Universidad del País Vasco
Euskal Herriko Unibertsitatea

Departamento de Arquitectura y Tecnología de Computadores
Konputagailuen Arkitektura eta Teknologia Saila

Universidad del País Vasco
Euskal Herriko Unibertsitatea
KONPUTAGAILUEN ARKITEKTURA ETA TEKNOLOGIA SAILA
DEPARTAMENTO DE ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

An update of the J48Consolidated WEKA's class: CTC algorithm enhanced with the notion of coverage

EHU-KAT-IK-02-14

Informe de Investigación

Igor Ibarguren
Jesús Mª Pérez
Javier Muguerza
Ibai Gurrutxaga
Olatz Arbelaitz

Ekaina 2014

An update of the J48Consolidated WEKA's class: CTC algorithm enhanced with the notion of coverage[¥]

Igor Ibarguren, Jesús Mª Pérez[*], Javier Muguerza, Ibai Gurrutxaga, Olatz Arbelaitz

Dept. of Computer Architecture and Technology, University of the Basque Country (UPV/EHU)
M. Lardizabal, 1, 20018 Donostia, Spain
{igor.ibarguren, txus.perez, j.muguerza, i.gurrutxaga, olatz.arbelaitz}@ehu.es
http://www.sc.ehu.es/aldapa

Index

Index of Tables

¥ If you want to refer to CTC in a publication, see “Citation policy” section.

* Corresponding author. E-mail address: txus.perez@ehu.es

Index of figures

Introduction

This document aims to describe an update of the implementation of the J48Consolidated class within WEKA platform [1]. The J48Consolidated class implements the CTC algorithm [2][3] which builds a unique decision tree based on a set of samples. The term “Inner Ensembles” was recently coined in [4] for this kind of techniques, applying ensemble techniques to build single models.

The J48Consolidated class extends WEKA’s J48 class which implements the well-known C4.5 algorithm [5]. This implementation was described in the technical report *"J48Consolidated: An implementation of CTC algorithm for WEKA"* [6].

The main, but not only, change in this update is the integration of the notion of coverage in order to determine the number of samples to be generated to build a consolidated tree. We define *coverage* as the percentage of examples of the training sample present in –or covered by– the set of generated subsamples. So, depending on the type of samples that we use, we will need more or less samples in order to achieve a specific value of coverage. More details about this issue can be seen in the “Coverage” section.

Before examining the coverage issue in depth, in the next section, “What is new?”, we will describe the rest of the changes in this update.

What is new?

All changes in this update are related to the way to generate the samples that will be used to build consolidated trees. So, the implementation of the CTC algorithm does not vary from the previous version.

We can classify the major changes in three sections:

- *Using the notion of coverage*
- *Enabling the change of class distribution of the generated samples for multi-class datasets*
- *Taking some exceptional characteristics of some datasets into account*

Using the notion of coverage

We have already described this concept briefly and we will devote the rest of the paper to deepen this issue.

Regarding this concept, we have added two new measures to the J48Consolidated class. For this purpose, we have redefined the `enumerateMeasures()` function which adds the two new measures below to the enumeration of the measure names of the J48 class:

- `measureNumberSamplesByCoverage`: the number of samples necessary to achieve the coverage indicated by the corresponding option.
- `measureTrueCoverage`: the true coverage of the examples of the original sample achieved by the set of samples generated for the consolidated tree. Although a coverage value had

been indicated as a parameter to determine the number of samples to be generated, this value is the minimum value to be guarantee in *the least favored class*, so the rest of classes would be better covered. As a consequence, the achieved true coverage with the generated samples should be bigger than the one indicated via parameter.

Enabling the change of class distribution of the generated samples for multi-class datasets

In the implementation of the J48Consolidated class, three types of samples can be generated to build consolidated trees according to the way the distribution of the minority class is changed (or is not) , i.e., the `newDistrMinClass` parameter:

- *Bootstrap samples* (-1 value): The instances will be randomly selected without taking their classes into account. This type of sample is used by the well-known Bagging ensemble, for example.
- *Stratified samples* (-2 value): The instances will be randomly selected but taking their classes into account and, therefore, the original class distribution will be maintained. In order to generate a set of different samples the size of these is reduced in a percentage value (by `bagSizePercent` parameter).
- *Class distribution modified samples* (a value between 0 and 100): The instances will be randomly selected taking their classes into account in order to achieve the specified proportion of examples of the minority class into the new subsamples.

In the above implementation, the third option, *class distribution modified samples*, was restricted to two-class datasets only. In this implementation we added the possibility to use multi-class datasets also. However, since a unique value can be set and the dataset has more than two classes, the only possibility accepted to change the original class distribution will be in order to balance all classes into the generated samples.

When the dataset has two classes, the value used to balance the generated samples is 50 (the proportion of the examples of the minority class). We will use this same value in order to indicate that we want to balance the samples to be generated also when we use multi-class datasets.

Taking some exceptional characteristics of some datasets into account

Based on our experiments along different works, the CTC algorithm achieves the best results when we use subsamples as big as possible and with a balanced class distribution using random sub-sampling without replacement. This occurs, in general terms, for a wide set of classification problems when the Area Under ROC Curve (AUC) is used as evaluation measure. Nowadays, AUC is widely used and is considered as one of the most robust measures in order to evaluate the classification capacity of classifiers [7][8]. Because of all this we set this kind of samples as default parameters for J48Consolidated classifiers, that is, `newDistrMinClass` parameter is set to 50 and `bagSizePercent` parameter is set to -2 (*maxSize*). This does not mean that other kind of sampling could not achieve better results for a specific dataset, specially if the evaluation criterion used is different, such as error rate, kappa or others.

When we use J48Consolidated classifier with the default parameters, that is to say, in order to balance the subsamples of a multi-class dataset (the most general case), the size of each class into the generated samples will be the same that the minority class has into the original sample. This means the minority class into the original sample is entirely copied and, in the case of the rest of the classes, random sub-sampling without replacement is used to complete the rest of the classes of the subsample.

In this sampling process, we can find two exceptional situations:

- *The original dataset is already balanced*: If the original dataset is already balanced (for example, the well-known *iris* dataset from *UCI repository* [9]), this sampling process would generate a set of samples all identical to the original sample. Thus, the CTC algorithm would build the same decision tree the C4.5 algorithm would. The CTC algorithm works but the result does not make much sense.

  In this situation, taking previous experiments into account, we decided to generate stratified samples (thus, in this case, balanced too) but reducing their size to 75% of the original sample size. This value is set in a member attribute of the J48Consolidated class, `m_bagSizePercentToReduce`, but it has not been considered a parameter (an option in WEKA's terminology) of J48Consolidated classifier in order to make easier the parameter's setting. Of course, the user will always have the possibility to set a different value indicating the corresponding value to use stratified samples.

- *The number of examples of the minority class is very low*: There are datasets which contain only one or two examples of the minority class (datasets such as *hypothyroid* or *audiology* from the *UCI repository*). If we generate balanced subsamples with this class size, we achieve very small samples and, as consequence, with very low knowledge, without forgetting the large amount of samples that would be required to cover most of examples.

  To mitigate this effect, we determined a minimum value of examples that the classes have to contain into subsamples as a percentage related to the original sample's size. It has been declared as a member attribute of the J48Consolidated class, `m_minExamplesPerClassPercent`, and initialized with 2%. If any class does not have enough examples to achieve this value, the examples will be oversampled randomly in order to get it.

  We carried out different trails comparing different values of this attribute, as well as with and without oversampling examples, and this choice seems a good option although we are aware a deeper work would be of interest.

Coverage

As we mentioned above, in this new version of the implementation of the J48Consolidated classifier, the user can determine the number of samples to be generated based on the percentage of examples of the original sample that is wanted to be covered, asymptotically, by the set of generated samples. We called this the *coverage* value. This value depends on the resampling method chosen to generate the samples. Therefore, we will have a different expression to calculate this value according to the kind of the obtained samples.

We want to know what percentage of examples from the original sample is contained in the set of generated samples. In other words, what probability has an example in the training sample of being in at least one sample of the set. Let us start with Bootstrap samples and, then, we will continue with the others.

Bootstrap samples

Bauer and Kohavi already determined in [10] the probability that an instance has of being selected at least once in a bootstrap sample as 63.2%. Let n be the number of examples of the original sample. Bootstrap sampling consist of selecting randomly n instances with replacement. So,

The probability of an example being selected from the original sample: $1/n$

The probability of an example not being selected from the original sample: $1-1/n$

The probability of an example not being selected from the original sample in n times: $(1-1/n)^n$

At last, the probability of an example being selected from the original sample in n times is $1-(1-1/n)^n$. For large n, this is about $1-1/e \approx 63.2\%$.

In our case, we want to know the probability of being selected for the set of samples, that is, if we generate m samples, the probability of being selected in $n \times m$ times, that is, $1-1/e^m = 1-e^{-m}$.

Therefore, the number of samples, m, which we also denote N_S, required to achieve a specific value of *coverage* is calculated by the next expression:

$$N\_S = -\ln(1-coverage) \quad (1)$$

Stratified samples

In this kind of samples, instances are randomly selected without replacement until the size of sample achieves a percentage (`bagSizePercent` parameter) of the training sample and, besides, taking into account that the class distribution has to be maintained. Let n be the number of examples of the original sample. Let s be the number of examples of the subsamples to be generated. We denote r as the ratio between the size the subsample and the size of the whole sample. Finally, let m be the number of samples to be generated. So,

The probability of an example being selected from the original sample: $1/n$

The probability of s examples being selected from the original sample: $s/n = r$

The probability of an example not being selected from the original sample: $1-r$

The probability of an example not being selected from the original sample in m samples: $(1-r)^m$

At last, the probability of an example being selected from the original sample in the m samples is $1-(1-r)^m$.

We do not take into account that the class distribution should be maintained into the generated subsamples. However, if we subsample each of the classes using the same r ratio, we will get a stratified sub-sample and with the desired size ($r \times n$). Thus, the expression of coverage for this kind of subsamples is the mentioned one.

Finally, we can calculate the number of samples, m or N_S, required to achieve a specific value of *coverage* as:

$$N\_S = \log_{(1-r)}(1-coverage) \quad (2)$$

On the other hand, if we use replacement in the process of selection of instances, then:

The probability of an example being selected from the original sample: $1/n$

The probability of an example not being selected from the original sample: $1-1/n$

The probability of an example not being selected from the original sample in $r \times n \times m$ times: $(1-1/n)^{rnm}$

At last, the probability of an example being selected from the original sample in the m samples is $1-(1-r)^{rnm}$. And for large n, this is about $1-1/e^{rm}$, that is, $1-e^{-rm}$.

And we can calculate the number of samples as below:

$$N\_S = -\ln(1-coverage)/r \quad (3)$$

That is, if the ratio between sample and sub-samples' size is of 100% ($r = 1$), we have Bootstrap samples and (3) expression becomes (1).

Class distribution modified samples

We generate class distribution modified samples sub- or under-sampling the original sample without replacement. For this purpose, we have to subsample each class adequately based on the new class distribution and the indicated size. Anyway, in order to achieve a given coverage with a set of this kind of subsamples we have to guarantee this value in *the least favored class*, that is, the class whose ratio between the amount of examples in the subsamples and the amount of examples in the original sample is the lowest one. With a given number of samples, we get the corresponding coverage value in the least favored class and, as a consequence, the coverage value for the rest of the classes will be equal or higher.

So, this case becomes the same case we have with stratified samples but using the ratio of the size of the least favored class between the subsample and the whole sample, instead of the sizes of the whole subsample and sample. Then, let r_{lfc} be the ratio between the amount of examples of the least favored class from the subsamples and the amount of examples of the same class from the original sample and let m be the number of samples to be generated, we get that the probability of an example being selected from the original sample in the m samples is $1-(1-r_{lfc})^m$.

Therefore, the expression to calculate the number of samples, *N_S*, is:

$$N\_S = \log_{(1-r_{lfc})}(1-coverage) \quad (4)$$

In this case, in contrast with the rest of the seen expressions, the value of number of samples varies, in addition with the coverage value, also with a characteristic of the dataset, that is, with the class distribution. For example, in order to generate balanced subsamples, the lower the proportion of the examples of the minority class is, the more subsamples will be necessary to achieve the same coverage (as we can observed in the next section).

Examples for a set of datasets

In this section we will present the values of number of samples to be generated to achieve a certain value of coverage which varies from 10% to 99.9% for the different expressions of coverage here presented. As we mentioned, when we use *class distribution modified samples*, since the number of subsamples changes based on the class distribution of the original sample, by way of example we have selected a well-known set of datasets whose subsample numbers for different coverage values will be shown in the next tables. In particular we downloaded the first jar file available in the section “Collections of Datasets” (http://www.cs.waikato.ac.nz/ml/weka/datasets.html) from the website of *Weka Data Mining Software*[1], containing 37 (actually 36!) classification problems, originally collected from the UCI repository [9].

Table 1 shows the characteristics of the datasets; name, number of instances, number of attributes and number of classes and, separately, number of instances and its proportion regarding with the size of the whole sample of minority and majority classes, respectively. The datasets are ordered based on the proportion of the number of examples into the minority class, from the smallest to the biggest. At the bottom of the table, a summary of these characteristics with the average value, the median, and the minimum and maximum values can be observed.

First of all, in order to determine the number of samples required for a specific coverage value, we need to know whether some of the exceptional situations mentioned in the above section (Taking some exceptional characteristics of some datasets into account) occur taking into account the characteristics of these datasets:

- *The original dataset is already balanced*: As it can be observed in Table 1 there are 3 datasets which have the same amount of examples in the minority class and in the majority

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

class (these values are marked in bold in the table), thus, they are balanced. Specifically, these datasets are: *Vowel*, *segment* and *iris*, and there are marked with the symbol (*). So, as we mentioned before, in these datasets the generated samples will be under-sampled to 75% of the size of the whole dataset maintaining their class distribution, that is, being stratified, and in this situation, being balanced.

- *The number of examples of the minority class is very low*: There are 8 datasets whose minority class' size does not reach the determined minimum, 2% of the dataset's size. They are located in the first 8 positions of Table 1. As we mentioned before, when we want to change the class distribution of the subsamples to be generated and the dataset is multi-class, this implementation only accepts to balance the classes. In this set of datasets, there are 18 two-class datasets and the other 18 datasets are multi-class. In normal conditions, the minority class' size determines the size of all of the classes into the subsamples in order to be balanced. When the minority class' size of a dataset does not reach the 2% set, this latter value (2%) will determine the size of the classes. Table 2 contains some values inferred from the characteristics of the datasets in order to determine the size of the subsamples to be generated, when we want to balance the subsamples and using the maximum possible size, *maxSize* value for `bagSizePercent` parameter. So, “MinCoverByClass” column indicates the minimum amount of examples to be covered by each class into the subsamples. If the number of examples of the minority class is lower than this value, the latter one will be set as the size of all classes, “SizeOfClasses” column, in otherwise, it will be the minority class' size. The next column, “Size”, refers to the subsamples' size (SizeOfClasses value multiplied by the number of classes) and, finally, “%Size” is the proportion of this size with respect to the whole sample's size. As it can be observed at the end of Table 2, the subsamples size is, in average, around a 60% of the whole datasets' size.

				Minority class		Majority class	
Dataset	#Instances	#Attributes	#Classes (Distinct)	#Instances	%	#Instances	%
hypothyroid	3772	30	4	2	0.05%	3481	92.29%
primary-tumor	339	18	21	1	0.29%	84	24.78%
audiology	226	70	24	1	0.44%	57	25.22%
anneal	898	39	5	8	0.89%	684	76.17%
anneal.ORIG	898	39	5	8	0.89%	684	76.17%
soybean	683	36	19	8	1.17%	92	13.47%
lymphography	148	19	4	2	1.35%	81	54.73%
autos	205	26	6	3	1.46%	67	32.68%
letter	20000	17	26	734	3.67%	813	4.07%
zoo	101	18	7	4	3.96%	41	40.59%
Glass	214	10	6	9	4.21%	76	35.51%
sick	3772	30	2	231	6.12%	3541	93.88%
balance-scale	625	5	3	49	7.84%	288	46.08%
Vowel (*)	990	14	11	**90**	**9.09%**	**90**	**9.09%**
segment (*)	2310	20	7	**330**	**14.29%**	**330**	**14.29%**
hepatitis	155	20	2	32	20.65%	123	79.35%
vehicle	846	19	4	199	23.52%	218	25.77%
splice	3190	62	3	767	24.04%	1655	51.88%
breast-cancer	286	10	2	85	29.72%	201	70.28%
german_credit	1000	21	2	300	30.00%	700	70.00%
waveform	5000	41	3	1653	33.06%	1692	33.84%
iris (*)	150	5	3	**50**	**33.33%**	**50**	**33.33%**
wisconsin-breast-cancer	699	10	2	241	34.48%	458	65.52%
pima_diabetes	768	9	2	268	34.90%	500	65.10%
labor	57	17	2	20	35.09%	37	64.91%
ionosphere	351	35	2	126	35.90%	225	64.10%
hungarian-14-heart-diseas	294	14	2	106	36.05%	188	63.95%
horse-colic	368	23	2	136	36.96%	232	63.04%
vote	435	17	2	168	38.62%	267	61.38%
horse-colic.ORIG	368	28	2	152	41.30%	214	58.15%
heart-statlog	270	14	2	120	44.44%	150	55.56%
credit-rating	690	16	2	307	44.49%	383	55.51%
cleveland-14-heart-diseas	303	14	2	138	45.54%	165	54.46%
sonar	208	61	2	97	46.63%	111	53.37%
kr-vs-kp	3196	37	2	1527	47.78%	1669	52.22%
mushroom	8124	23	2	3916	48.20%	4208	51.80%
Average	**1720.53**	**24.64**	**5.47**	**330.22**	**22.79%**	**662.64**	**50.90%**
Median	**530.00**	**19.50**	**2.50**	**113.00**	**26.88%**	**221.50**	**54.59%**
Min	**57**	**5**	**2**	**1**	**0.05%**	**37**	**4.07%**
Max	**20000**	**70**	**26**	**3916**	**48.20%**	**4208**	**93.88%**

Table 1: Characteristics of the datasets

Dataset	MinCover ByClass	SizeOfClasses	Size	%Size
Hypothyroid (*)	**76**	**76**	304	8.06%
primary-tumor (*)	**7**	**7**	147	43.36%
audiology (*)	**5**	**5**	120	53.10%
anneal (*)	**18**	**18**	90	10.02%
anneal.ORIG (*)	**18**	**18**	90	10.02%
soybean (*)	**14**	**14**	266	38.95%
lymphography (*)	**3**	**3**	12	8.11%
autos (*)	**5**	**5**	30	14.63%
letter	400	734	19084	95.42%
zoo	3	4	28	27.72%
Glass	5	9	54	25.23%
sick	76	231	462	12.25%
balance-scale	13	49	147	23.52%
vowel	20	67.5	742.5	75.00%
segment	47	247.5	1732.5	75.00%
hepatitis	4	32	64	41.29%
vehicle	17	199	796	94.09%
splice	64	767	2301	72.13%
breast-cancer	6	85	170	59.44%
german_credit	20	300	600	60.00%
waveform	100	1653	4959	99.18%
iris	3	37.5	112.5	75.00%
wisconsin-breast-cancer	14	241	482	68.96%
pima_diabetes	16	268	536	69.79%
labor	2	20	40	70.18%
ionosphere	8	126	252	71.79%
hungarian-14-heart-diseas	6	106	212	72.11%
horse-colic	8	136	272	73.91%
vote	9	168	336	77.24%
horse-colic.ORIG	8	152	304	82.61%
heart-statlog	6	120	240	88.89%
credit-rating	14	307	614	88.99%
cleveland-14-heart-diseas	7	138	276	91.09%
sonar	5	97	194	93.27%
kr-vs-kp	64	1527	3054	95.56%
mushroom	163	3916	7832	96.41%
Average	**34.83**	**330.10**	**1304.32**	**60.06%**
Median	**11.00**	**113.00**	**269.00**	**70.99%**

Table 2: Inferred characteristics of the datasets to determine the size of the subsamples to be generated using balancing and maxSize

Based on the values of Table 2, we know the amount of examples that the subsamples will have into each class, so, we can determine which class is the least favored class and, thus, we can calculate the ratio r_{lfc}, and, finally, the expression (4), the number of samples required according to a specific coverage value. Table 3 shows, in the first column, the obtained r_{lfc} ratio, and then, the required number of samples in order to achieve the indicated coverage value related to each column.

The number of samples in the table is rounded up to the next integer and it has been set to 3 when

the obtained value was lower than this because this is the minimum number of samples required to build a consolidated tree.

		Coverage									
Dataset	r_{lfc}	**10%**	**20%**	**30%**	**40%**	**50%**	**75%**	**90%**	**95%**	**99%**	**99.9%**
hypothyroid	2.18%	5	11	17	24	32	63	105	136	209	313
primary-tumor	8.33%	3	3	5	6	8	16	27	35	53	80
audiology	8.77%	3	3	4	6	8	16	26	33	51	76
anneal	2.63%	4	9	14	20	26	52	87	113	173	260
anneal.ORIG	2.63%	4	9	14	20	26	52	87	113	173	260
soybean	15.22%	3	3	3	4	5	9	14	19	28	42
lymphography	3.70%	3	6	10	14	19	37	62	80	123	184
autos	7.46%	3	3	5	7	9	18	30	39	60	90
letter	90.28%	3	3	3	3	3	3	3	3	3	3
zoo	9.76%	3	3	4	5	7	14	23	30	45	68
Glass	11.84%	3	3	3	5	6	11	19	24	37	55
sick	6.52%	3	4	6	8	11	21	35	45	69	103
balance-scale	17.01%	3	3	3	3	4	8	13	17	25	38
vowel	75.00%	3	3	3	3	3	3	3	3	4	5
segment	75.00%	3	3	3	3	3	3	3	3	4	5
hepatitis	26.02%	3	3	3	3	3	5	8	10	16	23
vehicle	91.28%	3	3	3	3	3	3	3	3	3	3
splice	46.34%	3	3	3	3	3	3	4	5	8	12
breast-cancer	42.29%	3	3	3	3	3	3	5	6	9	13
german_credit	42.86%	3	3	3	3	3	3	5	6	9	13
waveform	97.70%	3	3	3	3	3	3	3	3	3	3
iris	75.00%	3	3	3	3	3	3	3	3	4	5
wisconsin-breast-cancer	52.62%	3	3	3	3	3	3	4	5	7	10
pima_diabetes	53.60%	3	3	3	3	3	3	3	4	6	9
labor	54.05%	3	3	3	3	3	3	3	4	6	9
ionosphere	56.00%	3	3	3	3	3	3	3	4	6	9
hungarian-14-heart-diseas	56.38%	3	3	3	3	3	3	3	4	6	9
horse-colic	58.62%	3	3	3	3	3	3	3	4	6	8
vote	62.92%	3	3	3	3	3	3	3	4	5	7
horse-colic.ORIG	71.03%	3	3	3	3	3	3	3	3	4	6
heart-statlog	80.00%	3	3	3	3	3	3	3	3	3	5
credit-rating	80.16%	3	3	3	3	3	3	3	3	3	5
cleveland-14-heart-diseas	83.64%	3	3	3	3	3	3	3	3	3	4
sonar	87.39%	3	3	3	3	3	3	3	3	3	4
kr-vs-kp	91.49%	3	3	3	3	3	3	3	3	3	3
mushroom	93.06%	3	3	3	3	3	3	3	3	3	3
Average	**48.30%**	**3.11**	**3.67**	**4.44**	**5.39**	**6.47**	**10.86**	**16.97**	**21.64**	**32.58**	**48.47**
Median	**53.83%**	**3.00**	**3.00**	**3.00**	**3.00**	**3.00**	**3.00**	**3.00**	**4.00**	**6.00**	**9.00**

Table 3: Number of samples to be generated over a spectrum of different coverage values for balanced subsamples with maximum size

As it can be observed in Table 3, the lower the ratio of the *least favored class* is, the higher is the required number of samples to achieve a specific coverage value. In the case of the dataset which has the smallest ratio, the number of samples required to obtain a 99.9% of coverage is 313. At the

bottom of the table, the ratios are close to 90% and, thus, 3 samples will be enough to achieve this coverage (likely even more).

This implementation has another preconfigured option to generate balanced samples but, in this case, the size of the samples will be the same that minority class' size has in the original sample (*sizeOfMinClass* value for `bagSizePercent` parameter). This kind of samples were used by Weiss and Provost in [11] in order to guarantee that the proportion of the minority class may vary from 0% to 100% without using replacement for two-class problems. Regarding the CTC algorithm, this kind of subsampling was used in [12] and [13] in order to analyze the effect of changes in class distribution over this algorithm.

For this update, we have also adapted this kind of subsampling in order to be able to use it with multi-class datasets and, also, taking into account the minimum number of examples which determines classes size into the subsamples. Regarding the latter issue, nothing new in this case, that is, the same criterion will be used that we used for *maxSize* samples. On the other hand, for the multi-class datasets, we decided not to strictly use the size of the minority class into the training sample to generate the subsamples with this size (such as we do with two-class datasets). This would make impossible to build a subsample when the number of examples of the minority class in the training sample were lower than the number of classes (*hypothyroid*, for example, with 2 instances in its minority class and 4 classes or *audiology* with only one instance and 24 classes, in the most extreme case). Besides, in other cases, although it were possible, the representation of each class into the subsamples could be ridiculous. Because of this, we decided determine the size of all classes into the subsamples as half the minority class' size in the training sample. This makes that the size for the subsample in two-class datasets being the minority class' size in the sample, and getting a reasonable representation for multi-class datasets, in normal conditions.

Table 4 shows the values inferred for *sizeMax* samples in Table 2 but for *sizeOfMinClass* samples. As it can be seen, the same first 8 datasets continue to be where the minority class' size does not reach the required minimum, however, 3 more datasets do not satisfy this condition.

On the other hand, in this case, the average subsample size is around 34%, when it was around 60% for *maxSize* samples.

At last, Table 5 shows the number of samples to be generated for balanced and *sizeOfMinClass* sized samples using different values of coverage. Like in Table 3, the first column refers the ratio of the *least favored class* in this case. This values decrements, in average, to 26.14% (in contrast with 48.30% in the case of *maxSize* samples) and, so, the obtained values for number of samples are bigger than above. For example, for the most balanced datasets of the list more than 3 samples are required to achieve values of coverage equal or greater than 90%.

Regarding the other types of used resampling techniques (bootstrap and stratified samples), it is worth noting that, for bootstrap samples, the number of samples required for a specific coverage value does not depend on any dataset's characteristic and, for stratified samples, only the ratio between both samples (the original one's and the generated one's) will be taken into account. As a consequence, Table 6 summarizes the number of samples to be generated for both types of samples. In the case of stratified samples, four different percentages have been used indicating the size of the subsamples, S%, respect to the original sample's size: 90%, 75%, 50% and 25%. Besides, in order to notice the amount of coverage obtained with these kind of samples, we also have added in the second column the coverage value that would be obtained whether we use only three samples, the minimum number of samples necessary to build a consolidated tree.

Finally, we want to note that an extensive analysis of the results obtained by CTC algorithm with these resampling strategies has been reported in [14].

Dataset	MinCover ByClass	SizeOfClasses	Size	%Size
hypothyroid (*)	**76**	**76**	304	8.06%
primary-tumor (*)	**7**	**7**	147	43.36%
audiology (*)	**5**	**5**	120	53.10%
anneal (*)	**18**	**18**	90	10.02%
anneal.ORIG (*)	**18**	**18**	90	10.02%
soybean (*)	**14**	**14**	266	38.95%
lymphography (*)	**3**	**3**	12	8.11%
autos (*)	**5**	**5**	30	14.63%
letter (*)	**400**	**400**	10400	52.00%
zoo (*)	**3**	**3**	21	20.79%
Glass (*)	**5**	**5**	30	14.02%
sick	76	116	232	6.15%
balance-scale	13	25	75	12.00%
vowel	20	45	495	50.00%
segment	47	165	1155	50.00%
hepatitis	4	16	32	20.65%
vehicle	17	100	400	47.28%
splice	64	384	1152	36.11%
breast-cancer	6	43	86	30.07%
german_credit	20	150	300	30.00%
waveform	100	827	2481	49.62%
iris	3	25	75	50.00%
wisconsin-breast-cancer	14	121	242	34.62%
pima_diabetes	16	134	268	34.90%
labor	2	10	20	35.09%
ionosphere	8	63	126	35.90%
hungarian-14-heart-diseas	6	53	106	36.05%
horse-colic	8	68	136	36.96%
vote	9	84	168	38.62%
horse-colic.ORIG	8	76	152	41.30%
heart-statlog	6	60	120	44.44%
credit-rating	14	154	308	44.64%
cleveland-14-heart-diseas	7	69	138	45.54%
sonar	5	49	98	47.12%
kr-vs-kp	64	764	1528	47.81%
mushroom	163	1958	3916	48.20%
Average	**34.83**	**169.81**	**703.31**	**34.06%**
Median	**11.00**	**61.50**	**142.50**	**36.53%**

Table 4: Inferred characteristics of the datasets to determine the size of the subsamples to be generated using balancing and sizeOfMinClass

		Coverage									
Dataset	r_{lfc}	**10%**	**20%**	**30%**	**40%**	**50%**	**75%**	**90%**	**95%**	**99%**	**99.9%**
hypothyroid	2.18%	5	11	17	24	32	63	105	136	209	313
primary-tumor	8.33%	3	3	5	6	8	16	27	35	53	80
audiology	8.77%	3	3	4	6	8	16	26	33	51	76
anneal	2.63%	4	9	14	20	26	52	87	113	173	260
anneal.ORIG	2.63%	4	9	14	20	26	52	87	113	173	260
soybean	15.22%	3	3	3	4	5	9	14	19	28	42
lymphography	3.70%	3	6	10	14	19	37	62	80	123	184
autos	7.46%	3	3	5	7	9	18	30	39	60	90
letter	49.20%	3	3	3	3	3	3	4	5	7	11
zoo	7.32%	3	3	5	7	10	19	31	40	61	91
Glass	6.58%	3	4	6	8	11	21	34	45	68	102
sick	3.28%	4	7	11	16	21	42	70	90	139	208
balance-scale	8.68%	3	3	4	6	8	16	26	33	51	77
vowel	50.00%	3	3	3	3	3	3	4	5	7	10
segment	50.00%	3	3	3	3	3	3	4	5	7	10
hepatitis	13.01%	3	3	3	4	5	10	17	22	34	50
vehicle	45.87%	3	3	3	3	3	3	4	5	8	12
splice	23.20%	3	3	3	3	3	6	9	12	18	27
breast-cancer	21.39%	3	3	3	3	3	6	10	13	20	29
german_credit	21.43%	3	3	3	3	3	6	10	13	20	29
waveform	48.88%	3	3	3	3	3	3	4	5	7	11
iris	50.00%	3	3	3	3	3	3	4	5	7	10
wisconsin-breast-cancer	26.42%	3	3	3	3	3	5	8	10	16	23
pima_diabetes	26.80%	3	3	3	3	3	5	8	10	15	23
labor	27.03%	3	3	3	3	3	5	8	10	15	22
ionosphere	28.00%	3	3	3	3	3	5	8	10	15	22
hungarian-14-heart-diseas	28.19%	3	3	3	3	3	5	7	10	14	21
horse-colic	29.31%	3	3	3	3	3	4	7	9	14	20
vote	31.46%	3	3	3	3	3	4	7	8	13	19
horse-colic.ORIG	35.51%	3	3	3	3	3	4	6	7	11	16
heart-statlog	40.00%	3	3	3	3	3	3	5	6	10	14
credit-rating	40.21%	3	3	3	3	3	3	5	6	9	14
cleveland-14-heart-diseas	41.82%	3	3	3	3	3	3	5	6	9	13
sonar	44.14%	3	3	3	3	3	3	4	6	8	12
kr-vs-kp	45.78%	3	3	3	3	3	3	4	5	8	12
mushroom	46.53%	3	3	3	3	3	3	4	5	8	12
Average	**26.14%**	**3.14**	**3.78**	**4.72**	**5.86**	**7.14**	**12.83**	**20.97**	**27.06**	**41.36**	**61.81**
Median	**26.91%**	**3**	**3.00**	**3.00**	**3.00**	**3.00**	**5.00**	**8.00**	**10.00**	**15.00**	**22.50**

Table 5: Number of samples to be generated over a spectrum of different coverage values for balanced subsamples with minority class' size

	N_S	**Coverage**									
	3	**10%**	**20%**	**30%**	**40%**	**50%**	**75%**	**90%**	**95%**	**99%**	**99.9%**
Bootstrap	0.9502	3	3	3	3	3	3	3	3	5	7
Stratified S%=90	0.9990	3	3	3	3	3	3	3	3	3	4
Stratified S%=75	0.9844	3	3	3	3	3	3	3	3	4	5
Stratified S%=50	0.8750	3	3	3	3	3	3	4	5	7	10
Stratified S%=25	0.5781	3	3	3	3	3	5	9	11	17	25

Table 6: Coverage value for N_S=3 and Number of samples to be generated over a spectrum of different coverage values for bootstrap and stratified samples (with different sizes)

Using J48consolidated

The major changes done in this update of the J48consolidated class has been already mentioned in the above sections. However, this has also implied some changes in the user interfaces of this class. The main change is related to the way about how to determine the number of samples to be generated. We have added a new option, *RMnumberSamplesHowToSet* (`m_RMnumberSamplesHowToSet` member), which can be set to two values: "using a fixed value" (*`NumberSamples_FixedValue`* tag) or "based on a coverage value (%)" (*`NumberSamples_BasedOnCoverage`* tag).

The "using a fixed value" tag maintains the same semantic of the *RMnumberSamples* parameter that it had above, that is, the user can directly specify a particular value of number of samples to be generated for the consolidated tree's construction. On the other hand, if the user chooses the "based on a coverage value (%)" tag for the *RMnumberSamplesHowToSet* parameter, the value of the *RMnumberSamples* parameter must be the coverage value that the user wants to achieve with the generated set of samples (as a percentage; a value between 0 and 100). By default, the option based on a coverage value is selected and the default value for the coverage is 99%.

Graphical interface

The figure below shows the new graphical interface for J48Consolidated. As mentioned above, the *RMnumberSamplesHowToSet* option has been added. Only the options started by the “RM” prefix (Resampling Method) are specifically related to the CTC algorithm (J48Consolidated class); the rest of options (partially shown in the figure) are derived from the implementation of the C4.5 algorithm (J48 class in Weka).

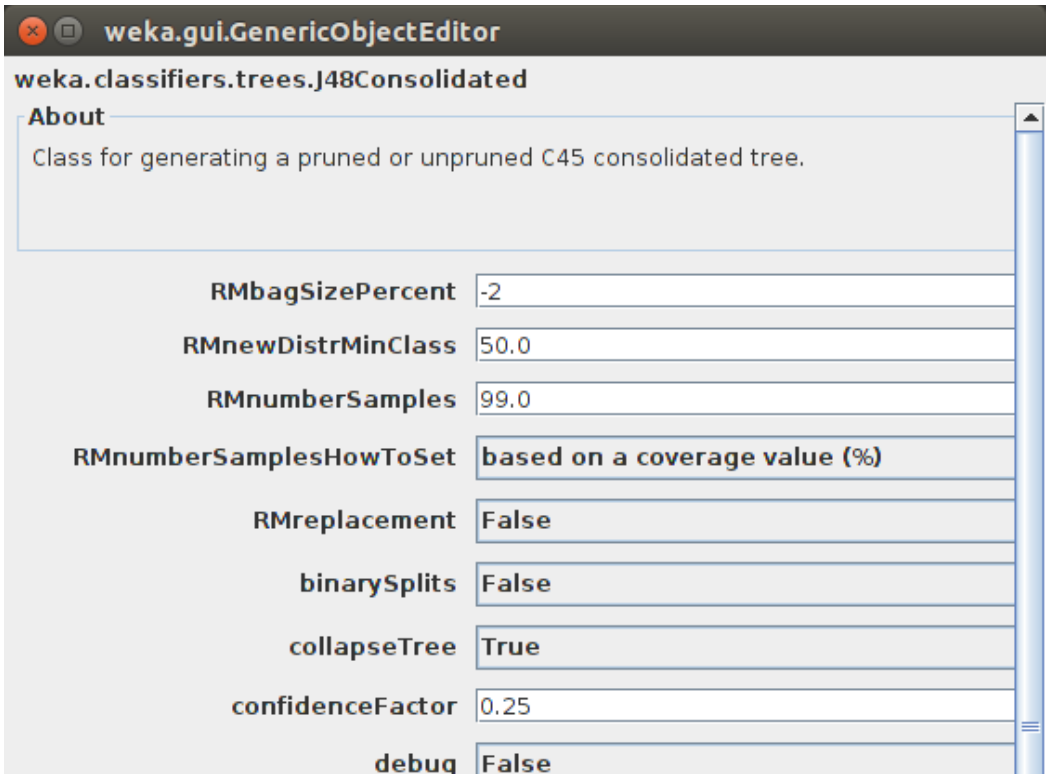

Figure 1: New graphical interface of the Weka Explorer for J48Consolidated classifier.

Command-line

Regarding the use of the J48Consolidated class from the command-line, a new option has been added in order to determine how to set the number of samples to be used. Actually, if the user wants

to set a fixed value of number of samples, nothing changes; only -RM-N option will have to be set followed by the specific value. However, if we want to determine the number of samples based on a coverage value, the new -RM-C option will have to be set followed by the -RM-N option and the desired coverage value. The figure below shows the help information related to both options.

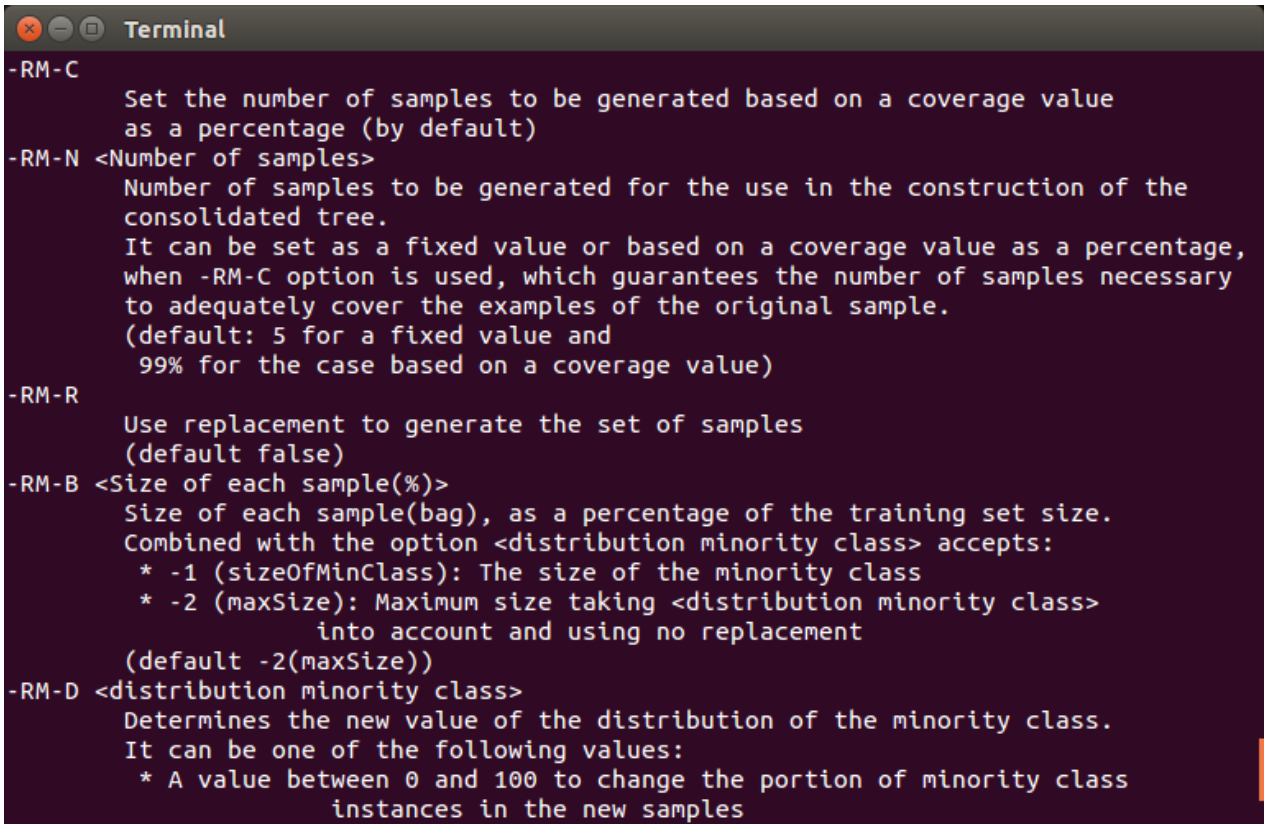

Figure 2: Changes in J48Consolidated options through the command-line

New measures shown

As it has been mentioned, two new measures has been implemented on the J48Consolidated class (see "Using the notion of coverage" section), both related to the coverage concept: `measureNumberSamplesByCoverage` and `measureTrueCoverage`. The values related to these measures will be shown with the built tree, as well as a brief summary of the used resampling method.

Figure 3 shows an example of the output achieved by J48Consolidated class with the default parameters for the *german_credit* dataset ("credit-g.arff"). As it can be seen, the used resampling method ([RM]) is to change the class distribution (-RM-D 50.0), in particular, balancing the classes, and using the maximum size for the subsamples (-RM-B -2) taking into account replacement is not selected (not -RM-R used). The number of samples is determined based on a coverage value (-RM-C appears) and the coverage value is 99.0% (-RM-N 99.0).

Regarding the new measures, the necessary number of samples to obtain a 99% of coverage is 9 (N_S=f(99% of coverage)=9) and the actual coverage achieved based on the dataset's characteristics, taking into account the mentioned resampling method, is around 99.5453%.

If an exceptional situation occurred, like those mentioned in the "Taking some exceptional characteristics of some datasets into account" section (for example, if the original dataset is already balanced), a brief message would also be added here in order to notice this situation.

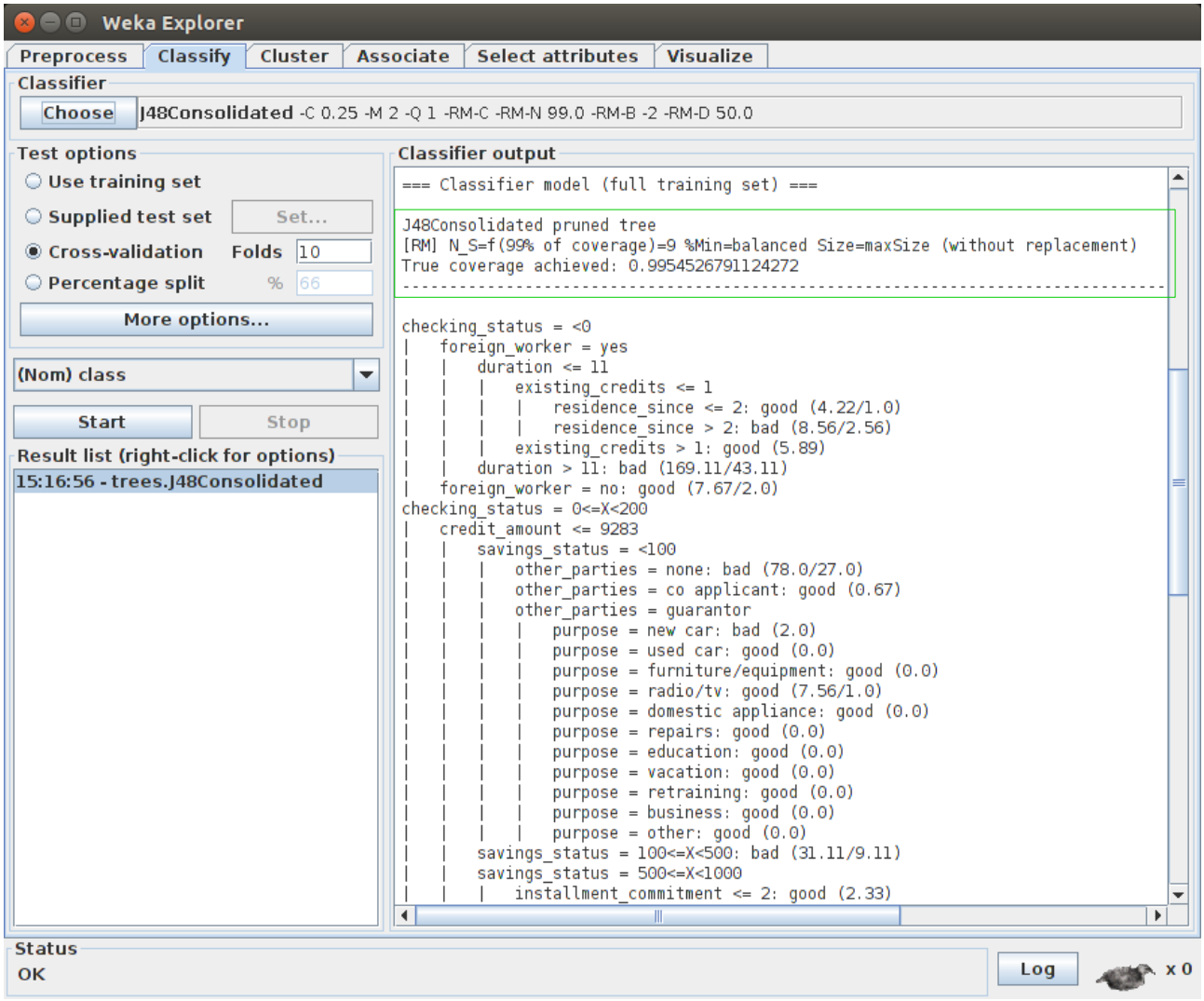

Figure 3: Output of J48Consolidated for the german_credit *dataset with the default parameters*

Source code

The implementation of the CTC algorithm for Weka consists on the J48Consolidated class and the j48Consolidated package which includes other 5 classes, all of them derived from its corresponding j48's original class (for more details see [6]). All changes done in the source code to achieve the functionality mentioned in this report were done, mainly, in the J48Consolidated class. Only two functions were added to the InstancesConsolidated class, basically for debugging purposes. The complete source code of both classes, "J48Consolidated.java" and "InstancesConsolidated.java" files, has been included in "Appendix 1: Source code" (page 21).

Downloading

A website exists containing the complete material related to the J48Consolidated classifier:

http://www.sc.ehu.es/aldapa/weka-ctc

Regarding this update, the material below can be found:

- *Documentation*:

  This technical report:

  Weka-CTC-v2.pdf

- *Downloading J48Consolidated*:

  Weka maintains two primary versions: the *stable version* corresponding to the latest edition of the data mining book and the *development version*, which exhibits a package management system that makes it easy for the Weka community to add new functionality to Weka (See Downloading and installing Weka).

  Thus, we also offers both versions for J48Consolidated package:

  Stable version

  The source code of the six classes that implement the J48Consolidated classifier can be found in (tested for weka-3-6-11):

  J48Consolidated-v2-stb.java.tar.gz

  In order to complete the whole source code of the implementation, download Weka source code from http://www.cs.waikato.ac.nz/ml/weka/downloading.html.

  And, finally, the whole executable file of Weka for stable-3-6-11 including J48Consolidated package can be found in:

  Weka-CTC-v2-stb.jar

Figure 4: Main window of Weka stable 3.6.11 version

Development version

The Weka package containing the J48Consolidated classifier (including compiled code, source code, javadocs and package description files) can be found in (tested for weka-3-7-11):

J48Consolidated-v2.zip

Also, this package has recently become an "official" Weka package and is accessible in the central package repository of Weka, as it can be seen in Figure 5.

In order to better understand how packages are structured for the package management system see here.

This package can be installed through the *package management system* of the development 3.7.11 version of Weka. However, since J48Consolidated class extends J48 class, some J48's members and functions have to be 'protected' (instead 'private') in order to be able to use them. Because of this and based on a question done in *Wekalist*, the Weka mailing list, Eibe Frank, one of the authors and developers of Weka platform, changed the access modifiers for (almost all) private member variables to protected in J48 and all classes in the j48 package (See "[Wekalist] How to build a package with a new classifier if I changed an original class of weka?").

These changes were done on Tue Apr 1 2014, therefore to be able to install the J48Consolidated package a version of dev-3-7-11 generated later than that date is required.

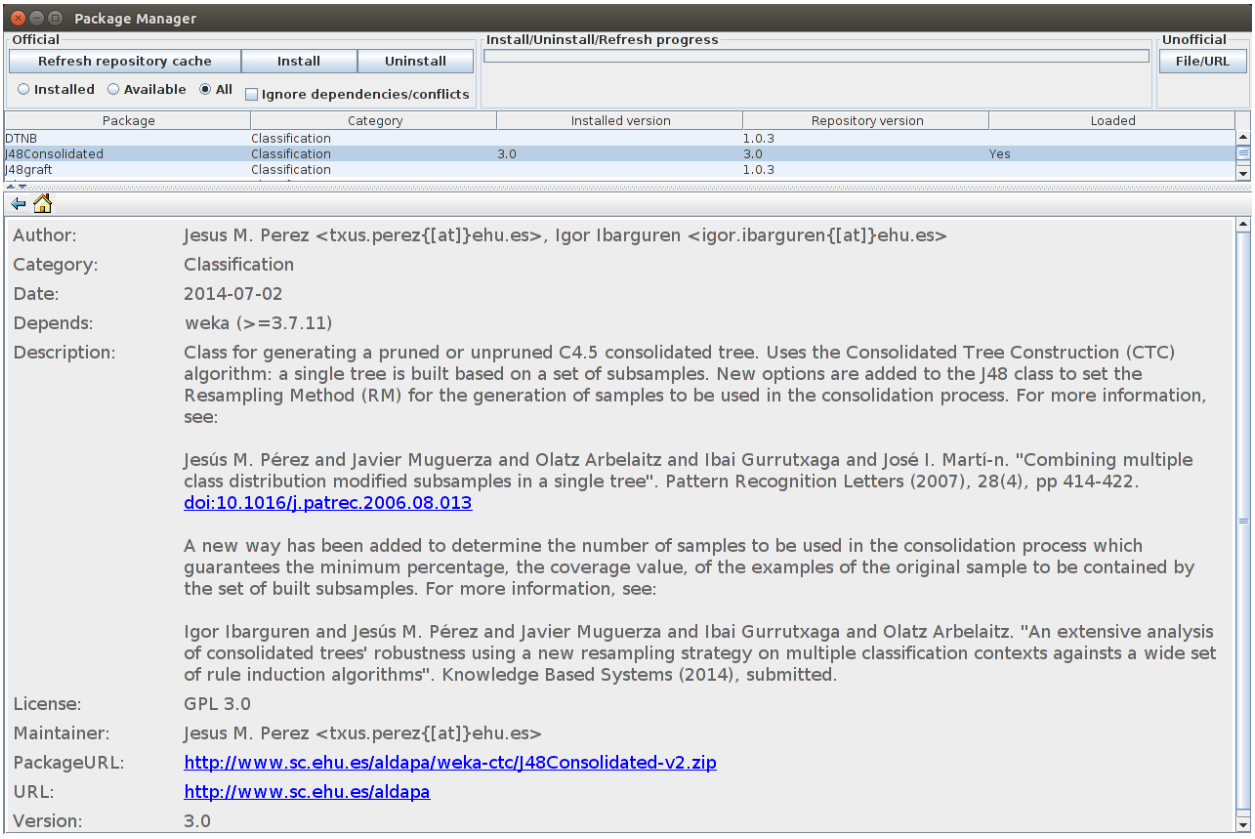

Figure 5: J48Consolidated Repository version package successfully installed in Weka dev-3-7-11

And, finally, the whole executable file of Weka for dev-3-7-11 including the J48Consolidated class (not the package) can be found in:

Weka-CTC-v2-dev.jar

Figure 6: Main window of Weka development 3.7.11 version

Citation policy

If you want to refer to CTC in a publication, please cite one of these papers:

> J.M. Pérez, J. Muguerza, O. Arbelaitz, I. Gurrutxaga, J.I. Martín. *"Combining multiple class distribution modified subsamples in a single tree"*. Pattern Recognition Letters. Vol. 28, Issue 4, 414-422 (2007). doi:10.1016/j.patrec.2006.08.013

We have also recently written this paper which explores the results of consolidated trees using the notion of coverage introduced in this technical report but it is pending for publication:

> I. Ibarguren, J.M. Pérez, J. Muguerza, I. Gurrutxaga, O. Arbelaitz. *"An extensive analysis of consolidated trees' robustness using a new resampling strategy on multiple classification contexts against a wide set of rule induction algorithms"*. Knowledge Based Systems, *submitted* (2014).

Further work

There are some things that can be done related to this work. All of then have been marked as `//TODO` in the source code:

- Implement the options `binarySplits` and `reducedErrorPruning` of the J48 implementation.
- Set the different options to generate the samples as a filter of Weka. In this way other strategies, already implemented in Weka, could be used for the generation of samples, as for example SMOTE method.
- Generalize the process of changing the class distribution to multi-class datasets. In this update it was enabled the possibility of balancing the classes of the subsamples to be generated, but not in a more general way.
- Make possible the use of replacement when changing the class distribution, even though we have not used this option in our previous works.

Bibliography

[1] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten. *"The WEKA Data Mining Software: An Update"*. SIGKDD Explorations, Vol. 11, Issue 1 (2009).

[2] J.M. Pérez, J. Muguerza, O. Arbelaitz, I. Gurrutxaga, J.I. Martín. *"Consolidated Tree Classifier in a Car Insurance Fraud Detection Domain with Class Imbalance"*. Lecture Notes in Computer Science 3686, Pattern Recognition and Data Mining. Springer-Verlag. S. Singh et al. (Eds.), 381-389 (2005). doi: 10.1007/11551188_41

[3] J.M. Pérez, J. Muguerza, O. Arbelaitz, I. Gurrutxaga, J.I. Martín. *"Combining multiple class distribution modified subsamples in a single tree"*. Pattern Recognition Letters. Vol. 28, Issue 4, 414-422 (2007). doi:10.1016/j.patrec.2006.08.013

[4] H. Abbasian, C. Drummond, N. Japkowicz, S. Matwin. *"Inner ensembles: Using ensemble methods inside the learning algorithm"*. Lecture Notes in Computer Science 8190, Machine Learning and Knowledge Discovery in Databases, H. Blockeel, K. Kersting, S. Nijssen, F. Zelezny (Eds.), European Conference, ECML/PKDD 2013, 33-48 (2013).

[5] J.R. Quinlan: *"C4.5: Programs for Machine Learning"*, Morgan Kaufmann Publishers Inc. (eds), San Mateo, California, (1993).

[6] O. Arbelaitz, I. Gurrutxaga, F. Lozano, J. Muguerza, J.M. Pérez. *"J48Consolidated: An implementation of CTC algorithm for WEKA"*. Technical Report EHU-KAT-IK-05-13, University of the Basque Country (UPV/EHU), 1-34 (2013). [http://www.sc.ehu.es/aldapa/weka-ctc/Weka-CTC.pdf]

[7] G.E.A.P.A. Batista, R.C. Prati, M.C. Monard. *"A study of the behavior of several methods for balancing machine learning training data"*. SIGKDD Explorations 6, 20–29 (2004) .

[8] C.X. Ling, J. Huang, H. Zhang. *"AUC: A Better Measure than Accuracy in Comparing Learning Algorithms"*. Canadian Conference on AI, 2671, 329-341 (2003).

[9] K. Bache, M. Lichman. *"UCI Machine Learning Repository"* [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science, (2013).

[10] E. Bauer, R. Kohavi. *"An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants"*. Machine Learning, Vol. 36, 105-139 (1999).

[11] G.M. Weiss, F. Provost *"Learning when training data are costly: The effect of class distribution on tree induction"*. Journal of Artificial Intelligence Research, Vol. 19 , 315–354 (2003).

[12] I. Albisua, O. Arbelaitz, I. Gurrutxaga, J.I. Martín, J. Muguerza, J.M. Pérez, I. Perona. *"Obtaining optimal class distribution for decision trees: comparative analysis of CTC and C4.5"*. Lecture Notes in Computer Science 5988, Current Topics in Artificial Intelligence, CAEPIA 2009 Selected Papers. Springer-Verlag. P. Meseguer, L. Mandow and R. M. Gasca (Eds.), 101-110 (2010). doi:10.1007/978-3-642-14264-2_11

[13] I. Albisua, O. Arbelaitz, I. Gurrutxaga, A. Lasarguren, J. Muguerza, J.M. Pérez. *"Analysis of the effect of changes in class distribution in C4.5 and consolidated C4.5 tree learners"*. Technical Report EHU-KAT-IK-01-12, University of the Basque Country (UPV/EHU), 1-80 (2012). [http://www.sc.ehu.es/aldapa/Argitalpenak/12.EHU-KAT-IK-01-12.pdf]

[14] I. Ibarguren, J.M. Pérez, J. Muguerza, I. Gurrutxaga, O. Arbelaitz. *"An extensive analysis of consolidated trees' robustness using a new resampling strategy on multiple classification contexts against a wide set of rule induction algorithms"*. Knowledge Based Systems, *submitted* (2014).

Appendix 1: Source code

This appendix contains the source code of the two only java classes that were changed in this update: "J48Consolidated.java" and "InstancesConsolidated.java". The rest of classes that compose the implementation of the J48Consolidated classifier (C45ConsolidatedPruneableClassifierTree.java, C45ConsolidatedModelSelection.java, C45ConsolidatedSplit.java and DistributionConsolidated.java) can be found in the previous technical report [6].

J48Consolidated.java

```
/*
 *    This program is free software; you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation; either version 2 of the License, or
 *    (at your option) any later version.
 *
 *    This program is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *
 *    You should have received a copy of the GNU General Public License
 *    along with this program; if not, write to the Free Software
 *    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/*
 *    J48Consolidated.java
 *    Copyright (C) 2014 ALDAPA Team (http://www.sc.ehu.es/aldapa)
 *    Computing Engineering Faculty, Donostia, 20018
 *    University of the Basque Country (UPV/EHU), Basque Country
 *
 */

package weka.classifiers.trees;

import java.util.Enumeration;
import java.util.Random;
import java.util.Vector;

import weka.classifiers.Sourcable;
import weka.classifiers.trees.j48.C45ModelSelection;
import weka.classifiers.trees.j48.C45PruneableClassifierTree;
import weka.classifiers.trees.j48.ModelSelection;
import weka.classifiers.trees.j48Consolidated.C45ConsolidatedModelSelection;
import weka.classifiers.trees.j48Consolidated.C45ConsolidatedPruneableClassifierTree;
import weka.classifiers.trees.j48Consolidated.InstancesConsolidated;
import weka.core.AdditionalMeasureProducer;
import weka.core.Capabilities;
import weka.core.Drawable;
import weka.core.Instances;
import weka.core.Matchable;
import weka.core.Option;
import weka.core.OptionHandler;
import weka.core.SelectedTag;
import weka.core.Summarizable;
import weka.core.Tag;
import weka.core.TechnicalInformation;
import weka.core.TechnicalInformationHandler;
import weka.core.Utils;
import weka.core.WeightedInstancesHandler;
import weka.core.TechnicalInformation.Field;
import weka.core.TechnicalInformation.Type;

/**
<!-- globalinfo-start -->
 * Class for generating a pruned or unpruned C4.5 consolidated tree. Uses the Consolidated Tree Construction
(CTC) algorithm: a single tree is built based on a set of subsamples. New options are added to the J48 class to
set the Resampling Method (RM) for the generation of samples to be used in the consolidation process. For more
```

```
information, see:<br/>
 * <br/>
 * Jes&uacute;s M. P&eacute;rez and Javier Muguerza and Olatz Arbelaitz and Ibai Gurrutxaga and Jos&eacute; I.
Mart&iacute;n.
 * "Combining multiple class distribution modified subsamples in a single tree". Pattern Recognition Letters
(2007), 28(4), pp 414-422.
 * <a href="http://dx.doi.org/10.1016/j.patrec.2006.08.013" target="_blank">doi:10.1016/j.patrec.2006.08.013</a>
 * <p/>
 * A new way has been added to determine the number of samples to be used in the consolidation process which
guarantees the minimum percentage, the coverage value, of the examples of the original sample to be contained by
the set of built subsamples. For more information, see:<br/>
 * <br/>
 * Igor Ibarguren and Jes&uacute;s M. P&eacute;rez and Javier Muguerza and Ibai Gurrutxaga and Olatz Arbelaitz.
 * "An extensive analysis of consolidated trees with a new resampling strategy to establish their robustness
across multiple classification contexts againsts a wide set of rule induction algorithms". Knowledge Based
Systems (2014), submitted.
 * <p/>
<!-- globalinfo-end -->
 *
<!-- technical-bibtex-start -->
 * BibTeX:
 * <pre>
 * &#64;article{Perez2007,
 *    title = "Combining multiple class distribution modified subsamples in a single tree",
 *    journal = "Pattern Recognition Letters",
 *    volume = "28",
 *    number = "4",
 *    pages = "414 - 422",
 *    year = "2007",
 *    doi = "10.1016/j.patrec.2006.08.013",
 *    author = "Jes\'us M. P\'erez and Javier Muguerza and Olatz Arbelaitz and Ibai Gurrutxaga and Jos\'e I.
Mart\'in"
 * }
 * </pre>
 * <p/>
 * <pre>
 * &#64;article{Ibarguren2014,
 *    title = "An extensive analysis of consolidated trees with a new resampling strategy to establish their
robustness across multiple classification contexts againsts a wide set of rule induction algorithms",
 *    journal = "Knowledge Based Systems (submitted)",
 *    year = "2014",
 *    author = "Igor Ibarguren and Jes\'us M. P\'erez and Javier Muguerza and Ibai Gurrutxaga and Olatz
Arbelaitz"
 * }
 * </pre>
 * <p/>
<!-- technical-bibtex-end -->
<!-- options-start -->
 * Valid options are: <p/>
 *
 * J48 options <br/>
 * ==========
 *
 * <pre> -U
 *  Use unpruned tree.</pre>
 *
 * <pre> -C &lt;pruning confidence&gt;
 *  Set confidence threshold for pruning.
 *  (default 0.25)</pre>
 *
 * <pre> -M &lt;minimum number of instances&gt;
 *  Set minimum number of instances per leaf.
 *  (default 2)</pre>
 *
 * <pre> -S
 *  Don't perform subtree raising.</pre>
 *
 * <pre> -L
 *  Do not clean up after the tree has been built.</pre>
 *
 * <pre> -A
 *  Laplace smoothing for predicted probabilities.</pre>
 *
 * <pre> -Q &lt;seed&gt;
 *  Seed for random data shuffling (default 1).</pre>
```

```
 *
 * Options to set the Resampling Method (RM) for the generation of samples
 *  to use in the consolidation process <br/>
 * ============================================================================================
 * <pre> -RM-C
 *  Determines the way to set the number of samples to be generated will be based on
 *  a coverage value as a percentage. In the case this option is not set, the number of samples
 *  will be determined using a fixed value.
 *  (set by default)</pre>
 *
 * <pre> -RM-N &lt;number of samples&gt;
 *  Number of samples to be generated for the use in the construction of the consolidated tree.
 *  It can be set as a fixed value or based on a coverage value as a percentage, when -RM-C option
 *  is used, which guarantees the number of samples necessary to adequately cover the examples
 *  of the original sample
 *  (default 5 for a fixed value or 99% for the case based on a coverage value)</pre>
 *
 * <pre> -RM-R
 *  Determines whether or not replacement is used when generating the samples.
 *  (default false)</pre>
 *
 * <pre> -RM-B &lt;Size of each sample(&#37;)&gt;
 *  Size of each sample(bag), as a percentage of the training set size.
 *  Combined with the option &lt;distribution minority class&gt; accepts:
 *  * -1 (sizeOfMinClass): The size of the minority class
 *  * -2 (Max): Maximum size taking &lt;distribution minority class&gt; into account
 *  *           and using no replacement
 *  (default -2)</pre>
 *
 * <pre> -RM-D &lt;distribution minority class&gt;
 *  Determines the new value of the distribution of the minority class, if we want to change it.
 *  It can be one of the following values:
 *  * A value between 0 and 100 to change the portion of minority class instances in the new samples
 *    (this option can only be used with binary problems (two-class datasets))
 *  * -1 (free): Works with the instances without taking their class into account
 *  * -2 (stratified): Maintains the original class distribution in the new samples
 *  (default 50.0)
 *
<!-- options-end -->
 *
 * @author Jes&uacute;s M. P&eacute;rez (txus.perez@ehu.es)
 * @author Igor Ibarguren (igor.ibarguren@ehu.es)
 *  (based on the previous version written in colaboration with Fernando Lozano)
 *  (based on J48.java written by Eibe Frank)
 * @version $Revision: 3.0 $
 */
public class J48Consolidated
   extends J48
   implements OptionHandler, Drawable, Matchable, Sourcable,
            WeightedInstancesHandler, Summarizable, AdditionalMeasureProducer,
            TechnicalInformationHandler {

   /** for serialization */
   private static final long serialVersionUID = -2647522302468491144L;

   /** The default value set for the percentage of coverage estimated necessary to adequately cover
    *  the examples of the original sample with the set of samples to be used in the consolidation process */
   private static float m_coveragePercent = (float)99;

   /** Number of samples necessary based on coverage (if this option is used) */
   int m_numberSamplesByCoverage = 0;

   /** The true value estimated for the coverage achieved with the set of samples generated
    *  for the construction of the consolidated tree */
   private double m_trueCoverage;

   /** Size of each sample(bag), as a percentage of the training set size, to be used in exceptional situations
    *  where the original class distribution and the distribution of the samples to be generated are the same
    *  and the size of samples has been set as the maximum possible (maxSize).
    *  In these cases, all generated samples would have the same examples that the original sample has.
    *  Due of this the size of samples will be reduced with this value. */
   private static int m_bagSizePercentToReduce = 75;

   /** Minimum percentage of cases required in each class for the samples to be generated when
    *  the distribution of the minority class is changed */
```

```
private static float m_minExamplesPerClassPercent = (float)2.0;

/** String containing a brief explanation of exceptional situations, if occur */
String m_stExceptionalSituationsMessage = "";

/** Ways to set the numberSamples option */
public static final int NumberSamples_FixedValue = 1;
public static final int NumberSamples_BasedOnCoverage = 2;

/** Strings related to the ways to set the numberSamples option */
public static final Tag[] TAGS_WAYS_TO_SET_NUMBER_SAMPLES = {
   new Tag(NumberSamples_FixedValue, "using a fixed value"),
   new Tag(NumberSamples_BasedOnCoverage, "based on a coverage value (%)"),
};

 /** Options to set the Resampling Method (RM) for the generation of samples
 *   to use in the consolidation process
 *   (Prefix RM added to the option names in order to appear together in the graphical interface)
 ********************************************************************************/
/** Selected way to set the number of samples to be generated; or using a fixed value;
 *   or based on a coverage value as a percentage (by default). */
private int m_RMnumberSamplesHowToSet = NumberSamples_BasedOnCoverage;

/** Number of samples to be generated for the use in the construction of the consolidated tree.
 * If m_RMnumberSamplesHowToSet = NumberSamples_BasedOnCoverage, the value of number of
 * samples to be used is calculated based on a coverage value in percentage (%), which guarantees
 * the number of samples necessary to adequately cover the examples of the original sample. */
private float m_RMnumberSamples = (float)m_coveragePercent; // default: f(99% of coverage)

/** Determines whether or not replacement is used when generating the samples.**/
private boolean m_RMreplacement = false;

/** Size of each sample(bag), as a percentage of the training set size.
 *  Combined with the option &lt;distribution minority class&gt; accepts:
 *  * -1 (sizeOfMinClass): The size of the minority class
 *  * -2 (maxSize): Maximum size taking &lt;distribution minority class&gt; into account
 *  *           and using no replacement */
private int m_RMbagSizePercent = -2; // default: maxSize

/** Value of the distribution of the minority class to be changed.
 * It can be one of the following values: <br>
 *  * A value between 0 and 100 to change the portion of minority class instances in the new samples
 *    (If the dataset is multi-class, only the special value 50.0 will be accepted to balance the classes)
 *  * -1 (free): Works with the instances without taking their class into account
 *  * -2 (stratified): Maintains the original class distribution in the new samples */
private float m_RMnewDistrMinClass = (float)50.0;

/**
 * Returns a string describing the classifier
 * @return a description suitable for
 * displaying in the explorer/experimenter gui
 */
public String globalInfo() {
   return "Class for generating a pruned or unpruned C45 consolidated tree. Uses the Consolidated "
      + "Tree Construction (CTC) algorithm: a single tree is built based on a set of subsamples. "
      + "New options are added to the J48 class to set the Resampling Method (RM) for "
      + "the generation of samples to be used in the consolidation process.\n"
      + "Recently, a new way has been added to determine the number of samples to be used "
      + "in the consolidation process which guarantees the minimum percentage, the coverage value, "
      + "of the examples of the original sample to be contained by the set of built subsamples. "
      + "For more information, see:\n\n"
      + getTechnicalInformation().toString();
}

/**
 * Returns an instance of a TechnicalInformation object, containing
 * detailed information about the technical background of this class,
 * e.g., paper reference or book this class is based on.
 *
 * @return the technical information about this class
 */
public TechnicalInformation getTechnicalInformation() {
   TechnicalInformation    result;

   result = new TechnicalInformation(Type.ARTICLE);
```

```
      result.setValue(Field.AUTHOR, "Jesús M. Pérez and Javier Muguerza and Olatz Arbelaitz and Ibai Gurrutxaga
and José I. Martín");
      result.setValue(Field.YEAR, "2007");
      result.setValue(Field.TITLE, "Combining multiple class distribution modified subsamples in a single
tree");
       result.setValue(Field.JOURNAL, "Pattern Recognition Letters");
       result.setValue(Field.VOLUME, "28");
       result.setValue(Field.NUMBER, "4");
       result.setValue(Field.PAGES, "414-422");
       result.setValue(Field.URL, "http://dx.doi.org/10.1016/j.patrec.2006.08.013");

       TechnicalInformation additional = new TechnicalInformation(Type.ARTICLE);
       additional.setValue(Field.AUTHOR, "Igor Ibarguren and Jesús M. Pérez and Javier Muguerza and Ibai
Gurrutxaga and Olatz Arbelaitz");
       additional.setValue(Field.YEAR, "2014");
       additional.setValue(Field.TITLE, "An extensive analysis of consolidated trees with a new resampling
strategy to establish their robustness across multiple classification contexts againsts a wide set of rule
induction algorithms");
       additional.setValue(Field.JOURNAL, "Knowledge Based Systems (submitted)");
       result.add(additional);

      return result;
   }

   /**
    * Returns default capabilities of the classifier.
    *
    * @return      the capabilities of this classifier
    */
   public Capabilities getCapabilities() {
      Capabilities      result;

      try {
         result = new C45PruneableClassifierTree(null, !m_unpruned, m_CF, m_subtreeRaising, !m_noCleanup,
m_collapseTree).getCapabilities();
      }
      catch (Exception e) {
         result = new Capabilities(this);
      }

      result.setOwner(this);

      return result;
   }

   /**
    * Generates the classifier.
    * (Implements the original CTC algorithm, so it
    *  does not implement the options binarySplits and reducedErrorPruning of J48,
    *  only what is based on C4.5 algorithm)
    *
    * @param instances the data to train the classifier with
    * @throws Exception if classifier can't be built successfully
    */
   public void buildClassifier(Instances instances)
         throws Exception {

      ModelSelection modSelection;
      // TODO Implement the option binarySplits of J48
      modSelection = new C45ConsolidatedModelSelection(m_minNumObj, instances,
                        m_useMDLcorrection, m_doNotMakeSplitPointActualValue);
      // TODO Implement the option reducedErrorPruning of J48
      m_root = new C45ConsolidatedPruneableClassifierTree(modSelection, !m_unpruned,
            m_CF, m_subtreeRaising, !m_noCleanup, m_collapseTree);

      // remove instances with missing class before generating samples
      instances = new Instances(instances);
      instances.deleteWithMissingClass();

      //Generate as many samples as the number of samples with the given instances
      Instances[] samplesVector = generateSamples(instances);
       //if (m_Debug)
       //   printSamplesVector(samplesVector);

      ((C45ConsolidatedPruneableClassifierTree)m_root).buildClassifier(instances, samplesVector);
```

```
    ((C45ModelSelection) modSelection).cleanup();
 }

 /**
  * Generate as many samples as the number of samples based on Resampling Method parameters
  *
  * @param instances the training data which will be used to generate the sample set
  * @return Instances[] the vector of generated samples
  * @throws Exception if something goes wrong
  */
protected Instances[] generateSamples(Instances instances) throws Exception {
   Instances[] samplesVector = null;
   // can classifier tree handle the data?
   getCapabilities().testWithFail(instances);

   // remove instances with missing class
   InstancesConsolidated instancesWMC = new InstancesConsolidated(instances);
   instancesWMC.deleteWithMissingClass();
   if (m_Debug) {
      System.out.println("=== Generation of the set of samples ===");
      System.out.println(toStringResamplingMethod());
   }
   /** Original sample size */
   int dataSize = instancesWMC.numInstances();
   if(dataSize==0)
      System.err.println("Original data size is 0! Handle zero training instances!");
   else
      if (m_Debug)
         System.out.println("Original data size: " + dataSize);
   /** Size of samples(bags) to be generated */
   int bagSize = 0;

   // Some checks done in set-methods
   //@ requires  0 <= m_RMnumberSamples
   //@ requires -2 <= m_RMbagSizePercent && m_RMbagSizePercent <= 100
   //@ requires -2 <= m_RMnewDistrMinClass && m_RMnewDistrMinClass < 100
   if(m_RMbagSizePercent >= 0 ){
      bagSize =  dataSize * m_RMbagSizePercent / 100;
      if(bagSize==0)
         System.err.println("Size of samples is 0 (" + m_RMbagSizePercent + "% of " + dataSize
               + ")! Handle zero training instances!");
   } else if (m_RMnewDistrMinClass < 0) { // stratified OR free
      throw new Exception("Size of samples, m_RMbagSizePercent, (" + m_RMbagSizePercent +
               ") has to be between 0 and 100, when m_RMnewDistrMinClass < 0 (stratified or free)!!!");
   }

   Random random;
   if (dataSize == 0) // To be OK when testing to Handle zero training instances!
      random = new Random(m_Seed);
   else
      random = instancesWMC.getRandomNumberGenerator(m_Seed);

   // Generate the vector of samples with the given parameters
   // TODO Set the different options to generate the samples like a filter and then use it here.
   if(m_RMnewDistrMinClass == (float)-2)
      // stratified: Maintains the original class distribution in the new samples
      samplesVector = generateStratifiedSamples(instancesWMC, dataSize, bagSize, random);
   else if (m_RMnewDistrMinClass == (float)-1)
      // free: It doesn't take into account the original class distribution
      samplesVector = generateFreeDistrSamples(instancesWMC, dataSize, bagSize, random);
   else
      // RMnewDistrMinClass is between 0 and 100: Changes the class distribution to the indicated value
      samplesVector = generateSamplesChangingMinClassDistr(instancesWMC, dataSize, bagSize, random);
   if (m_Debug)
      System.out.println("=== End of Generation of the set of samples ===");
   return samplesVector;
}

/**
 * Generate a set of stratified samples
 *
 * @param instances the training data which will be used to generate the sample set
 * @param dataSize Size of original sample (instances)
 * @param bagSize Size of samples(bags) to be generated
```

```
 * @param random a random number generator
 * @return Instances[] the vector of generated samples
 * @throws Exception if something goes wrong
 */
private Instances[] generateStratifiedSamples(
      InstancesConsolidated instances, int dataSize, int bagSize, Random random) throws Exception{
   int numClasses = instances.numClasses();
   // Get the classes
   InstancesConsolidated[] classesVector =  instances.getClasses();
   // What is the minority class?
   /** Vector containing the size of each class */
   int classSizeVector[] = instances.getClassesSize(classesVector);
   /** Index of the minority class in the original sample */
   int iMinClass = Utils.minIndex(classSizeVector);
   if (m_Debug)
      instances.printClassesInformation(dataSize , iMinClass, classSizeVector);

   // Determine the sizes of each class in the new samples
   /** Vector containing the size of each class in the new samples */
   int newClassSizeVector[] = new int [numClasses];
   // Check the bag size
   int bagSizePercent;
   if((dataSize == bagSize) && !m_RMreplacement){
      System.out.println("It doesn't make sense that the original sample's size and " +
            "the size of samples to be generated are the same without using replacement" +
            "because all the samples will be entirely equal!!!\n" +
            m_bagSizePercentToReduce + "% will be used  as the bag size percentage!!!");
      bagSizePercent = m_bagSizePercentToReduce;
      bagSize =  dataSize * m_bagSizePercentToReduce / 100;
   }
   else
      bagSizePercent = m_RMbagSizePercent;
   /** Partial bag size */
   int localBagSize = 0;
   for(int iClass = 0; iClass < numClasses; iClass++)
      if(iClass != iMinClass){
         /** Value for the 'iClass'-th class size of the samples to be generated */
         int newClassSize = Utils.round(classSizeVector[iClass] * (double)bagSizePercent / 100);
         newClassSizeVector[iClass] = newClassSize;
         localBagSize += newClassSize;
      }
   /** Value for the minority class size of the samples to be generated */
   // (Done in this way to know the exact size of the minority class in the generated samples)
   newClassSizeVector[iMinClass] = bagSize - localBagSize;
   if (m_Debug) {
      System.out.println("New bag size: " + bagSize);
      System.out.println("Classes sizes of the new bag:");
      for (int iClass = 0; iClass < numClasses; iClass++){
         System.out.print(newClassSizeVector[iClass]);
         if(iClass < numClasses - 1)
            System.out.print(", ");
      }
      System.out.println("");
   }
   // Determine the size of samples' vector; the number of samples
   int numberSamples;
   /** Calculate the ratio of the sizes for each class between the sample and the subsample */
   double bagBySampleClassRatioVector[] = new double[numClasses];
   for(int iClass = 0; iClass < numClasses; iClass++)
      if (classSizeVector[iClass] > 0)
         bagBySampleClassRatioVector[iClass] = newClassSizeVector[iClass] / (double)classSizeVector[iClass];
      else // The size of the class is 0
         // This class won't be selected
         bagBySampleClassRatioVector[iClass] = Double.MAX_VALUE;
   if(m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage) {
      // The number of samples depends on the coverage to be guaranteed for the most disfavored class.
      double coverage = m_RMnumberSamples / (double)100;
      /** Calculate the most disfavored class in respect of coverage */
      int iMostDisfavorClass = Utils.minIndex(bagBySampleClassRatioVector);
      if (m_Debug) {
         System.out.println("Ratio bag:sample by each class:");
         System.out.println("(*) The most disfavored class based on coverage");
         for (int iClass = 0; iClass < numClasses; iClass++){
            System.out.print(Utils.doubleToString(bagBySampleClassRatioVector[iClass],2));
            if(iClass == iMostDisfavorClass)
```

```
            System.out.print("(*)");
         if(iClass < numClasses - 1)
            System.out.print(", ");
      }
      System.out.println("");
   }
   if(m_RMreplacement)
      numberSamples = (int) Math.ceil((-1) * Math.log(1 - coverage) /
                              bagBySampleClassRatioVector[iMostDisfavorClass]);
   else
      numberSamples = (int) Math.ceil(Math.log(1 - coverage) /
                              Math.log(1 - bagBySampleClassRatioVector[iMostDisfavorClass]));
   System.out.println("The number of samples to guarantee at least a coverage of " +
                      Utils.doubleToString(100*coverage,0) + "% is " + numberSamples + ".");
   m_numberSamplesByCoverage = numberSamples;
   if (numberSamples < 3){
      numberSamples = 3;
      System.out.println("(*) Forced the number of samples to be 3!!!");
      m_stExceptionalSituationsMessage += " (*) Forced the number of samples to be 3!!!\n";
   }
} else // m_RMnumberSamplesHowToSet == NumberSamples_FixedValue
   // The number of samples has been set by parameter
   numberSamples = (int)m_RMnumberSamples;

// Calculate the true coverage achieved
m_trueCoverage = (double)0.0;
for (int iClass = 0; iClass < numClasses; iClass++){
   double trueCoverageByClass;
   if(classSizeVector[iClass] > 0){
      if(m_RMreplacement)
         trueCoverageByClass = 1 - Math.pow(Math.E,
                                            (-1) * bagBySampleClassRatioVector[iClass] * numberSamples);
      else
         trueCoverageByClass = 1 - Math.pow((1 - bagBySampleClassRatioVector[iClass]), numberSamples);
   } else
      trueCoverageByClass = (double)0.0;
   double ratioClassDistr = classSizeVector[iClass] / (double)dataSize;
   m_trueCoverage += ratioClassDistr * trueCoverageByClass;
}

// Set the size of the samples' vector
Instances[] samplesVector = new Instances[numberSamples];

// Generate the vector of samples
for(int iSample = 0; iSample < numberSamples; iSample++){
   InstancesConsolidated bagData = null;
   InstancesConsolidated bagClass = null;
   for(int iClass = 0; iClass < numClasses; iClass++){
      // Extract instances of the iClass-th class
      if(m_RMreplacement)
         bagClass = new InstancesConsolidated(classesVector[iClass].resampleWithWeights(random));
      else
         bagClass = new InstancesConsolidated(classesVector[iClass]);
      // Shuffle the instances
      bagClass.randomize(random);
      if (newClassSizeVector[iClass] < classSizeVector[iClass]) {
         InstancesConsolidated newBagData = new InstancesConsolidated(bagClass, 0,
                                                                      newClassSizeVector[iClass]);
         bagClass = newBagData;
         newBagData = null;
      }
      if(bagData == null)
         bagData = bagClass;
      else
         bagData.add(bagClass);
      bagClass = null;
   }
   // Shuffle the instances
   bagData.randomize(random);
   samplesVector[iSample] = (Instances)bagData;
   bagData = null;
}
classesVector = null;
classSizeVector = null;
newClassSizeVector = null;
```

```
   return samplesVector;
}

/**
 * Generate a set of samples without taking the class distribution into account
 * (like in the meta-classifier Bagging)
 *
 * @param instances the training data which will be used to generate the sample set
 * @param dataSize Size of original sample (instances)
 * @param bagSize Size of samples(bags) to be generated
 * @param random a random number generator
 * @return Instances[] the vector of generated samples
 * @throws Exception if something goes wrong
 */
 private Instances[] generateFreeDistrSamples(
        InstancesConsolidated instances, int dataSize, int bagSize, Random random) throws Exception{
     // Check the bag size
     if((dataSize == bagSize) && !m_RMreplacement){
        System.out.println("It doesn't make sense that the original sample's size and " +
              "the size of samples to be generated are the same without using replacement" +
              "because all the samples will be entirely equal!!!\n" +
              m_bagSizePercentToReduce + "% will be used  as the bag size percentage!!!");
        bagSize =  dataSize * m_bagSizePercentToReduce / 100;
     }
     if (m_Debug)
        System.out.println("New bag size: " + bagSize);
     // Determine the size of samples' vector; the number of samples
     int numberSamples;
     double bagBySampleRatio = bagSize / (double) dataSize;
     if(m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage) {
        // The number of samples depends on the coverage to be guaranteed for the most disfavored class.
        double coverage = m_RMnumberSamples / (double)100;
        if(m_RMreplacement)
           numberSamples = (int) Math.ceil((-1) * Math.log(1 - coverage) /
                                        bagBySampleRatio);
        else
           numberSamples = (int) Math.ceil(Math.log(1 - coverage) /
                                    Math.log(1 - bagBySampleRatio));
        System.out.println("The number of samples to guarantee at least a coverage of " +
                           Utils.doubleToString(100*coverage,0) + "% is " + numberSamples + ".");
        m_numberSamplesByCoverage = numberSamples;
        if (numberSamples < 3){
           numberSamples = 3;
           System.out.println("(*) Forced the number of samples to be 3!!!");
           m_stExceptionalSituationsMessage += " (*) Forced the number of samples to be 3!!!\n";
        }
     } else // m_RMnumberSamplesHowToSet == NumberSamples_FixedValue
        // The number of samples has been set by parameter
        numberSamples = (int)m_RMnumberSamples;

     // Calculate the true coverage achieved
     if(m_RMreplacement)
        m_trueCoverage = 1 - Math.pow(Math.E, (-1) * bagBySampleRatio * numberSamples);
     else
        m_trueCoverage = 1 - Math.pow((1 - bagBySampleRatio), numberSamples);

     // Set the size of the samples' vector
     Instances[] samplesVector = new Instances[numberSamples];

     // Generate the vector of samples
     for(int iSample = 0; iSample < numberSamples; iSample++){
        Instances bagData = null;
        if(m_RMreplacement)
           bagData = new Instances(instances.resampleWithWeights(random));
        else
           bagData = new Instances(instances);
        // Shuffle the instances
        bagData.randomize(random);
        if (bagSize < dataSize) {
           Instances newBagData = new Instances(bagData, 0, bagSize);
           bagData = newBagData;
           newBagData = null;
        }
        samplesVector[iSample] = bagData;
```

```
            bagData = null;
        }
        return samplesVector;
    }

    /**
     * Generate a set of samples changing the distribution of the minority class
     *
     * @param instances the training data which will be used to generate the sample set
     * @param dataSize Size of original sample (instances)
     * @param bagSize Size of samples(bags) to be generated
     * @param random a random number generator
     * @return Instances[] the vector of generated samples
     * @throws Exception if something goes wrong
     */
    private Instances[] generateSamplesChangingMinClassDistr(
            InstancesConsolidated instances, int dataSize, int bagSize, Random random) throws Exception{
        int numClasses = instances.numClasses();
        // Some checks
        if((numClasses > 2) && (m_RMnewDistrMinClass != (float)50.0))
            throw new Exception("In the case of multi-class datasets, the only posibility to change the
distribution of classes is to balance them!!!\n" +
                    "Use the special value '50.0' in <distribution minority class> for this purpose!!!");
        // TODO Generalize the process to multi-class datasets to set different new values of distribution for
each classs.
        // Some checks done in set-methods
        //@ requires m_RMreplacement = false
        // TODO Accept replacement

        // Get the classes
        InstancesConsolidated[] classesVector =  instances.getClasses();

        // What is the minority class?
        /** Vector containing the size of each class */
        int classSizeVector[] = instances.getClassesSize(classesVector);
        /** Index of the minority class in the original sample */
        int iMinClass, i_iMinClass;
        /** Prevent the minority class from being empty (we hope there is one non-empty!) */
        int iClassSizeOrdVector[] = Utils.sort(classSizeVector);
        for(i_iMinClass = 0; ((i_iMinClass < numClasses) &&
                              (classSizeVector[iClassSizeOrdVector[i_iMinClass]] == 0)); i_iMinClass++);
        if(i_iMinClass < numClasses)
            iMinClass = iClassSizeOrdVector[i_iMinClass];
        else // To be OK when testing to Handle zero training instances!
            iMinClass = 0;

        /** Index of the majority class in the original sample */
        int iMajClass = Utils.maxIndex(classSizeVector);
        /** Determines whether the original sample is balanced or not */
        boolean isBalanced = false;
        if (iMinClass == iMajClass){
            isBalanced = true;
            // If the sample is balanced, it is determined, by convention, that the majority class is the last one
            iMajClass = numClasses-1;
        }
        if (m_Debug)
            instances.printClassesInformation(dataSize , iMinClass, classSizeVector);

        /** Distribution of the minority class in the original sample */
        float distrMinClass;
        if (dataSize == 0)
            distrMinClass = (float)0;
        else
            distrMinClass = (float)100 * classSizeVector[iMinClass] / dataSize;

        /** Guarantee the minimum number of examples in each class based on m_minExamplesPerClassPercent */
        int minExamplesPerClass = (int) Math.ceil(dataSize * m_minExamplesPerClassPercent / (double)100.0) ;
        /** Guarantee to be at least m_minNumObj */
        if (minExamplesPerClass < m_minNumObj)
            minExamplesPerClass = m_minNumObj;
        if (m_Debug)
            System.out.println("Minimum number of examples to be guaranteed in each class: " +
                                                                                    minExamplesPerClass);
        for(int iClass = 0; iClass < numClasses; iClass++){
            if((classSizeVector[iClass] < minExamplesPerClass) && // if number of examples is smaller than the
```

```
minimum
               (classSizeVector[iClass] > 0)){              // but, at least, it has to exist any example.
            // Oversample the class ramdonly
            System.out.println("The " + iClass + "-th class has too few examples (" +
                           classSizeVector[iClass]+ ")!!!\n" +
                           "It will be oversampled ranmdoly up to " + minExamplesPerClass + "!!!");
            m_stExceptionalSituationsMessage += " (*) Forced the " + iClass +
                                                "-th class to be oversampled!!!\n";
            // based on the code of the function 'resample(Random)' of the class 'Instances'
            InstancesConsolidated bagClass = classesVector[iClass];
             while (bagClass.numInstances() < minExamplesPerClass) {
                bagClass.add(classesVector[iClass].instance(random.nextInt(classSizeVector[iClass])));
             }
             // Update the vectors with classes' information and the new data size
             dataSize = dataSize - classSizeVector[iClass] + minExamplesPerClass;
             classesVector[iClass] = bagClass;
             classSizeVector[iClass] = minExamplesPerClass;
         }
      }

      /** Maximum values for classes' size on the samples to be generated taking RMnewDistrMinClass into
account
       *   and without using replacement */
      int maxClassSizeVector[] = new int[numClasses];
      if (numClasses == 2){
         // the dataset is two-class
         if(m_RMnewDistrMinClass > distrMinClass){
            // Maintains the whole minority class
            maxClassSizeVector[iMinClass] = classSizeVector[iMinClass];
            maxClassSizeVector[iMajClass] = Utils.round(classSizeVector[iMinClass] *
                                                (100 - m_RMnewDistrMinClass) / m_RMnewDistrMinClass);
         } else {
            // Maintains the whole majority class
            maxClassSizeVector[iMajClass] = classSizeVector[iMajClass];
            maxClassSizeVector[iMinClass] = Utils.round(classSizeVector[iMajClass] *
                                                m_RMnewDistrMinClass / (100 - m_RMnewDistrMinClass));
         }
      } else {
         // the dataset is multi-class
         /** The only accepted option is to change the class distribution is to balance the samples */
         for(int iClass = 0; iClass < numClasses; iClass++)
            maxClassSizeVector[iClass] = classSizeVector[iMinClass];
      }

      // Determine the sizes of each class in the new samples
      /** Vector containing the size of each class in the new samples */
      int newClassSizeVector[] = new int[numClasses];
      /** Determines whether the size of samples to be generated will be forced to be reduced in exceptional
situations */
      boolean forceToReduceSamplesSize = false;
      if(m_RMbagSizePercent == -2){
         // maxSize : Generate the biggest samples according to the indicated distribution
(RMnewDistrMinClass),
         //  that is, maintaining the whole minority (majority) class
         if (numClasses == 2){
            // the dataset is two-class
            if(Utils.eq(m_RMnewDistrMinClass, distrMinClass)){
               System.out.println("It doesn't make sense that the original distribution and " +
                  "the distribution to be changed (RMnewDistrMinClass) are the same and " +
                  "the size of samples to be generated is maximum (RMbagSizePercent=-2) " +
                  "(without using replacement) " +
                  "because all the samples will be entirely equal!!!\n" +
                  m_bagSizePercentToReduce + "% will be used  as the bag size percentage!!!");
               forceToReduceSamplesSize = true;
            }
         } else
            // the dataset is multi-class
            if (isBalanced){
               System.out.println("In the case of multi-class datasets, if the original sample is balanced, " +
                  "it doesn't make sense that " +
                  "the size of samples to be generated is maximum (RMbagSizePercent=-2) " +
                  "(without using replacement) " +
                  "because all the samples will be entirely equal!!!\n" +
                  m_bagSizePercentToReduce + "% will be used  as the bag size percentage!!!");
               forceToReduceSamplesSize = true;
```

```
            }
         if(!forceToReduceSamplesSize){
            bagSize = 0;
            for(int iClass = 0; iClass < numClasses; iClass++)
               if (classSizeVector[iClass] == 0)
                  newClassSizeVector[iClass] = 0;
               else {
                  newClassSizeVector[iClass] = maxClassSizeVector[iClass];
                  bagSize += maxClassSizeVector[iClass];
               }
         }
      } else {
         if (m_RMbagSizePercent == -1)
            // sizeOfMinClass: the generated samples will have the same size that the minority class
            bagSize = classSizeVector[iMinClass];
         else
            // m_RMbagSizePercent is between 0 and 100. bagSize is already set.
            if(dataSize == bagSize){
               System.out.println("It doesn't make sense that the original sample's size and " +
                     "the size of samples to be generated are the same" +
                     "(without using replacement) " +
                     "because all the samples will be entirely equal!!!\n" +
                     m_bagSizePercentToReduce + "% will be used  as the bag size percentage!!!");
               forceToReduceSamplesSize = true;
            }
      }
      if((m_RMbagSizePercent != -2) || forceToReduceSamplesSize)
      {
         if (numClasses == 2){
            // the dataset is two-class
            if(forceToReduceSamplesSize){
               bagSize =  dataSize * m_bagSizePercentToReduce / 100;
               m_stExceptionalSituationsMessage += " (*) Forced to reduce the size of the generated
samples!!!\n";
            }
            newClassSizeVector[iMinClass] = Utils.round(m_RMnewDistrMinClass * bagSize / 100);
            newClassSizeVector[iMajClass] = bagSize - newClassSizeVector[iMinClass];
         } else {
            // the dataset is multi-class
            /** The only accepted option is to change the class distribution is to balance the samples */
            /** All the classes will have the same size, classSize, based on bagSizePercent applied
             *  on minority class, that is, the generated samples will be bigger than expected,
             *  neither sizeofMinClass nor original sample's size by bagSizePercent. Otherwise,
             *  the classes of the samples would be too unpopulated */
            int bagSizePercent;
            if (m_RMbagSizePercent == -1)
               /** sizeOfMinClass (-1) is a special case, where the bagSizePercent to be applied will be
                *  the half of the minority class, the same that it would be achieved if the dataset was two-
class */
               bagSizePercent = 50;
            else
               if (forceToReduceSamplesSize)
                  bagSizePercent = m_bagSizePercentToReduce;
               else
                  bagSizePercent = m_RMbagSizePercent;
            int classSize = (int)(bagSizePercent * classSizeVector[iMinClass] / (float)100);
            bagSize = 0;
            for(int iClass = 0; iClass < numClasses; iClass++)
               if (classSizeVector[iClass] == 0)
                  newClassSizeVector[iClass] = 0;
               else {
                  newClassSizeVector[iClass] = classSize;
                  bagSize += classSize;
               }
         }
      }

      if (m_Debug) {
         System.out.println("New bag size: " + bagSize);
         System.out.println("New minority class size: " + newClassSizeVector[iMinClass] + " (" +
                                 (int)(newClassSizeVector[iMinClass] / (double)bagSize * 100) + "%)");
         System.out.print("New majority class size: " + newClassSizeVector[iMajClass]);
         if (numClasses > 2)
            System.out.print(" (" + (int)(newClassSizeVector[iMajClass] / (double)bagSize * 100) + "%)");
         System.out.println();
```

```
}
// Some checks
for(int iClass = 0; iClass < numClasses; iClass++)
   if(newClassSizeVector[iClass] > classSizeVector[iClass])
      throw new Exception("There aren't enough instances of the " + iClass +
            "-th class (" +   classSizeVector[iClass] +
            ") to extract " + newClassSizeVector[iClass] +
            " for the new samples whithout replacement!!!");

// Determine the size of samples' vector; the number of samples
int numberSamples;
/** Calculate the ratio of the sizes for each class between the sample and the subsample */
double bagBySampleClassRatioVector[] = new double[numClasses];
for(int iClass = 0; iClass < numClasses; iClass++)
   if (classSizeVector[iClass] > 0)
      bagBySampleClassRatioVector[iClass] = newClassSizeVector[iClass] / (double)classSizeVector[iClass];
   else // The size of the class is 0
      // This class won't be selected
      bagBySampleClassRatioVector[iClass] = Double.MAX_VALUE;
if(m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage) {
   // The number of samples depends on the coverage to be guaranteed for the most disfavored class.
   double coverage = m_RMnumberSamples / (double)100;
   /** Calculate the most disfavored class in respect of coverage */
   int iMostDisfavorClass = Utils.minIndex(bagBySampleClassRatioVector);
   if (m_Debug) {
      System.out.println("Ratio bag:sample by each class:");
      System.out.println("(*) The most disfavored class based on coverage");
      for (int iClass = 0; iClass < numClasses; iClass++){
         System.out.print(Utils.doubleToString(bagBySampleClassRatioVector[iClass],2));
         if(iClass == iMostDisfavorClass)
            System.out.print("(*)");
         if(iClass < numClasses - 1)
            System.out.print(", ");
      }
      System.out.println("");
   }
   numberSamples = (int) Math.ceil(Math.log(1 - coverage) /
                           Math.log(1 - bagBySampleClassRatioVector[iMostDisfavorClass]));
   System.out.println("The number of samples to guarantee at least a coverage of " +
                     Utils.doubleToString(100*coverage,0) + "% is " + numberSamples + ".");
   m_numberSamplesByCoverage = numberSamples;
   if (numberSamples < 3){
      numberSamples = 3;
      System.out.println("(*) Forced the number of samples to be 3!!!");
      m_stExceptionalSituationsMessage += " (*) Forced the number of samples to be 3!!!\n";
   }
} else // m_RMnumberSamplesHowToSet == NumberSamples_FixedValue
   // The number of samples has been set by parameter
   numberSamples = (int)m_RMnumberSamples;

// Calculate the true coverage achieved
m_trueCoverage = (double)0.0;
for (int iClass = 0; iClass < numClasses; iClass++){
   double trueCoverageByClass;
   if(classSizeVector[iClass] > 0){
      if(m_RMreplacement)
         trueCoverageByClass = 1 - Math.pow(Math.E, (-1) * bagBySampleClassRatioVector[iClass] *
                                                                                  numberSamples);
      else
         trueCoverageByClass = 1 - Math.pow((1 - bagBySampleClassRatioVector[iClass]), numberSamples);
   } else
      trueCoverageByClass = (double)0.0;
   double ratioClassDistr = classSizeVector[iClass] / (double)dataSize;
   m_trueCoverage += ratioClassDistr * trueCoverageByClass;
}

// Set the size of the samples' vector
Instances[] samplesVector = new Instances[numberSamples];

// Generate the vector of samples
for(int iSample = 0; iSample < numberSamples; iSample++){
   InstancesConsolidated bagData = null;
   InstancesConsolidated bagClass = null;

   for (int iClass = 0; iClass < numClasses; iClass++)
```

```
         if (classSizeVector[iClass] > 0){
            // Extract instances of the i-th class
            bagClass = new InstancesConsolidated(classesVector[iClass]);
            // Shuffle the instances
            bagClass.randomize(random);
            if (newClassSizeVector[iClass] < classSizeVector[iClass]) {
               InstancesConsolidated newBagData = new InstancesConsolidated(bagClass, 0,
                                                                            newClassSizeVector[iClass]);
                bagClass = newBagData;
                newBagData = null;
            }
            // Add the bagClass (i-th class) to bagData
            if (bagData == null)
               bagData = bagClass;
            else
               bagData.add(bagClass);
            bagClass = null;
         }
      // Shuffle the instances
      if (bagData == null) // To be OK when testing to Handle zero training instances!
         bagData = instances;
      else
         bagData.randomize(random);
      samplesVector[iSample] = (Instances)bagData;
      bagData = null;
   }
   classesVector = null;
   classSizeVector = null;
   maxClassSizeVector = null;
   newClassSizeVector = null;

   return samplesVector;
}

/**
 * Print the generated samples. Only for testing purposes.
 *
 * @param samplesVector the vector of samples
 */
protected void printSamplesVector(Instances[] samplesVector){

   for(int iSample=0; iSample<samplesVector.length; iSample++){
      System.out.println("==== SAMPLE " + iSample + " ====");
      System.out.println(samplesVector[iSample]);
      System.out.println(" ");
   }
}

/**
 * Returns an enumeration describing the available options.
 *
 * Valid options are:<br/>
 *
 * J48 options<br/>
* ==============<br/>
*
* <pre>-U
* Use unpruned tree.</pre>
*
* <pre>-C confidence
* Set confidence threshold for pruning. (Default: 0.25)</pre>
*
* <pre>-M number
* Set minimum number of instances per leaf. (Default: 2)</pre>
*
* <pre>-S
* Don't perform subtree raising.</pre>
*
* <pre>-L
* Do not clean up after the tree has been built.</pre>
*
* <pre>-A
* If set, Laplace smoothing is used for predicted probabilites.</pre>
*
* <pre>-Q seed
```

```
 * Seed for random data shuffling (Default: 1)</pre>
 *
 * Options to set the Resampling Method (RM) for the generation of samples
 *  to use in the consolidation process
 * ================================================================================
 * <pre>-RM-C
 * Determines the way to set the number of samples to be generated will be based on
 * a coverage value as a percentage. In the case this option is not set, the number of samples
 * will be determined using a fixed value.
 * (set by default)</pre>
 *
 * <pre>-RM-N &lt;number of samples&gt;
 * Number of samples to be generated for the use in the construction of the consolidated tree.
 * It can be set as a fixed value or based on a coverage value as a percentage, when -RM-C option
 * is used, which guarantees the number of samples necessary to adequately cover the examples
 * of the original sample
 * (Default 5 for a fixed value or 99% for the case based on a coverage value)</pre>
 *
 * <pre>-RM-R
 * Determines whether or not replacement is used when generating the samples.
 * (Default: false)</pre>
 *
 * <pre>-RM-B percentage
 * Size of each sample(bag), as a percentage of the training set size.
 * Combined with the option &lt;distribution minority class&gt; accepts:
 *  * -1 (sizeOfMinClass): The size of the minority class
 *  * -2 (maxSize): Maximum size taking &lt;distribution minority class&gt; into account
 *              and using no replacement
 * (Default: -2(maxSize))</pre>
 *
 * <pre>-RM-D distribution minority class
 * Determines the new value of the distribution of the minority class, if we want to change it.
 * It can be one of the following values:
 *  * A value between 0 and 100 to change the portion of minority class instances in the new samples
 *    (If the dataset is multi-class, only the special value 50.0 will be accepted to balance the classes)
 *  * -1 (free): Works with the instances without taking their class into account
 *  * -2 (stratified): Maintains the original class distribution in the new samples
 * (Default: -1(free))</pre>
 *
 * @return an enumeration of all the available options.
 */
public Enumeration listOptions() {

  Vector<Option> newVector = new Vector<Option>();

  // J48 options
  // ===========
   Enumeration en;
   en = super.listOptions();
   while (en.hasMoreElements())
      newVector.addElement((Option) en.nextElement());

  // Options to set the Resampling Method (RM) for the generation of samples
  //  to use in the consolidation process
  // ================================================================================
  newVector.
  addElement(new Option("\tSet the number of samples to be generated based on a coverage value\n" +
        "\tas a percentage (by default)",
        "RM-C", 0, "-RM-C"));
  newVector.
  addElement(new Option("\tNumber of samples to be generated for the use in the construction of the\n" +
        "\tconsolidated tree.\n" +
        "\tIt can be set as a fixed value or based on a coverage value as a percentage, \n" +
        "\twhen -RM-C option is used, which guarantees the number of samples necessary \n" +
        "\tto adequately cover the examples of the original sample.\n" +
        "\t(default: 5 for a fixed value and \n" +
        "\t " + Utils.doubleToString(m_coveragePercent,0) + "% for the case based on a coverage value)",
        "RM-N", 1, "-RM-N <Number of samples>"));
  newVector.
  addElement(new Option("\tUse replacement to generate the set of samples\n" +
        "\t(default false)",
        "RM-R", 0, "-RM-R"));
  newVector.
  addElement(new Option("\tSize of each sample(bag), as a percentage of the training set size.\n" +
        "\tCombined with the option <distribution minority class> accepts:\n" +
```

```
        "\t * -1 (sizeOfMinClass): The size of the minority class\n" +
        "\t * -2 (maxSize): Maximum size taking <distribution minority class>\n" +
        "\t             into account and using no replacement\n" +
        "\t(default -2(maxSize))",
        "RM-B", 1, "-RM-B <Size of each sample(%)>"));
   newVector.
   addElement(new Option(
        "\tDetermines the new value of the distribution of the minority class.\n" +
        "\tIt can be one of the following values:\n" +
        "\t * A value between 0 and 100 to change the portion of minority class\n" +
        "\t              instances in the new samples\n" +
        "\t   (If the dataset is multi-class, only the special value 50.0 will\n" +
        "\t              be accepted to balance the classes)\n" +
        "\t * -1 (free): Works with the instances without taking their class\n" +
        "\t              into account\n" +
        "\t * -2 (stratified): Maintains the original class distribution in the\n" +
        "\t              new samples\n" +
        "\t(default 50.0)",
        "RM-D", 1, "-RM-D <distribution minority class>"));

   return newVector.elements();
}

/**
 * Parses a given list of options.
 *
<!-- options-start -->
 * Valid options are: <p/>
 *
 * Options to set the Resampling Method (RM) for the generation of samples
 *  to use in the consolidation process
 * ================================================================================
 * <pre> -RM-C
 *  Determines the way to set the number of samples to be generated will be based on
 *  a coverage value as a percentage. In the case this option is not set, the number of samples
 *  will be determined using a fixed value.
 *  (set by default)</pre>
 *
 * <pre> -RM-N &lt;number of samples&gt;
 *  Number of samples to be generated for the use in the construction of the consolidated tree.
 *  It can be set as a fixed value or based on a coverage value as a percentage, when -RM-C option
 *  is used, which guarantees the number of samples necessary to adequately cover the examples
 *  of the original sample
 *  (default 5 for a fixed value or 99% for the case based on a coverage value)</pre>
 *
 * <pre> -RM-R
 *  Determines whether or not replacement is used when generating the samples.
 *  (default true)</pre>
 *
 * <pre> -RM-B &lt;Size of each sample(&#37;)&gt;
 *  Size of each sample(bag), as a percentage of the training set size.
 *  Combined with the option &lt;distribution minority class&gt; accepts:
 *  * -1 (sizeOfMinClass): The size of the minority class
 *  * -2 (maxSize): Maximum size taking &lt;distribution minority class&gt; into account
 *  *           and using no replacement
 *  (default -2(maxSize))</pre>
 *
 * <pre> -RM-D &lt;distribution minority class&gt;
 *  Determines the new value of the distribution of the minority class, if we want to change it.
 *  It can be one of the following values:
 *  * A value between 0 and 100 to change the portion of minority class instances in the new samples
 *    (If the dataset is multi-class, only the special value 50.0 will be accepted to balance the classes)
 *  * -1 (free): Works with the instances without taking their class into account
 *  * -2 (stratified): Maintains the original class distribution in the new samples
 *  (default 50.0)</pre>
 *
<!-- options-end -->
 *
 * @param options the list of options as an array of strings
 * @throws Exception if an option is not supported
 */
public void setOptions(String[] options) throws Exception {

   // Options to set the Resampling Method (RM) for the generation of samples
   //  to use in the consolidation process
```

```
    // ===================================================================
     if (Utils.getFlag("RM-C", options))
         setRMnumberSamplesHowToSet(new SelectedTag(NumberSamples_BasedOnCoverage,
                                                    TAGS_WAYS_TO_SET_NUMBER_SAMPLES));
     else
         setRMnumberSamplesHowToSet(new SelectedTag(NumberSamples_FixedValue,
                                                    TAGS_WAYS_TO_SET_NUMBER_SAMPLES));
    String RMnumberSamplesString = Utils.getOption("RM-N", options);
    if (RMnumberSamplesString.length() != 0) {
       setRMnumberSamples(new Float(RMnumberSamplesString).floatValue());
    }
    else {
       if (m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage)
          setRMnumberSamples((float)m_coveragePercent); // default: f(99% of coverage)
       else
          setRMnumberSamples((float)5.0);  // default: 5 samples
    }
    String RMbagSizePercentString = Utils.getOption("RM-B", options);
    if (RMbagSizePercentString.length() != 0)
       setRMbagSizePercent(Integer.parseInt(RMbagSizePercentString), false);
    else
       setRMbagSizePercent(-2, false); // default: maxSize
    String RMnewDistrMinClassString = Utils.getOption("RM-D", options);
    if (RMnewDistrMinClassString.length() != 0)
       setRMnewDistrMinClass(new Float(RMnewDistrMinClassString).floatValue(), false);
    else
       setRMnewDistrMinClass((float)50.0, false);
    // Only checking the combinations of the three options RMreplacement, RMbagSizePercent and
    //  RMnewDistrMinClass when they all are set.
    setRMreplacement(Utils.getFlag("RM-R", options), true);
    // J48 options
    // ===========
     super.setOptions(options);
}

/**
 * Gets the current settings of the Classifier.
 *
 * @return an array of strings suitable for passing to setOptions
 */
public String [] getOptions() {

   Vector<String> result = new Vector<String>();

   // J48 options
   // ===========
   String[] options = super.getOptions();
   for (int i = 0; i < options.length; i++)
      result.add(options[i]);
   // In J48 m_Seed is added only if m_reducedErrorPruning is true,
   // but in J48Consolidated it is necessary to the generation of the samples
   result.add("-Q");
   result.add("" + m_Seed);

   // Options to set the Resampling Method (RM) for the generation of samples
   //  to use in the consolidation process
   // ===================================================================
    if (m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage)
       result.add("-RM-C");
    result.add("-RM-N");
   result.add(""+ m_RMnumberSamples);
   if (m_RMreplacement)
      result.add("-RM-R");
   result.add("-RM-B");
   result.add("" + m_RMbagSizePercent);
   result.add("-RM-D");
   result.add("" + m_RMnewDistrMinClass);

   return (String[]) result.toArray(new String[result.size()]);
}

/**
 * Returns a description of the classifier.
 *
 * @return a description of the classifier
```

```
 */
public String toString() {

   if (m_root == null) {
      return "No classifier built";
   }
   if (m_unpruned)
      return "J48Consolidated unpruned tree\n" +
            toStringResamplingMethod() +
            m_root.toString();
   else
      return "J48Consolidated pruned tree\n" +
             toStringResamplingMethod() +
            m_root.toString();
}

/**
 * Returns a description of the Resampling Method used in the consolidation process.
 *
 * @return a description of the used Resampling Method (RM)
 */
public String toStringResamplingMethod() {
   String st;
   st = "[RM] N_S=";
   if (m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage){
      st += "f(" + Utils.doubleToString(m_RMnumberSamples,2) + "% of coverage)";
      if (m_numberSamplesByCoverage != 0)
         st += "=" + m_numberSamplesByCoverage;
   }
   else // m_RMnumberSamplesHowToSet == NumberSamples_FixedValue
      st += (int)m_RMnumberSamples;
   if (m_RMnewDistrMinClass == -2)
      st += " stratified";
   else if (m_RMnewDistrMinClass == -1)
      st += " free distribution";
   else {
      st += " %Min=";
      if (Utils.eq(m_RMnewDistrMinClass, (float)50))
         st += "balanced";
      else
         st += m_RMnewDistrMinClass;
   }
   st += " Size=";
   if (m_RMbagSizePercent == -2)
      st += "maxSize";
   else if (m_RMbagSizePercent == -1)
      st += "sizeOfMinClass";
   else
      st += m_RMbagSizePercent + "%";
   if (m_RMreplacement)
      st += " (with replacement)";
   else
      st += " (without replacement)";
   st += "\n";
   st += m_stExceptionalSituationsMessage;
   // Add the true coverage achieved
   st += "True coverage achieved: " + m_trueCoverage + "\n";
   // Add a separator
   char[] ch_line = new char[st.length()];
   for (int i = 0; i < ch_line.length; i++)
      ch_line[i] = '-';
   String line = String.valueOf(ch_line);
   line += "\n";
   st += line;
   return st;
}

/**
 * Returns the tip text for this property
 * @return tip text for this property suitable for
 * displaying in the explorer/experimenter gui
 */
public String RMnumberSamplesHowToSetTipText() {
   return "Way to set the number of samples to be generated:\n" +
         " * using a fixed value which directly indicates the number of samples to be generated\n" +
```

```
            " * based on a coverage value as a percentage (by default)\n";
    }

    /**
     * Get the value of RMnumberSamplesHowToSet.
     *
     * @return Value of RMnumberSamplesHowToSet.
     */
    public SelectedTag getRMnumberSamplesHowToSet() {
        return new SelectedTag(m_RMnumberSamplesHowToSet,
              TAGS_WAYS_TO_SET_NUMBER_SAMPLES);
    }

    /**
     * Set the value of RMnumberSamplesHowToSet. Values other than
     * NumberSamples_FixedValue, or NumberSamples_BasedOnCoverage will be ignored.
     *
     * @param newWayToSetNumberSamples the way to set the number of samples to use
     * @throws Exception if an option is not supported
     */
    public void setRMnumberSamplesHowToSet(SelectedTag newWayToSetNumberSamples) throws Exception {
         if (newWayToSetNumberSamples.getTags() == TAGS_WAYS_TO_SET_NUMBER_SAMPLES)
         {
            int newEvWay = newWayToSetNumberSamples.getSelectedTag().getID();

           if (newEvWay == NumberSamples_FixedValue || newEvWay == NumberSamples_BasedOnCoverage)
               m_RMnumberSamplesHowToSet = newEvWay;
           else
              throw new IllegalArgumentException("Wrong selection type, value should be: "
                                                 + "between 1 and 2");
        }
    }

    /**
     * Returns the tip text for this property
     * @return tip text for this property suitable for
     * displaying in the explorer/experimenter gui
     */
    public String RMnumberSamplesTipText() {
       return "Number of samples to be generated for the use in the consolidation process (fixed value) or based
on a coverage value as a %.\n" +
              " * if RMnumberSamplesHowToSet == " + (new
SelectedTag(NumberSamples_FixedValue,TAGS_WAYS_TO_SET_NUMBER_SAMPLES)).getSelectedTag().getReadable() + "\n" +
              "    A positive value which directly indicates the number of samples to be generated\n" +
              " * if RMnumberSamplesHowToSet == " + (new
SelectedTag(NumberSamples_BasedOnCoverage,TAGS_WAYS_TO_SET_NUMBER_SAMPLES)).getSelectedTag().getReadable() +
"\n" +
              "    A positive value as a percentage, the coverage value, which guarantees the number of samples
necessary\n" +
              "    to adequately cover the examples of the original sample\n" +
              " (default: 5 for a fixed value or " + Utils.doubleToString(m_coveragePercent,0) + "% for the case
based on a coverage value)";
    }

    /**
     * Get the value of RMnumberSamples.
     *
     * @return Value of RMnumberSamples.
     */
    public float getRMnumberSamples() {

       return m_RMnumberSamples;
    }

    /**
     * Set the value of RMnumberSamples.
     *
     * @param v  Value to assign to RMnumberSamples.
     * @throws Exception if an option is not supported
     */
    public void setRMnumberSamples(float v) throws Exception {
       if (m_RMnumberSamplesHowToSet == NumberSamples_FixedValue) {
          if (v < 0)
             throw new Exception("Number of samples has to be greater than zero!");
          if (v == (float)0)
```

```
            System.err.println("Number of samples is 0. It doesn't make sense to build a consolidated tree
without set of samples. Handle zero training instances!");
         if ((v == (float)1) || (v == (float)2))
            System.out.println("It doesn't make sense to build a consolidated tree with 1 or 2 samples, but it's
possible!");
      } else { // m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage
         if (v < -1)
            throw new Exception("Coverage value has to be greater than zero!");
         if (v == (float)0)
            System.err.println("Coverage value is 0. It doesn't make sense to build a consolidated tree without
set of samples. Handle zero training instances!");
      }
      m_RMnumberSamples = v;
   }

   /**
    * Returns the tip text for this property
    * @return tip text for this property suitable for
    * displaying in the explorer/experimenter gui
    */
   public String RMreplacementTipText() {
      return "Whether replacement is performed to generate the set of samples.";
   }

   /**
    * Get the value of RMreplacement
    *
    * @return Value of RMreplacement
    */
   public boolean getRMreplacement() {

      return m_RMreplacement;
   }

   /**
    * Set the value of RMreplacement.
    * Checks the combinations of the options RMreplacement, RMbagSizePercent and RMnewDistrMinClass
    *
    * @param v  Value to assign to RMreplacement.
    * @throws Exception if an option is not supported
    */
   public void setRMreplacement(boolean v) throws Exception {

      setRMreplacement(v, true);
   }

   /**
    * Set the value of RMreplacement, but, optionally,
    * checks the combinations of the options RMreplacement, RMbagSizePercent and RMnewDistrMinClass.
    * This makes possible only checking in the last call of the method setOptions().
    *
    * @param v  Value to assign to RMreplacement.
    * @param checkComb true to check some combinations of options
    * @throws Exception if an option is not supported
    */
   public void setRMreplacement(boolean v, boolean checkComb) throws Exception {

      if(checkComb)
         checkBagSizePercentAndReplacementAndNewDistrMinClassOptions(v, m_RMbagSizePercent,
                                                                      m_RMnewDistrMinClass);
      m_RMreplacement = v;
   }

   /**
    * Returns the tip text for this property
    * @return tip text for this property suitable for
    * displaying in the explorer/experimenter gui
    */
   public String RMbagSizePercentTipText() {
      return "Size of each sample(bag), as a percentage of the training set size/-1=sizeOfMinClass/-
2=maxSize.\n" +
            "Combined with the option <distribution minority class>, RMnewDistrMinClass, accepts:\n" +
            " * -1 (sizeOfMinClass): The size of the minority class\n" +
            " * -2 (maxSize): Maximum size taking <distribution minority class> into account \n" +
            "             and using no replacement." +
```

```
            " (default: -2 (maxSize))";
   }

   /**
    * Get the value of RMbagSizePercent.
    *
    * @return Value of RMbagSizePercent.
    */
   public int getRMbagSizePercent() {

      return m_RMbagSizePercent;
   }

   /**
    * Set the value of RMbagSizePercent.
    * Checks the combinations of the options RMreplacement, RMbagSizePercent and RMnewDistrMinClass
    *
    * @param v  Value to assign to RMbagSizePercent.
    * @throws Exception if an option is not supported
    */
   public void setRMbagSizePercent(int v) throws Exception {

      setRMbagSizePercent(v, true);
   }

   /**
    * Set the value of RMbagSizePercent, but, optionally,
    * checks the combinations of the options RMreplacement, RMbagSizePercent and RMnewDistrMinClass.
    * This makes possible only checking in the last call of the method setOptions().
    *
    * @param v  Value to assign to RMbagSizePercent.
    * @param checkComb true to check some combinations of options
    * @throws Exception if an option is not supported
    */
   public void setRMbagSizePercent(int v, boolean checkComb) throws Exception {

      if ((v < -2) || (v > 100))
         throw new Exception("Size of sample (%) has to be greater than zero and smaller " +
               "than or equal to 100 " +
               "(or combining with the option <distribution minority class> -1 for 'sizeOfMinClass' " +
               "or -2 for 'maxSize')!");
      else if (v == 0)
         throw new Exception("Size of sample (%) has to be greater than zero and smaller "
               + "than or equal to 100!");
      else {
         if(checkComb)
            checkBagSizePercentAndReplacementAndNewDistrMinClassOptions(m_RMreplacement, v,
                                                                        m_RMnewDistrMinClass);
         m_RMbagSizePercent = v;
      }
   }

   /**
    * Returns the tip text for this property
    * @return tip text for this property suitable for
    * displaying in the explorer/experimenter gui
    */
   public String RMnewDistrMinClassTipText() {
      return "Determines the new value of the distribution of the minority class, if we want to change it/-
1=free/-2=stratified.\n" +
            "It can be one of the following values:\n" +
            " * A value between 0 and 100 to change the portion of minority class instances in the new
samples\n" +
            "   (If the dataset is multi-class, only the special value 50.0 will be accepted to balance the
classes)\n" +
            " * -1 (free): Works with the instances without taking their class into account.\n" +
            " * -2 (stratified): Maintains the original class distribution in the new samples.\n" +
            " (default: 50.0)";
   }

   /**
    * Get the value of RMnewDistrMinClass
    *
    * @return Value of RMnewDistrMinClass
    */
```

```
   public float getRMnewDistrMinClass() {
      return m_RMnewDistrMinClass;
   }

   /**
    * Set the value of RMnewDistrMinClass
    * Checks the combinations of the options RMreplacement, RMbagSizePercent and RMnewDistrMinClass
    *
    * @param v Value to assign to RMnewDistrMinClass
    * @throws Exception if an option is not supported
    */
   public void setRMnewDistrMinClass(float v) throws Exception {

      setRMnewDistrMinClass(v, true);
   }

   /**
    * Set the value of RMnewDistrMinClass, but, optionally,
    * checks the combinations of the options RMreplacement, RMbagSizePercent and RMnewDistrMinClass.
    * This makes possible only checking in the last call of the method setOptions().
    *
    * @param v Value to assign to RMnewDistrMinClass
    * @param checkComb true to check
    * @throws Exception if an option is not supported
    */
   public void setRMnewDistrMinClass(float v, boolean checkComb) throws Exception {

      if ((v < -2) || (v == 0) || (v >= 100))
           throw new Exception("Minority class distribution has to be greater than zero and smaller " +
                 "than 100 (or -1 for 'sizeOfMinClass' or -2 for 'maxSize')!");
      else {
         if (checkComb)
            checkBagSizePercentAndReplacementAndNewDistrMinClassOptions(m_RMreplacement, m_RMbagSizePercent, v);
         m_RMnewDistrMinClass = v;
      }
   }

   /**
    * Checks the combinations of the options RMreplacement, RMbagSizePercent and RMnewDistrMinClass
    *
    * @throws Exception if an option is not supported
    */
   private void checkBagSizePercentAndReplacementAndNewDistrMinClassOptions(
         boolean replacement, int bagSizePercent, float newDistrMinClass) throws Exception{

      if((newDistrMinClass > (float)0) && (newDistrMinClass < (float)100))
         // NewDistrMinClass is a valid value to change the distribution of the sample
         if(replacement)
            throw new Exception("Using replacement isn't contemplated to change the distribution of minority
class!");
      if((newDistrMinClass == (float)-1) || (newDistrMinClass == (float)-2)){
         // NewDistrMinClass = free OR stratified
         if(bagSizePercent < 0)
            throw new Exception("Size of sample (%) has to be greater than zero and smaller " +
                    "than or equal to 100!");
         if((!replacement) && (bagSizePercent==100))
            System.err.println("It doesn't make sense that size of sample (%) is 100, when replacement is
false!");
      }
   }

   /**
    * Returns the tip text for this property
    * (Rewritten to indicate this option is not implemented for J48Consolidated)
    *
    * @return tip text for this property suitable for
    * displaying in the explorer/experimenter gui
    */
   public String reducedErrorPruningTipText() {
      return "J48 option not implemented for J48Consolidated";
   }

   /**
    * Set the value of reducedErrorPruning. Turns
    * unpruned trees off if set.
```

```
 * (Rewritten to maintain the default value of J48)
 *
 * @param v  Value to assign to reducedErrorPruning.
 */
public void setReducedErrorPruning(boolean v){

   m_reducedErrorPruning = false;
   throw new RuntimeException("J48 option not implemented for J48Consolidated");
}

/**
 * Returns the tip text for this property
 * (Rewritten to indicate this option is not implemented for J48Consolidated)
 *
 * @return tip text for this property suitable for
 * displaying in the explorer/experimenter gui
 */
public String numFoldsTipText() {
   return "J48 option not implemented for J48Consolidated";
}

/**
 * Set the value of numFolds.
 * (Rewritten to maintain the default value of J48)
 *
 * @param v  Value to assign to numFolds.
 */
public void setNumFolds(int v) {

   m_numFolds = 3;
   throw new RuntimeException("J48 option not implemented for J48Consolidated");
}

/**
 * Returns the tip text for this property
 * (Rewritten to indicate this option is not implemented for J48Consolidated)
 *
 * @return tip text for this property suitable for
 * displaying in the explorer/experimenter gui
 */
public String binarySplitsTipText() {
   return "J48 option not implemented for J48Consolidated";
}

/**
 * Set the value of binarySplits.
 * (Rewritten to maintain the default value of J48)
 *
 * @param v  Value to assign to binarySplits.
 */
public void setBinarySplits(boolean v) {

   m_binarySplits = false;
   throw new RuntimeException("J48 option not implemented for J48Consolidated");
}

/**
 * Returns the tip text for this property
 * (Rewritten to indicate the true using of the seed in this class)
 *
 * @return tip text for this property suitable for
 * displaying in the explorer/experimenter gui
 */
public String seedTipText() {
   return "Seed for random data shuffling in the generation of samples";
}

/**
 * Returns a superconcise version of the model
 *
 * @return a summary of the model
 */
public String toSummaryString() {
   String stNumberSamplesByCoverage;
   if (m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage)
```

```
            stNumberSamplesByCoverage = "Number of samples based on coverage: " +
                                        measureNumberSamplesByCoverage() + "\n";
         else
            stNumberSamplesByCoverage = "";
         return super.toSummaryString() +
               stNumberSamplesByCoverage +
               "True coverage: " + measureTrueCoverage() + "\n";
      }

      /**
       * Returns the number of samples neccesary to achieve the indicated coverage
       * @return number of samples based on coverage
       */
      public double measureNumberSamplesByCoverage() {
         return m_numberSamplesByCoverage;
      }

      /**
       * Returns the true coverage of the examples of the original sample
       * achieved by the set of samples generated for the consolidated tree
       * @return the true coverage achieved
       */
      public double measureTrueCoverage() {
         return m_trueCoverage;
      }

      /**
       * Returns an enumeration of the additional measure names
       * produced by the J48 algorithm, plus the true coverage achieved
       * by the set of samples generated
       * @return an enumeration of the measure names
       */
      public Enumeration enumerateMeasures() {
         Enumeration enm = super.enumerateMeasures();
         Vector measures = new Vector();
         while (enm.hasMoreElements())
            measures.add(enm.nextElement());
         if (m_RMnumberSamplesHowToSet == NumberSamples_BasedOnCoverage)
            measures.add("measureNumberSamplesByCoverage");
         measures.add("measureTrueCoverage");
         return measures.elements();
      }

      /**
       * Returns the value of the named measure
       * @param additionalMeasureName the name of the measure to query for its value
       * @return the value of the named measure
       * @throws IllegalArgumentException if the named measure is not supported
      */
      public double getMeasure(String additionalMeasureName) {
         if (additionalMeasureName.compareToIgnoreCase("measureTrueCoverage") == 0)
            return measureTrueCoverage();
         else
            if (additionalMeasureName.compareToIgnoreCase("measureNumberSamplesByCoverage") == 0)
               return measureNumberSamplesByCoverage();
            else
               return super.getMeasure(additionalMeasureName);
      }

      /**
       * Main method for testing this class
       *
       * @param argv the commandline options
       */
      public static void main(String [] argv){
         runClassifier(new J48Consolidated(), argv);
      }
}
```

InstancesConsolidated.java

```
 package weka.classifiers.trees.j48Consolidated;

import weka.core.Instance;
import weka.core.Instances;
import weka.core.Utils;

/**
 * Class for extending the Instances class in order to add some methods
 * (These methods can be added to the class 'Instances').
 * ************************************************************************
 *
 * @author Jes&uacute;s M. P&eacute;rez (txus.perez@ehu.es)
 * @version $Revision: 2.0 $
 */
public class InstancesConsolidated extends Instances {

/** for serialization */
   private static final long serialVersionUID = 8452710983684965074L;

   /**
    * Constructor calling the constructor of the superclass
    * (Not necessary if the above methods are moved to the official class 'Instances')
    *
    * @param dataset the set to be copied
    */
   public InstancesConsolidated(Instances dataset) {
      super(dataset);
   }

   /**
    * Constructor calling the constructor of the superclass
    * (Not necessary if the above methods are moved to the official class 'Instances')
    *
    * @param source the set of instances from which a subset
    * is to be created
    * @param first the index of the first instance to be copied
    * @param toCopy the number of instances to be copied
    */
   public InstancesConsolidated(Instances source, int first, int toCopy) {
      super(source, first, toCopy);
   }

   /**
    * Gets the vector of classes of the dataset like a set of samples
    *
    * @return the vector of classes
    */
   public InstancesConsolidated[] getClasses(){
      int numClasses = numClasses();
      InstancesConsolidated[] classesVector = new InstancesConsolidated[numClasses];
      // Sort instances based on the class to extract the set of classes
      sort(classIndex());
      // Determine where each class starts in the sorted dataset
      int[] classIndices = getClassIndices();

      for (int iClass = 0; iClass < numClasses; iClass++) {
         int classSize;
         if (iClass == numClasses - 1) // if the last class
            classSize = numInstances() - classIndices[iClass];
         else
            classSize = classIndices[iClass + 1] - classIndices[iClass];
         classesVector[iClass] = new InstancesConsolidated(this, classIndices[iClass], classSize);
      }
      classIndices = null;
      return classesVector;
   }

   /**
    * Creates an index containing the position where each class starts in
    * the dataset. The dataset must be sorted by the class attribute.
    * (based on the method 'createSubsample()' of the class 'SpreadSubsample'
```

```
 *  of the package 'weka.filters.supervised.instances')
 *
 * @return the positions
 */
private int[] getClassIndices() {

   // Create an index of where each class value starts
   int [] classIndices = new int [numClasses() + 1];
   int currentClass = 0;
   classIndices[currentClass] = 0;
   for (int i = 0; i < numInstances(); i++) {
      Instance current = instance(i);
      if (current.classIsMissing()) {
         for (int j = currentClass + 1; j < classIndices.length; j++) {
            classIndices[j] = i;
         }
         break;
      } else if (current.classValue() != currentClass) {
         for (int j = currentClass + 1; j <= current.classValue(); j++) {
            classIndices[j] = i;
         }
         currentClass = (int) current.classValue();
      }
   }
   if (currentClass <= numClasses()) {
      for (int j = currentClass + 1; j < classIndices.length; j++) {
         classIndices[j] = numInstances();
      }
   }
   return classIndices;
}

/**
 * Gets the vector with the size of each class of the dataset
 *
 * @param classesVector the vector of classes of the dataset like a set of samples
 * @return the vector of classes' size
 */
public int[] getClassesSize(InstancesConsolidated[] classesVector){
   int numClasses = numClasses();
   int classSizeVector[] = new int [numClasses];
   for (int iClass = 0; iClass < numClasses; iClass++)
      classSizeVector[iClass] = classesVector[iClass].numInstances();
   return classSizeVector;
}

/**
 * Adds a set of instances to the end of the set.
 *
 * @param instances the set of instances to be added
 */
public void add(InstancesConsolidated instances) {
   for(int i = 0; i < instances.numInstances(); i++)
      add(instances.instance(i));
}

/**
 * Prints information about the size of the classes and their proportions
 * and indicates which is the minority class of the sample
 *
 * @param dataSize the size of original sample
 * @param iMinClass the index of the minority class in the original sample
 * @param classSizeVector the vector with the size of each class of the dataset
 */
public void printClassesInformation(int dataSize, int iMinClass, int[] classSizeVector){
   int numClasses = numClasses();
   System.out.println("Minority class value (" + iMinClass +
         "): " + classAttribute().value(iMinClass));
   System.out.println("Classes sizes:");
   for (int iClass = 0; iClass < numClasses; iClass++){
      /** Distribution of the 'iClass'-th class in the original sample */
      float distrClass;
      if (dataSize == 0)
         distrClass = (float)0;
      else
```

```
            distrClass = (float)100 * classSizeVector[iClass] / dataSize;
         System.out.print(classSizeVector[iClass] + " (" + Utils.doubleToString(distrClass,2) + "%)");
         if(iClass < numClasses - 1)
            System.out.print(", ");
      }
      System.out.println("");
   }
}
```