Collective and Semi-supervised classification

Bernhard Pfahringer
Kurt Driessens
Peter Reutemann

February 27, 2015

Contents

List of Figures

Chapter 1

Command-line

The command-line for collective classifiers is similar to the ones of normal WEKA classifiers. Here is a list of the general options:

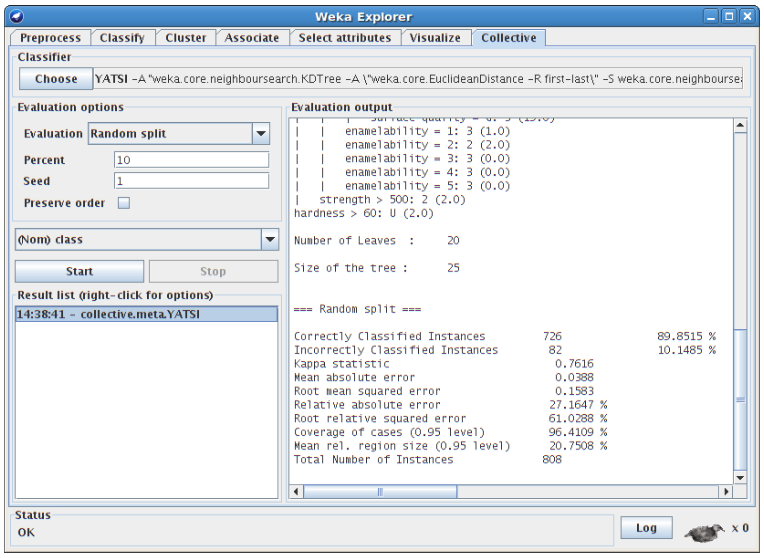
```
-t <file>
Training set

-T <file>
Test set

-c <index>
Class index (1-based or 'first' or 'last')
default: last

-l <file>
Serialized model to use, requires '-T' but does not allow '-t'

-d <file>
Serializes a built model, not available when using cross-validation

-x <folds>
The number of folds for cross-validation (if '-T' is omitted)
Use -1 for leave-one-out cross-validation
default: 10

-swap-folds
Swaps train and test folds in case of cross-validation

-s <number>
The seed value for randomization
default: 1

-split-percentage <0-100>
Splits the training set into train and test set with the
specified amount set aside for training

-preserve-order
Turns off randomization when using '-split-percentage'
```

Chapter 2

Explorer

Since the collective classifiers should get built using labeled and unlabeled dataset, they cannot be run in the usual *Classify* tab in the Explorer. Hence the package provides a custom tab, to perform experiments with the collective classifiers, called *Collective*. It is a lot simpler compared to the *Classify* tab, but still offers the following evaluation options:

- cross-validation[1]
- percentage split (random or order preserved)
- supplied test set



Classifier: YATSI -A "weka.core.neighboursearch.KDTree -A \"weka.core.EuclideanDistance -R first-last\" -S weka.core.neighbourse

Evaluation: Random split
Percent: 10
Seed: 1
Preserve order: ☐
(Nom) class

Result list: 14:38:41 - collective.meta.YATSI

Status: OK

```
|   |   enamelability = 1: 3 (1.0)
|   |   enamelability = 2: 2 (2.0)
|   |   enamelability = 3: 3 (0.0)
|   |   enamelability = 4: 3 (0.0)
|   |   enamelability = 5: 3 (0.0)
|   strength > 500: 2 (2.0)
hardness > 60: U (2.0)

Number of Leaves  :     20

Size of the tree :      25


=== Random split ===

Correctly Classified Instances         726               89.8515 %
Incorrectly Classified Instances        82               10.1485 %
Kappa statistic                          0.7616
Mean absolute error                      0.0388
Root mean squared error                  0.1583
Relative absolute error                 27.1647 %
Root relative squared error             61.0288 %
Coverage of cases (0.95 level)          96.4109 %
Mean rel. region size (0.95 level)      20.7508 %
Total Number of Instances              808
```

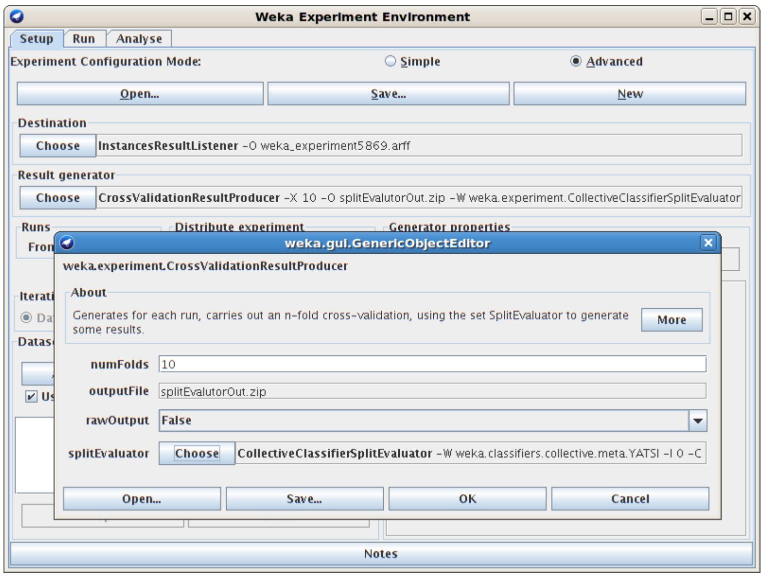
Figure 2.1: Split evaluator for the collective classifiers.

[1]You can invert training and testing folds as well, i.e., for 10-fold CV you train with 10% of the data and test it against 90% instead of the normal 90/10.

Chapter 3

Experimenter

In order to run experiments with the collective classifiers, you need to use the advanced interface and select the `CollectiveClassifierSplitEvaluator` rather than the usual `ClassifierSplitEvaluator` (package `weka.experiment`). Figure 3.1 shows a cross-validation setup.



Experiment Configuration Mode: Simple [✓ Advanced]

Destination: InstancesResultListener -O weka_experiment5869.arff

Result generator: CrossValidationResultProducer -X 10 -O splitEvalutorOut.zip -W weka.experiment.CollectiveClassifierSplitEvaluator

weka.experiment.CrossValidationResultProducer

About: Generates for each run, carries out an n-fold cross-validation, using the set SplitEvaluator to generate some results.

numFolds: 10

outputFile: splitEvalutorOut.zip

rawOutput: False

splitEvaluator: CollectiveClassifierSplitEvaluator -W weka.classifiers.collective.meta.YATSI -I 0 -C

Figure 3.1: Split evaluator for the collective classifiers.

Some example experiment setups used in the literature ([1], [2]) are available here:

```
src/site/resources/experiments
```

The accompanying datasets are located here:

```
src/site/resources/datasets
```

Chapter 4

API and classifiers

4.1 Classes and Interfaces

Source code can be found at this location:

`https://code.google.com/p/collective-classification/`

All classifiers, interfaces and helper classes are located below the following package:

`weka.classifiers.collective`

There exist two Interfaces for collective classifiers:

- `CollectiveClassifier`
  general interface for setting test/training set, number of folds in case no test file is provided and the train is then split (new train = 1/folds of train, new test = (folds - 1)/folds of train)
- `RestartableCollectiveClassifier`
  if the classifier needs restarts and iterations (inherits from `CollectiveClassifier` and `Comparable`)

The following abstract classifiers all implement the CollectiveClassifer interface:

- `CollectiveRandomizableClassifier`
  derived from RandomizableClassifier, for simple collective classifiers
- `CollectiveRandomizableSingleClassifierEnhancer`
  derived from `RandomizableSingleClassifierEnhancer`, for meta classifiers that take one classifier as input
- `CollectiveRandomizableMultipleClassifiersCombiner`
  derived from `RandomizableMultipleClassifiersCombiner`, for meta classifiers that take several classifiers as input

4.2 Other classes

- `CollectiveWrapper`
  a simple Wrapper for non-collective classifiers. used for comparison.

- `CollectiveInstances`
  used by some collective classifiers to produce the training and test file, as well as used for initializing the labels and flipping them (in case they are or type `RestartableCollectiveClassifier`)
- `Splitter`
  splits the training set into two, according to the specified number of folds and whether it should be inverted (cf. `StratifiedRemoveFolds` filter)
- `Flipper, ..., FlipHistory`
  used in classifiers that flip labels during iterations to improve performance

4.3 Classifiers

4.3.1 SimpleCollective

```
extends CollectiveRandomizableSingleClassifierEnhancer
implements RestartableCollectiveClassifier
package weka.classifiers.collective.meta
```

- trains with training + "new" test set
- the "new" test set is randomly labeled according to class distribution in training set
- three ways to get prediction model
  1. random walk (current model flips labels) / last model for prediction
  2. random walk (current model flips labels) / best model for prediction
  3. hill climbing (best model so far flips labels) / best model for prediction
- three ways of comparing models
  1. RMS on training data
  2. RMS on test data (lower prediction of binary problem is taken as error)
  3. overall RMS (normalized train+test RMS)

4.3.2 AdvancedCollective

```
extends SimpleCollective
package weka.classifiers.collective.meta
```

- splits test file into two T1 and T2
- T1 and T2 are randomly labeled according to class distribution in training set
- does two training runs
  - trains with training + T1 and tests against T2
  - trains with training + T2 and tests against T1
- the root mean squared error of both tests is averaged for the overall performance
- prediction model and comparing models as with `SimpleCollective`

4.3.3 TwoStageCollective

`extends CollectiveRandomizableMultipleClassifiersCombiner`
`implements RestartableCollectiveClassifier`
`package weka.classifiers.collective.meta`

- takes 2 base classifiers
- 1. classifier is trained on training set
- 1. classifier does the initial labeling of the test set and creates the "new" test set
- 2. classifier is built upon training and the "new" test set
- prediction model and comparing models as with `SimpleCollective`

4.3.4 CollectiveIBk

`extends CollectiveRandomizableClassifier`
`package weka.classifiers.collective.lazy`

- uses internally `weka.classifiers.lazy.IBk` to determine the best K on the training set
- builds for all instances from the test set a neighborhood consisting of K instances (the previously determined K value) from the pool of train and test set (either a nave search over the complete set of instances or a `KDTree` is used to determine the neighbors)
- all neighbors in such a neighborhood (`CollectiveIBkNeighbors`) are sorted according to their distance to the test instance they belong to
- the neighboorhoods are sorted according to their "rank", where "rank" means the difference occurrences of the two classes in the neighborhood. e.g. class A appears 5 times in the neighborhood, class B 3 times and 2 unlabeled instances, which leads to a rank of (5 - 3) = 2.
- for all (still unlabeled) test instances with the highest rank the class label is determined by majority vote or in case of a tie the first class. due to this labeling of an instance of the test set other neighborhoods change their rank (since they might contain this test instance)
- this is done until no further unlabeled test instances remain
- classification is done by returning the class label of the instance that is about to be classified

Note

- the `ReplaceMissingValues` filter is applied to data sets, since the Euclidean-Distance returns the highest possible distance between two instances if they both have a missing value at the same position. an instance with at least one missing value then never has the distance 0 to itself!
- formerly distinct instances may become equal due to the `ReplaceMissingValues` filter. In order to get the correct classification for an instance the original instance is saved along the one with the filter applied to. This ensures that the correct instance is found during the search, and therefore the correct classification.

4.3.5 CollectiveWrapper

extends `CollectiveRandomizableSingleClassifierEnhancer`
package `weka.classifiers.collective.meta`

- takes any WEKA classifier as input
- used to enable comparison between "real" collective classifiers and other WEKA classifiers

4.3.6 CollectiveForest

extends `CollectiveRandomizableClassifier`
package `weka.classifiers.collective.trees`

- uses `weka.classifiers.trees.RandomTree` as base classifier, just like `weka.classifiers.trees.RandomForest`
- this classifier divides the test set into folds containing the same number of elements
- if the number of folds is 0 then one has a `weka.classifiers.trees.RandomForest` classifier
- if bagging is used (simulates the
- weka.classifiers.meta.Bagging Meta-Classifier) the out of bag error is computed in addition
- the first iteration trains on the original training set and generates the distribution for all instances in the test set
- the best instances are then added to the original training set (the number of instances being chosen is the same as in a fold)
- the next iteration trains with the new "enriched" training set and generates then the distributions for the remaining instances in the test set

4.3.7 CollectiveTree

extends `CollectiveRandomizableClassifier`
package `weka.classifiers.collective.trees`

- similar to the `weka.classifiers.trees.RandomTree` classifier
- splits the attribute at that position that divides the current subset of instances (of training and test instances) into (roughly) 2 halves
  1. nominal attributes

	a	b	c	d	e	
p	10	8	2	0	2	p=pos, n=neg, ?=inst. from test set
n	1	2	7	0	0	
?	10	20	5	10	5	
	11/13	9/12	3/11	1/2	3/4	dist: $(p+1)/(p+n+2)$
	1	2	4	3	2	ranking
	21	30	14	10	7	overall instances

  order according to rank and equally ranked ones are merged

```
->          21  37   10  14
-> split:   21  37 | 10  14
-> bins:   (a b e)  (d e)
```

  2. numeric attributes
  find numerical value of attribute that splits the instances into two equal sized halves

   3. splitting on an attribute that contains missing values:
      split training instances according to their class into these two branches, test instances randomly according to the class-proportion previously determined
- the tree is stopped from growing, if one of the following conditions is met
   1. only training instances would be covered (the labels for these instances are already known!)
   2. only test instances in the leaf $\rightarrow$ taking the distribution from the parent node
   3. only training instances of one class $\rightarrow$ all test instances are considered to have this class
- calculation of distributions (for complete set or a subset)
   1. class distribution
      according to the weights in the training set the weights are summed up and normalized
   2. attribute distribution
      nominal: for each distinct value the weights are summed up, then normalized
      numeric: binary split based on median is calculated and then the weights are summed up for the two bins, and finally normalized

4.3.8 CollectiveEM

`extends CollectiveRandomizableSingleClassifierEnhancer`
`package weka.classifiers.collective.meta`

- the first step is to train a classifier on the training set
- the trained classifier is used to determine the distributions and hence the class labels for the test set
- the classified test set is then duplicated and then the one set gets the labels and weights of the first class, the other one of the second class; together with the training set those two sets present the new training set
- the classifier determines the distributions always on the original test set
- we keep track of all the distributions we get for each instance in the test set ("mean")
- the new weight for an instance is calculated as follows
  $mean_{n+1} = q \times mean_n + (1 - q) \times x_n$ with x_n as the current distribution from the classifier

optional

- the training set is updated like the test set (i.e. splitting in two and updating labels/weights)
- randomizing the data before it is presented to the clasifier

4.3.9 CollectiveWoods

`extends CollectiveForest`
`package weka.classifiers.collective.trees`

- works like `CollectiveForest`
- uses `CollectiveTree` instead of `RandomTree`

4.3.10 LLGC

`extends CollectiveRandomizableClassifier`
`package weka.classifiers.collective.functions`

- works similar to spreading activation networks
- based on [1]
- in addition to the paper, the number of atttributes can be used as an additional normalization factor in the calculation of the affinity matrix, i.e., dividing the distance between two instances by $2\sigma^2 \times \#attributes$ instead of $2\sigma^2$.

4.3.11 YATSI

`extends CollectiveRandomizableSingleClassifierEnhancer`
`package weka.classifiers.collective.meta`

- YATSI – Yet Another Two Stage Idea [2]
- The first step is to train a classifier on the training set.
- Predictions for unlabelled instances are performed using k-nearest neighbour approach, using both the training instances and the test-instances as labeled by the classfier obtained in Step 1. (the instance under investigation is left out)
- To reduce the influence of the test-set, a $weight = p \times (\#traininstances / \#testinstances)$ is given to each of the test instances. (with p a user defined parameter that can be used to raise or lower the importance of the test-set)
- The k is a user supplied parameter, but one can also use IBk's (`weka.classifiers.lazy.IBk`) internal cross-validation to determine the optimal k on the training set
- To investigate the impact of weighting of the unlabeled instances, one can also disable any weighting
- The `ReplaceMissingValues` filter is applied to data sets, since the Euclidean-Distance returns the highest possible distance between two instances if they both have a missing value at the same position. an instance with at least one missing value then never has the distance 0 to itself!

4.3.12 FilteredCollectiveClassifier

`extends CollectiveRandomizableSingleClassifierEnhancer`
`package weka.classifiers.collective.meta`

- takes, just like the `weka.classifiers.meta.FilteredClassifier`, a filter and a classifier as input
- the filter is only "trained" on the training set, but applied to all instances, train, test and other instance objects that get passed to the `FilteredCollectiveClassifier`, before they are passed on to the collective base classifier

4.3.13 Chopper

`extends CollectiveRandomizableMultipleClassifiersCombiner`
`package weka.classifiers.collective.meta`

- only works on two-class-problems
- similar to `CollectiveForest`
- **first stage:** trains a base classifier on the training set and determines the distributions for all instances in the test set; the difference between the confidences is used as ranking criterium
- **second stage:** the ranked test set is divided in a number of equal sized folds and the best fold is added to the current training set, which is again input for another classifier
- Adding the best fold is done until no more instances are left or the cut-off fold number is reached (i.e., only x number of best folds will be added, in order to keep the classifier from deteriorating with not-so-good instances)
- if there is more than one classifier defined for the second stage, then these classifiers will be used in a rotating manner

4.3.14 Weighting

`extends CollectiveRandomizableMultipleClassifiersCombiner`
`package weka.classifiers.collective.meta`

- only works on two-class-problems
- all base classifiers must implement the `weka.core.WeightedInstancesHandler` interface
- **initialization step:** the first classifier is trained on the training set plus the test set (all instances of this set have a weight of 0) and used to set the initial labels of the test set
- **training step:** if there are more than 2 classifiers specified, classifiers 2 to n are then used in turns to train on the combined dataset (training set plus weighted and labeled test set)
- the training step is performed x times (defined by the user), but the training can also be stopped after a certain cut-off step (user-defined again). In each training step the weights of the test instances are calculated like this: $\#step/\#steps$. The labels of the test instances are determined anew in each training step, by using the previously built classifier (if step 1 this is the classifier from the initialization step).

Bibliography

[1] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schoelkopf. *Learning with local and global consistency.* In 18th Annual Conf. on Neural Information Processing Systems, 2003.
`http://research.microsoft.com/en-us/um/people/denzho/papers/LLGC.pdf`

[2] Kurt Driessens, Peter Reutemann, Bernhard Pfahringer, and Claire Leschi (2006). *Using weighted nearest neighbor to benefit from unlabeled data.* Advances in Knowledge Discovery and Data Mining, 10th Pacific-Asia Conference, PAKDD 2006, LNCS 3918, 60-69.
`http://dx.doi.org/10.1007/11731139_10`