Java Classes for MDL-Based Attribute Ranking and Clustering

Zdravko Markov
Computer Science Department
Central Connecticut State University
New Britain, CT 06050, USA
http://www.cs.ccsu.edu/~markov/

1. Introduction 2
2. Attribute Ranking 2
3. Attribute Discretization 4
4. Clustering 5
5. Utilities 7
    5.1. ARFFstring class 7
    5.2. Increasing the heap size for the Java virtual machine 8
6. References 8

1. Introduction

This document describes algorithms for attribute ranking and clustering originally described in [1, 2, 3], an algorithm for attribute discretization briefly outlined hereafter, and a utility for creating string ARFF files. The algorithms are available as Java classes from a JAR file, which can be downloaded at http://www.cs.ccsu.edu/~markov/MDLclustering/MDL.jar. The archive also includes all Weka classes [4]. The archive is executable and starts the Weka command-line interface (Simple CLI), which provides access to these algorithms and to the full functionality of the Weka data mining software as well (see the Weka manual at http://www.cs.waikato.ac.nz/ml/weka/documentation.html). The data files mentioned in this document are available from a zip archive at http://www.cs.ccsu.edu/~markov/MDLclustering/data.zip.

2. Attribute Ranking

The original idea of the attribute ranking proposed in [1, 2, 3] is the following. First we split the data using the values of an attribute, i.e. create a clustering such that each cluster contains the instances that have the same value for that attribute. Then we compute the MDL of this clustering and use it as a measure to evaluate the attribute assuming that the best attribute minimizes MDL. This approach works for nominal attributes. To apply it to numeric attributes we first split the range of the attribute values into two intervals, which then can be used as nominal values for ranking. The idea is to find a breakpoint that minimizes the MDL of the resulting split of data. Hereafter we illustrate this process with the weather data example (data file weather.arff).

outlook	temperature	humidity	windy	play
sunny	85	85	FALSE	no
sunny	80	90	TRUE	no
overcast	83	86	FALSE	yes
rainy	70	96	FALSE	yes
rainy	68	80	FALSE	yes
rainy	65	70	TRUE	no
overcast	64	65	TRUE	yes
sunny	72	95	FALSE	no
sunny	69	70	FALSE	yes
rainy	75	80	FALSE	yes
sunny	75	70	TRUE	yes
overcast	72	90	TRUE	yes
overcast	81	75	FALSE	yes
rainy	71	91	TRUE	no

Two of the attributes in this dataset are numeric – temperature and humidity. Assume we want to evaluate humidity. First we order the values of this attribute (excluding repetitions) and start looking for a breakpoint, which will split them in two subsets. We consider each value as a possible breakpoint and compute the MDL for the split of data based on this value. When computing MDL we exclude the class attribute (play). The result of this process is shown in the table below, where the first row shows the ordered values (*x*), the second – the distribution of instances based on the split of data using that value (the number of instances with humidity less than or equal to *x* / the number of instances with humidity greater than *x*), and the third – the MDL for the split.

x	65	70	75	**80**	85	86	90	91	95	96
#[65, *x*] / #(*x*, 95]	1 / 13	4 / 10	5 / 9	**7 / 7**	8 / 6	9 / 5	11 / 3	12 / 2	13 / 1	–
MDL([65, *x*], (*x*, 96])	201.70	203.77	203.61	**200.48**	201.28	202.03	205.65	204.04	201.70	–

Choosing a value of *x* as a breakpoint defines a possible discretization of the attribute in two intervals, [65, *x*] and (*x*, 96]. The best split is the one that minimizes MDL, which is obtained with breakpoint 80, or the intervals [65, 80] and (80, 96]. Thus we compute the MDL of humidity as 200.48 and then rank it along with the rest of the attributes.

The general format for running this algorithm in CLI is:

```
java MDLranker <input file>.arff [# of attributes] [<output file>.{arff|csv}] [de]
```

The input file must be in ARFF format. The second argument specifies the number of attributes from the top of the ranked list to be displayed or written to the output file. Note that the last attribute in the data set (the default class attribute, "play" in the weather data) is not used for ranking and is excluded from this list (but added in the output file). The output file format may be either ARFF or CSV and is determined by the file name extension. When the fourth argument (de) is specified the numeric attributes are first transformed by applying nonparametric density estimation. For this purpose we use equal-interval binning and calculate the width of the interval *h* using the Scott's method as $h = \frac{3.5\sigma}{n^{1/3}}$, where *n* is the number of values of the attribute and σ – its standard deviation. Below is the output produced by the MDLranker algorithm applied to the weather data. The data set is written on the output file weather.ranked.csv, where the attributes humidity and temperature are switched.

```
> java MDLranker data/weather.arff 4 data/weather.ranked.csv

Attributes: 5
Ignored attribute: play
Instances: 14 (non-sparse)
Attribute-values in original data: 27
Numeric attributes with missing values (replaced with mean): 0
Minimum encoding length of data: 197.39
-------------------------------------------------------------------
Top 4 attributes ranked by MDL:
199.47 @attribute outlook {sunny,overcast,rainy}
200.48 @attribute humidity numeric
201.29 @attribute temperature numeric
202.51 @attribute windy {TRUE,FALSE}
---------------------------------------
Time(ms): 15
```

Note that the minimum encoding length of data is less than the MDL of the best attribute. This usually happens when there are too many attribute values and indicates that the MDL encoding does not help to reduce the code length of data. Applying density estimation reduces the number of attribute values and lowers the MDL encoding length as shown below.

```
> java MDLranker data/weather.arff 4 de

Attributes: 5
Ignored attribute: play
Instances: 14 (non-sparse)
Attribute-values in original data: 27
Numeric attributes with missing values (replaced with mean): 0
Attribute-values after density estimation: 9
Minimum encoding length of data: 97.68
-------------------------------------------------------------------
Top 4 attributes ranked by MDL:
92.07 @attribute outlook {sunny,overcast,rainy}
94.15 @attribute temperature numeric
94.15 @attribute humidity numeric
94.15 @attribute windy {TRUE,FALSE}
```

3. Attribute Discretization

The attribute discretization algorithm processes the numeric attributes in the same way as the MDLranker (described above). The general format for running this algorithm in CLI is:

```
java MDLdiscretize <input file>.arff <output file>.{arff|csv} [de]
```

The input file must be in ARFF format, while the output file format is specified by the file extension. When the third argument (de) is specified the numeric attributes are first transformed by density estimation (as in the MDLranker algortihm). Below is the output produced by the MDLdiscretize algorithm applied to the weather data:

```
> java MDLdiscretize data/weather.arff weather.discretized.arff

Attributes: 5
Ignored attribute: play
Instances: 14 (non-sparse)
Attribute-values in original data: 27
Numeric attributes with missing values (replaced with mean): 0
Minimum encoding length of data: 197.39
---------------------------------------
@attribute temperature {[64-80],(80-85]}
@attribute humidity {[65-80],(80-96]}
2 numeric attributes discretized
Time(ms): 31
```

The contents of the output file is the following:

```
@relation weather.MDLdiscretize

@attribute outlook {sunny,overcast,rainy}
@attribute temperature {[64-80],(80-85]}
@attribute humidity {[65-80],(80-96]}
@attribute windy {TRUE,FALSE}
@attribute play {yes,no}

@data
sunny,(80-85],(80-96],FALSE,no
sunny,[64-80],(80-96],TRUE,no
overcast,(80-85],(80-96],FALSE,yes
rainy,[64-80],(80-96],FALSE,yes
rainy,[64-80],[65-80],FALSE,yes
rainy,[64-80],[65-80],TRUE,no
overcast,[64-80],[65-80],TRUE,yes
sunny,[64-80],(80-96],FALSE,no
sunny,[64-80],[65-80],FALSE,yes
rainy,[64-80],[65-80],FALSE,yes
sunny,[64-80],[65-80],TRUE,yes
overcast,[64-80],(80-96],TRUE,yes
overcast,(80-85],[65-80],FALSE,yes
rainy,[64-80],(80-96],TRUE,no
```

Using the density estimation option changes the intervals for the values of temperature and humidity:

```
> java MDLdiscretize data/weather.arff weather.discretized.arff de

Attributes: 5
Ignored attribute: play
Instances: 14 (non-sparse)
Attribute-values in original data: 27
```

```
Numeric attributes with missing values (replaced with mean): 0
Attribute-values after density estimation: 9
Minimum encoding length of data: 97.68
---------------------------------------
@attribute temperature {[64.0-74.5],(74.5-85.0]}
@attribute humidity {[65.0-80.5],(80.5-96.0]}
2 numeric attributes discretized
Time(ms): 125
```

4. Clustering

The MDL clustering algorithm is described in [2]. It starts with the data split produced by the attribute that minimizes MDL and then recursively applies the same procedure to the resulting splits, thus generating a hierarchical clustering. For nominal attributes the number of splits is equal to the number of attribute values. Numeric attributes are treated in the same way as in the previous algorithms, and then the breakpoint that minimizes MDL is used to split the data in two subsets. The process of growing the clustering tree is controlled by a parameter evaluating the information compression at each node. The information compression is computed as the difference between the code length of the data at the current node of the tree and the MDL of the attribute that produces the data split. If the compression becomes lower than a specified cutoff value the process of growing the tree stops and a leaf node is created. An experimentally determined value of 20% of the information compression at the root of the tree is used as a default cutoff.

The general format for running this algorithm in CLI is the following:

```
java MDLcluster <input file>.arff [compr. cutoff] [de] [<output file>.{arff|csv}]
```

The input file must be in ARFF format. When specified the output file (in ARFF or CSV format) contains the cluster assignments as an additional attribute added as a last attribute (after the class attribute). If the compression cutoff is omitted the default value is used (20% of the initial compression). The meaning of the **de** argument is the same as in the previous algorithms.

An example of running MDLcluster with the iris data set is shown below:

```
> java MDLcluster data/iris.arff

Attributes: 5
Ignored attribute: class
Instances: 150 (non-sparse)
Attribute-values in original data: 123
Numeric attributes with missing values (replaced with mean): 0
Minimum encoding length of data: 3467.11
---------------------------------------------------------------
(238.38) (47.68)
#petallength<=1.9 (49.44)
  #sepalwidth<=3.2 (10.37) [18,0,0] Iris-setosa
  #sepalwidth>3.2 (22.63) [32,0,0] Iris-setosa
#petallength>1.9 (78.29)
  #sepallength<=6.2 (28.20) [0,36,13] Iris-versicolor
  #sepallength>6.2 (30.90) [0,14,37] Iris-virginica

---------------------------------------
Number of clusters (leaves): 4
Correctly classified instances: 123 (82%)
Time(ms): 16
```

The number in the parentheses represents the information compression at the corresponding node of the clustering tree (the second number at the root is the default cutoff), and the numbers in square brackets – the distribution of the class labels at the tree leaves. For each leaf the majority class label is also shown. The class distribution in the leaves provides information for evaluating the clustering quality when class labels are known (but ignored for the purposes of clustering) by using the classes-to-clusters evaluation measure (also used in Weka). It is a comparison to the "true" cluster membership specified by the class attribute and is computed as the total number of majority labels in the leaves divided by the total number of instances. This measure is reported as "Correctly classified instances" (123 out of 150, or 82%).

Below is another example that illustrates the use of the compression cutoff.

```
> java MDLcluster data/soybean-small.arff

Attributes: 36
Ignored attribute: class
Instances: 47 (non-sparse)
Attribute-values in original data: 72
Numeric attributes with missing values (replaced with mean): 0
Minimum encoding length of data: 3221.60
---------------------------------------------------------------
(951.79) (190.36)
stem-cankers=0 (51.15) [0,10,0,0] D2
stem-cankers=1 (153.77) [0,0,10,8] D3
stem-cankers=2 (53.56) [0,0,0,9] D4
stem-cankers=3 (46.84) [10,0,0,0] D1
---------------------------------------
Number of clusters (leaves): 4
Correctly classified instances: 39 (82%)
```

This clustering is produced with the default cutoff of 190.36, so all leaves of the tree have a smaller compression. The class distribution information suggests that we may get a better clustering by splitting cluster "stem-cankers=1", because it is not "pure" (as the other three). If we don't know the class labels (in this case we have to use a dummy class attribute), we still may be able to arrive at the same conclusion just by comparing the compression values of different clusters – the compression of "stem-cankers=1" is substantially larger than the compression at the other clusters, so it may be possible to expand the tree at this node and still have a good compression. To do this we have to use a cutoff value less than 153.77 and greater than 53.56 (not to split any of the other clusters). The result of this run is shown below.

```
> java MDLcluster data/soybean-small.arff 100

Attributes: 36
Ignored attribute: class
Instances: 47 (non-sparse)
Attribute-values in original data: 72
Numeric attributes with missing values (replaced with mean): 0
Minimum encoding length of data: 3221.60
---------------------------------------------------------------
(951.79)
stem-cankers=0 (51.15) [0,10,0,0] D2
stem-cankers=1 (153.77)
  canker-lesion=1 (78.09) [0,0,10,0] D3
  canker-lesion=2 (42.95) [0,0,0,8] D4
stem-cankers=2 (53.56) [0,0,0,9] D4
stem-cankers=3 (46.84) [10,0,0,0] D1
---------------------------------------
Number of clusters (leaves): 5
Correctly classified instances: 47 (100%)
```

5. Utilities

5.1. ARFFstring class

By using the ARFFstring class large collections of text/HTML files may be transformed into ARFF files with string attributes and then converted to TF/TFIDF type by applying the StringToWordVector filter. The general format of using this class is the following:

```
java ARFFstring <input directory> <class label> <output file>
```

This command creates a text file with as many lines as files in the input directory, where each line contains the following:

```
"file name", "file content", "class label"
```

The files in the input directory must contain plain text or HTML, where all HTML tags are removed.

Below we describe the steps for creating an ARFF file from the departments document collection available from http://www.cs.ccsu.edu/~markov/MDLclustering/data.zip:

1. Create file `deptA` with the files in folder `data/departments/A` with class label `A`:

```
java ARFFstring data/departments/A A deptA
```

2. Create file `deptB` with the files in folder `data/departments/B` with class label `B`:

```
java ARFFstring data/departments/B B deptB
```

3. Merge `deptA` and `deptB`. This can be done in a command prompt window with the copy command:

```
copy deptA + deptB departments-string.arff
```

4. Add the following ARFF file header in the beginning of `departments-string.arff`:

```
@relation departments_string

@attribute document_name string
@attribute document_content string
@attribute document_class string

@data
```

5. Convert the first and the third attribute into nominal and write the output on `temp1.arff`:

```
java  weka.filters.unsupervised.attribute.StringToNominal
      -i departments-string.arff
      -o temp1.arff -R 1,3
```

6. Transform the second attribute (document_content) into a set of numeric attributes representing the word presence (0/1) in the documents:

```
java  weka.filters.unsupervised.attribute.StringToWordVector
      -i temp1.arff
      -o temp2.arff
      -S
      -tokenizer weka.core.tokenizers.AlphabeticTokenizer
```

7. Move the document_class attribute to the end and write the output to `departments.arff`:

```
java weka.filters.unsupervised.attribute.Reorder
      -i temp2.arff
      -o departments.arff
      -R 1,3-last,2
```

8. The data set `departments.arff` can now be used for clustering, classification and other experiments. For example:

```
> java MDLcluster departments.arff

Attributes: 612
Ignored attribute: document_class
Instances: 20 (sparse)
Attribute-values in original data: 1234
Numeric attributes with missing values (replaced with mean): 0
Minimum encoding length of data: 24569.11
---------------------------------------------------------------
(4305.58) (861.12)
#major<=0 (1498.23)
  #computer<=0 (247.68) [2,3] B
  #computer>0 (255.52) [4,1] A
#major>0 (1452.01)
  #offers<=0 (265.72) [3,2] A
  #offers>0 (264.13) [2,3] B

---------------------------------------
Number of clusters (leaves): 4
Correctly classified instances: 13 (65%)
Time(ms): 375
```

5.2. Increasing the heap size for the Java virtual machine

The parameter **`-Xmx<heap size>`** may be used in the command line to set the maximum heap size used by the Java virtual machine. This can be done when starting the MDL.jar in the Windows command prompt. For example, the following command will initialize the Java virtual machine with 1600 Mb maximum heap size and start the Weka CLI.

```
java -Xmx1000m -jar MDL.jar
```

Similarly, any class from MDL.jar may be started in the command prompt window after extracting the archive. For example, the following command will initializes the Java virtual machine with 1000 Mb maximum heap size and start the MDLcluster class.

```
java -Xmx1000m MDLcluster data/weather.arff
```

6. References

1. Zdravko Markov. MDL-based Unsupervised Attribute Ranking, Proceedings of the 26th International Florida Artificial Intelligence Research Society Conference (FLAIRS-26), St. Pete Beach, Florida, USA, May 22-24, 2013, AAAI Press 2013, pp. 444-449. http://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS13/paper/view/5845/6115.

2. Zdravko Markov. MDL-Based Hierarchical Clustering, Proceedings of the IEEE 14th International Conference on Machine Learning and Applications (ICMLA 2015), December 9-11, 2015, Miami, Florida, USA, pp. 464-467. PDF

3. Zdravko Markov and Daniel T. Larose. MDL-Based Model and Feature Evaluation, in Chapter 4 of *Data Mining the Web: Uncovering Patterns in Web Content, Structure, and Usage*, Wiley, April 2007, ISBN: 978-0-471-66655-4.

4. Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten (2009). The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.