Universidad del País Vasco
Euskal Herriko Unibertsitatea

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

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

KONPUTAGAILUEN ARKITEKTURA ETA TEKNOLOGIA SAILA
DEPARTAMENTO DE ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

J48Consolidated: An implementation of CTC algorithm for WEKA

EHU-KAT-IK-05-13

Informe de Investigación

Olatz Arbelaitz
Ibai Gurrutxaga
Fernando Lozano
Javier Muguerza
Jesús Mª Pérez

Marzo 2013

J48Consolidated: An implementation of CTC algorithm for WEKA[¥]

Olatz Arbelaitz, Ibai Gurrutxaga, Fernando Lozano, Javier Muguerza, Jesús Mª Pérez[*]

Dept. of Computer Architecture and Technology, University of the Basque Country (UPV/EHU)
M. Lardizabal, 1, 20018 Donostia, Spain
{olatz.arbelaitz, i.gurrutxaga}@ehu.es, flozano002@ikasle.ehu.es, {j.muguerza, txus.perez}@ehu.es
http://www.sc.ehu.es/aldapa

Index

Table of figures

CTC algorithm

The CTC algorithm, Consolidated Tree Construction algorithm, is a machine learning paradigm that was designed to solve a class imbalance problem, a fraud detection problem in the area of car insurance [1] where, besides, an explanation about the classification made was required. The algorithm is based on a decision tree construction algorithm, in this case the well-known C4.5 [2], but it extracts knowledge from data using a set of samples instead of a single one as C4.5 does. In contrast to other methodologies based on several samples to build a classifier, such as bagging, the CTC builds a single tree and as a consequence, it obtains comprehensible classifiers .

The CTC algorithm has been used in different contexts. A good description of this algorithm and the results obtained for a set of different datasets from the UCI repository [3] can be found in this paper [4] (if you want to refer to CTC in a publication, see Citation policy section).

¥ If you want to refer to CTC in a publication, see Citation policy section.
* Corresponding author. E-mail address: txus.perez@ehu.es

The motivation

The main motivation of this implementation is to make public and available an implementation of the CTC algorithm. With this purpose we have implemented the algorithm within the well-known WEKA data mining environment [5] (http://www.cs.waikato.ac.nz/ml/weka/). WEKA is an open source project that contains a collection of machine learning algorithms written in Java for data mining tasks.

J48 is the implementation of C4.5 algorithm within the WEKA package. We called J48Consolidated to the implementation of CTC algorithm based on the J48 Java class.

J48Consolidated class

The implementation of J48Consolidated consists on the J48Consolidated class and the J48Consolidated package (the same way J48 consist on J48 class and J48 package), both included in weka.classifiers.trees package.

The J48Consolidated class extends the J48 class and, basically, redefines the buildClassifier() method to build a consolidated tree. With this aim it generates a set of samples and, in order to carry out the consolidation process, calls a method of a new class, called C45ConsolidatedPruneableClassifierTree, which is an extension of the original class C45PruneableClassifierTree. The J48Consolidated package contains the definition of the next five new classes, each of them extends its corresponding J48's original class:

- C45ConsolidatedPruneableClassifierTree
- C45ConsolidatedModelSelection
- C45ConsolidatedSplit
- DistributionConsolidated
- InstancesConsolidated

Actually, the implementation of the consolidation process of a decision tree is carried out only in the first three classes and, so, the methods included in the other two classes, DistributionConsolidated and InstancesConsolidated, could be a part of the class they extend, Distribution and Instances, respectively.

The whole source code of these classes can be found in Appendix 1 (see Downloading section to download).

The implementation of J48Consolidated class follows the guidelines of section 17.1 “Writing a new Classifier” of the document “WEKA Manual for Version 3-6-8” [6] (which comes with the latest stable version of Weka, available in http://www.cs.waikato.ac.nz/ml/weka/downloading.html) and of the HOWTO article “Writing your own Classifier” of the Weka Wiki (http://weka.wikispaces.com). The four tests proposed to ensure correct basic functionality of the classifier returned a positive answer.

Using J48Consolidated

The implementation of the J48Consolidated class adds a set of options to J48. These options set the Resampling Method (RM) for the generation of samples to use in the consolidation process:

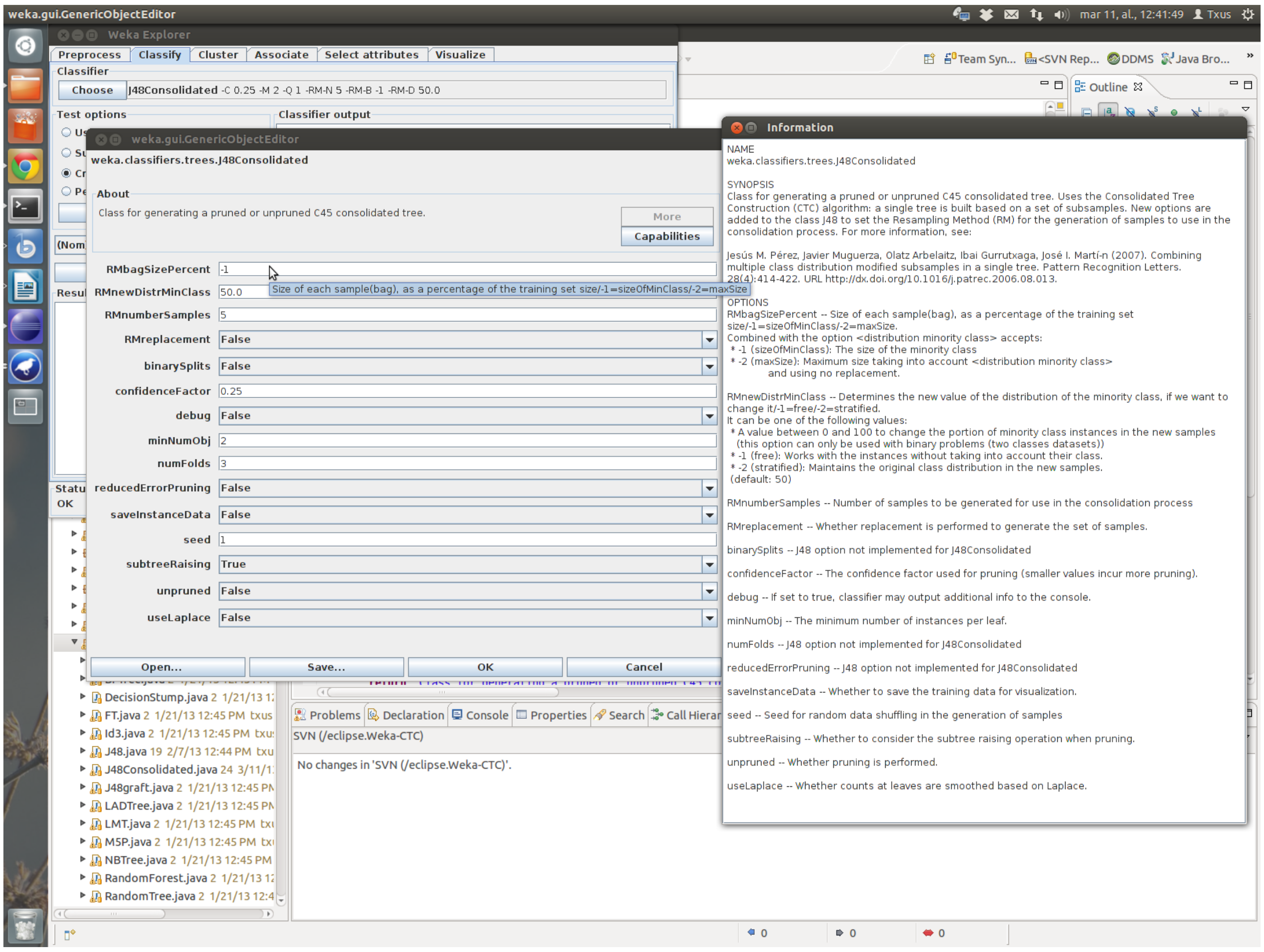
- *Number of samples*: This option indicates the number of samples to be generated for use in the construction of the consolidated tree. The default value is 5.
- *Replacement*: It is a boolean option which determines whether replacement is used or not for

the generation of samples. Its default value is false.

- *Size of samples*: This option makes an integer value between 0 and 100 indicating the size of each sample (bag) as a percentage of the training set size. It can also take two special values:
    - -1 (sizeOfMinClass): The size of the samples will be equal to the size of the minority class.
    - -2 (maxSize): The maximum size to generate the samples taking into account the distribution of the minority class to be achieved without replacement.
- *Distribution of the minority class*: This option can take a real value between 0 and 100 which determines the new value of the distribution of the minority class in the samples to be generated (this possibility is not valid for multi-classes datasets). It can also take two special values (for any dataset):
    - -1 (free): The instances will be selected randomly without taking into account their classes.
    - -2 (stratified): The instances will be selected randomly but taking into account their classes and, therefore, the original class distribution will be maintained.

Graphical interface

The figure below shows the options of this classifier as they are shown in the Weka Explorer, as well as the information window related to them. Only the first four options are related to the resampling method used to generate the samples (we added the prefix RM to the option names so that they appear together in the graphical interface) and the rest of the options are derived from the J48 class.



Classifier: J48Consolidated -C 0.25 -M 2 -Q 1 -RM-N 5 -RM-B -1 -RM-D 50.0

weka.classifiers.trees.J48Consolidated

About: Class for generating a pruned or unpruned C45 consolidated tree.

RMbagSizePercent	-1
RMnewDistrMinClass	50.0
RMnumberSamples	5
RMreplacement	False
binarySplits	False
confidenceFactor	0.25
debug	False
minNumObj	2
numFolds	3
reducedErrorPruning	False
saveInstanceData	False
seed	1
subtreeRaising	True
unpruned	False
useLaplace	False

Size of each sample(bag), as a percentage of the training set size/-1=sizeOfMinClass/-2=maxSize

DecisionStump.java 2 1/21/13
FT.java 2 1/21/13 12:45 PM txus
Id3.java 2 1/21/13 12:45 PM txu
J48.java 19 2/7/13 12:44 PM txu
J48Consolidated.java 24 3/11/1
J48graft.java 2 1/21/13 12:45 PM
LADTree.java 2 1/21/13 12:45 PM
LMT.java 2 1/21/13 12:45 PM txu
MSP.java 2 1/21/13 12:45 PM txu
NBTree.java 2 1/21/13 12:45 PM
RandomForest.java 2 1/21/13
RandomTree.java 2 1/21/13 12:4

SVN (/eclipse.Weka-CTC)

No changes in 'SVN (/eclipse.Weka-CTC)'.

Information

```
NAME
weka.classifiers.trees.J48Consolidated

SYNOPSIS
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 class J48 to set the Resampling Method (RM) for the generation of samples to use in the
consolidation process. For more information, see:

Jesús M. Pérez, Javier Muguerza, Olatz Arbelaitz, Ibai Gurrutxaga, José I. Martí-n (2007). Combining
multiple class distribution modified subsamples in a single tree. Pattern Recognition Letters.
28(4):414-422. URL http://dx.doi.org/10.1016/j.patrec.2006.08.013.

OPTIONS
RMbagSizePercent -- Size of each sample(bag), as a percentage of the training set
size/-1=sizeOfMinClass/-2=maxSize.
Combined with the option <distribution minority class> accepts:
 * -1 (sizeOfMinClass): The size of the minority class
 * -2 (maxSize): Maximum size taking into account <distribution minority class>
          and using no replacement.

RMnewDistrMinClass -- Determines the new value of the distribution of the minority class, if we want to
change it/-1=free/-2=stratified.
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 classes datasets))
 * -1 (free): Works with the instances without taking into account their class.
 * -2 (stratified): Maintains the original class distribution in the new samples.
 (default: 50)

RMnumberSamples -- Number of samples to be generated for use in the consolidation process

RMreplacement -- Whether replacement is performed to generate the set of samples.

binarySplits -- J48 option not implemented for J48Consolidated

confidenceFactor -- The confidence factor used for pruning (smaller values incur more pruning).

debug -- If set to true, classifier may output additional info to the console.

minNumObj -- The minimum number of instances per leaf.

numFolds -- J48 option not implemented for J48Consolidated

reducedErrorPruning -- J48 option not implemented for J48Consolidated

saveInstanceData -- Whether to save the training data for visualization.

seed -- Seed for random data shuffling in the generation of samples

subtreeRaising -- Whether to consider the subtree raising operation when pruning.

unpruned -- Whether pruning is performed.

useLaplace -- Whether counts at leaves are smoothed based on Laplace.
```

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

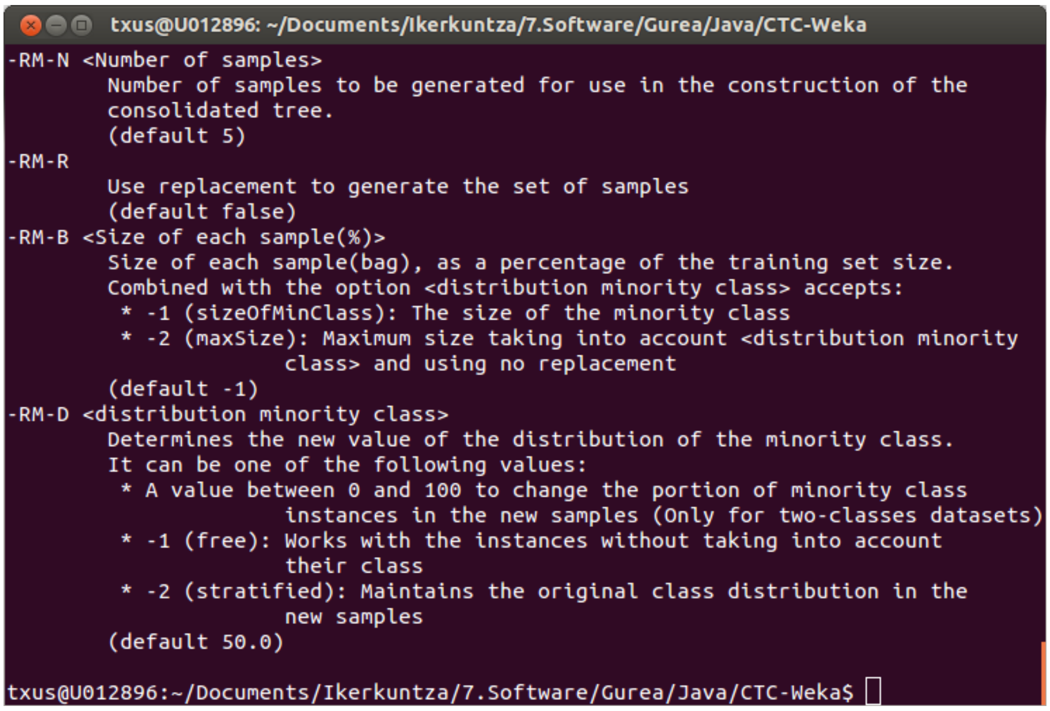
Because the CTC algorithm is based on Quinlan's C4.5 algorithm, two of the options implemented in J48 were not implemented for J48Consolidated (the default value is false in J48 for both):

- binarySplits: Whether to use binary splits on nominal attributes when building the trees.
- reducedErrorPruning: Whether reduced-error pruning is used instead of C4.5 pruning.

Besides, the numFolds and seed options are related to the reducedErrorPruning option, so they are not implemented. However, seed has been used for random data shuffling in the generation of samples in J48Consolidated.

Command-line

Another option is to use J48Consolidated class from the command-line. The picture below shows the options related to the resampling method and how to use them.



```
txus@U012896: ~/Documents/Ikerkuntza/7.Software/Gurea/Java/CTC-Weka
-RM-N <Number of samples>
        Number of samples to be generated for use in the construction of the
        consolidated tree.
        (default 5)
-RM-R
        Use replacement to generate the set of samples
        (default false)
-RM-B <Size of each sample(%)>
        Size of each sample(bag), as a percentage of the training set size.
        Combined with the option <distribution minority class> accepts:
         * -1 (sizeOfMinClass): The size of the minority class
         * -2 (maxSize): Maximum size taking into account <distribution minority
                      class> and using no replacement
        (default -1)
-RM-D <distribution minority class>
        Determines the new value of the distribution of the minority class.
        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 (Only for two-classes datasets)
         * -1 (free): Works with the instances without taking into account
                      their class
         * -2 (stratified): Maintains the original class distribution in the
                      new samples
        (default 50.0)

txus@U012896:~/Documents/Ikerkuntza/7.Software/Gurea/Java/CTC-Weka$
```

Figure 2: J48Consolidated options through the command-line

The command to get the whole help information of J48Consolidated classifier (partially seen in the Figure 2) is below:

```
java -cp Weka-CTC.jar weka.classifiers.trees.J48Consolidated -h
```

Case studies

In order to be able to use J48Consolidated classifiers the same way we used them our previous works, we have implemented all the different possibilities to set the resampling method that were used in the consolidation process of the decision trees. We can distinguish three types of sampling in these works:

- *Samples with modified class distribution whose size is the size of the minority class in the training sample*

  This kind of samples were used by Weiss and Provost in [7]. This size guarantees that the proportion of the minority class may vary from 0% to 100% without using replacement. This can only be used for binary problems (two classes datasets).

This is the type of samples used in J48Consolidated by default.

We used this type of samples in works such as [1], [8] and [9].

The values for the options should be as follows:

`numberSamples (-RM-N)`	<Indifferent>
`replacement (-RM-R)`	false
`bagSizePercent (-RM-B)`	-1
`newDistrMinClass (-RM-D)`	0 .. 100

- *Samples with modified class distribution whose size is the biggest that can be obtained for the new distribution given and without using replacement*

  This type of samples can only be used for binary problems (two classes datasets).

  We used this type of samples in works such as [4].

  The values for the options should be as follows:

`numberSamples (-RM-N)`	<Indifferent>
`replacement (-RM-R)`	false
`bagSizePercent (-RM-B)`	-2
`newDistrMinClass (-RM-D)`	0 .. 100

- *Bootstrap samples (such as those used by bagging) and stratified samples whose size is set as a percentage of the training sample size and without using replacement*

  We used this type of samples in works such as [10], [11] and [12].

  The values for the options to generate bootstrap samples should be as follows:

`numberSamples (-RM-N)`	<Indifferent>
`replacement (-RM-R)`	true
`bagSizePercent (-RM-B)`	100
`newDistrMinClass (-RM-D)`	-1

And the values for the options to generate stratified samples should be as follows:

`numberSamples (-RM-N)`	<Indifferent>
`replacement (-RM-R)`	false
`bagSizePercent (-RM-B)`	0 .. 100
`newDistrMinClass (-RM-D)`	-2

Downloading

- *Source code*:

  The source code of the six classes that implement the J48Consolidated classifier can be found in (also included in Appendix 1 in this document):

  http://www.sc.ehu.es/aldapa/weka-ctc/J48Consolidated.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

- *Executable file*:

The executable file in Weka is a .jar file. The file with the current J48Consolidated implementation can be found in:

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

To run Weka type:

```
java -Xmx1000M -jar Weka-CTC.jar
```

(see http://www.cs.waikato.ac.nz/ml/weka/downloading.html for more information)

Citation policy

If you want to refer to CTC in a publication, please cite the following paper:

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).

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, yet implemented in Weka, could be used for the generation of samples, as for example SMOTE method.
- Generalize the process to change the class distribution to multi-class datasets.
- 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] 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

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

[3] A. Frank, A. Asuncion. *"UCI Machine Learning Repository"* [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science, (2010).

[4] 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

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

[6] R.R. Bouckaert , E. Frank , M. Hall , R. Kirkby , P. Reutemann , A. Seewald , D. Scuse . *"WEKA Manual for Version 3-6-8"* [http://www.cs.waikato.ac.nz/ml/weka/downloading.html]. The university of Waikato, New Zealand. August (2012).

[7] 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).

[8] 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

[9] 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).

[10] J.M. Pérez, J. Muguerza, O. Arbelaitz, I. Gurrutxaga. *"A New Algorithm to Build Consolidated Trees: Study of the Error Rate and Steadiness"*. Advances in Soft Computing (Springer-Verlag). Proceedings of the International Intelligent Information Processing and Web Mining Conference (IIS: IIPWM´04), Zakopane, Poland, 79-88 (2004).

[11] I. Gurrutxaga, J.M. Pérez, O. Arbelaitz, J. Muguerza, J.I. Martín, A. Ansuategi. *"CTC: An Alternative to Extract Explanation from Bagging"*. Lecture Notes in Computer Science 4788, Current Topics in Artificial Intelligence. Springer-Verlag. D. Borrajo, L. Castillo and J.M. Corchado (Eds.), 90-99 (2007). doi: 10.1007/978-3-540-75271-4_10

[12] J.M. Pérez, I. Albisua, O. Arbelaitz, I. Gurrutxaga, J.I. Martín, J. Muguerza, I. Perona. *"Consolidated trees versus bagging when explanation is required"*. Computing, Vol. 89, 113-145 (2010). doi:10.1007/s00607-010-0094-z

Appendix 1: Source code

This appendix contains the whole source code that implements the J48Consolidated class.

J48Consolidated.java

```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) 2013 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.Summarizable;
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 class J48 to
set the Resampling Method (RM) for the generation of samples to use 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/>
```

```
<!-- 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/>
<!-- technical-bibtex-end -->
 * **************************************************************************************<br/>
 * Attention! The visibility of the following members of the class 'J48' changed
 *     to 'protected' instead of 'private' in order to use them here:
 * <ul>
 *       <li>protected ClassifierTree m_root;</li>
 *       <li>protected boolean m_unpruned = false;</li>
 *       <li>protected float m_CF = 0.25f;</li>
 *       <li>protected int m_minNumObj = 2;</li>
 *       <li>protected boolean m_useLaplace = false;</li>
 *       <li>protected boolean m_reducedErrorPruning = false;</li>
 *       <li>protected int m_numFolds = 3;</li>
 *       <li>protected boolean m_binarySplits = false;</li>
 *       <li>protected boolean m_subtreeRaising = true;</li>
 *       <li>protected boolean m_noCleanup = false;</li>
 *       <li>protected int m_Seed = 1;</li>
 * </ul>
 * Attention! Exceptions added to the following methods of the class 'J48'
 *     in order to use them here:
 * <ul>
 *       <li>public void setReducedErrorPruning(boolean v) throws Exception {}</li>
 *       <li>public void setNumFolds(int v) throws Exception {}</li>
 *       <li>public void setBinarySplits(boolean v) throws Exception {}</li>
 * </ul>
<!-- 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> -N &lt;number of samples&gt;
 *  Number of samples to be generated for use in the construction of the consolidated tree.
 *  (default 5)</pre>
 *
 * <pre> -R
 *  Determines whether or not replacement is used when generating the samples.
 *  (default false)</pre>
 *
 * <pre> -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 into account &lt;distribution minority class&gt;
 *  *           and using no replacement
 *  (default -1)</pre>
 *
 * <pre> -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 classes datasets))
 *  * -1 (free): Works with the instances without taking into account their class
 *  * -2 (stratified): Maintains the original class distribution in the new samples
 *  (default 50.0)
 *
<!-- options-end -->
 *
 * @author Jesús M. Pérez (txus.perez@ehu.es)
 * @author Fernando Lozano (flozano002@ikasle.ehu.es)
 *  (based on J48.java written by Eibe Frank)
 * @version $Revision: 2.0 $
 */
public class J48Consolidated
   extends J48
   implements OptionHandler, Drawable, Matchable, Sourcable,
            WeightedInstancesHandler, Summarizable, AdditionalMeasureProducer,
            TechnicalInformationHandler {

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

   /** 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)
    ********************************************************************************/
   /** Number of samples to be generated for use in the construction of the consolidated tree.**/
   private int m_RMnumberSamples = 5;

   /** 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 into account &lt;distribution minority class&gt;
    *  *           and using no replacement */
   private int m_RMbagSizePercent = -1; // default: sizeOfMinClass

   /** Value of the distribution of the minority class to be changed.
    *  It also accepts:
    *  * -1 (free): Works with the instances without taking into account their class
    *  * -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 class J48 to set the Resampling Method (RM) for "
           + "the generation of samples to use in the consolidation process. "
           + "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");

    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).getCapabilities();
    }
    catch (Exception e) {
      result = new Capabilities(this);
    }

    result.setOwner(this);

    return result;
  }

  /**
   * Generates the classifier.
   * (Implements the original CTC algorithm, then
   *  does not implement the options binarySplits and reducedErrorPruning of J48,
   *  so 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);
    // TODO Implement the option reducedErrorPruning of J48
    m_root = new C45ConsolidatedPruneableClassifierTree(modSelection, !m_unpruned,
          m_CF, m_subtreeRaising, !m_noCleanup);

    // remove instances with missing class before generate 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 samples set
  * @return Instances[] the vector of generated samples
  * @throws Exception if something goes wrong
  */
 private 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 -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 + "% of " + dataSize
             + ") has to be greater than 0!");
    }

    Random random;
    if (dataSize == 0) // To be OK the test 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: 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);
    return samplesVector;
 }

 /**
  * Generate a set of stratified samples
  *
  * @param instances the training data which will be used to generate the samples 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{
    Instances[] samplesVector = new Instances[m_RMnumberSamples];
    int numClasses = instances.numClasses();
    /** Partial bag size */
    int localBagSize = 0;
    // Get the classes
    InstancesConsolidated[] classesVector =  instances.getClasses();
    // What is the minority class?
    /** Index of the minority class in the original sample */
    int iMinClass;
    /** Vector containing the size of each class */
    int classSizeVector[] = new int [numClasses];
    for (int iClass = 0; iClass < numClasses; iClass++)
       classSizeVector[iClass] = classesVector[iClass].numInstances();
    iMinClass = Utils.minIndex(classSizeVector);
    if (m_Debug){
       System.out.println("Minority class value (" + iMinClass +
             "): " + instances.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] + " (" + distrClass + "%)");
          if(iClass < numClasses - 1)
             System.out.print(", ");
       }
       System.out.println("");
    }
    // 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];
    for(int iClass = 0; iClass < numClasses; iClass++)
       if(iClass != iMinClass){
          /** Value for the 'iClass'-th class size of the samples to have to be generated */
          int newClassSize = classSizeVector[iClass] * m_RMbagSizePercent / 100;
          newClassSizeVector[iClass] = newClassSize;
          localBagSize += newClassSize;
       }
    /** Value for the minority class size of the samples to have to be generated */
    // (Done in this way to know the exact size to be generated of the minority class)
    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("");
    }

    // Generate the vector of samples
    for(int iSample = 0; iSample < m_RMnumberSamples; 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;
         bagClass = null;
      }
      return samplesVector;
   }

   /**
    * Generate a set of samples without taking into account the class distribution
    * (like in the meta-classifier Bagging)
    *
    * @param instances the training data which will be used to generate the samples 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{
      Instances[] samplesVector = new Instances[m_RMnumberSamples];
      if (m_Debug)
         System.out.println("New bag size: " + bagSize);
      for(int iSample = 0; iSample < m_RMnumberSamples; 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;
      }
      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 samples 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{
      Instances[] samplesVector = new Instances[m_RMnumberSamples];
      // Some checks
      if(instances.classAttribute().numValues() != 2)
         throw new Exception("Only binary problems (two classes datasets) can be used to change the 
distribution of classes!!!\n" +
                     "Use 'free' or 'stratified' values in <distribution minority class> for multi-class 
datasets!!!");
         //throw new Exception("Multi-class datasets aren't contempled to change the distribution of 
classes!");
      // TODO Generalize the process to multi-class datasets
      // Some checks done in set-methods
      //@ requires m_RMreplacement = false
      // TODO Accept replacement

      // Get the two classes
      InstancesConsolidated[] classesVector = instances.getClasses();
      // What is the minority class?
      int minClassSize, majClassSize; /** Sizes of the minority and majority classes */
      int iMinClass, iMajClass; /** Index of the minority and majority classes in the original sample */
      if(classesVector[0].numInstances() <= classesVector[1].numInstances())
         // The minority class is the first
         iMinClass = 0;
      else
         // The minority class is the second
         iMinClass = 1;
      iMajClass = 1 - iMinClass;
      minClassSize = classesVector[iMinClass].numInstances();
      majClassSize = classesVector[iMajClass].numInstances();

      /** Distribution of the minority class in the original sample */
      float distrMinClass;
      if (dataSize == 0)
         distrMinClass = (float)0;
      else
         distrMinClass = (float)100 * minClassSize / dataSize;
      if (m_Debug) {
         System.out.println("Minority class value (" + iMinClass +
               "): " + instances.classAttribute().value(iMinClass));
         System.out.println("Minority class size: " + minClassSize + " (" + distrMinClass + "%)");
         System.out.println("Majority class size: " + majClassSize);
      }
      /** Maximum values for minority and majority classes' size taking into account RMnewDistrMinClass
       *   and using not replacement */
      int maxMinClassSize, maxMajClassSize;
      if(m_RMnewDistrMinClass > distrMinClass){
         // Maintains the whole minority class
         maxMinClassSize = minClassSize;
         maxMajClassSize = (int) (minClassSize * (100 - m_RMnewDistrMinClass) / m_RMnewDistrMinClass);
      } else {
         // Maintains the whole mayority class
         maxMajClassSize = majClassSize;
         maxMinClassSize = (int) (majClassSize * m_RMnewDistrMinClass / (100 - m_RMnewDistrMinClass));
      }
      /** Values for minority and majority classes' sizes of the samples to have to be generated */
      int newMinClassSize, newMajClassSize;
      if(m_RMbagSizePercent == -2){
         // maxSize : Generate the biggest samples according the indicated distribution (RMnewDistrMinClass)
         if(m_RMnewDistrMinClass == distrMinClass)
            System.err.println("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)!!!");
         newMinClassSize = maxMinClassSize;
         newMajClassSize = maxMajClassSize;
         bagSize = maxMinClassSize + maxMajClassSize;
      } else {
         if (m_RMbagSizePercent == -1) {
            // sizeOfMinClass: the samples to be generated will have the same size that the minority class
            bagSize = minClassSize;
            newMinClassSize = (int) (m_RMnewDistrMinClass * bagSize / 100);
         } else {
            // m_RMbagSizePercent is between 0 and 100. bagSize is already set.
            newMinClassSize = (int) (m_RMnewDistrMinClass * bagSize / 100);
         }
         newMajClassSize = bagSize - newMinClassSize;
      }
      if (m_Debug) {
         System.out.println("New bag size: " + bagSize);
         System.out.println("New minority class size: " + newMinClassSize + " (" + (int)(newMinClassSize /
(double)bagSize * 100) + "%)");
         System.out.println("New majority class size: " + newMajClassSize);
      }
      // Some checks
      if(newMinClassSize > minClassSize)
         throw new Exception("There isn't enough instances of the minority class (" +
               minClassSize + ") to extract " + newMinClassSize + " for the new samples " +
               "whithout replacement!!!");
```

```
   if(newMajClassSize > majClassSize)
      throw new Exception("There isn't enough instances of the majority class (" +
               majClassSize + ") to extract " + newMajClassSize + " for the new samples " +
               "whithout replacement!!!");
   // Generate the vector of samples
   for(int iSample = 0; iSample < m_RMnumberSamples; iSample++){
      InstancesConsolidated bagData = null;
      InstancesConsolidated bagClass = null;
      // Extract instances of the minority class
      bagClass = new InstancesConsolidated(classesVector[iMinClass]);
      // Shuffle the instances
      bagClass.randomize(random);
      if (newMinClassSize < minClassSize) {
         InstancesConsolidated newBagData = new InstancesConsolidated(bagClass, 0, newMinClassSize);
          bagClass = newBagData;
          newBagData = null;
      }
      // Save the minority class to bagData
      bagData = bagClass;
      bagClass = null;
      // Extract instances of the majority class
      bagClass = new InstancesConsolidated(classesVector[iMajClass]);
      // Shuffle the instances
      bagClass.randomize(random);
      if (newMajClassSize < majClassSize) {
         InstancesConsolidated newBagData = new InstancesConsolidated(bagClass, 0, newMajClassSize);
          bagClass = newBagData;
          newBagData = null;
      }
      // Add the second bagClass (majority class) to bagData
      bagData.add(bagClass);
      // Shuffle the instances
      bagData.randomize(random);
      samplesVector[iSample] = (Instances)bagData;
      bagData = null;
      bagClass = null;
   }
   return samplesVector;
}

/**
 * Print the generated samples. Only for test purposes.
 *
 * @param samplesVector the vector of samples
 */
private 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: <p>
 *
 * J48 options
 * =============
 *
 * -U <br>
 * Use unpruned tree.<p>
 *
 * -C confidence <br>
 * Set confidence threshold for pruning. (Default: 0.25) <p>
 *
 * -M number <br>
 * Set minimum number of instances per leaf. (Default: 2) <p>
 *
 * -S <br>
 * Don't perform subtree raising. <p>
 *
 * -L <br>
```

```java
 * Do not clean up after the tree has been built. <p>
 *
 * -A <br>
 * If set, Laplace smoothing is used for predicted probabilites. <p>
 *
 * -Q seed <br>
 * Seed for random data shuffling (Default: 1) <p>
 *
 * Options to set the Resampling Method (RM) for the generation of samples
 *  to use in the consolidation process
 * ============================================================================
 * -N number <br>
 * Number of samples to be generated for use in the construction of the consolidated tree. <br>
 * (Default: 5) <p>
 *
 * -R <br>
 * Determines whether or not replacement is used when generating the samples. <br>
 * (Default: true)<p>
 *
 * -B percent <br>
 * Size of each sample(bag), as a percentage of the training set size. <br>
 * Combined with the option &lt;distribution minority class&gt; accepts: <br>
 *  * -1 (sizeOfMinClass): The size of the minority class <br>
 *  * -2 (maxSize): Maximum size taking into account &lt;distribution minority class&gt;
 *             and using no replacement <br>
 * (Default: -1(sizeOfMinClass)) <p>
 *
 * -D distribution minority class <br>
 * Determines the new value of the distribution of the minority class, if we want to change it. <br>
 * 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 <br>
 *    (this option can only be used with binary problems (two classes datasets)) <br>
 *  * -1 (free): Works with the instances without taking into account their class <br>
 *  * -2 (stratified): Maintains the original class distribution in the new samples <br>
 * (Default: -1(free)) <p>
 *
 * @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("\tNumber of samples to be generated for use in the construction of the\n" +
         "\tconsolidated tree.\n" +
         "\t(default 5)",
         "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 into account <distribution minority\n" +
         "\t              class> and using no replacement\n" +
         "\t(default -1)",
         "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 (Only for two-classes datasets)\n" +
         "\t * -1 (free): Works with the instances without taking into account\n" +
         "\t              their class\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> -N &lt;Number of samples&gt;
 *  Number of samples to be generated for use in the construction of the consolidated tree.
 *  (default 5)</pre>
 *
 * <pre> -R
 *  Determines whether or not replacement is used when generating the samples.
 *  (default true)</pre>
 *
 * <pre> -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 into account &lt;distribution minority class&gt;
 *  *           and using no replacement
 *  (default -1(sizeOfMinClass))</pre>
 *
 * <pre> -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 classes datasets))
 *  * -1 (free): Works with the instances without taking into account their class
 *  * -2 (stratified): Maintains the original class distribution in the new samples
 *  (default -1(free))
 *
<!-- 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
   // =========================================================================
   String RMnumberSamplesString = Utils.getOption("RM-N", options);
   if (RMnumberSamplesString.length() != 0) {
      setRMnumberSamples(Integer.parseInt(RMnumberSamplesString));
   } else {
      setRMnumberSamples(5);
   }
   String RMbagSizePercentString = Utils.getOption("RM-B", options);
   if (RMbagSizePercentString.length() != 0) {
      setRMbagSizePercent(Integer.parseInt(RMbagSizePercentString), false);
   } else {
      setRMbagSizePercent(-1, false); // default: sizeOfMinClass
   }
   String RMnewDistrMinClassString = Utils.getOption("RM-D", options);
   if (RMnewDistrMinClassString.length() != 0) {
      setRMnewDistrMinClass(new Float(RMnewDistrMinClassString).floatValue(), false);
   } else {
      setRMnewDistrMinClass((float)50, 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 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
   // =========================================================================
   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() +
//               "-------------------------------\n" +
            m_root.toString();
   else
      return "J48Consolidated pruned tree\n" +
             toStringResamplingMethod() +
//               "-----------------------------\n" +
            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=" + m_RMnumberSamples;
   if (m_RMnewDistrMinClass == -2)
      st += " stratified";
   else if (m_RMnewDistrMinClass == -1)
      st += " free distribution";
   else
      st += " %Min=" + 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";
   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 RMnumberSamplesTipText() {
   return "Number of samples to be generated for use in the consolidation process";
}

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

   return m_RMnumberSamples;
}

/**
 * Set the value of RMnumberSamples.
 *
 * @param v  Value to assign to RMnumberSamples.
 */
public void setRMnumberSamples(int v) {
   // Doesn't make sense to build a consolidated tree with 1 or 2 samples, but it's possible!
   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> accepts:\n" +
            " * -1 (sizeOfMinClass): The size of the minority class\n" +
            " * -2 (maxSize): Maximum size taking into account <distribution minority class>\n" +
            "             and using no replacement.";
   }

   /**
    * 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;
      }
   }

   /**
    * 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 contempled 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("Doesn't make sense that size of sample (%) is 100, when replacement is
false!");
      }
   }

   /**
    * 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" +
            "   (this option can only be used with binary problems (two classes datasets))\n" +
            " * -1 (free): Works with the instances without taking into account their class.\n" +
            " * -2 (stratified): Maintains the original class distribution in the new samples.\n" +
            " (default: 50)";
   }

   /**
    * 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("Distribution minority class 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;
      }
   }

   /**
    * 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.
    * @throws Exception if an option is not supported
    */
   public void setReducedErrorPruning(boolean v) throws Exception {

      m_reducedErrorPruning = false;
      throw new Exception("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.
    * @throws Exception if an option is not supported
    */
   public void setNumFolds(int v) throws Exception {

      m_numFolds = 3;
      throw new Exception("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.
    * @throws Exception if an option is not supported
    */
   public void setBinarySplits(boolean v) throws Exception {

      m_binarySplits = false;
      throw new Exception("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";
   }

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

C45ConsolidatedPruneableClassifierTree.java

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

import weka.classifiers.trees.j48.C45PruneableClassifierTree;
import weka.classifiers.trees.j48.ClassifierTree;
import weka.classifiers.trees.j48.ModelSelection;
import weka.core.FastVector;
import weka.core.Instances;
import weka.core.Utils;

/**
 * Class for handling a consolidated tree structure that can
 * be pruned using C4.5 procedures.
 * *********************************************************************************
 * <p/>*** Attention! The visibility of the following members of the class 'C45PruneableClassifierTree'
 *     changed to 'protected' instead of 'private' in order to use them here:
 * <ul>
 *    <li>protected boolean m_pruneTheTree = false;</li>
 *    <li>protected float m_CF = 0.25f;</li>
 *    <li>protected boolean m_subtreeRaising = true;</li>
 *    <li>protected boolean m_cleanup = true;</li>
 * </ul>
 *
 * @author Jesús M. Pérez (txus.perez@ehu.es)
 * @version $Revision: 1.0 $
 */
public class C45ConsolidatedPruneableClassifierTree extends
      C45PruneableClassifierTree {

   /** for serialization */
   private static final long serialVersionUID = 2660972525647728377L;

   /**
    * Constructor for pruneable consolidated tree structure. Calls
    * the superclass constructor.
    *
    * @param toSelectLocModel selection method for local splitting model
    * @param pruneTree true if the tree is to be pruned
    * @param cf the confidence factor for pruning
    * @param raiseTree true if subtree raising has to be performed
    * @param cleanup true if cleanup has to be done
    * @throws Exception if something goes wrong
    */
   public C45ConsolidatedPruneableClassifierTree(
         ModelSelection toSelectLocModel, boolean pruneTree, float cf,
         boolean raiseTree, boolean cleanup) throws Exception {
      super(toSelectLocModel, pruneTree, cf, raiseTree, cleanup);
   }

   /**
    * Method for building a pruneable classifier consolidated tree.
    *
    * @param data the data for pruning the consolidated tree
    * @param samplesVector the vector of samples for building the consolidated tree
    * @throws Exception if something goes wrong
    */
   public void buildClassifier(Instances data, Instances[] samplesVector) throws Exception {

      // can classifier tree handle the data?
      getCapabilities().testWithFail(data);

      // remove instances with missing class
      // Already removed for J48Consolidated!

      buildTree(data, samplesVector, m_subtreeRaising);
      collapse();
      if (m_pruneTheTree) {
         prune();
      }
      if (m_cleanup) {
         cleanup(new Instances(data, 0));
      }
```

```java
	}

	/**
	 * Returns a newly created tree.
	 *
	 * @param data the data to work with
	 * @param samplesVector the vector of samples for building the consolidated tree
	 * @return the new consolidated tree
	 * @throws Exception if something goes wrong
	 */
	protected ClassifierTree getNewTree(Instances data, Instances[] samplesVector) throws Exception {

		C45ConsolidatedPruneableClassifierTree newTree =
				new C45ConsolidatedPruneableClassifierTree(m_toSelectModel, m_pruneTheTree, m_CF,
						m_subtreeRaising, m_cleanup);
		newTree.buildTree(data, samplesVector, m_subtreeRaising);

		return newTree;
	}

	/**
	 * Builds the consolidated tree structure.
	 * (based on the method buildTree() of the class 'ClassifierTree')
	 *
	 * @param data the data for pruning the consolidated tree
	 * @param samplesVector the vector of samples used for consolidation
	 * @param keepData is training data to be kept?
	 * @throws Exception if something goes wrong
	 */
	public void buildTree(Instances data, Instances[] samplesVector, boolean keepData) throws Exception {
		/** Number of Samples. */
		int numberSamples = samplesVector.length;
		if (keepData) {
			m_train = data;
		}
		m_test = null;
		m_isLeaf = false;
		m_isEmpty = false;
		m_sons = null;
		m_localModel = ((C45ConsolidatedModelSelection)m_toSelectModel).selectModel(data, samplesVector);
		if (m_localModel.numSubsets() > 1) {
			/** Vector storing the obtained subsamples after the split of data */
			Instances [] localInstances;
			/** Vector storing the obtained subsamples after the split of each sample of the vector */
			FastVector localInstancesVector = new FastVector(); // = Instances [][] localInstancesVector;

			localInstances = m_localModel.split(data);
			for (int iSamples = 0; iSamples < numberSamples; iSamples++)
				localInstancesVector.addElement(m_localModel.split(samplesVector[iSamples]));
			data = null;
			samplesVector = null;
			m_sons = new ClassifierTree [m_localModel.numSubsets()];
			for (int iSon = 0; iSon < m_sons.length; iSon++) {
				/** Vector storing the subsamples related to the iSon-th son */
				Instances[] localSamplesVector = new Instances[numberSamples];
				for (int iSamples = 0; iSamples < numberSamples; iSamples++)
					localSamplesVector[iSamples] =
							((Instances[]) localInstancesVector.elementAt(iSamples))[iSon];
				m_sons[iSon] = getNewTree(localInstances[iSon], localSamplesVector);
				localInstances[iSon] = null;
				localSamplesVector = null;
			}
			localInstances = null;
			localInstancesVector.removeAllElements();
		}else{
			m_isLeaf = true;
			if (Utils.eq(m_localModel.distribution().total(), 0))
				m_isEmpty = true;
			data = null;
			samplesVector = null;
		}
	}

}
```

C45ConsolidatedModelSelection.java

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


import weka.classifiers.trees.j48.*;
import weka.core.Instances;
import weka.core.Utils;
import weka.core.matrix.DoubleVector;

/**
 * Class for selecting a C4.5Consolidated-type split for a given dataset.
 * *************************************************************************************<br/>
 * <p/>*** Attention! Method 'splitPoint()' created in the class 'C45Split' of the package 'J48'
 *     to get the value of the member 'm_splitPoint' (similar to attIndex()).
 * <ul><li>public final double splitPoint(){ return m_splitPoint; }</li></ul>
 * <p/>*** Attention! The visibility of the following members of the class 'C45ModelSelection'
 *     changed to 'protected' instead of 'private'  in order to use them here:
 * <ul><li>protected int m_minNoObj;</li></ul>
 *
 * @author Jesús M. Pérez (txus.perez@ehu.es)
 * @version $Revision: 1.0 $
 */

public class C45ConsolidatedModelSelection extends C45ModelSelection {


   /** for serialization */
   private static final long serialVersionUID = 970984256023901098L;

   /** The model selection method to consolidate. */
   protected ModelSelection m_toSelectModelToConsolidate;

   /**
    * Initializes the split selection method with the given parameters.
    * At the moment, only accepted C45ModelSelection
    *
    * @param minNoObj minimum number of instances that have to occur in at least two
    * subsets induced by split
    * @param allData FULL training dataset (necessary for
    * selection of split points).
    */
   public C45ConsolidatedModelSelection(int minNoObj, Instances allData) {
      super(minNoObj, allData);

      m_toSelectModelToConsolidate = new C45ModelSelection(minNoObj, allData);
   }

   /**
    * Selects Consolidated-type split based on C4.5 for the given dataset.
    *
    * @param data the data to train the classifier with
    * @param samplesVector the vector of samples
    * @return the consolidated model to be used to split
    * @throws Exception  if something goes wrong
    */
   public ClassifierSplitModel selectModel(Instances data, Instances[] samplesVector) throws Exception{

      /** Number of Samples. */
      int numberSamples = samplesVector.length;
      /** Vector storing the chosen attribute to split in each sample */
      int[] attIndexVector = new int[numberSamples];
      /** Vector storing the split point to use to split, if numerical, in each sample */
      double[] splitPointVector = new double[numberSamples];

      // Select C4.5-type split for each sample and
      //  save the chosen attribute (and the split point if numerical) to split
      for (int iSample = 0; iSample < numberSamples; iSample++) {
         ClassifierSplitModel localModel = m_toSelectModelToConsolidate.selectModel(samplesVector[iSample]);
         if(localModel.numSubsets() > 1){
            attIndexVector[iSample] = ((C45Split) localModel).attIndex();
            splitPointVector[iSample] = ((C45Split) localModel).splitPoint();
         }else{
            attIndexVector[iSample] = -1;
            splitPointVector[iSample] = -1;
         }
      }
      // Get the most voted attribute (index)
      int votesCountByAtt[] = new int[data.numAttributes()];
      int numberVotes = 0;
      for (int iAtt = 0; iAtt < data.numAttributes(); iAtt++)
         votesCountByAtt[iAtt] = 0;
      for (int iSample = 0; iSample < numberSamples; iSample++)
         if(attIndexVector[iSample]!=-1){
            votesCountByAtt[attIndexVector[iSample]]++;
            numberVotes++;
         }
      int mostVotedAtt = Utils.maxIndex(votesCountByAtt);

      Distribution checkDistribution = new DistributionConsolidated(samplesVector);
      NoSplit noSplitModel = new NoSplit(checkDistribution);
      // if all nodes are leafs,
      if(numberVotes==0)
         //  return a consolidated leaf
         return noSplitModel;

      // Consolidate the split point (if numerical)
      double splitPointConsolidated = consolidateSplitPoint(
                                        mostVotedAtt, attIndexVector, splitPointVector, data);
      // Creates the consolidated model
      C45ConsolidatedSplit consolidatedModel =
            new C45ConsolidatedSplit(mostVotedAtt, m_minNoObj, checkDistribution.total(),
                  data, samplesVector, splitPointConsolidated);

//       // Set the split point analogue to C45 if attribute numeric.
//       // // It is not necessary for the consolidation process because the median value
//       // //  is already one of the proposed split points.
//       consolidatedModel.setSplitPoint(data);

      if(!consolidatedModel.checkModel())
         return noSplitModel;
      return consolidatedModel;
   }

   /**
    * Calculates the median of the split points related to 'mostVotedAtt' attribute, if this is numerical
    *  (MAX_VALUE otherwise).
    *
    * @param mostVotedAtt the most voted attribute (index)
    * @param attIndexVector Vector storing the chosen attribute to split in each sample
    * @param splitPointVector Vector storing the split point to use to split, if numerical, in each sample
    * @param data the training sample. Only to know if mostVotedAtt is numerical
    * @return the consolidated split point
    */
   private final double consolidateSplitPoint(int mostVotedAtt,
         int[] attIndexVector, double[] splitPointVector, Instances data){
      int numberSamples = attIndexVector.length;
      double consolidatedSplitPoint = Double.MAX_VALUE;
      if(data.attribute(mostVotedAtt).isNumeric()){
         DoubleVector splitPointChosenAttVector = new DoubleVector();
         for (int iSample = 0; iSample < numberSamples; iSample++)
            if(attIndexVector[iSample] == mostVotedAtt)
               splitPointChosenAttVector.addElement(splitPointVector[iSample]);
         // Number of split points related to chosen attribute
         int numberSplitPoints = splitPointChosenAttVector.size();
         // Get the median of the split points vector
         // // TODO median could be a method to be inserted in the class 'DoubleVector'
         splitPointChosenAttVector.sort();
         consolidatedSplitPoint = splitPointChosenAttVector.get(((numberSplitPoints+1)/2)-1);
      }
      return consolidatedSplitPoint;
   }

}
```

C45ConsolidatedSplit.java

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

import weka.classifiers.trees.j48.C45Split;
import weka.core.Instances;

/**
 * Class implementing a C4.5-type split on a consolidated attribute based on a set of samples.
 * *************************************************************************************
 * <p/>*** Attention! The visibility of the following members of the class 'C45Split'
 *     changed to 'protected' instead of 'private' in order to use them here:
 * <ul>
 *    <li>protected int m_complexityIndex;</li>
 *    <li>protected int m_attIndex;</li>
 *    <li>protected int m_minNoObj;</li>
 *    <li>protected double m_splitPoint;</li>
 *    <li>protected double m_infoGain;</li>
 *    <li>protected double m_gainRatio;</li>
 *    <li>protected double m_sumOfWeights;</li>
 *    <li>protected int m_index;</li>
 *    <li>protected static InfoGainSplitCrit infoGainCrit = new InfoGainSplitCrit();</li>
 *    <li>protected static GainRatioSplitCrit gainRatioCrit = new GainRatioSplitCrit();</li>
 * </ul>
 *
 * @author Jesús M. Pérez (txus.perez@ehu.es)
 * @version $Revision: 1.0 $
 */
public class C45ConsolidatedSplit extends C45Split {

   /** for serialization */
   private static final long serialVersionUID = 1174832141695586851L;

   /**
    * Creates a split model to be used to consolidate the decision around the set of samples,
    *  but with a null distribution
    *
    * @param attIndex attribute to split on
    * @param minNoObj minimum number of objects
    * @param sumOfWeights sum of the weights
    * @param data the training sample. Only to get information about the attributes
    * @param splitPointConsolidated the split point to use to split, if numerical.
    */
   public C45ConsolidatedSplit(int attIndex, int minNoObj, double sumOfWeights,
         Instances data, double splitPointConsolidated) {
      super(attIndex, minNoObj, sumOfWeights);

      // Initialize the remaining instance variables.
      m_splitPoint = splitPointConsolidated;
      m_infoGain = 0;
      m_gainRatio = 0;
      m_distribution = null;

      // Different treatment for enumerated and numeric attributes.
       if (data.attribute(m_attIndex).isNominal()) {
         m_complexityIndex = data.attribute(m_attIndex).numValues();
         m_index = m_complexityIndex;
         m_numSubsets = m_complexityIndex;
       }else{
         m_complexityIndex = 2;
         m_index = 2;
         m_numSubsets = 2;
       }
   }

   /**
    * Creates a split model based on the consolidated decision
    *
    * @param attIndex attribute to split on
    * @param minNoObj minimum number of objects
    * @param sumOfWeights sum of the weights
    * @param data the training sample. Only to get information about the attributes
    * @param samplesVector the vector of samples used for consolidation
    * @param splitPointConsolidated the split point to use to split, if numerical.
    * @exception Exception if something goes wrong
    */
   public C45ConsolidatedSplit(int attIndex, int minNoObj, double sumOfWeights,
         Instances data, Instances[] samplesVector, double splitPointConsolidated) throws Exception {
      this(attIndex, minNoObj, sumOfWeights, data, splitPointConsolidated);
      // Create a null model with the consolidated decision to calculate the consolidated distribution
      C45ConsolidatedSplit nullModelToConsolidate =
            new C45ConsolidatedSplit(m_attIndex, m_minNoObj, m_sumOfWeights, data, splitPointConsolidated);
      m_distribution = new DistributionConsolidated(samplesVector, nullModelToConsolidate);
      m_infoGain = infoGainCrit.splitCritValue(m_distribution, m_sumOfWeights);
      m_gainRatio = gainRatioCrit.splitCritValue(m_distribution, m_sumOfWeights, m_infoGain);
   }
}
```

DistributionConsolidated.java

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

import weka.classifiers.trees.j48.C45Split;
import weka.classifiers.trees.j48.ClassifierSplitModel;
import weka.classifiers.trees.j48.Distribution;
import weka.core.Instances;

/**
 * Class for handling a distribution of class values based on a consolidation process
 * ***************************************************************************************
 * </p>*** Attention! The visibility of the following members of the class 'Distribution'
 *     changed to 'protected' instead of 'private' in order to use them here:
 * <ul>
 *    <li>protected double m_perClassPerBag[][];</li>
 *    <li>protected double m_perBag[];</li>
 *    <li>protected double m_perClass[];</li>
 *    <li>protected double totaL;</li>
 * </ul>
 *
 * @author Jesús M. Pérez (txus.perez@ehu.es)
 * @version $Revision: 1.0 $
 */
public class DistributionConsolidated extends Distribution {

   /** for serialization */
   private static final long serialVersionUID = -6386302948424098805L;

   /**
    * Creates a distribution with only one bag according
    * to the vector of samples by calculating the average of the distributions.
    *
    * @param samplesVector the vector of samples used for consolidation
    */
   public DistributionConsolidated(Instances[] samplesVector) throws Exception {
      // Create the distribution object
      super(1, samplesVector[0].numClasses());
      int numberSamples = samplesVector.length;

      DistributionConsolidated[] distributionVector = new DistributionConsolidated[numberSamples];
      // Create the distribution related to each sample
      for(int iSample = 0; iSample < numberSamples; iSample++)
         distributionVector[iSample] = new DistributionConsolidated(samplesVector[iSample]);
      calculateMeanDistribution(distributionVector);
   }

   /**
    * Creates a distribution by calculating the average of the distributions according
    *  to each sample and given split model.
    *
    * @param samplesVector the vector of samples used for consolidation
    * @param modelToUse the split model to be used to split each sample
    */
   public DistributionConsolidated(Instances[] samplesVector,
         ClassifierSplitModel modelToUse) throws Exception {
      // Create the distribution object
      super(modelToUse.numSubsets(), samplesVector[0].numClasses());
      int numberSamples = samplesVector.length;
      int attIndex = ((C45Split)modelToUse).attIndex();

      /** Vector storing the distribution according to each sample */
      DistributionConsolidated[] distributionVector = new DistributionConsolidated[numberSamples];
      // Create the distribution related to each sample using the given split model
      for(int iSample = 0; iSample < numberSamples; iSample++){
         // Only Instances with known values are relevant.
         Instances sampleWithoutMissing = samplesVector[iSample];
         sampleWithoutMissing.deleteWithMissing(attIndex);
         distributionVector[iSample] = new DistributionConsolidated(sampleWithoutMissing, modelToUse);
         // Add all Instances with unknown values for the corresponding
         // attribute to the distribution for the model, so that
         // the complete distribution is stored with the model.
         distributionVector[iSample].addInstWithUnknown(samplesVector[iSample], attIndex);
      }
      calculateMeanDistribution(distributionVector);
   }

   /**
    * Constructor calling the constructor of the superclass
    * (No necessary if the above methods are moved to the official class 'Distribution')
    *
    * @param instances instances to be taken in account
    */
   public DistributionConsolidated(Instances instances) throws Exception {
      super(instances);
   }

   /**
    * Constructor calling the constructor of the superclass
    * (No necessary if the above methods are moved to the official class 'Distribution')
    *
    * @param instances instances to be taken in account
    * @param modelToUse the split model to be used to split each sample
    */
   public DistributionConsolidated(Instances instances, ClassifierSplitModel modelToUse) throws Exception {
      super(instances, modelToUse);
   }

   /**
    * Calculates the distribution by calculating the average of the distributions.
    *
    * @param distributionVector vector storing the distributions according to a set of samples
    */
   private void calculateMeanDistribution(DistributionConsolidated[] distributionVector){
      int numberSamples = distributionVector.length;
      int numberClasses = distributionVector[0].numClasses();
      // Add the distributions
      for(int iSample = 0; iSample < numberSamples; iSample++)
         add(distributionVector[iSample]);
      // Calculate the mean
      for(int iBag = 0; iBag < numBags(); iBag++){
          m_perBag[iBag] /= numberSamples;
         for(int iClass = 0; iClass < numberClasses; iClass++)
            m_perClassPerBag[iBag][iClass] /= numberSamples;
      }
      for(int iClass = 0; iClass < numberClasses; iClass++)
         m_perClass[iClass] /= numberSamples;
       totaL /= numberSamples;
   }

   /**
    * Adds the given distribution to this one.
    *
    * @param distribution distribution to be added
    */
   private final void add(DistributionConsolidated distribution) {
      for(int iBag = 0; iBag < numBags(); iBag++)
         add(iBag, distribution.getPerClassPerBag(iBag));
   }

   /**
    * Gets the weights of instances per class related to given bag.
    *
    * @param bagIndex index of the bag
    * @return the weights of instances per class
    */
   private final double [] getPerClassPerBag(int bagIndex){
       return m_perClassPerBag[bagIndex];
   }
}
```

InstancesConsolidated.java

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

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

/**
 * Class for extending the class Instances in order to add some methods
 * (These methods can be added to the class 'Instances')
 * *************************************************************************
 *
 * @author Jesús M. Pérez (txus.perez@ehu.es)
 * @version $Revision: 1.0 $
 */
public class InstancesConsolidated extends Instances {

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

   /**
    * Constructor calling the constructor of the superclass
    * (No 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
    * (No 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 on the class attribute.
    * (based on the method 'createSubsample()' of the class 'SpreadSubsample'
    *  of the package 'weka.filters.supervised.instances')
```

```java
     *
     * @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;
   }

   /**
    * 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));
   }
}
```