== Summary ===

Correctly Classified Instances	141	94 %
Incorrectly Classified Instances	9	6 %
Kappa statistic	0.91	
Mean absolute error	0.0482	
Root mean squared error	0.1794	
Relative absolute error	10.8379 %	
Root relative squared error	38.0567 %	
Total Number of Instances	150	

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate
setosa	0.98	0.00
versicolor	0.94	0.06
virginica	0.9	0.03
Weighted Avg.	0.94	0.03

=== Confusion Matrix ===

a	b	c	<-- classified as
49	1	0	a = Iris-setosa
0	47	3	b = Iris-versicolor
0	5	45	c = Iris-virginica

En el pantallazo anterior, no hemos visto el modelo (clasificador) construido, que en este caso es un conjunto de reglas. Para verlo tenemos que subir la pantalla un poco hacia arriba, usando la barra de scroll. Hay tres reglas. Los números (a/b) entre paréntesis indican cuantos datos clasifica la regla (a) y cuantos de ellos lo hace incorrectamente (b). La primera de las reglas es:

IF (petalwidth <= 0.6) THEN clase = Iris-setosa (50.0)

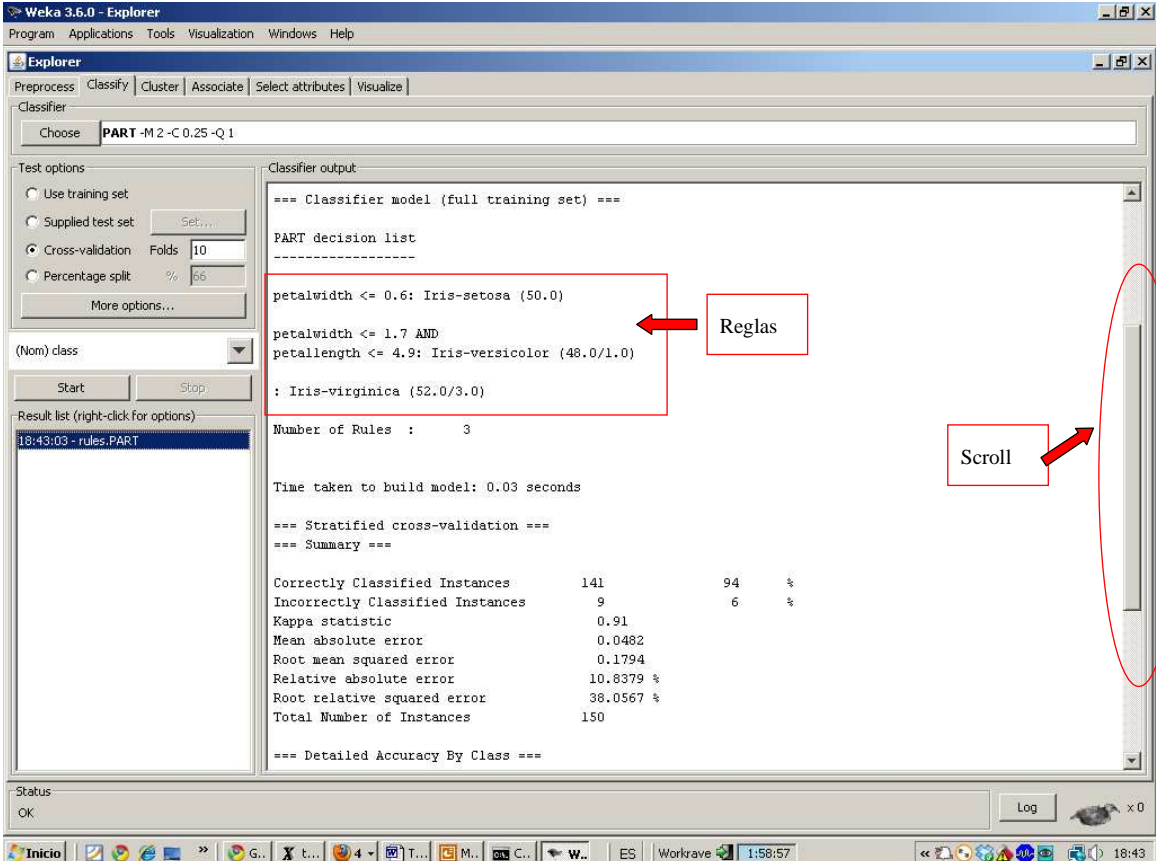
La segunda:

**IF (petalwidth <= 1.7) AND (petalength <= 4.9)
THEN clase = Iris-versicolor (48.0/1.0)**

Y si no se cumplen ninguna de las condiciones, entonces clasificar por omisión como:

Clase = Iris-virginica (52.0/3.0)

Observese que las reglas sólo utilizan los atributos petalength y petalwidth, que son los que habíamos comprobado antes que eran los mejores (al menos, considerados de manera individual cada uno de ellos).

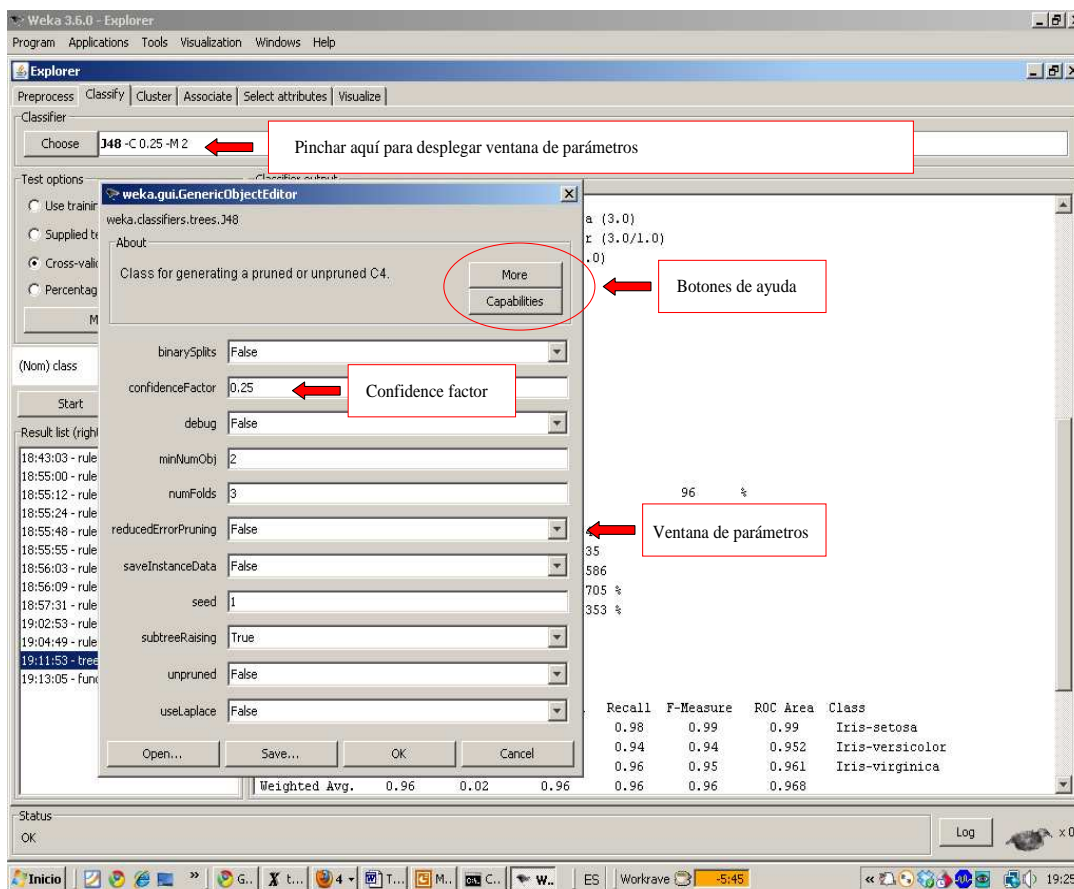


The screenshot shows the Weka 3.6.0 Explorer interface. The 'Classifier output' window displays the following text:

```
=== Classifier model (full training set) ===  
PART decision list  
-----  
petalwidth <= 0.6: Iris-setosa (50.0)  
petalwidth <= 1.7 AND  
petalength <= 4.9: Iris-versicolor (48.0/1.0)  
: Iris-virginica (52.0/3.0)  
  
Number of Rules : 3  
  
Time taken to build model: 0.03 seconds  
  
=== Stratified cross-validation ===  
=== Summary ===  
  
Correctly Classified Instances 141 94 %  
Incorrectly Classified Instances 9 6 %  
Kappa statistic 0.91  
Mean absolute error 0.0462  
Root mean squared error 0.1794  
Relative absolute error 10.8379 %  
Root relative squared error 38.0567 %  
Total Number of Instances 150  
  
=== Detailed Accuracy By Class ===
```

Red annotations highlight the rules section and the scroll bar. A box labeled 'Reglas' points to the decision list, and a box labeled 'Scroll' points to the vertical scrollbar on the right side of the output window.

Todos los algoritmos tienen parámetros que se pueden ajustar. Para poder cambiar sus valores, tenemos que pinchar con el ratón sobre el nombre del algoritmo. El más importante es el que tiene que ver con la complejidad del clasificador construido, y que en el caso de J48, es el confidence factor, el cual controla el tamaño del árbol de decisión construido. Recordemos que la complejidad del clasificador tiene que ver con el sobreaprendizaje (overfitting) En la gráfica anterior, J48 construyó un árbol de 5 hojas (**leaves**) y 9 nodos (**size of the tree**). No podemos controlar directamente el número de nodos, pero cuanto más pequeño es este parámetro, más simple tiende a ser el clasificador (menos nodos), y viceversa. Este parámetro varía entre 0 y 1, siendo el valor por omisión de 0.25. Podemos cambiar los parámetros de un algoritmo pinchando justo sobre el nombre del clasificador (J48 en este caso). En la ventana de selección de parámetros existe también un botón que explica el algoritmo y el significado de sus parámetros (**More**).



Vamos a ver si podemos construir clasificadores más simples y más complejos, variando el confidence factor. Veamos que ocurre si ponemos 0.001 (árboles simples) y 1.0 (árboles complejos) en ese parámetro.

Con 0.001 ocurre lo siguiente:

J48 pruned tree

petalwidth <= 0.6: Iris-setosa (50.0)

petalwidth > 0.6

| **petalwidth <= 1.7**

| | **petallength <= 4.9: Iris-versicolor (48.0/1.0)**

| | **petallength > 4.9: Iris-virginica (6.0/2.0)**

| **petalwidth > 1.7: Iris-virginica (46.0/1.0)**

Number of Leaves : 4

Size of the tree : 7

Correctly Classified Instances 141 94 %

Y con 1.0:

J48 pruned tree

petalwidth <= 0.6: Iris-setosa (50.0)

petalwidth > 0.6

| **petalwidth <= 1.7**

| | **petallength <= 4.9: Iris-versicolor (48.0/1.0)**

| | **petallength > 4.9**

| | | **petalwidth <= 1.5: Iris-virginica (3.0)**

| | | **petalwidth > 1.5: Iris-versicolor (3.0/1.0)**

| **petalwidth > 1.7: Iris-virginica (46.0/1.0)**

Number of Leaves : 5

Size of the tree : 9

Correctly Classified Instances 144 96 %

Con CF = 0.001 el árbol se simplifica (4 hojas y 7 nodos), pero el porcentaje de aciertos disminuye (94%). Con CF = 1.0, el árbol queda igual que antes (5 hojas y 9 nodos), con lo que se mantiene el porcentaje de aciertos (96%).

Podemos concluir que en este caso no estaba habiendo sobreaprendizaje y que la complejidad del modelo era la adecuada. Caso de que hubiese habido sobreaprendizaje, un modelo más simple hubiera obtenido un porcentaje de aciertos mejor, aunque hay que recordar que si el modelo es excesivamente simple, se producirá underfitting (es decir, que el modelo no es lo suficientemente complejo para llevar a cabo el aprendizaje de manera correcta). En este caso, la única ventaja del árbol más pequeño es que es (ligeramente) más sencillo de entender, pero tiene la contraprestación de ser menos preciso.

Vamos a volver ahora a los resultados obtenidos inicialmente por PART con los parámetros por omisión, que se copian más abajo. Se puede ver que se obtiene un porcentaje de aciertos global del 94%, pero que no todas las clases se aprenden igual, aunque si muy parecido. Esto es lógico, puesto que este problema estaba balanceado en la cantidad de datos por clase, por lo que es de esperar que las tres se aprendan mas o menos igual (aunque no tiene por que ocurrir necesariamente). Aún así, vamos a ver si podemos mejorar la precisión de la clase peor, la iris-virginica, que obtiene sólo un 90%:

=== Detailed Accuracy By Class ===

TP Rate	Class
0.98	Iris-setosa
0.94	Iris-versicolor
0.9	Iris-virginica

Clasificador de reglas PART

Time taken to build model: 0 seconds

=== Stratified cross-validation ===
 === Summary ===

Correctly Classified Instances	141	94 %
Incorrectly Classified Instances	9	6 %
Kappa statistic	0.91	
Mean absolute error	0.0482	
Root mean squared error	0.1794	
Relative absolute error	10.8379 %	
Root relative squared error	38.0567 %	
Total Number of Instances	150	

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	
0.98	0.06	Iris-setosa	
0.94	0.03	Iris-versicolor	
0.9	0.03	Iris-virginica	
Weighted Avg.	0.94	0.03	0.941 0.94 0.94 0.968

=== Confusion Matrix ===

a	b	c	<-- classified as
49	1	0	a = Iris-setosa
0	47	3	b = Iris-versicolor
0	5	45	c = Iris-virginica

Observemos primero la matriz de confusión para ver dónde se producen los errores. Vemos que PART asigna erróneamente 5 datos de la clase *c* a la clase *b*, 3 datos de la *b* a la clase *c* y 1 dato de *a*, a *b*. Ya sabemos que la clase que peor clasifica es la iris-virginica (la *c*) y en la matriz de confusión vemos que tiende a confundirla con la *b* (iris-versicolor). Vamos a intentar mejorar el porcentaje de aciertos de la clase *c*, utilizando un meta-clasificador, el **cost-sensitive-classifier**. Este clasificador se puede encontrar donde muestra la gráfica.

=== Confusion Matrix ===

```

a b c <-- classified as
49 1 0 | a = Iris-setosa
0 47 3 | b = Iris-versicolor
0 5 45 | c = Iris-virginica

```

The screenshot shows the Weka 3.6.0 Explorer interface. In the 'Classifier' list on the left, 'CostSensitiveClassifier' is highlighted with a red circle and a red arrow pointing to it. A text box next to the arrow says 'Meta clasificador sensible al coste'. The main window displays the following information:

Command: `1.0; 1.0 0.0 2.0; 1.0 2.0 0.0] -S 1 -W weka.classifiers.rules.PART --M2 -C 0.25 -Q 1`

ed cross-validation ===

```

====
Classified Instances      141      94 %
Classified Instances      9         6 %
Statistic                  0.91

```

Mean squared error: 38.0567 % of Instances: 150

Accuracy By Class ===

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.98	0	1	0.98	0.99	0.99	Iris-setosa
0.94	0.06	0.887	0.94	0.913	0.954	Iris-versicolor
0.9	0.03	0.938	0.9	0.918	0.959	Iris-virginica
0.94	0.03	0.941	0.94	0.94	0.968	

Confusion Matrix ===

```

a b c <-- classified as
49 1 0 | a = Iris-setosa
0 47 3 | b = Iris-versicolor
0 5 45 | c = Iris-virginica

```

The bottom status bar shows 'Log' and 'x 0'.

El **cost-sensitive-classifier** es un meta-clasificador, porque utiliza a otro clasificador base. En este caso, utilizaremos como clasificador base a PART. El **cost-sensitive-classifier** permite introducirle una matriz similar a la matriz de costes, de tal manera que podemos forzar a que el clasificador base (PART en este caso). Seleccionad PART como clasificador base. En la siguiente página veremos como introducir una matriz de costes.

Pinchar aquí para desplegar ventana de parámetros

Aquí se selecciona el clasificador base

Aquí se selecciona la matriz de costes

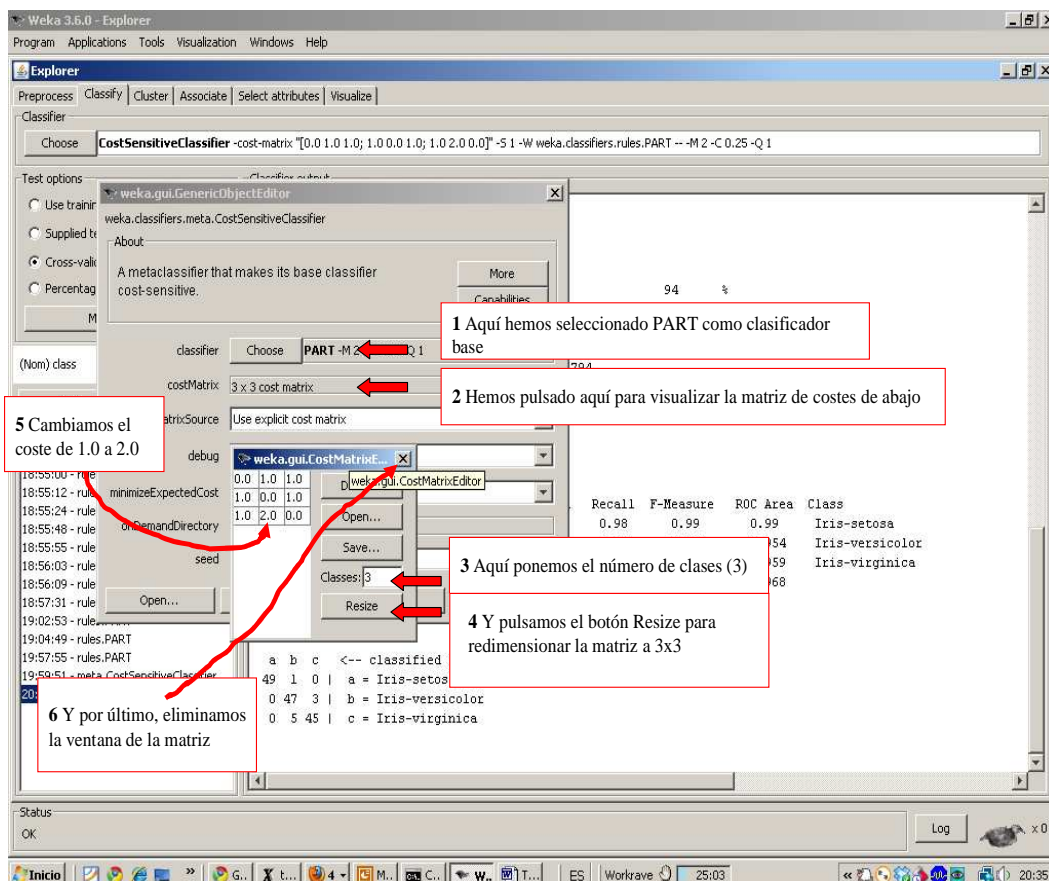
Recall	F-Measure	ROC Area	Class
0.98	0.99	0.99	Iris-setosa
0.94	0.913	0.954	Iris-versicolor
0.9	0.918	0.959	Iris-virginica
0.94	0.94	0.968	

```
a b c <-- classified as
49 1 0 | a = Iris-setosa
0 47 3 | b = Iris-versicolor
0 5 45 | c = Iris-virginica
```

Para seleccionar la matriz de costes, pinchar en **1x1 cost matrix (paso 2)**. En el campo **classes** pone 1, pero nuestro problema tiene 3 clases. Poner 3 en ese campo (**paso 3**) y pulsar el botón **Resize (paso 4)**. Aparecerá una matriz de 3x3, cuyos valores en la diagonal son ceros, y los de fuera de la diagonal son unos. Esto quiere decir que clasificar un dato correctamente no tiene coste (son los que están en la diagonal) y clasificarlo incorrectamente tiene un coste de 1. Sin embargo, según la matriz de confusión, sabemos que la clase que peor clasifica es la tercera (c), por lo que en la matriz del cost-sensitive-classifier pondremos un 2.0 en la posición en la que en la matriz de confusión aparece un 5 (**paso 5**). De esta manera, forzaremos a que PART intente minimizar los errores producidos al confundir la clase iris-virginica (c) con la clase iris-versicolor (b). Una vez escrito el 2.0 en la posición adecuada pulsar enter y eliminar la ventana de la matriz (**paso 6**). Finalizar pulsando **OK**.

=== Confusion Matrix ===

a	b	c	<-- classified as
49	1	0	a = Iris-setosa
0	47	3	b = Iris-versicolor
0	5	45	c = Iris-virginica



Ejecutamos el meta algoritmo cost-sensitive-classifier pulsando como de costumbre el botón **Star** y veremos que ahora los porcentajes de aciertos desglosados por clase son distintos a los anteriores:

Antes:

=== Detailed Accuracy By Class ===

TP Rate	Class
0.98	Iris-setosa
0.94	Iris-versicolor
0.9	Iris-virginica

Después:

=== Detailed Accuracy By Class ===

TP Rate	Class
0.98	Iris-setosa
0.88	Iris-versicolor
0.96	Iris-virginica

Como era de esperar, hemos mejorado la clase iris-virginica (*c*), pero a costa de empeorar la clase iris-versicolor (*b*). La matriz de confusión nos da información mas precisa sobre que clases se están confundiendo:

=== Confusion Matrix ===

```
a b c <-- classified as
49 1 0 | a = Iris-setosa
0 44 6 | b = Iris-versicolor
0 2 48 | c = Iris-virginica
```

Vemos que ahora 6 datos pertenecientes a la clase **iris-versicolor** se clasifican erróneamente como **iris-virginica**. Ejecutar de nuevo el cost-sensitive-classifier con otra matriz de costes, para intentar solventar este problema. En general, se puede comprobar que solucionar una clase suele ser a costa de perjudicar otra clase.

Otro metaclassificador que también permite especificar los costes mediante una matriz es **meta-cost**. Probadlo.

2.2 El Explorer, selección de atributos

Trataremos ahora el tema de la selección de atributos (eliminación de atributos redundantes e irrelevantes). Si hay un número excesivo de atributos, esto puede hacer que el modelo sea demasiado complejo y se produzca overfitting. En Weka, la selección de atributos se puede hacer de varias maneras. La más directa es usando la pestaña de **attribute selection**. Tenemos que seleccionar un método de búsqueda y un método de evaluación. A grandes rasgos existen tres posibilidades:

Evaluación de atributos. Por ejemplo:

Método de búsqueda = Ranker

Método de evaluación = InfoGainAttributeEval

Evaluación de conjuntos de atributos

Filter. Por ejemplo:

Método de búsqueda = Greedy Stepwise

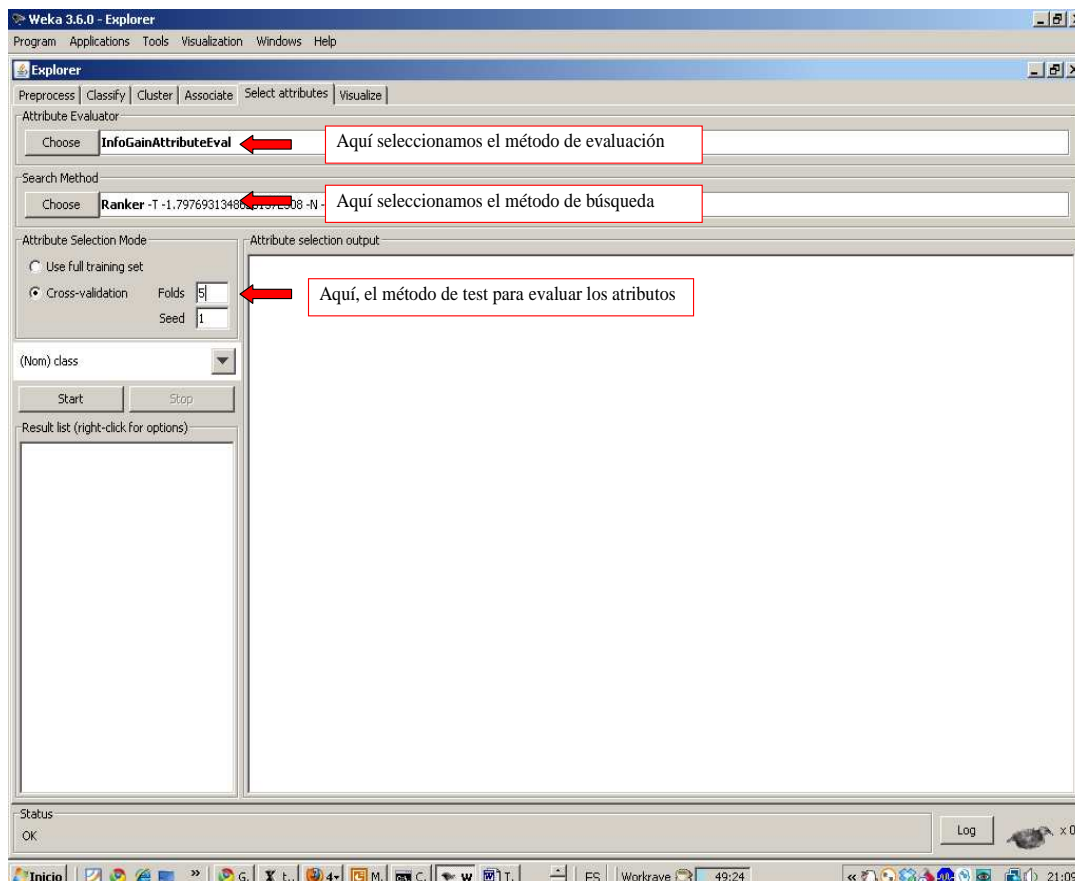
Método de evaluación = CfsSubsetEval

Método Wrapper. Por ejemplo:

Método de búsqueda = Greedy Stepwise

Método de evaluación = ClassifierSubsetEval

De momento vamos a seleccionar **Ranker**, con **InfoGain**. El método de test para evaluar los atributos será de **crossvalidation** de 5 hojas. Esto hará que la evaluación sea 5 veces más lenta, pero más precisa. En dominios con muchos datos o atributos, habrá que reducir este número, o usar **Use full training set**, para que Weka tarde un tiempo razonable (aunque eso reducirá la confianza que tengamos en el subconjunto de atributos seleccionado).



Pulsamos **Start**, y observamos los resultados:

=== Attribute selection 5 fold cross-validation (stratified), seed: 1 ===

average merit	average rank	attribute
1.386 +- 0.022	1.4 +- 0.49	4 petalwidth
1.368 +- 0.017	1.6 +- 0.49	3 petallength
0.711 +- 0.053	3 +- 0	1 sepallength
0.323 +- 0.059	4 +- 0	2 sepalwidth

Ranker nos ordenaba los atributos, y ha considerado que petalwidth y petallength son los mejores, cosa que ya sabíamos desde el principio, cuando visualizamos en la pestaña de Preprocess, el desglose de los valores de cada atributo por clase. Weka nos da dos informaciones: el **average merit** (y su desviación típica) y el **average rank** (y su desviación típica). El primero se refiere a la media de las correlaciones (medidas con InfoGain) en los cinco ciclos de validación cruzada. El average rank se refiere al orden medio en el que quedó cada atributo en cada uno de los cinco ciclos. Por ejemplo, como para sepallength y sepalwidth, la desviación típica es de cero, eso quiere decir que sepallength quedó como tercer atributo en los cinco ciclos de validación cruzada, y que sepalwidth quedó siempre el cuarto. Petalwidth y petallength debieron de quedar ambos a veces primero y a veces segundo, por eso el orden medio es de 1.4. En resumen, petalwidth y petallength son los mejores atributos, a bastante diferencia de los 2 siguientes. Sepalwidth se ve que es particularmente malo.

Vamos a probar ahora un método filter. Eso quiere decir que habrá que seleccionar, por ejemplo:

Método de búsqueda = Greedy Stepwise
Método de evaluación = CfsSubsetEval

Aunque podríamos probar con cualquier otro método de búsqueda (genético, best first, ...). El resultado es:

=== Attribute selection 5 fold cross-validation (stratified), seed: 1 ===

```
number of folds (%) attribute
0( 0 %) 1 sepallength
0( 0 %) 2 sepalwidth
5(100 %) 3 petallength
5(100 %) 4 petalwidth
```

Recordemos que CfsSubsetEval selecciona subconjuntos de atributos. La información anterior quiere decir que petallength y petalwidth fueron seleccionados en cada uno de los 5 folds de validación cruzada (o sea, siempre), mientras que sepallength y sepalwidth no fueron seleccionados en ningún fold (o sea, nunca). Nuevamente, petallength y petalwidth vuelven a ser seleccionados.

Por último, probaremos con un Wrapper, por ejemplo seleccionar lo siguiente:

Método de búsqueda = Greedy Stepwise
Método de evaluación = ClassifierSubsetEval

Mucho cuidado, porque un método Wrapper, requiere que se seleccione un clasificador base. Para ello, pincharemos sobre **ClassifierSubsetEval** para ver sus parámetros, y veremos que hay uno que es **Classifier** y que por omisión tiene el valor **ZeroR**. Pongamos por ejemplo el clasificador **PART**. Y pulsemos **Start**. El resultado ahora es:

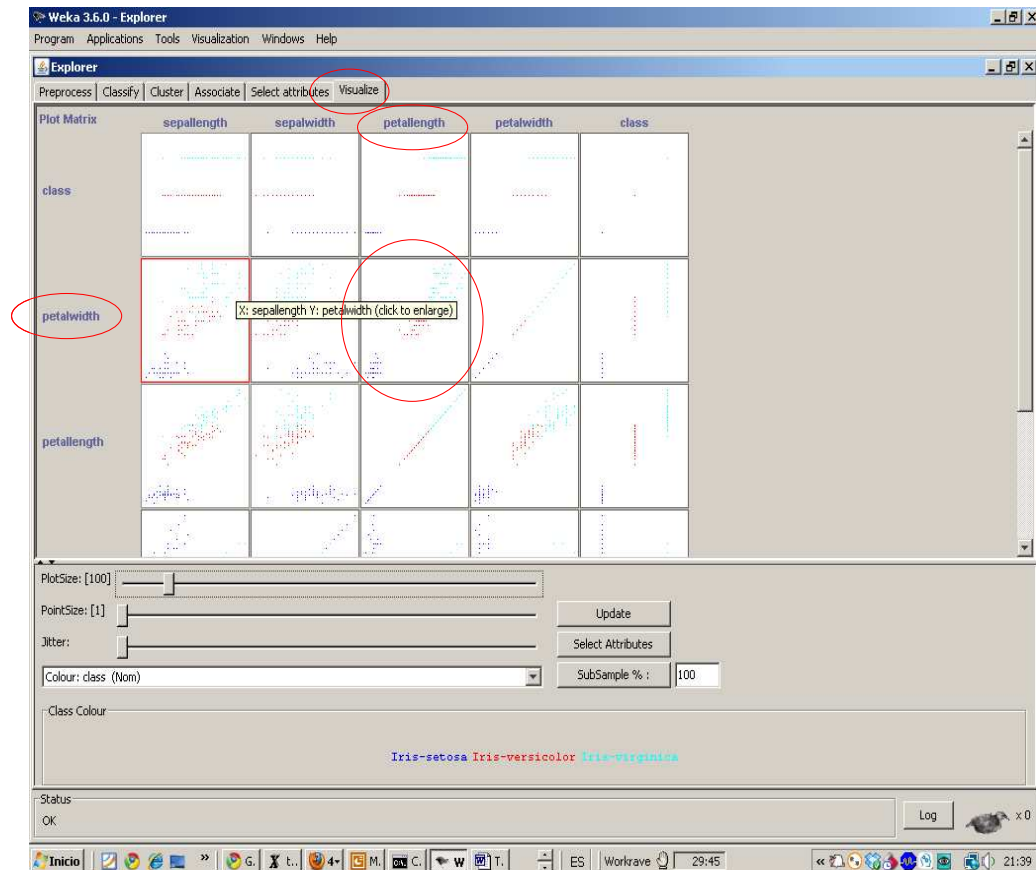
=== Attribute selection 5 fold cross-validation (stratified), seed: 1 ===

```
number of folds (%) attribute
1( 20 %) 1 sepallength
0( 0 %) 2 sepalwidth
5(100 %) 3 petallength
4( 80 %) 4 petalwidth
```

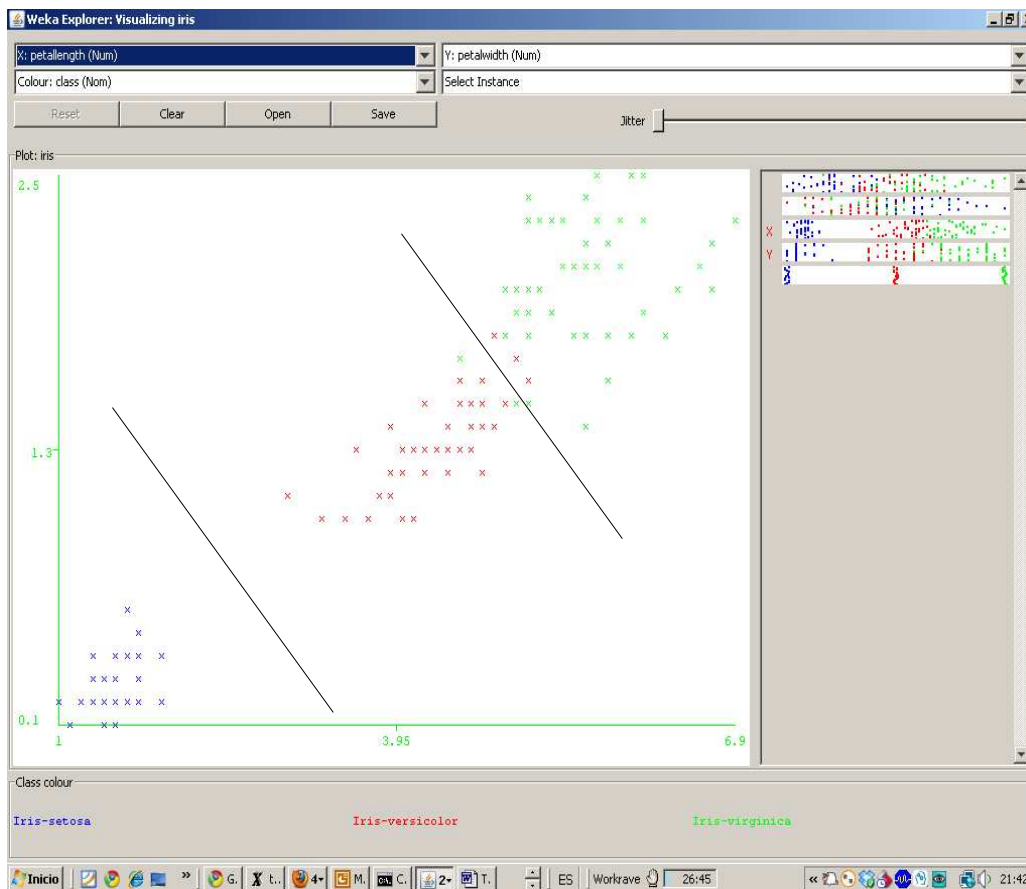
Nuevamente considera importantes los atributos del pétalo, aunque ahora el petalwidth sólo lo selecciona en 4 folds de los 5 de la validación cruzada. Aún así, petallength y petalwidth siguen siendo los mejores atributos.

De los varios métodos probados, se puede ver que los tres (Ranker, Filter y Wrapper) nos dan los mismos resultados. En este caso, no parece haber atributos que en bloque funcionen bien y que de manera aislada funcionen mal, por lo que Wrapper no merecería la pena (es el más lento y obtiene los mismos resultados que los demás).

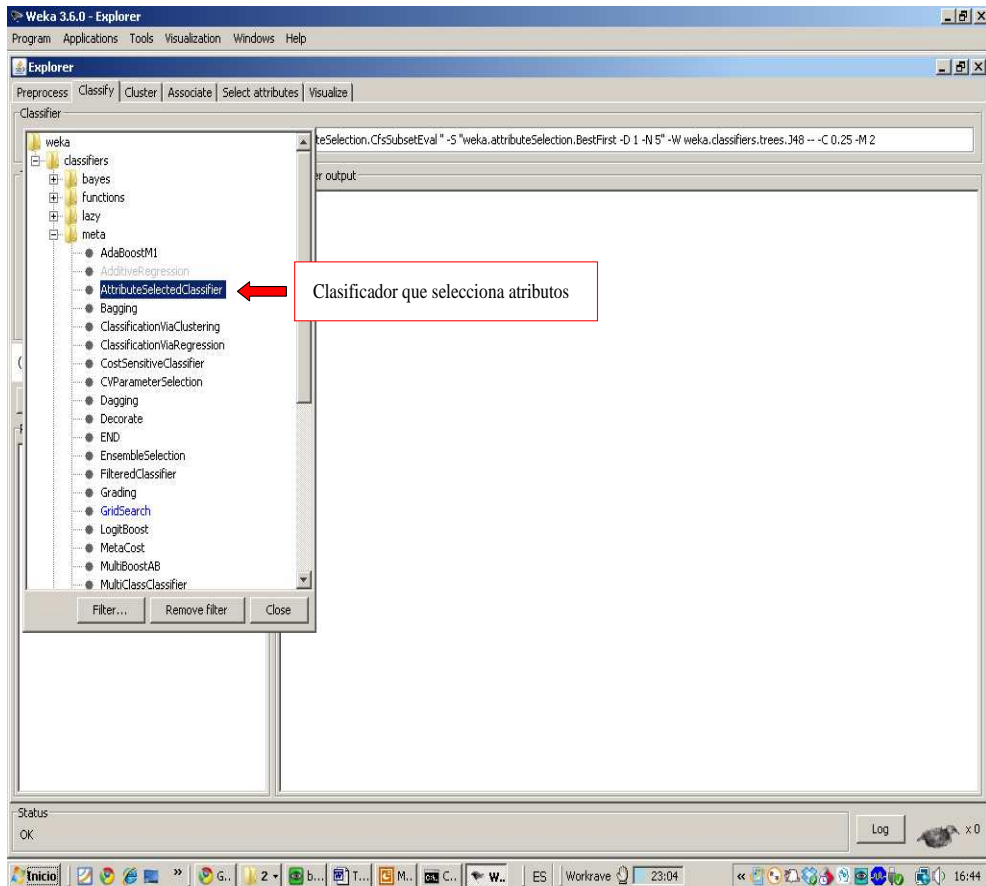
Una cuestión importante a tener en cuenta en este momento es que los datos originales no han sido modificados. La pestaña **attribute selection** en la que estamos, simplemente nos permite ver cuales son los atributos más relevantes, pero los datos siguen teniendo todos los atributos. Mas adelante veremos como seleccionar los atributos modificando realmente los datos. De momento, vamos a visualizar los atributos por parejas, utilizando los atributos más relevantes encontrados hasta el momento: petalwidth y petalength. Para ello, nos vamos a la pestaña de **Visualize**, donde se pueden visualizar todas las posibles parejas de atributos. Pincharemos sobre la pareja de atributos más relevante:



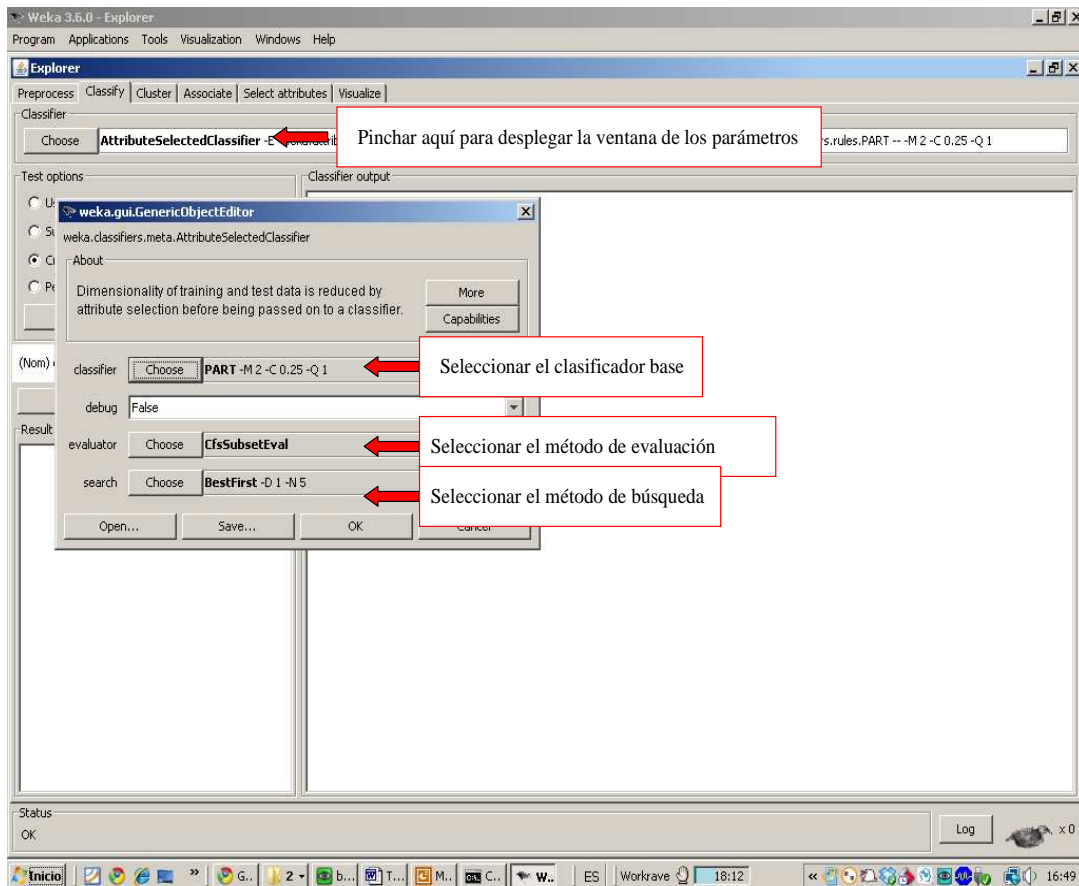
En la gráfica podemos ver porqué los atributos de los pétalos eran los más relevantes. Usándolos juntos, separamos casi perfectamente las tres clases. Ahora podemos ver porque la clase iris-setosa (azul oscuro) era la que se clasificaba mejor y porqué las otras dos (roja y verde) presentaban ciertos errores (es casi imposible trazar una frontera sencilla entre ambas clases que las separe de manera perfecta). Si se prueba con los dos atributos de los sépalos, se verá que hay mucho más solape entre las clases. A la hora de visualizar, es importante que los dos atributos no sean redundantes. Si lo son, ambos aportarán la misma información, con lo que la separación será menos clara.



Hasta ahora, hemos podido ver cuales son los atributos más relevantes, y hemos podido visualizar que tal separan las clases, por parejas. Pero todavía no hemos observado si la selección de atributos mejora (o empeora) el porcentaje de aciertos esperado, o si los modelos que se construyen son más complejos o más simples. Lo que necesitamos es un meta-clasificador que primero pase un filtro de selección de atributos, y después realice aprendizaje (y test) utilizando exclusivamente los atributos seleccionados. Dicho meta-clasificador se llama **attribute selected classifier** y lo encontraremos entre los classifiers meta, volviendo a la pestaña **classify** (donde usamos antes el PART y el J48).



Al tratarse de un metaclasificador, hay que especificar un clasificador base, además de los parámetros que definen la selección de atributos en Weka (un método de búsqueda y un método de evaluación de subconjuntos de atributos). Como clasificador base seleccionaremos PART y para la selección de atributos utilizaremos un método filter con búsqueda mejor primero (búsqueda = CfsSubsetEval y evaluación = BestFirst). Podéis utilizar cualquier otro método de búsqueda (genética, greedy, ...).



Pulsemos **Start**, y observaremos que el porcentaje de aciertos ha mejorado ligeramente. Pasamos de 94% a 95.3333%.

=== Stratified cross-validation ===

=== Summary ===

Correctly Classified Instances	143	95.3333 %
---------------------------------------	------------	------------------

Usando la barra de scroll, podemos subir hacia arriba y ver que el subconjunto de atributos seleccionado es el:

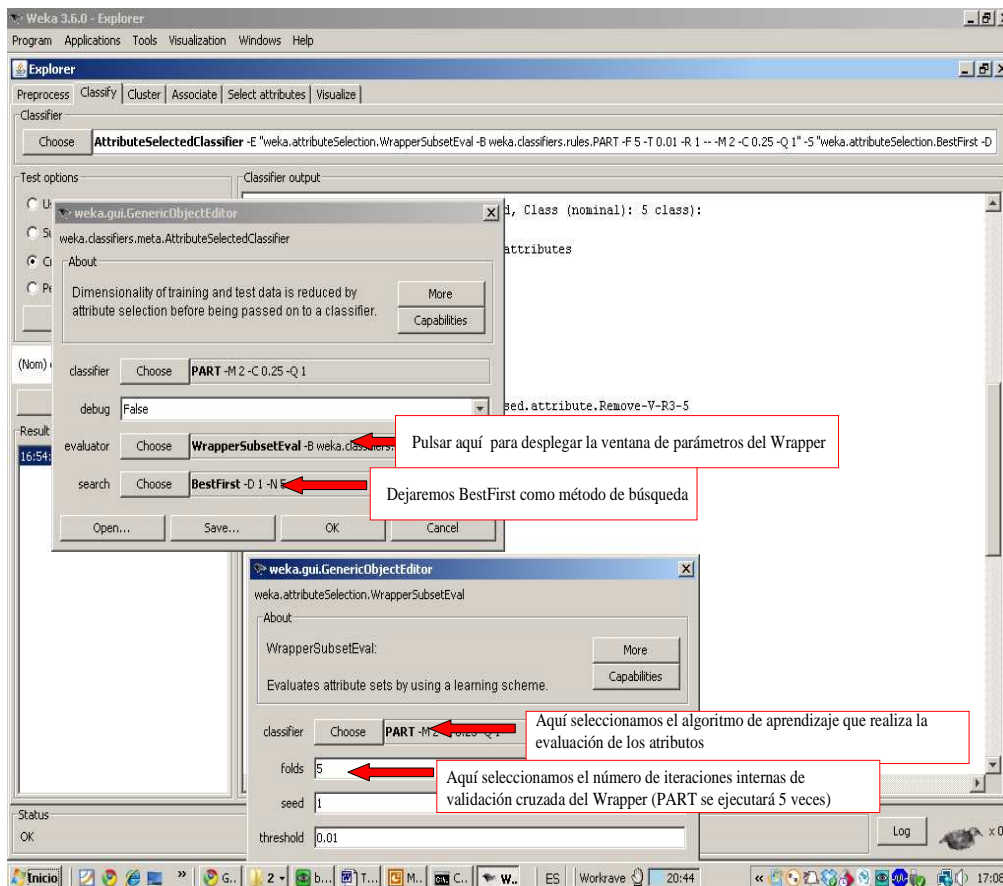
Selected attributes: 3,4 : 2

petallength

petalwidth

Probaremos ahora con un método de selección Wrapper. Para seleccionar los parámetros, mirar el gráfico en esta misma página. En un Wrapper, es necesario especificar:

- Evaluador de subconjuntos de atributos (**evaluator**): puede ser **ClassifierSubsetEval** o **WrapperSubsetEval**. La única diferencia entre ambos es que el primero no hace una validación cruzada interna y el segundo sí. En este caso elegiremos el segundo: será más preciso con validación cruzada, pero también más lento. Como hay pocos datos, será un tiempo razonable. El WrapperSubsetEval tiene sus propios parámetros, por lo que pincharemos encima y seleccionaremos:
 - El **classifier** que se usa para evaluar los subconjuntos de atributos (pondremos también PART).
 - El número de ciclos de validación cruzada (pondremos 5)
- Método de búsqueda (**search**): dejaremos el valor por omisión (**best-first**).



Ahora pulsemos **Start**, y observemos los resultados. Wrapper ha seleccionado sólo un atributo:

Selected attributes: 4 : 1
Petalwidth

Y el porcentaje de aciertos es menor que antes:

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances	139	92.6667 %
---------------------------------------	------------	------------------

A pesar de que en teoría Wrapper es el mejor selector de atributos, puede ocurrir en ocasiones que si hay pocos datos, los resultados obtenidos sean peores.

Por último, probaremos a usar **attribute selected classifier** con un método de búsqueda **Ranker**, el cual ordena los atributos según alguna medida de correlación, que vamos a fijar en **InfoGainAttributeEval**. Hay que tener cuidado, porque Ranker no es realmente un método de selección, sino un método de ordenación. Es decir, simplemente ordena los atributos de mejor a peor, pero los vuelve a seleccionar todos. Para que seleccione los n primeros, tenemos que cambiar un parámetro de Ranker. Para desplegar los parámetros de Ranker, pincharemos sobre Ranker. El parámetro a cambiar es **numToSelect**, que por omisión vale -1 (o sea, seleccionar todos). Pondremos aquí un 2, para que seleccione los dos mejores (porque ya sabemos que los dos atributos de pétalos son los mejores).

The screenshot shows the Weka 3.6.0 Explorer interface. The main window displays the configuration for the **AttributeSelectedClassifier**. The classifier is set to **PART -M2-C 0.25**. The evaluator is set to **InfoGainAttributeEval**. The search method is set to **Ranker -T**. A red box highlights the classifier and evaluator settings with the text: "Aquí seleccionamos Ranker y aquí InfoGainAttributeEval". A red arrow points to the search method dropdown, with a red box containing the text: "Pinchamos sobre Ranker para abrir su ventana de parámetros, que aparece más abajo". A second dialog box, **weka.gui.GenericObjectEditor**, is open for the **Ranker** class. In this dialog, the **numToSelect** parameter is set to **2**, highlighted by a red arrow and a red box with the text: "Cambiamos el -1 por 2, para seleccionar sólo los dos mejores atributos". The background shows a table of results with columns: Precision, Recall, F-Measure, ROC Area, and Class. The table contains three rows of data for different classes: Iris-setosa, Iris-versicolor, and Iris-virginica.

Precision	Recall	F-Measure	ROC Area	Class
0.98	0.99	0.99	0.99	Iris-setosa
0.9	0.891	0.941	0.945	Iris-versicolor
0.9	0.9	0.9	0.945	Iris-virginica

Pulsamos Start, y efectivamente vemos que selecciona:

Ranked attributes:

1.418 3 petallength

1.378 4 petalwidth

Selected attributes: 3,4 : 2

Y que los resultados son:

=== Stratified cross-validation ===

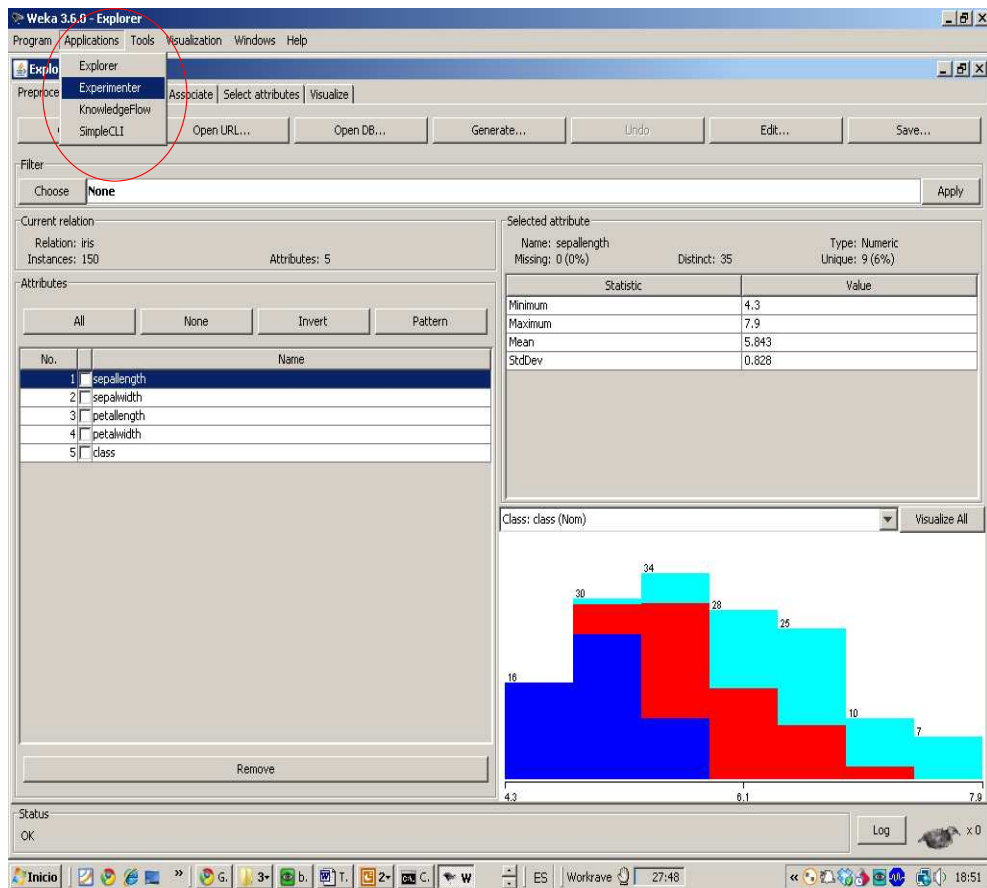
=== Summary ===

Correctly Classified Instances	143	95.3333 %
---------------------------------------	------------	------------------

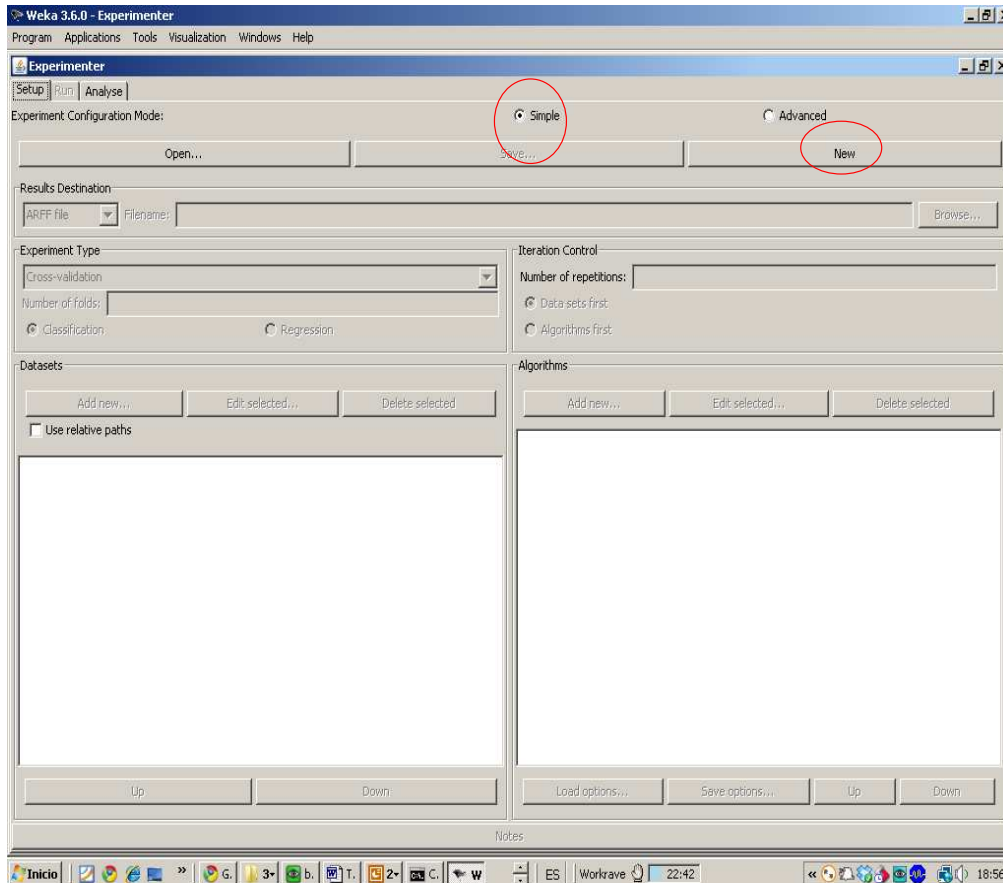
Ligeramente mejores que el 94% inicial.

3.0 El Experimenter

El Experimenter sirve para aplicar varios algoritmos de aprendizaje automático sobre distintos conjuntos de datos y determinar de manera estadística cual se comporta mejor. Es decir, el Experimenter nos dirá si las diferencias aparentes en porcentajes de aciertos de distintos algoritmos son estadísticamente significativas, o son debidas al azar. Para arrancar el Experimenter, debemos ir a la barra de herramientas de Weka, que está en la parte superior, y en el menú **Applications**, seleccionar **Experimenter**.



Las fases de uso de Experimenter son **Setup** (configura), **Run** (ejecuta) y **Analyse** (análisis estadístico), las cuales se pueden ver en las pestañas superiores del Experimenter. Comenzaremos configurando nuestro experimento. Tenemos dos opciones: **Simple** y **Advanced**. Usaremos la **Simple**, que ya está marcada por omisión. Ahora podemos o bien abrir un fichero de configuración de un experimento anterior (**Open**) o bien crear un nuevo experimento (**New**), que es lo que vamos a hacer.



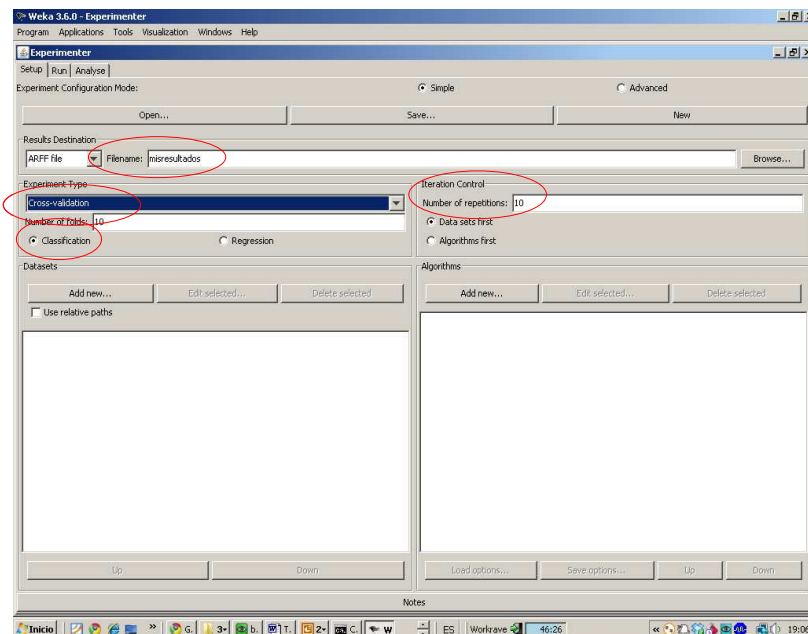
Ahora hay que configurar el experimento. Primero, le doy un nombre al fichero que va a contener los resultados de las distintas validaciones cruzadas del experimento (**filename** = misresultados). Por omisión, el tipo del fichero es arff. Realmente, nosotros no vamos a trabajar directamente con este fichero, aunque es conveniente que los resultados se vayan almacenando en algún sitio. A continuación hay que definir el tipo de experimento, que puede ser:

- **Crossvalidation** (de 10 hojas, aunque esto último se puede cambiar)
- **Train-test percentage split (data randomized)**: primero desordena aleatoriamente los datos y después coge el primer 66% para construir el clasificador y el 34% restante para hacer el test (calcular el porcentaje de aciertos esperado). El 66% es un parámetro que puede ser cambiado.
- **Train-test percentage split (order preserved)**: hace lo mismo que el anterior pero no desordena el conjunto de datos antes de dividirlos en entrenamiento y test

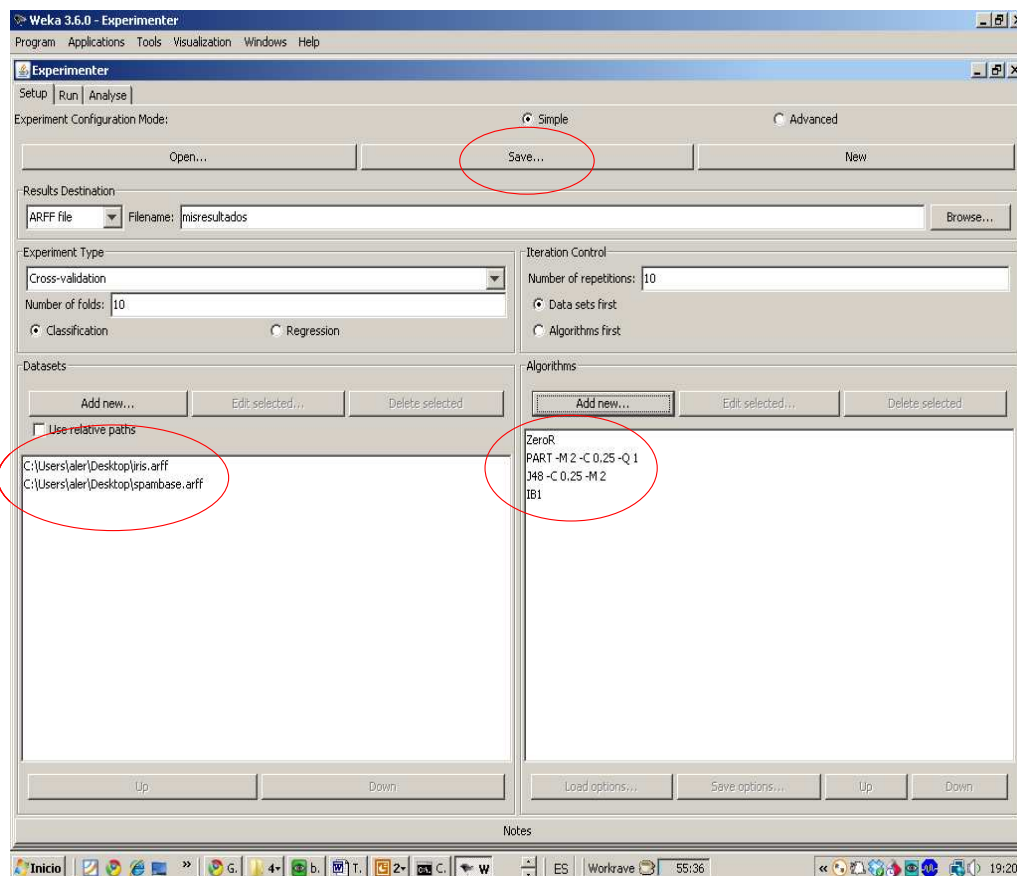
Después hay que elegir si el problema es de **classification** o de **regression** (se trata del primero). A continuación, hay que especificar el iteration control. Para realizar los tests estadísticos, Weka repite la validación cruzada n veces (por omisión 10 veces). Es decir, realiza 10 validaciones cruzadas de 10 hojas cada una. Esto implica que el algoritmo de aprendizaje será ejecutado 100 veces.

En un momento, pondremos en la configuración los conjuntos de datos (**datasets**) con los que vamos a trabajar y los algoritmos que vamos a aplicar sobre ellos. El algoritmo que sigue el Experimenter es (si seleccionamos algorithms first, el bucle externo será el de los algoritmos, y el interno el de los datasets, pero ahora mismo es irrelevante):

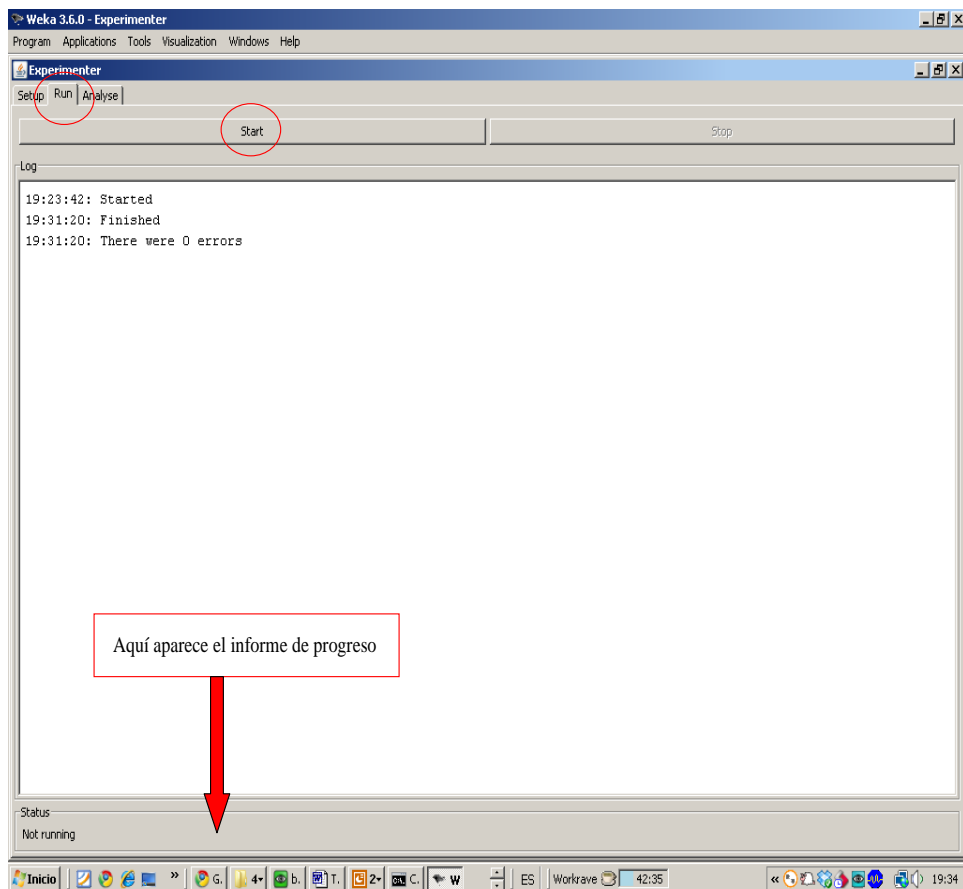
- Repite para cada dataset
 - Repite para cada algoritmo
 - Repite “Number of iterations”
 - Haz validación cruzada



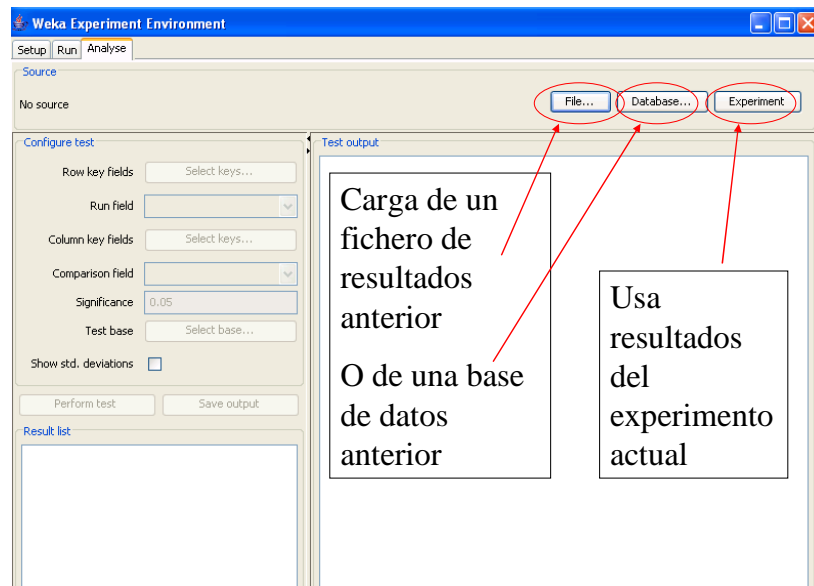
Ahora, añadiremos dos ficheros de datos (**datasets**) con el botón **New** de la izquierda y cuatro algoritmos con el botón **New** de la derecha. Los datasets serán Iris.arff y SpamBase.arff. NOTA: SpamBase.arff tiene bastantes datos y resulta extraordinariamente lento, con lo que en una primera pasada de este tutorial, es mejor no ponerlo (poned sólomente iris.arff). Si se ponen varios datasets, es importante que la palabra que aparece tras @relation en la cabecera del fichero sea distinta, ya que si es la misma, el Experimenter considerará que se trata del mismo dataset (aunque sean distintos ficheros) y sólo mostrará los resultados para uno de los datasets. Los algoritmos serán: ZeroR (rules), J48 (trees), PART (rules), IB1 (lazy). A los algoritmos podemos cambiarle los parámetros si queremos, pinchando sobre ellos como de costumbre. De hecho, podemos poner el mismo algoritmo varias veces, con parámetros distintos. No nos olvidemos de salvar el fichero de configuración del experimento (**Save**).



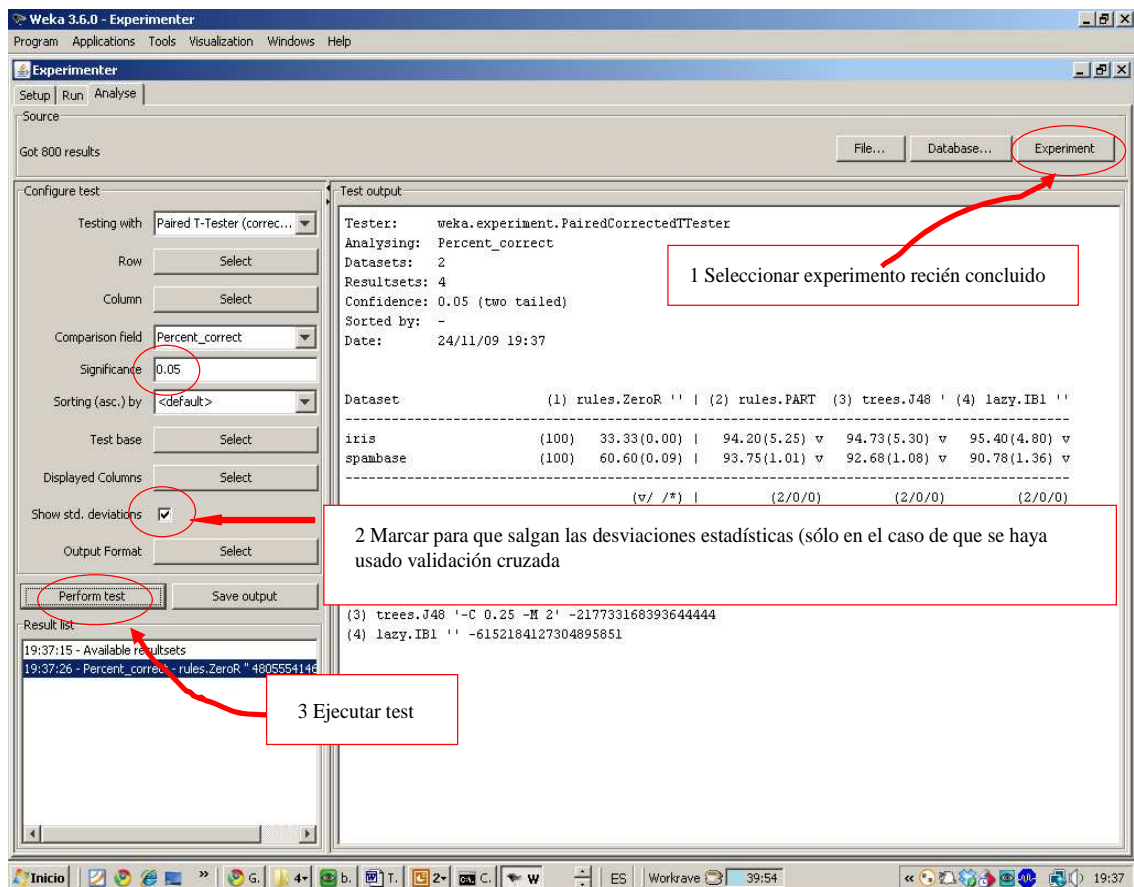
Ahora ejecutemos el experimento: pestaña **Run**, y pulsar **Start**.



Por último, pasaremos a la pestaña de **Analyse**, para realizar el análisis estadístico. Los resultados sobre los que realizar este análisis pueden venir de un fichero que contenga resultados de un experimento antiguo (**File**), de una base de datos (**Database**), o del experimento actual (**Experiment**). Seleccionaremos **Experiment**.



Para terminar, ejecutamos el test estadístico. Primero pulsamos en **Experiment** (caso de que no lo hayamos hecho antes), para seleccionar los resultados del experimento que acabamos de realizar. Segundo, marcamos la opción de mostrar las desviaciones típicas **show std. deviations** (caso de que se haya usado crossvalidation y queramos ver las desviaciones con respecto a la media). El resto de parámetros se pueden dejar como están. Comentar que Weka utiliza un t-test por parejas, corregido, basado en las 10 validaciones cruzadas de 10 hojas. Para los detalles, consultar el libro de Weka. El Experimenter encuentra que las diferencias son estadísticamente significativas con un nivel alfa (**significance**) del 0.05. Esto mas o menos quiere decir que la probabilidad de que el Experimenter os diga que una diferencia es significativa cuando realmente es debida al azar, es del 5%. Tercero, ejecutamos el test propiamente dicho (**Perform Test**). Aparecerá una tabla mostrando los resultados.



La tabla mostrada por el Experimenter tiene este aspecto:

Dataset	(1)	rules.ZeroR		(2) rules.PART	(3) trees.J48	(4) lazy.IB1 "
iris	(100)	33.33(0.00)		94.20(5.25) v	94.73(5.30) v	95.40(4.80) v
spambase	(100)	60.60(0.09)		93.75(1.01) v	92.68(1.08) v	90.78(1.36) v
		(v/ *)		(2/0/0)	(2/0/0)	(2/0/0)

En las filas aparecen los dos datasets (iris y spambase), mientras que en las columnas aparecen los 4 algoritmos utilizados (ZeroR, PART, J48 e IB1). El algoritmo con respecto al cual se hace la comparación, es el de más a la izquierda (en este caso, el ZeroR), aunque eso se puede cambiar. Tiene sentido comparar con ZeroR, puesto que necesariamente hay que superar los resultados del clasificador de la clase mayoritaria. En este caso, para el dominio Iris, ZeroR obtiene un 33.33% de aciertos, mientras que PART obtiene un 94.20%, con una desviación típica de 5.25. Al lado de los resultados de cada algoritmo, puede aparecer una **v**, un *****, o nada. La **v** del 94.20% significa que la mejora de PART sobre ZeroR es estadísticamente significativa. En este caso, todos los resultados tienen **v**, por lo que todos representan diferencias estadísticamente significativas. Si apareciera un *****, eso querría decir que el empeoramiento es estadísticamente significativo. Y si no apareciera nada, eso significaría que la diferencia no es estadísticamente significativa. Es decir, que la diferencia podría ser debida al azar, y que no podemos afirmar que un algoritmo sea mejor que el otro, o al contrario.

La última línea de la tabla:

(v/|*) | (2/0/0) (2/0/0) (2/0/0)
(mejor/igual/peor)

simplemente cuenta en cuantos dominios el algoritmo es significativamente mejor que ZeroR, en cuantos es igual, y en cuantos es significativamente peor. En este caso, PART es mejor en los dominios, y por tanto es igual o peor en 0 dominios (2/0/0). Lo mismo ocurre con el resto de algoritmos.

Si queremos, podemos cambiar el algoritmo con el que se comparan todos los demás. Esto se puede hacer pulsando el botón **Test Base** y cambiando el algoritmo. Por ejemplo, la siguiente tabla compara todos los algoritmos con IB1.

Dataset	(4)	lazy.IB1 "	-6	(1) rules.ZeroR	(2) rules.PART	(3) trees.J48 '
iris	(100)	95.40(4.80)		33.33(0.00) *	94.20(5.25)	94.73(5.30)
spambase	(100)	90.78(1.36)		60.60(0.09) *	93.75(1.01) v	92.68(1.08) v
		(v/ *)		(0/0/2)	(1/1/0)	(1/1/0)

Aquí podemos ver que ZeroR es significativamente peor que IB1 (33.33*), y que no hay diferencias estadísticamente significativas PART/IB1 y J48/IB1, en el dominio iris. Sin embargo, PART y J48 son estadísticamente mejores que IB1 en el dominio SpamBase.

