utiml: Utilities for multi-label learning

Adriano Rivolli

2016-04-07

Version: 0.1.0.9000

The utiml package is a framework to support multi-label processing, like Mulan on Weka. It is simple to use and extend. This tutorial explain the main topics related with the utiml package. More details and examples are available on utiml repository.

Note: *Currently, just few one-agains-all transformation methods are available, but in the future we intend remove this note and have a full range of multi-label classification methods. If you want to contribute with your code or your tallent, read the last section about how to contribute.*

1. Introduction

The general prupose of **utiml** is be an alternative to processing multi-label in R. The main methods available on this package are organized in the groups:

- Classification methods
- Evaluation methods
- Pre-process utilities
- Sampling methods
- Threshold methods

The **utiml** package needs of the mldr package to handle multi-label datasets. It will be installed together with the **utiml**[1].

The installation process is similar to other packages available on CRAN:

```
install.packages("utiml")
```

After installed, you can now load the **utiml** package (The mldr package will be also loaded):

```
library("utiml")
```

The **utiml** brings a synthetic multi-label dataset called `toyml`, all examples illustrated in this tutorial use it. To understand how to load your own dataset, we suggest the read of mldr documentation. The `toyml` contains 100 instances, 10 features and 5 labels, its prupose is to be used for small tests and examples.

```
head(toyml)
```

In the following section, an overview of how to conduct a multi-label experiment are explained. Next, we explores each group of methods and its particularity. Finally, how to extend the **utiml** is aborded and illustrated, then the final considerations are made.

[1] You may also be interested in mldr.datasets

2. Overview

After load the multi-label dataset some data processing may be necessary. The pre-processing methods are utilities that manipulate the `mldr` datasets. Suppose that we want to normalize the attributes values (between 0 and 1), we can do:

```
mytoy <- normalize_mldata(toyml)
```

Next, we want to stratification the dataset in two partitions (train and test), containing 65% and 35% of instances respectively, then we can do:

```
ds <- create_holdout_partition(mytoy, c(train=0.65, test=0.35), "iterative")
```

Now, the `ds` object has two elements `ds$train` and `ds$test`, where the first will be used to create a model and the second to test the model. For example, using the *Binary Relevance* multi-label method with the base classifier *Random Forest*[2], we can do:

```
brmodel <- br(ds$train, "RF", seed=123)
prediction <- predict(brmodel, ds$test)
```

The `prediction` is an object of class `mlresult` that contains the probability (also called confidence or score) and the bipartitions values:

```
head(as.bipartition(prediction))
head(as.probability(prediction))
head(as.ranking(prediction))
```

A threshold strategy can be applied and generate a refined prediction:

```
newpred <- rcut_threshold(prediction, 2)
```

Now we can evaluate the model and compare if the use of MCUT threshold improve the results:

```
result <- multilabel_evaluate(ds$tes, prediction, "bipartition")
thresres <- multilabel_evaluate(ds$tes, newpred, "bipartition")

measures <- c("accuracy", "F1", "precision", "recall", "subset-accuracy")
round(cbind(Default=result, RCUT=thresres), 3)
```

3. Pre-processing

The pre processing methods were developed to facilitate some operation with the multi-label data. All pre-processing methods receive a mldr dataset and return other mldr dataset. You can use them as needed.

Here, a overview of the pre-processing methods:

[2]Requires the randomForest package.

```r
# Fill sparce data
mdata <- fill_sparce_mldata(toyml)

# Remove unique attributes
mdata <- remove_unique_attributes(toyml)

# Remove the attributes "iatt8", "iatt9" and "ratt10"
mdata <- remove_attributes(toyml, c("iatt8", "iatt9", "ratt10"))

# Remove labels with less than 10 positive or negative examples
mdata <- remove_skewness_labels(toyml, 10)

# Remove the labels "y2" and "y3"
mdata <- remove_labels(toyml, c("y2", "y3"))

# Remove the examples without any labels
mdata <- remove_unlabeled_instances(toyml)

# Replace nominal attributes
mdata <- replace_nominal_attributes(toyml)

# Normalize the predictive attributes between 0 and 1
mdata <- normalize_mldata(mdata)
```

4. Sampling

4.1 Subsets

If you want to create a specific or a random subset of a dataset, you can use the methods `create_subset` and `create_random_subset`, respectively. In the first case, you should specify which rows and optionally attributes, do you want. In the second case, you just define the number of instances and optionally the number of attributes.

```r
# Create a subset of toyml dataset with the even instances and the first five attributes
mdata <- create_subset(toyml, seq(1, 100, 2), 1:5)

# Create a subset of toyml dataset with the ten first instances and all attributes
mdata <- create_subset(toyml, 1:10)

# Create a random subset of toyml dataset with 30 instances and 6 attributes
mdata <- create_random_subset(toyml, 30, 6)

# Create a random subset of toyml dataset with 7 instances and all attributes
mdata <- create_random_subset(toyml, 7)
```

4.2 Holdout

To create two or more partitions of the dataset, we use the method `create_holdout_partition`. The first argument is a mldr dataset, the second is the size of partitions and the third is the partition method. The options are: `random`, `iterative` and `stratified`. The `iterative` is a stratification by label and the `stratified` is a stratification by labelset. The return of the method is a list with the names defined by the second parameter. See some examples:

```r
# Create two equal partitions using the 'iterative' method
toy <- create_holdout_partition(toyml, c(train=0.5, test=0.5), "iterative")
## toy$train and toy$test is a mldr object

# Create three partitions using the 'random' method
toy <- create_holdout_partition(toyml, c(a=0.4, b=0.3, c=0.3))
## Use toy$a, toy$b and toy$c

# Create two partitions using the 'stratified' method
toy <- create_holdout_partition(toyml, c(0.6, 0.4), "stratified")
## Use toy[[1]] and toy[[2]]
```

4.3 k-Folds

Finally, to run a k-fold cross validation we can use the `create_kfold_partition`. The return of this method is an object of type `kFoldPartition` that will be used with the method `partition_fold` to create the datasets:

```r
# Create 3-fold object
kfcv <- create_kfold_partition(toyml, k=3, "iterative")
result <- lapply(1:3, function (k) {
  toy <- partition_fold(kfcv, k)
  model <- br(toy$train, "RF")
  predict(model, toy$test)
})

# Create 5-fold object and use a validation set
kfcv <- create_kfold_partition(toyml, 5, "stratified")
result <- lapply(1:5, function (k) {
  toy <- partition_fold(kfcv, k, has.validation=TRUE)
  model <- br(toy$train, "RF")

  list(
    validation = predict(model, toy$validation),
    test = predict(model, toy$test)
  )
})
```

5. Classification Methods

The multi-label classification is a supervised learning task that seeks to learn and predict one or more labels together. This task can be grouped in: problem transformation and algorithm adaptation. Next, we provide more details about the methods and their specifities.

5.1 Transformation methods and Base Learners

The transformation methods require a base learner (binary or multi-class) and use their predictions to compose the multi-label result. In the **utiml** package there are some default base learners that are accepted, but if you need another you can easily developement your own[3].

[3]see the section *Create a new base Learner*

Each base learner requires a specific package, you need to install manually this packages, because they are not installed together with **utiml**. The follow base learners are supported:

Use	Name	Package	Call
CART	Classification and regression trees	rpart	rpart::rpart(...)
C5.0	C5.0 Decision Trees and Rule-Based Models	C50	C50::C5.0(...)
J48	Java implementation of the C4.5	RWeka and rJava	RWeka::J48(...)
KNN	K Nearest Neighbor	kknn	kknn::kknn(...)
MAJORITY	Majority class prediction	-	-
NB	Naive Bayes	e1071	e1071::naiveBayes(...)
RANDOM	Random prediction	-	-
RF	Random Forest	randomForest	randomForest::randomForest(...)
SVM	Support Vector Machine	e1071	e1071::svm(...)

To realize a classification first is necessary create a multi-label model, the available methods are:

Method	Name	Approach
br	Binary Relevance (BR)	one-agains-all
brplus	BR+	one-agains-all; stacking
cc	Classifier Chains	one-agains-all; stacking
ctrl	ConTRolled Label correlation exploitation (CTRL)	one-agains-all; binary-ensemble
dbr	Dependent Binary Relevance (DBR)	one-agains-all; stacking
ebr	Ensemble of Binary Relevance (EBR)	one-agains-all; ensemble
ecc	Ensemble of Classifier Chains (ECC)	one-agains-all; ensemble; stacking
mbr	Meta-Binary Relevance (MBR or 2BR)	one-agains-all; stacking
ns	Nested Stacking (NS)	one-agains-all; stacking
prudent	Pruned and Confident Stacking Approach (Prudent)	one-agains-all; binary-ensemble; stacking
rdbr	Recursive Dependent Binary Relevance (RDBR)	one-agains-all; stacking

The first and second parameters of each multi-label method is always the same: The multi-label dataset and the base classifier, respectively. However, they may have specific parameters, examples:

```
#Classifier chain with a specific chain
ccmodel <- cc(toyml, "J48", chain = c("y5", "y4", "y3", "y2", "y1"))

# Ensemble with 5 models using 60% of sampling and 75% of attributes
ebrmodel <- ebr(toyml, "C5.0", m = 5, subsample=0.6, attr = 0.75)
```

Beyond the parameters of each multi-label methods, you can define the parameters for the base method, like this:

```
# Specific parameters for SVM
brmodel <- br(toyml, "SVM", gamma = 0.1, scale=FALSE)

# Specific parameters for KNN
ccmodel <- cc(toyml, "KNN", c("y5", "y4", "y3", "y2", "y1"), k=5)

# Specific parameters for Random Forest
ebrmodel <- ebr(toyml, "RF", 5, 0.6, 0.75, proximity=TRUE, ntree=100)
```

After build the model, To predict new data use the `predict` method. Here, some predict methods require specific arguments and you can assign arguments for the base method too. For default, all base learner will

predict the probability of prediciton, then do not use these parameters. Instead of, use the `probability` parameter defined by the multi-label prediction method.

```
# Predict the BR model
result <- predict(brmodel, toyml)

# Specific parameters for KNN
result <- predict(ccmodel, toyml, kernel="triangular", probability = FALSE)

# Specific parameters for ebr predict method
result <- predict(ebrmodel, toyml, vote.schema = "avg", probability = TRUE)
```

An object of type `mlresult` is the return of predict method. It always contains the bipartitions and the probabilities values. So you can use: `as.bipartition`, `as.probability` and `as.ranking` for specific values.

5.2 Algorithm adapatation

Any method available yet!

5.3 Seed and Multicores

Almost all multi-label methods can run in parallel, but it requires the installation of *parallel* package. The train and prediction methods receive a parameter called `cores` that specify the number of cores used to run the method. For some multi-label methods are not possible running in multi-core, then read the documentation of each method, for more details[4].

```
# Running Binary Relevance method using 4 cores
brmodel <- br(toyml, "SVM", cores=4)
prediction <- predict(brmodel, toyml, cores=4)
```

If you need of reproducibility, you can set a specific seed:

```
# Running Binary Relevance method using 4 cores
brmodel <- br(toyml, "SVM", cores=4, seed=1984)
prediction <- predict(brmodel, toyml, seed=1984, cores=4)
```

6. Thresholds

The threshold methods receive a `mlresult` object and return a new `mlresult`, except for `scut` that returns the threshold values. These methods, change mainly the bipartitions values using the probabilities values.

```
# Use a fixed threshold for all labels
newpred <- fixed_threshold(prediction, 0.4)

# Use a specific threshold for each label
newpred <- fixed_threshold(prediction, c(0.4, 0.5, 0.6, 0.7, 0.8))

# Use the MCut approch to define the threshold
newpred <- mcut_threshold(prediction)
```

[4]Base learner `J48` do not work very well with multicore

```r
# Use the PCut threshold
newpred <- pcut_threshold(prediction, ratio=0.65)

# Use the RCut threshold
newpred <- rcut_threshold(prediction, k=3)

# Choose the best threshold values based on a Mean Squared Error
thresholds <- scut_threshold(prediction, toyml, cores = 5)
newpred <- fixed_threshold(prediction, thresholds)

#Predict only the labelsets present in the train data
newpred <- subset_correction(prediction, toyml)
```

7. Evaluation

To evaluate multi-label models you can use the method `multilabel_evaluate`. There are two ways of call this method:

```r
toy <- create_holdout_partition(toyml)
brmodel <- br(toy$train, "SVM")
prediction <- predict(brmodel, toy$test)

# Using the test dataset and the prediction
result <- multilabel_evaluate(toy$test, prediction)
print(round(result, 3))
```

```
##          accuracy average-precision          coverage                F1
##             0.583             0.858             1.900             0.721
##      hamming-loss         macro-AUC          macro-F1   macro-precision
##             0.213             0.640             0.337             0.297
##      macro-recall       margin-loss         micro-AUC          micro-F1
##             0.392             0.933             0.804             0.729
##   micro-precision      micro-recall         one-error         precision
##             0.729             0.729             0.133             0.733
##      ranking-loss            recall   subset-accuracy
##             0.167             0.794             0.100
```

```r
# Build a confusion matrix
confmat <- multilabel_confusion_matrix(toy$test, prediction)
result <- multilabel_evaluate(confmat)
print(confmat)
```

```
## Multi-label Confusion Matrix
##
## Absolute Matrix:
## -------------------------------------
##              Expected_1 Expected_0 TOTAL
## Prediction_1         43         16    59
## Predicion_0          16         75    91
## TOTAL                59         91   150
##
```

```
## Proportinal Matrix:
## -------------------------------------
##              Expected_1 Expected_0 TOTAL
## Prediction_1      0.287      0.107 0.393
## Predicion_0       0.107      0.500 0.607
## TOTAL             0.393      0.607 1.000
##
## Label Matrix
## -------------------------------------
##    TP FP FN TN Correct Wrong  %TP  %FP  %FN  %TN %Correct %Wrong
## y1  0  0  6 24      24     6 0.00 0.00 0.20 0.80     0.80   0.20
## y2 23  6  1  0      23     7 0.77 0.20 0.03 0.00     0.77   0.23
## y3  0  1  6 23      23     7 0.00 0.03 0.20 0.77     0.77   0.23
## y4 20  9  0  1      21     9 0.67 0.30 0.00 0.03     0.70   0.30
## y5  0  0  3 27      27     3 0.00 0.00 0.10 0.90     0.90   0.10
##    MeanRanking MeanScore
## y1        4.30      0.15
## y2        1.47      0.72
## y3        4.53      0.15
## y4        1.63      0.69
## y5        3.07      0.20
```

The confusion matrix summarizes a lot of data, and can be merged. For example, using a k-fold experiment:

```
kfcv <- create_kfold_partition(toyml, k=3)
confmats <- lapply(1:3, function (k) {
  toy <- partition_fold(kfcv, k)
  model <- br(toy$train, "RF")
  multilabel_confusion_matrix(toy$test, predict(model, toy$test))
})
result <- multilabel_evaluate(merge_mlconfmat(confmats))
```

Its possible choose which measures will be computed:

```
# Example-based measures
result <- multilabel_evaluate(confmat, "example-based")
print(names(result))
```

```
## [1] "accuracy"        "F1"              "hamming-loss"    "precision"
## [5] "recall"          "subset-accuracy"
```

```
# Subset accuracy, F1 measure and hamming-loss
result <- multilabel_evaluate(confmat, c("subset-accuracy", "F1", "hamming-loss"))
print(names(result))
```

```
## [1] "F1"              "hamming-loss"    "subset-accuracy"
```

```
# Ranking and label-basedd measures
result <- multilabel_evaluate(confmat, c("label-based", "ranking"))
print(names(result))
```

```
##  [1] "average-precision" "coverage"          "macro-AUC"
##  [4] "macro-F1"          "macro-precision"   "macro-recall"
##  [7] "margin-loss"       "micro-AUC"         "micro-F1"
## [10] "micro-precision"   "micro-recall"      "one-error"
## [13] "ranking-loss"
```

```
# To see all the supported measures you can try
multilabel_measures()
```

```
##  [1] "accuracy"          "all"               "average-precision"
##  [4] "bipartition"       "coverage"          "example-based"
##  [7] "F1"                "hamming-loss"      "label-based"
## [10] "macro-AUC"         "macro-based"       "macro-F1"
## [13] "macro-precision"   "macro-recall"      "margin-loss"
## [16] "micro-AUC"         "micro-based"       "micro-F1"
## [19] "micro-precision"   "micro-recall"      "one-error"
## [22] "precision"         "ranking"           "ranking-loss"
## [25] "recall"            "subset-accuracy"
```

8. How to Contribute

The **utiml** repository is available on (https://github.com/rivolli/utiml). If you want to contribute with the development of this package, contact us and you will be very welcome.

Please, report any bugs or suggestions on CRAN mail or git hub page.