Tutorial. MEKA 1.7.3

Jesse Read

September 2014

A Multilabel/multitarget Extension to WEKA.
http://meka.sourceforge.net

Contents

1 Introduction

This is a tutorial for the open source machine learning framework MEKA. MEKA is closely based upon the WEKA framework [2]; providing support for development, running and evaluation of *multi-label* and *multi-target* classifiers (which WEKA does not).

In the *multi-label* problem, a data instance may be associated with multiple labels. This is as opposed to the traditional task of single-label classification (i.e., multi-class, or binary) where each instance is only associated with a single class label. The multi-label context is receiving increased attention and is applicable to a wide variety of domains, including text, music, images and video, and bioinformatics. A good introduction can be found in [7] and [3].

The multi-label problem is in fact a special case of *multi-target* learning. In multi-target, or *multi-dimensional* learning, a data instance is associated with multiple target variables, where each variable takes a number of values. In the multi-label case, all variables are binary, indicating label relevance (1) or irrelevance (0). The multi-target case has been investigated by, for example, [9] and [10].

MEKA can also includes *incremental* classifiers suitable for the *data streams* context. An overview of some of the methods included in MEKA for learning from incremental data streams is given in [4].

MEKA is released under the GNU GPL licence. The latest release, source code, API reference, this tutorial, and further information and links to additional material, can be found at the website: `http://meka.sourceforge.net`.

This tutorial applies to MEKA version `1.7.3`.

2 Getting Started

MEKA can be download from: `http://sourceforge.net/projects/meka/files/`. This tutorial is written for version `1.7.3`; and assumes that you have downloaded and extracted the `meka-release-1.7.3` and that `meka-1.7.3`, found within, is your current working directory.

2.1 Requirements

MEKA requires:

- Java version 1.6 or above

MEKA comes bundled with other packages such as WEKA's `weka.jar`, and also MULAN's `mulan.jar` for running classifiers from this framework. These files are found in the `lib` directory. See `lib/README.txt` for version information.

2.2 Running

MEKA can be used very easily from the command line. For example, to run the Binary Relevance (BR) classifier on the *Music* dataset; type:

```
java -cp "./lib/*" meka.classifiers.multilabel.BR -t data/Music.arff
```

If you are on a Microsoft Windows system, you need to use back slashes instead of forward slashes (`.\lib\*`). If you add the `jar` files to the system's `CLASSPATH`, you do not need to supply the `-cp` option at runtime. For the remainder of examples in this tutorial we will assume that this is the case.

Since Version 1.2 MEKA has a graphical user interface (GUI). Run this with either the `run.sh` script (under Linux, OSX) as follows:

```
bash run.sh
```

Run `run.bat` instead of you are using Microsoft windows.

3 MEKA's Dataset Format

MEKA uses WEKA's ARFF file format. See `http://weka.wikispaces.com/ARFF` to learn about this format. MEKA uses multiple attributes – one for each target or label – rather than a single class attribute. The *number* of target attributes is specified with either `-C` or `-c`; *unlike* in WEKA where the `-c` flag indicates the position of the *class index*. MEKA uses the reference to the `classIndex` internally to denote the number of target attributes.

Since the number of target attributes tends to vary with each dataset, for convenience MEKA allows this option (as well as other dataset options like the train/test split percentage) to be stored in the `@relation` name of an ARFF file, where a colon (`:`) is used to separate the dataset name and the options. The following is an example ARFF header for multi-target classification with three target variables and four attributes:

```
@relation 'Example_Dataset: -C 3'

@attribute category {A,B,C,NEG}
@attribute label {0,1}
@attribute rank {1,2,3}
@attribute X1 {0,1}
@attribute X2 {0,1}
@attribute X3 numeric
@attribute X4 numeric

@data
```

Note that the format of the `label` attribute (binary) is the *only* kind of target attribute in multi-*label* datasets. For more examples of MEKA ARFF files; see the `data/` directory for several multi-label and multi-target datasets (some of these are in a compressed format).

MEKA can also read ARFF files in the MULAN format where target attributes are the *last* attributes, rather than the first ones. This format can also be read by MEKA by specifying a minus sign '`-`' before the number of target attributes in the `-C` option. For example, `-C -3` will set the *last* `3` attributes as the target attributes automatically when the file is loaded. Alternatively, the class attributes can be moved using WEKA's `Reorder` filter, or in the GUI, as in the following.

3.1 Manipulating Datasets in the GUI

A good way to set up an ARFF file for multi-dimensional classification is using the GUI. Open an ARFF file with 'Open' from the File menu. In the Preprocess tab in the right-hand column

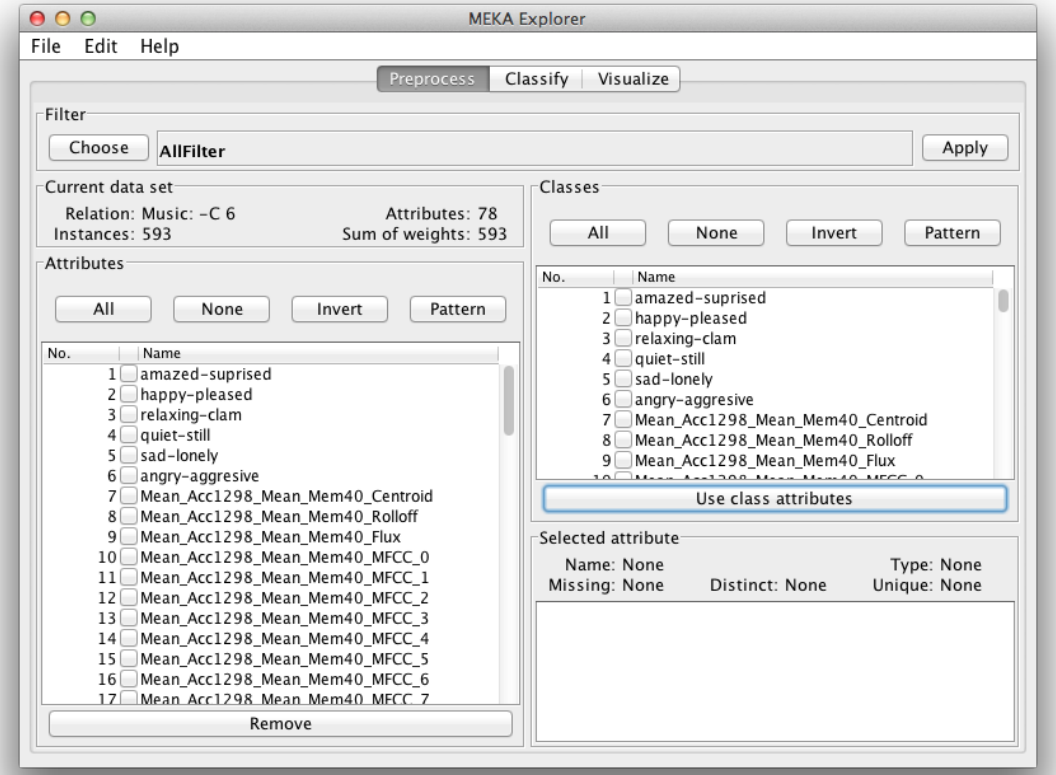

Figure 1: MEKA's GUI interface; having loaded the *Music* dataset.

(see Figure 1), simply select the attributes you wish to use as class attributes and click the button 'Use class attributes'. You can then save this file using 'Save' from the File menu, which will also save the `-C` flag into the `@relation` tag as described above (displayed under 'Relation:' in the GUI), so next time the classes will be set automatically.

The datasets that come with MEKA already come with the `-C` flag specified correctly, so you do not need to set this information.

You can also run any of WEKA's filters on the dataset with the Choose button. See the WEKA documentation for more information.

4 Using MEKA

A suitable dataset is the only requirement to begin running experiments with MEKA.

4.1 Command Line Interface

With the exception of the different use of the `-c` flag (see the previous section), many of WEKA's command line options for evaluation work identically in MEKA too. You can obtain a list of them

by running any classifier with the -h flag, for example: `java meka.classifiers.multilabel.BR -h` displays the following:

```
Evaluation Options:

-h
        Output help information.
-t <name of training file>
        Sets training file.
-T <name of test file>
        Sets test file.
-x <number of folds>
        Do cross-validation with this many folds.
-R
        Randomise the dataset (done after a range is removed, but before
           the train/test split).
-split-percentage <percentage>
        Sets the percentage for the train/test set split, e.g., 66.
-split-number <number>
        Sets the number of training examples, e.g., 800
-i
        Invert the specified train/test split.
-s <random number seed>
        Sets random number seed.
-threshold <threshold>
        Sets the type of thresholding; where
        - 'PCut1' automatically calibrates a threshold (the default);
        - 'PCutL' automatically calibrates one threshold for each label;
        - any double number, e.g. '0.5', specifies that threshold.
-C <number of classes/labels>
        Sets the number of target attributes (classes/labels) to expect (
           indexed from the beginning).
-f <results_file>
        Specify a file to output results and evaluation statistics into.
-d <classifier_file>
        Specify a file to dump classifier into.
-l <classifier_file>
        Specify a file to load classifier from.
-verbosity <verbosity level>
        Specify more/less evaluation output


Classifier Options:

-W
        Full name of base classifier.
        (default: weka.classifiers.rules.ZeroR)
-output-debug-info
        If set, classifier is run in debug mode and
        may output additional info to the console
--do-not-check-capabilities
        If set, classifier capabilities are not checked before classifier
            is built
```

```
        (use with caution).
```

The only required options are `-t` to specify the dataset, and `-C` to specify the number of target attributes; the latter is typically included within the dataset, as explained in the previous section.

The `Classifier Options` are specific to each classifier, which in this case (for `java meka.classifiers.multilabel.BR`) are not very extensive. However, to get decent results with this classifier, we will have to specify a more competitive *base classifier* with the `-W` option, for example Naive Bayes. To run this on the *Music* data, we would type[1] on the command line:

```
java meka.classifiers.multilabel.BR -t data/Music.arff \
  -W weka.classifiers.bayes.NaiveBayes
```

4.2 Graphical User Interface

The CLI is the most powerful way to work with MEKA, but the GUI is a good way to get started. Refer to Section 2.2 on how to open the GUI. Once opened, you will see three tabs: Preprocess, Classify, Visualize. The following process will guide you through a simple experiment.

1. Load a dataset file using Open from the file menu.
2. Click on the Classify tab.
3. Choose a multi-label or multi-target classifier and (in most cases) an appropriate WEKA base classifier, as well as its options. For MEKA's meta classifiers, you will need to choose a MEKA base classifier, and a single-label WEKA base classifier for this classifier. See, for example, Figure 2, using Bagging of Classifier Chains with SMO.
4. In the Evaluation panel you configure what type of evaluation you want to do, and some of the options given in the previous section are available here. For example, a 50/50 train/test split, as being specified in Figure 3.
5. When you click Start the experiment will be run. When finished, the result will appear in the History panel; or multiple results in the case of incremental validation. This is the same output as would be seen on the command line, and explained in the following section.
6. Optionally, you can click on the Visualize tab and visualize the results.

4.3 Evaluation

Running a BR classifier with Naive Bayes on the *Music* data will output the following:

```
    Classifier_name : meka.classifiers.multilabel.BR
     Classifier_ops : [-W, weka.classifiers.bayes.NaiveBayes, --, , , ]
    Classifier_info :
       Dataset_name : Music
               Type : ML
          Threshold : 0.9974578524138343
          Verbosity : 1
```

[1] If typed on one line, the bar '\' should be omitted

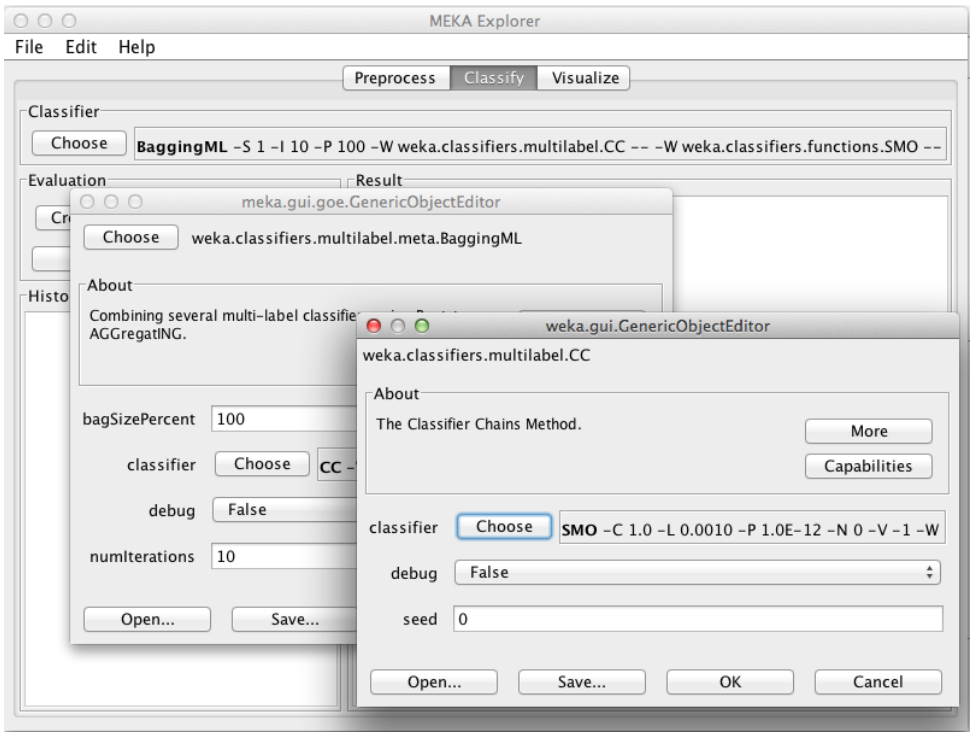

Figure 2: MEKA's GUI interface; setting Bagging of Classifier Chains with SMO.

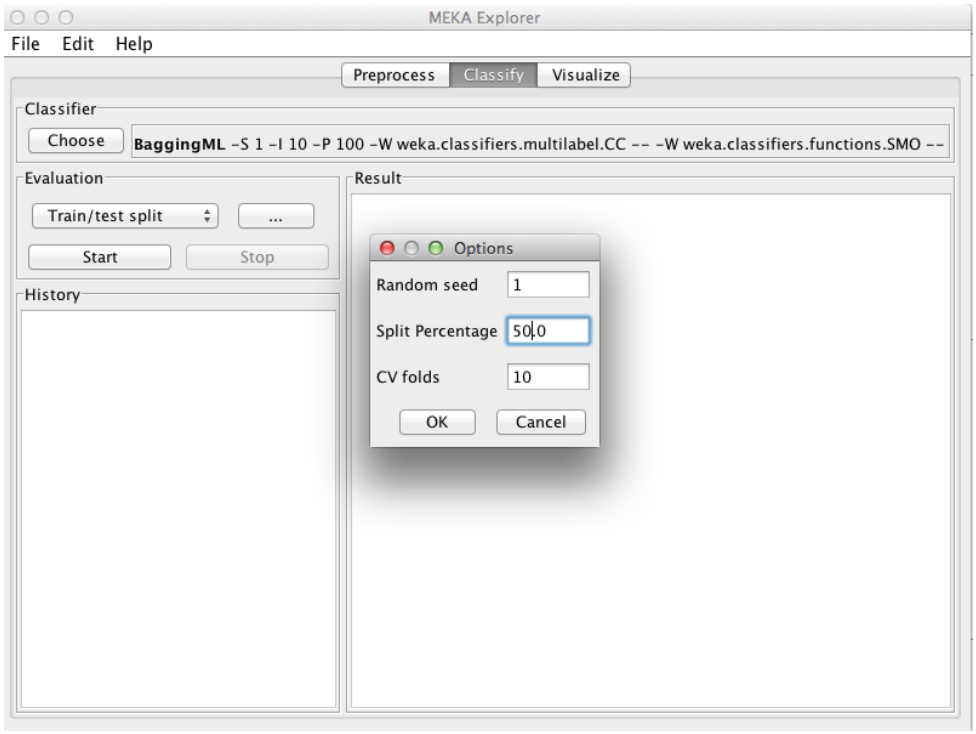

Figure 3: MEKA's GUI interface; setting a train/test split.

```
                L : 6
         Accuracy : 0.436
  Hamming score : 0.745
    Exact match : 0.215

        N_train : 355
         N_test : 237
    LCard_train : 1.789
     LCard_test : 1.992
     Build_time : 0.113
      Test_time : 0.067
     Total_time : 0.18
```

Note that by increasing the `verbosity` level, you can get more output: `-verbosity <n>` where `<n>` =

`verbosity`	Output
`1` (default)	basic output
`2`	plus more metrics
`3`	plus individual metrics (for each label)
`4`	plus more individual metrics (for each label)
`5`	plus individual classifications
`6`	plus individual confidence outputs (rounded to 1 d.p.)
`7`	... (rounded to 2 d.p.)
`8`	... (rounded to 3 d.p.)

Most of these measures are described in [3, 6, 7]. The most common measures in the multi-label literature are *Hamming Loss* (`H_loss`), which is the accuracy for each label, averaged across all labels; *Exact Match* (`Exact_match`), which is the is the accuracy of each *example* – where all label relevances must match exactly for an example to be correct; and *Accuracy* (`Accuracy`), which is neither as strict as Exact Match nor as 'easy' as Hamming Loss.

Note that a `Threshold` has been calibrated automatically; to minimise the difference between the label cardinality of the training set and the predictions on the test set; a practice described in [6]. To calibrate a threshold for *each* label, add the `-threshold PCutL` option[2]. This gives a vector of thresholds which, in this case, increases `Accuracy` slightly (to `0.456`). Different thresholds are calculated for different methods and datasets.

MEKA also supports *cross validation*; for example:

```
java meka.classifiers.multilabel.BR -x 10 -R -t data/Music.arff \
  -W weka.classifiers.bayes.NaiveBayes
```

conducts 10 fold cross validation on a randomised version of the *Music.arff* data and outputs the average results across all folds with standard deviation.

4.4 Examples

Binary Relevance (BR) On the *Music* data, loading from two separate sets, using Naive Bayes as a base classifier, calibrating a separate threshold automatically for each label:

[2]Note: this was `-T C` in previous versions of MEKA

```
java meka.classifiers.multilabel.BR \
  -t data/Music_train.arff \
  -T data/Music_test.arff \
  -threshold PCutL \
  -W weka.classifiers.bayes.NaiveBayes
```

Ensembles of Pruned Sets (EPS; see [5]) With 10 ensemble members (the default) on the *Enron* dataset with Support Vector Machines as the base classifier; each `PS` model is set with `N=1` and `P` to a random selection of {`1,2,3,4,5`}:

```
java meka.classifiers.multilabel.meta.EnsembleML \
  -t data/Yeast.arff \
  -W meka.classifiers.multilabel.PS -- \
    -P 1-5 -N 1 -W weka.classifiers.functions.SMO
```

Ensembles of Classifier Chains (ECC; see [6]) With 50 ensemble members (`-I 50`), and some textual output (`-output-debug-info`) on the *Enron* dataset with Support Vector Machines as a base classifier:

```
java meka.classifiers.multilabel.meta.BaggingML -I 50 -P 100 \
  -output-debug-info -t data/Enron.arff -W meka.classifiers.multilabel.CC
     \
  -- -W weka.classifiers.functions.SMO
```

Mulan Classifier (`RAkEL` see [8]) With parameters `k=3, m=2C` where `C` is the number of labels (these options are hardwired; you need to edit `MULAN.java` to specify new parameter configurations) on the *Scene* dataset with Decision Trees as the base classifier (remember `mulan.jar` must be in the classpath):

```
java meka.classifiers.multilabel.MULAN -t data/Scene.arff -verbosity 5 \
  -S RAkEL2 -W weka.classifiers.trees.J48
```

the `-verbosity 5` options increases the amount of evaluation output.

Incremental Classification: Ensembles of Binary Relevance (see [6, 4]) With 10 ensemble members (default) on the *Enron* dataset with `NaiveBayesUpdateable` as a base classifier; using over 20 evaluation windows:

```
java meka.classifiers.multilabel.meta.BaggingMLUpdateable -B 20\
  -t data/Enron.arff \
  -W meka.classifiers.multilabel.BRUpdateable -- \
    -W weka.classifiers.bayes.NaiveBayesUpdateable
```

Evaluating incremental classifiers will carry out evaluation and display statistics for the data over $\texttt{B} - 1$ evaluation windows (an initial window is used for the initial training; making `B` windows in total).

Multi-target: Ensembles of Class Relevance (see [9]) The multi-target version of the Binary Relevance classifier) on the *solar flare* dataset with Logistic Regression as a base classifier under 5-fold cross-validation:

```
java weka.classifiers.multitarget.meta.BaggingMT -x 10 -R \
  -t data/solar_flare.arff \
  -W weka.classifiers.multitarget.CR -- \
    -W weka.classifiers.functions.Logistic
```

5 Development

The following sections explain a bit more in detail of how to obtain MEKA's source code, how to compile it and how to develop new algorithms.

5.1 Source code

For obtaining the source code of MEKA, you have two options:

- Using subversion[3]
- Release archive

In the case of *subversion*, you can obtain the source code using the following command in the console (or *command prompt* for Windows users):

```
svn checkout svn://svn.code.sf.net/p/meka/code/trunk meka
```

This will create a new directory called `meka` in the current directory, containing the source code and build scripts.

Instead of using subversion, you can simply use the source code that is part of each MEKA release, contained in the `meka-src-X.Y.jar` Java archive[4]. A Java archive can be opened with any archive manager that can handle ZIP files.

5.2 Compiling

Using `ant`
MEKA uses *Apache ant*[5] as build tool. You can compile MEKA as follows:

```
ant clean compile jar
```

If you want to generate an archive containing all source code, pdfs and binary jars, then you can use the following command:

```
ant clean compile release
```

Please note, if you develop new algorithms, you should also create a unit test for it, to ensure that it is working properly. See section 5.3 for more details.

Using Eclipse
After obtaining the source code, you can use the Eclipse[6] template files that come with MEKA. You only need to rename two files, found in the same directory as the `build.xml` ant build script, as follows:

[3] `http://subversion.apache.org/`
[4] `http://en.wikipedia.org/wiki/JAR_(file_format)`
[5] `http://ant.apache.org/`
[6] `http://eclipse.org/`

- `.classpath.default` → `.classpath`
- `.project.default` → `.project`

Then you can simply import MEKA as *Existing Java Project*, pointing the import wizard to the directory containing the above mentioned files.

5.3 Unit tests

MEKA uses the JUnit 3.8.x unit testing framework.
A classifier test case is derived from the following abstract super class:

```
meka.classifiers.AbstractMekaClassifierTest
```

A filter test case is derived from the following abstract super class:

```
meka.filters.AbstractMekaFilterTest
```

You can execute all the unit tests by calling the following class from the command-line:

```
meka.MekaTests
```

5.4 Extending MEKA

Writing MEKA classifiers involves writing regular WEKA classifiers that extend either the `MultilabelClassifi` or `MultiTargetClassifier` class, and expect the `classIndex()` of `Instances` and `Instancess` to indicate the number of target attributes (indexed at the beginning) rather than the class index (as explained in Section 3).

The following is an example of a functioning (but extremely minimalistic) classifier, `TestClassifier`, that predicts 0-relevance for all labels:

```
package meka.classifiers.multilabel;
import weka.core.*;

public class TestClassifier extends MultilabelClassifier {

    public void buildClassifier(Instances D) throws Exception {
        testCapabilities(D);
        int C = D.classIndex();
    }

    public double[] distributionForInstance(Instance x) throws Exception
        {
        int C = x.classIndex();
        return new double[C];
    }

    public static void main(String args[]) {
        MultilabelClassifier.runClassifier(new TestClassifier(),args);
    }
}
```

This shows how easy it is to create a new classifier. However, for more useful examples see the source code of existing MEKA classifiers. The `testCapabilities(D)` line is optional but highly recommended. Note that the `distributionForInstance` method returns

a double[] array exactly like in WEKA. However, whereas in WEKA, there is one value in the array for each possible value of the single target attribute, in MEKA this function returns an array of C values, where C is the *number* of target attributes, and the jth value of the array is the *value* corresponding to the jth target attribute.

5.4.1 Building Classifiers

In the buildClassifier(Instances) method, you build your classifier. Here is where you can take advantage of all of Weka's libraries. You can use any Wekaclassifier to your needs. The m_Classifier variable is already available for this, which *already contains* the Weka classifier you specify on the command line. You can simply do:

```
public void buildClassifier(Instances D) throws Exception {
    testCapabilities(D);
    int C = D.classIndex();
    D.setClassIndex(0);
    m_Classifier.buildClassifier(D);
}
```

to train a classifier to learn the first label of your data (using the other labels, and all other input-space feature attributes). So if you then run on the command line

```
java meka.classifiers.multilabel.TestClassifier -t data/Music.arff \
    -W weka.classifiers.functions.SMO
```

an instantiation of SMO will already be be available in m_Classifier. You can also do this explicitly with

```
...
    m_Classifier = new SMO();
    m_Classifier.buildClassifier(D);
...
```

5.4.2 Classifying New Instances

In the multi-label case, for a test Instance x, the double[] array returned by the method distributionForInstance(x) might contain the 0/1 label relevances, for example (assuming -C 5):

```
    [0.0, 0.0, 1.0, 1.0, 0.0]
```

or it might contain posterior probabilities / prediction confidences / votes for each label, for example:

```
    [0.1, 0.0, 0.9, 0.9, 0.2]
```

where clearly the third and fourth labels are most relevant. Under a threshold of 0.5 the final classification for x would be [0,0,1,1,0]. MEKA will by default automatically calibrate a threshold to convert all values into 0/1 relevances like these (see Section 4.3).

In the multi-target case, the double[] values returned by the method distributionForInstance must indicate the *relevant class value*; for example (assuming -C 3):

```
    [3.0, 1.0, 0.0]
```

If this were the dataset exemplified in 3, this classification would be `C`, `1`, and `1` for the class attributes `category`, `label`, and `rank`, respectively.

Note that no threshold is calibrated. However, any associated voting or probabilistic values may be stored in the following `C+1,...,2C` values; for example (again assuming `-C 6`):

```
[3.0, 1.0, 0.0, 0.5, 0.9, 0.9]
```

where `C` is predicted as the value of the first target attribute with confidence 0.9, and so on. However these values are currently only for use at classification time (for example the voting scheme of an ensemble method, see `weka.classifiers.multitarget.BaggingMT`); and not taken into account for evaluation. Note also that we intend to deprecate this method in the future, so avoid if possible.

5.4.3 Incremental Classifiers

MEKA comes with incremental versions of many classifiers, as well as incremental evaluation methods. Incremental classifiers implement WEKA's `UpdatebleClassifier` interface and therefore must implement the `updateClassifier(Instance)` method. The following extends `TestClassifier` for incremental learning.

```
package meka.classifiers.multilabel;
import weka.core.*;

public class TestClassifierUpdateable extends TestClassifier
    implements UpdateableClassifier{

    public void updateClassifier(Instance x) throws Exception {
        int L = D.classIndex();
    }

    public static void main(String args[]) {
        IncrementalEvaluation.runExperiment(new TestClassifierUpdateable
            (),args);
    }
}
```

Note that the `IncrementalEvaluation` class is called for evaluation in this case; see Section 4.3. The MOA framework [1] for learning in data streams contains a number of incremental classifiers that Weka does not, for example Hoeffding trees. It is included in MEKA, and you can run these classifiers using Weka's `MOA` meta classifier (wrapper), e.g.,

```
java meka.classifiers.multilabel.BRUpdateable -output-debug-info -t data/
   Music.arff \
        -W weka.classifiers.meta.MOA -- -B moa.classifiers.trees.
           HoeffdingTree
```

Note that MOA now also supports MEKA classifiers via a wrapper class; and is currently being developed to support a range of incremental multi-label classification and evaluation for data streams.

6 Getting Help / Reporting Bugs / Contributing

If you need help with MEKA, you can post your problem on Meka's Mailing List: `http://sourceforge.net/p/meka/mailman/`.

If you have found a bug with MEKA, you can report in via the Tracker of MEKA's `SourceForge.net` site: `http://sourceforge.net/p/meka/bugs/`.

If you would like to contribute to MEKA, such as adding new classifiers, please get in touch with the developers.

More more information (such as contact information) can be found at the MEKA website: `http://meka.sourceforge.net`.

A Methods Implemented in Meka

Below is a list of the major methods supported in MEKA, along with their abbreviations (in some cases methods are known by more than one name), and the class name (given from the `classifiers` directory). Problem transformation methods expect a `-W` option. For example, to run classifier chains:

```
java meka.classifiers.multilabel.meta.BaggingML -P 100 \
    -W meka.classifiers.multilabel.CC -- -W weka.classifiers.functions.
       SMO
```

and to run CLR:

```
java meka.classifiers.multilabel.MULAN -S CLR\
    -W weka.classifiers.functions.SMO
```

Other parameters are mostly set to the default found in the literature where the method was introduced. The methods have been grouped approximately together. Citations are included within the `TechnicalInformation` of the source code files. *U* indicates that an `Updateable` version is available. *M* indicates that a multi-target version (or can be used directly) is available.

Algorithm	**Class Name**
Majority Labelset [U,M]	`MajorityLabelset`
Binary Relevance (BR) [U,M]	`BR`
Classifier Chains (CC) [U,M]	`CC`
Ensembles of CC (ECC) [U,M]	`meta.BaggingML -P 100 -W CC`
Ensembles of CC 'quick' (ECCq) [U,M]	`meta.RandomSubspaceML -W CCq`
Probabilistic Classifier Chains (PCC) [M]	`PCC`
Monte Carlo Optimization of CC (MCC) [M]	`MCC`
Bayesian Classifier Chains (BCC) [M]	`BCC`
Ranking + Threshold [M] (RT, a.k.a. PT5)	`RT`
Fourclass Pairwise (FW, a.k.a. PW)	`FW`
Calibrated Label Ranking [R]	`MULAN -S CLR`
Learning EM-style from unlabeled examples	`meta.EM`
Conditional Dependency Networks [M]	`CDN`
Label Combination (LC, a.k.a. LP)	`LC`
Random k-labEL Sets (RAkEL)	`MULAN -S RAkEL1`
RAkEL combined with `PS`	`RAkEL`
... disjoint subsets	`RAkELd`
Pruned Problem Transformation (PPT)	`PSt`
Pruned Sets (PS) [U]	`PS`
Ensembles of Pruned Sets (EPS	`meta.EnsembleML -W PS`
Nearest Set Replacement (NSR) [M]	`NSR`

References

[1] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa massive online analysis, 2010. `http://mloss.org/software/view/258/`.

[2] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Reutemann Peter, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[3] Jesse Read. *Scalable Multi-label Classification*. PhD thesis, University of Waikato, 2010.

[4] Jesse Read, Albert Bifet, Bernhard Pfahringer, and Geoffrey Holmes. Scalable and efficient multi-label classification for evolving data streams. *Machine Learning*, 2012. Accepted for publication.

[5] Jesse Read, Bernhard Pfahringer, and Geoff Holmes. Multi-label classification using ensembles of pruned sets. In *ICDM'08: Eighth IEEE International Conference on Data Mining*, pages 995–1000. IEEE, 2008.

[6] Jesse Read, Bernhard Pfahringer, Geoffrey Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Machine Learning*, 85(3):333–359, 2011.

[7] G. Tsoumakas, I. Katakis, and I. Vlahavas. Mining multi-label data. In O. Maimon and L. Rokach, editors, *Data Mining and Knowledge Discovery Handbook*. 2nd edition, Springer, 2010.

[8] Grigorios Tsoumakas and Ioannis P. Vlahavas. Random k-labelsets: An ensemble method for multilabel classification. In *ECML '07: 18th European Conference on Machine Learning*, pages 406–417. Springer, 2007.

[9] Julio H. Zaragoza, Luis Enrique Sucar, Eduardo F. Morales, Concha Bielza, and Pedro Larrañaga. Bayesian chain classifiers for multidimensional classification. In *24th International Conference on Artificial Intelligence (IJCAI '11)*, pages 2192–2197, 2011.

[10] Bernard Zenko and Saso Dzeroski. Learning classification rules for multiple target attributes. In *PAKDD '08: Twelfth Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 454–465, 2008.