LyS Research Group, Departamento de Computación

MIOPIA-SD

Sistema de Minería de Opiniones en Múltiples Idiomas mediante Análisis Sintáctico de Dependencias

User manual

David Vilares Calvo

Miguel A. Alonso Pardo

Carlos Gómez Rodríguez

October 10, 2014

Contents

1 Introduction

MIOPIA-SD (Sistema de Minería de Opiniones en Múltiples Idiomas mediante Análisis Sintáctico de Dependencias) is a library which provides you the capabilities for analysing the perception of the public with respect to a product, service, event or a celebrity, given a collection of related messages.

This manual is a guide to describe how to install and exploit this software, including how to run the supervised and unsupervised models and the corresponding input formats. Thus, is a practical-oriented manual, where the theory of the models is not included. The user should be familiar with supervised an unsupervised learning approaches for sentiment analysis. He also should be familiar with the basics of language programming in Python and the paradigm of Oriented Object Programming.

MIOPIA-SD has a web page (`miopia.grupolys.org`) where an online demo and an API are available too. It is important to point out that this page will allow access to MIOPIA-SD as well as improvements and versions of MIOPIA developed after the writing of this manual. Therefore, there may be differences between the results obtained by running MIOPIA-SD and those obtained by accesssing the web page.

2 Terms and conditions

This system 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 3 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 MIOPIA. If not, see ¡http://www.gnu.org/licenses/¿.

3 Installing the library

MIOPIA is a library for Sentiment Analysis purposes written in Python, so it can be used as long as the user has installed a Python language interpreter. However, the configuration files included with MIOPIA are intended for UNIX systems and the package provided with the

guide is intended for UNIX platforms too. The user is responsible to adapt them to other platforms, such as Windows.

3.1 Dependencies and resources

The system requirements and their dependencies can be found in the README.txt file, although they are also detailed below these lines:

- You should have installed a Python interpreter (version 2.7.*).
- You are required to install Setuptools (which is needed to install MIOPIA). It can be downloaded from `https://pypi.python.org/pypi/setuptools`.
- You should download and install the Natural Language Toolkit (NLTK) from `http://nltk.org/install.html` and the *nltk_data* from `http://nltk.org/data.html` (in particular, the tokenizer module). Do not modify their default location.
- MaltParser version 1.7.* is required to run the parsers provided by MIOPIA. It can be downloaded from `http://www.maltparser.org/`. Install it in the desired directory (e.g. /opt/maltparser/maltparser-1.7.2/).
- The WEKA data mining software is needed to be able to train and run a supervised model. It can be downloaded from `http://www.cs.waikato.ac.nz/ml/weka/`. Both 3.6.* and 3.7.* (developer) versions are valid. Install it in your desired directory (e.g. /opt/weka/weka-3-7-10/).
- Installing *textblob* is required to use the part-of-speech tagger for English language included with MIOPIA. `http://textblob.readthedocs.org/en/dev/` and *sudo pip-install -U textblob-aptagger* on your command line.

After downloading and installing all these dependencies, unzip the file *miopia-1.0.0.tar.gz* included with this guide. Inside the `miopia-1.0.0/miopia/` directory you will find a file named *miopia-1.0.0_conf.yaml*. If you have followed our suggestions about where to install the WEKA and MaltParser jar's, you do not need to edit this file. In other case, adapt the *path_weka* and *path_maltparser* entries to indicate the paths where you have located the WEKA and MaltParser jar's. Next, use the command line as follows:

cd miopia-1.0.0 # Change to the miopia-1.0.0 directory

sudo python setup.py install # It will install MIOPIA in the default directory for Python libraries: `/usr/local/lib/python2.7/dist-packages/`

Now MIOPIA should be installed in your computer and available to use from any Python program. The configuration file, *miopia-1.0.0_conf.yaml*, will be located inside the */etc/* folder.

3.2 Other requirements

- CPU and memory requirements depends on the complexity of the model that the user plans to build. The memory requirements of the unsupervised model are strongly determined by the dependency parser (Java software). In this respect, any machine able to run a Java Runtime Environment should be capable to run this kind of model. On the other hand, the memory bottle neck of the supervised models mainly relies on the vector space (also called feature space), which depends on the number of features that are used to feed a particular model.

- MIOPIA is intended to work with different encodings. However, in order to avoid problems, we encourage to the users to employ *utf-8* encoding for all their files. If they present a different encoding, change it before starting to work with them.

4 The MIOPIA library

MIOPIA contains a number of packages with different purposes. A detailed description for each class and method can be found in the API documentation provided with the code. The list below these lines describes the most relevant functionalities and classes of the library:

- *adapter*: A wrapper to represent instances according to the WEKA data format (ARFF format, see `http://www.cs.waikato.ac.nz/ml/weka/arff.html` for the details.).

- *analyzer*: This package contains the functionalities responsible of analysing the content of the texts.

  - *SentimentAnalyzer*: A rule-based classifier which is able to obtain the semantic orientation of a text. A detailed explanation of this system is provided in Section 6. The analyzer evaluates the *SentimentDependencyGraph's* provided by the *parser* package and determines the semantic orientation based on linguistic rules.

  - *analyzer.counter*: This sub-package provides the capabilities to create different instances able to collect different features present in a text. It includes *counters* for: n-grams, generalised n-grams, dependency triplets or generalised dependency triplets, among other ones.

- *classifier*: The package is a wrapper for different classification strategies. In particular, it contains a wrapper for the *AttributeSelectedClassifier* available in WEKA, called *MetaStrategy*, which make possible to train a WEKA classifier, by firstly applying a feature selection filter (*e.g.* information gain or chi-squared). MIOPIA supports at the moment up to four different classifiers: *Naives Bayes*, *J48*, *LibLinear* and *SMO*.

- *parser*: The package is responsible of: (1) loading/running the parser models using MaltParser, (2) reading/writting file in ConLL-X format and (3) creating instances of the *SentimentDependencyGraph* class that will be analysed by the supervised and unsupervised systems.

- *preparator*: This package contains the classes that allow to carry out the natural language pipeline, including sentence and word tokenisation, and interacts with the tagger to obtain the part-of-speech tags of each term of the text.

- *preprocessor*: It provides the capabilities to carry out an *ad-hoc* preprocessing to web texts, including preprocesors such as: *EmoticonPreprocessor* for smileys, *HashTagPreprocessor* and *TwitterUserNamePreprocessor* for Twitter slang (hashtags and usernames) or a *URLPreprocessor* to normalise links.

- *tagger*: This package make possible to train a tagger based on the Brill implementation provided by the NLTK.

5 Creating a natural language processing pipeline for web text classification

Before creating either a supervised or an unsupervised model, it will be required to instantiate some Natural Language Processing (NLP) tools which will serve as the starting point to analyse the texts. They are described below these lines.

5.1 Preprocessor

To create a naive preprocessor in MIOPIA you just need to import:

```
from miopia.preprocessor.PreProcessor import PreProcessor
```

and instantiate and object as follows:

```
preprocessor_es = PreProcessor() #for Spanish, PreProcessor(lang='es') is also valid.
preprocessor_en = PreProcessor(lang='en') #for English.
```

This will provide a preprocessor with capabilities such as: identification of composed words (according to path referred at the parameter *path_composedwords* of the file *miopia-1.0.0_conf.yaml*), replacement of frequent abbreviations, (according to path referred at the parameter *path_abbreviations* of the file *miopia-1.0.0_conf.yaml*) or unification of different number representations. Composed and frequent abbreviations can be also specified via parameters. For example:

```
cw_dict = {'menos que': 'menos_que'}
ab_dict = {'x': 'por'}
preprocessor_es = PreProcessor(composite_words=cw_dict,
abbreviations=ab_dict,
lang="es"))
```

It is possible to create more complex preprocessors to handle different web phenomena, including: URL's, emoticons or the slang of particular social networks (such as usernames or hashtags in Twitter). MIOPIA preprocessors are implemented according to the Decorator pattern design. Thus, to instantiate a preprocessor able to manage the particularities described above these lines, we will need to write the following piece of code:

```
decorated_preprocessor_es = URLPreProcessor(
                            EmoticonPreProcessor(
                            HashTagProcessor(
                            TwitterUserNameProcessor(
                            PreProcessor(composite_words=cw_dict,
                                         abbreviations=ab_dict,
                                         lang="es")))))
```

Some of these preprocessors have different configuration options. Read the API documentation provided with the code for a detailed explanation.

5.2 Lexical processor

Before obtaining the syntactic structure of the texts via dependency parsing, we need to process them by applying tokenisation and part-of-speech tagging.

Sentence and word tokenisation

MIOPIA relies on the tokenizers provided by the NLTK to split texts into sentences and sentences into words. To obtain the sentences of a given text, you can use the following code:

```
#Sentence tokeniser for Spanish language
sentence_tokenizer_es = nltk.data.load('tokenizers/punkt/spanish.pickle')
#Sentence tokeniser for English language
sentence_tokenizer_en = nltk.data.load('tokenizers/punkt/english.pickle')
```

To obtain the words of a particular sentence, we encourage to the users to instantiate the following tokenisers:

```
#Word tokeniser for Spanish language
tokenizer_es = PunktWordTokenizer()
#Recommended word tokeniser for English language
tokeniser_en = TreebankWordTokenizer()
```

Users can rely on other tokenisers provided by the NLTK software (as long as they implement the same interface), although it may drop the performance of the next steps of the NLP pipeline.

Part-of-speech tagging

We provide a trained tagger (serialised) for Spanish language within MIOPIA. It can be loaded as follows:

```
tagger_es = pickle.load(open(
                        ConfigurationManager().getParameter(
                        "path_pickle_taggers")+"spanish_brill.pickle",'r'))
```

Users must rely on this tagger to create the unsupervised classifier (see Section 6), since some of the rules need of the tag set provided by this PoS-tagger. This tagger provides the same set of features that employs the parser for Spanish language included within MIOPIA.

With respect to the English system, it is needed to instantiate an object of the class *PerceptronTwitTagger*. This class is just a wrapper for a *PerceptronTagger* provided by the library *textblob_aptagger*, that you should have installed, as indicated in Section 3. The tagger uses the Penn Treebank tagset and it is compatible with the parser provided by MIOPIA for English language too:

```
from miopia.tagger.PerceptronTwitTagger import PerceptronTwitTagger
tagger_en = PerceptronTwitTagger()
```

Creating the lexical processor

Once we have instantiate the tokenisers and the part-of-speech tagger, we can create the instance of the *LexicalProcessor* that will be used by our sentiment classifiers.

```
lexical_processor = LexicalProcessor(sentence_tokenizer_es,
                                     tokenizer_es,
                                     tagger_es) #For Spanish
```

5.3 Parser

The *Parser* class acts as a Python wrapper for the MaltParser models. MIOPIA already includes a trained parser both for Spanish and English, so users don't need to train any dependency parser, just use the following code to have access to the parsers:

```
parser_es = Parser(lang='es') #For Spanish
parser_en = Parser(lang='en') #For English
```

6 The unsupervised system

Unsupervised systems involve the use of dictionaries where different kinds of words are tagged with their semantic orientation (SO). To classify polarity, these methods obtain the words present in a text and aggregate their SO in a given way. In contrast with machine learning approaches, semantic-based methods are more domain independent, although their performance can still vary from one domain to another.

MIOPIA provides a unsupervised system based on syntactic rules and it is only available for Spanish at the moment. Next section describe how to create a model following these approach and how to run them.

6.1 Creating an unsupervised model

After creating the NLP tools, it is possible to instantiate an object of the class *SentimentAnalyzer*:

```
from miopia.analyzer.Dictionary import Dictionary
from miopia.analyzer.SentimentAnalyzer import SentimentAnalyzer
from miopia.analyzer.AnalyzerConfiguration import AnalyzerConfiguration

#We create the dictionary:
#It gives access to the dictionaries included with MIOPIA
dictionary = Dictionary()
sentiment_analyzer =  SentimentAnalyzer(parser_es,
                       dictionary,
                       AnalyzerConfiguration(final_sentences_weight=1.),
                       preprocessor_es,
                       lexical_processor)
```

The class *AnalyzerConfiguration* manages the parameters to weight the relevance of different phenomena, such as negation, adversative subordinate clauses or the relevance of the final sentences of a message. For a detailed explanation, consult the API documentation provided with MIOPIA.

6.2 Running an unsupervised model

After creating the unsupervised classifier, we are ready to obtain the semantic orientation of a particular text. The *SentimentAnalyzer* class provides two main methods to obtain its semantic orientation:

- *analyze(self,text)*: It receives a string *text* as a parameter. This text will be preprocessed, tagged and parsed to then finally obtain the results.

```
# -*- coding: utf-8 -*-
...
text = "Esta máquina es muy barata, pero pésima".decode('utf-8')
graphs,sentiment_info = sentiment_analyzer.analyze(running_example)
```

- *analyze_from_conll(self,path_to_parsed_text)*: Since the parsing step is the bottle neck of the system, MIOPIA allows to analyze a parsed text in CoNLL format. We encourage to the users to employ this method when they are performing experiments to set up the optimal configuration of the classifier.

```
path_file = "/tmp/example.conll"
graphs,sentiment_info = sentiment_analyzer.analyze_from_conll(path_file)
```

In both cases, we obtain as a result a tuple containing: (1) a list of *SentimentDependencyGraph* (one dependency graph for each sentence) and (2) a *SentimentInfo* object, which contain the semantic orientation, among other information. To obtain the semantic orientation of the text:

```
sentiment_info.get_so()
```

7 The supervised system

Machine learning solutions involve building classifiers from a collection of annotated texts, where each text is usually represented as a bag-of-words. It is also common to include some linguistic-related processing for preparing features, such as lemmatisation, stemming or stop word removal. Classifiers of this kind perform well in the domain where they have been trained, but their accuracy drops markedly in other areas, because they are highly domain dependent.
MIOPIA provides the capabilities to extract a number of features, based on rich linguistic knowledge, from different kinds of texts. It also allows to train and run a supervised classifier, based on the WEKA data mining software. Users are responsible to obtain the corpus which will serve them to feed their models.

7.1 The MIOPIA data format

To train and run the MIOPIA supervised classifiers, users need to be familiar with the MIOPIA data format. It consist of a tabular file, preceded of a header. The header of the format, indicates the path to the directory where are stored the parsed files. It also considers the possibility of indicating the path of a directory which contains files with metainformation

about the text, although miopia-1.0.0 does not provide functionalities for mananing metadata-based features. Finally, it contains the token *DATA*, which indicates the beginning of the data set. The form of the header is as follows:

```
BASE_PATH_PARSED_FILES path_to_the_directory_of_parsed_files
BASE_PATH_METADATA_FILES path_to_the_directory_with_the_metadata_files
DATA
```

The content of the data set must be represented with a tabular format. Each row represents an instance of the data, where each column indicates:

1. The file identifier.
2. The category of the files. A *'?'*, if the MIOPIA data file represents the test set.
3. The raw text.
4. The path to the relative path of the parsed text file in the directory of the parsed files. A *'_'* if no *BASE_PATH_PARSED_FILES* provided.
5. The name of the relative path of the metadata information about the text. At the moment, users should fill it always with a *'_'*.

A simple example of the content of a MIOPIA data file could be (without including the paths to the parsed files):

```
BASE_PATH_PARSED_FILES
BASE_PATH_METADATA_FILES
DATA
file1\tP\tI like you\t_\t_
file2\tP\tThe camera of that mobile is amazing\t_\t_
file3\tN\tThree people died in an accident\t_\t_
```

The same example, but providing the paths to the parsed files:

```
BASE_PATH_PARSED_FILES /tmp/parsed-dir/
BASE_PATH_METADATA_FILES
DATA
file1\tP\tI like you\tP/file1.conll\t_
```

```
file2\tP\tThe camera of that mobile is amazing\tP/file2.conll\t_
file3\tN\tThree people died in an accident\tN/file3.conll\t_
```

7.2 Training a supervised system

To train a supervised classifier, we need to instantiate the NLP tools explained in Section 5. This is the starting point to be able to run the *Counter's*, *i.e.* the objects responsible of counting a particular kind of feature present in a text. The API documentation lists all the different counters provided by MIOPIA. This user manual poses below these lines how to instantiate different counters, to be able to train complex models:

- *NGramCounter*: MIOPIA supports features based on n-grams of words. It also provides the capabilities to create *generalised n-grams*, features where the n element is abstracted to reduce feature sparsity. The following *generalisations*, also known as *back-off*, are available: *word*, *lemma*, *part-of-speech tag*, *psychometric properties* or *none* (to completely remove a element). The constructor of any *NGramCounter* is as follows:

  ```
  def __init__(self,ftc, preprocessor, lexical_processor, back_off,
               stop_words=set([]), lowercase=True):
  ```

  where:

  - *ftc*: It represents an instance of the *FeatureTypeConfiguration*. See the API documentation for a detailed explanation.
  - *preprocessor*: See Section 5 to know how to instantiate this kind of object.
  - *lexical_processor*: The Section 5 explains how to obtain an instance of the *LexicalProcessor* class.
  - *back_off*. The class is responsible of obtaining the generalisations. The example below these lines shows how to create an object of this kind.
  - *stop_words*: A set of words that should be ignored when counting features.

  ```
  #Creating unigram (psychometric) counter
  psychometric_key = '-'.join([FeatureLevelBackOff.TYPE_BACK_OFF_PSYCHOMETRIC])
  u_psychometric_counter = NGramCounter(FeatureTypeConfiguration(
  ```

```
                        n_gram_back_off= psychometric_key,
                        n_gram=1),
                        preprocessor,
                        lexical_processor,
                        BackOff(dictionary),
                        set([]))

#Creating a bigram (lemma, part-of-speech) counter
lemma_postag_key = '-'.join([FeatureLevelBackOff.TYPE_BACK_OFF_LEMMA,
                             FeatureLevelBackOff.TYPE_BACK_OFF_FINE_TAG])
b_lemma_postag_counter = NGramCounter(FeatureTypeConfiguration(
                        n_gram_back_off= lemma_postag_key,
                        n_gram=2),
                        preprocessor,
                        lexical_processor,
                        BackOff(dictionary),
                        set([]))
```

- *DependencyTripletsCounter*: It counts triplets present in a dependency graph. As the *NGramCounter*, generalisation functionalities are available. The constructor is as follows:

```
def __init__(self, ftc, back_off,stop_words=set([])):


t_lemma_postag_counter = DependencyTripletsCounter(FeatureTypeConfiguration(
                    back_off_head=FeatureLevelBackOff.TYPE_BACK_OFF_FINE_LEMMA,
                    back_off_dependent = FeatureLevelBackOff.TYPE_BACK_OFF_FINE_TAG,
                    add_dependency_type = False),
                    BackOff(dictionary))
```

- *AbstractedLexiconCounter*: Counters of this kind are intended to create input features for the classifier using external knowledge via dictionaries.

```
def __init__(self, ftc, preprocessor, lexical_processor,
             dict_lexicon,lowercase=True):
```

The *lowercase* option just ignores capitalised characters. The *dict_lexicon* parameter should be a nailed dictionary able to manage long expressions. For example:

```
dict_lexicon ={'mal': [Negative_term,
                       {pagado: [Negative term],
                        uso: [Negative term]}
                      ]
                }
```

MIOPIA will take only into account the longest match.

Afterwards, instantiating an object of the *AbstractedLexiconCounter* is similar to instantiate an *NGramCounter*:

```
abs_counter= AbstractedLexiconsCounter(FeatureTypeConfiguration(),
                                       preprocessor, lexical_processor,
                                       dict_lexicon)
```

Once we have created the counters, they must be wrapped up into the corresponding *Adapter* instance:

```
PATH_WEKA = "/opt/weka/weka-3-7-10/weka.jar"
#Adapters support both binary and total occurrence weighting factors
u_psychometric_adapter = NGramAdapter(PATH_WEKA,
                                      u_psychometric_counter,
                                      Adapter.BINARY_WEIGHTING_FACTOR)
b_lemma_postag_adapter = NGramAdapter(PATH_WEKA,
                                      b_lemma_postag_counter,
                                      Adapter.BINARY_WEIGHTING_FACTOR)
t_lemma_postag_adapter = DependencyTripletsAdapter(PATH_WEKA,
                                                   t_lemma_postag_counter,
                                                   Adapter.BINARY_WEIGHTING_FACTOR)
```

```
abs_adapter = AbstractedLexiconsAdapter(PATH_WEKA,
                                        abs_counter,
                                        Adapter.BINARY_WEIGHTING_FACTOR)
```

It is possible to combine different counters, using the *CompositeAdapter*, in order to build more complex models.

```
composite_counter = CompositeAuxiliaryCounter(FeatureTypeConfiguration())
composite_adapter = CompositeAdapter(PATH_WEKA, composite_counter)

composite_adapter.add(u_psychometric_adapter)
composite_adapter.add(b_lemma_postag_adapter)
composite_adapter.add(t_lemma_postag_adapter)
composite_adapter.add(abs_adapter)
```

Next step consist of transforming our MIOPIA data file to the ARFF format, using the *Adapter* method *to_arff*.

```
PATH_TRAINING_MIOPIA = "/tmp/training.miopia"
PATH_TRAINING_ARFF = "/tmp/training.arff"
composite_adapter.to_arff(PATH_TRAINING_MIOPIA,PATH_TRAINING_ARFF)
```

Now we are ready to create and train our strategy for the supervised classifier.

```
PATH_WEKA_MODEL = "/tmp/weka.model" #The file where will be stored the WEKA model
PATH_OUTPUT_RESULTS = "/tmp/output.results" # File where our trained model
                                            # will print the evaluation
                                            # over the training set
evaluator = InformationGainAttributeEvaluator()
search_method = RankerSearchMethod(0)
strategy = MetaStrategy(evaluator, search_method,
                        ClassifierWeka.NAIVE_BAYES, PATH_WEKA)

strategy.train(PATH_WEKA_MODEL,
               PATH_OUTPUT_RESULTS,
               PATH_TRAINING_ARFF)
```

And our strategy should be trained. Time to train the model will depend on the complexity of the selected classifier and the feature space.

7.3 Running a supervised system

To successfully run a trained classifier, we need to know the ARFF header employed to train the model, to then be able to create compatible test sets. MIOPIA provides the method *arff_header_from_arff_file* for this purpose.

```
arff_header = composite_adapter.arff_header_from_arff_file(PATH_TRAINING_ARFF)
```

We then transform our test data (stored in a MIOPIA data file) to the ARFF format, using as parameters the *arff_header* and a boolean set to *True* (to indicate that is a test set).

```
PATH_TEST_MIOPIA = "/tmp/test.miopia"
PATH_TEST_ARFF = "/tmp/test.arff"
d_position_id = composite_adapter.to_arff(PATH_TEST_MIOPIA
                                          PATH_TEST_ARFF, arff_header, True)
```

This methods returns a dictionary (named *d_position_id* in the example above this line), which associate each position in the ARFF file, with the corresponding text identifier.
We then associate the trained *strategy* with an instance of a *SimpleClassifier* which acts as a wrapper for handling different strategies and formating the results:

```
classifier = SimpleClassifier(strategy)
```

We can now classify our test set represented in ARFF format:

```
PATH_WEKA_RESULTS = "/tmp/classifications.results"
classifications = classifier.classify(PATH_TEST_ARFF,PATH_WEKA_RESULTS,
                                       d_position_id)
```

Finally, it is possible to write the results in a file, *PATH_QREL_RESULTS*, following a *key-value* format:

```
PATH_QREL_RESULTS = "/tmp/classifications.qrel"
classifier.to_key_value_format(classifications, PATH_QREL_RESULTS)
```

8 Available examples and executables

The MIOPIA package provides a number of examples (executables) inside the `demo` folder:

- *demo_unsupervised_analyzer.py:* This script creates an unsupervised sentiment classifier. It shows how to instantiate the natural language resources required by this kind of classifier. The classifier is then used to obtain the sentiment of both raw and parsed texts in CoNLL-X format, based on an dummy set provided with the MIOPIA package.

- *demo_supervised_analyzer_es.py:* The script shows how to create and run a supervised classifier for Spanish language. In addition to the natural language pipeline, this kind of classifier relies on instances responsible of counting features to then feed the classifier. The model of the script is fed by words, psychometric properties and part-of-speech tags. The MIOPIA data format is used to train and evaluate the model.

- *demo_supervised_analyzer_en.py:* This script is similar to *demo_supervised_analyzer_es.py*, but intended for English language. The difference of the script relies on how to instantiate resources for English.

These three examples contain a number of global variables:

- *PATH_TRAINING_ARFF:* The path where the training set will be stored in the ARFF format.

- *PATH_TEST_ARFF:* The path where the test set will be stored in the ARFF format.

- *PATH_OUTPUT_RESULTS:* The path where the output results for the evaluation over the training set will be printed.

- *PATH_RANKING_FILE_FEATURES:* It indicates the file where it will be stored the ranking of the most relevant features according to the feature selection filter.

- *PATH_WEKA_MODEL:* The path where the trained WEKA model will be stored.

- *PATH_TRAINING_MIOPIA_FILE:* The path where the training set, represented in MIOPIA data format, will be stored.

- *PATH_TEST_MIOPIA_FILE:* The path where the test set, represented in MIOPIA data format, will be stored.

- *PATH_WEKA_RESULTS:* The path where the results of the test set will be printed, following the WEKA output format.
- *PATH_QREL_RESULTS:* The path where the results of the test set will be printed, following a key-value format.

9 Acknowledgements

Research reported in this article has been partially funded by Ministerio de Economía y Competitividad and FEDER (Grant TIN2010-18552-C03-02), Xunta de Galicia (Grants CN2012/008, CN2012/319) and Ministerio de Educación, Cultura y Deporte (FPU13/01180).