Crowd Environment and its Knowledge Analysis (CEKA)

(Version 1.0)

Programming Guide

Jing Zhang
Bryce Nicolson
Victor S. Sheng
Xindong Wu

Southeast University, China
University of Central Arkansas, UAS
Hefei University of Technology, China

CEKA is a software package for developers and researchers to mine the wisdom of crowds. It makes the entire knowledge discovery procedure much easier, including analyzing qualities of workers, simulating labeling behaviors, inferring true class labels of instances, filtering and correcting mislabeled instances (noise), building learning models and evaluating them. It integrates a set of state-of-the-art inference algorithms, a set of general noise handling algorithms, and abundant functions for model training and evaluation. CEKA is written in Java with core classes being compatible with the well-known machine learning tool WEKA, which makes the utilization of the functions in WEKA much easier.

Contents

1 System Overview

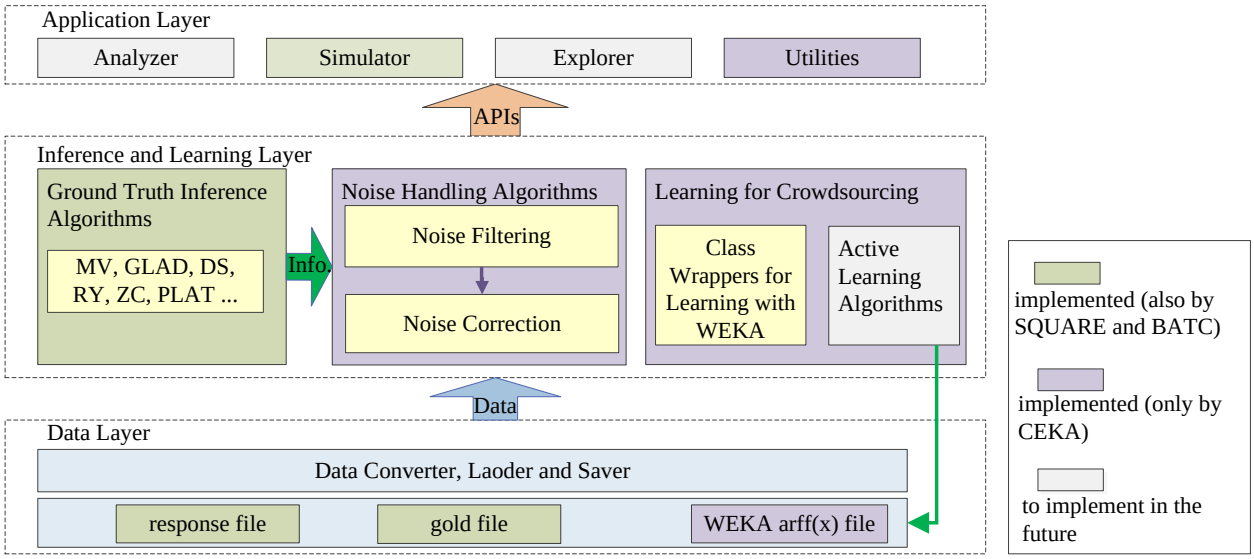
1.1 Introduction

The emergence of crowdsourcing (Howe, 2006) has changed the way of knowledge acquisition. It has already attracted vast attentions of the machine learning and data mining research community in the past several years. Researchers show great interests in utilizing crowdsourcing as a new approach to acquire class labels of objects from common users, which costs much less than the traditional way\textemdash annotating by domain experts. In order to improve the labeling quality, an object usually obtains multiple labels from different non-expert annotators. Then, inference algorithms will be introduced to estimate the ground truths of these objects. Many inference algorithms have been proposed in recent years. Besides, building learning models from the inferred crowdsourced data is another research issue with great challenges, which aims at lifting the quality of a learned model to the level that can be achieved by training with the data labeled by domain experts.

To facilitate the research on mining the wisdom of crowds, we develop a novel software package named Crowd Environment and its Knowledge Analysis (CEKA).

1.2 System Architecture

The following figure illustrates the hierarchical architecture of CEKA, in which it is also compared with the two other tools for crowdsourcing SQUARE (Sheshadri and Lease, 2013) and BATC (Nguyen et al., 2013). Generally, SQUARE and BATC only provide some inference algorithms and several simple analysis functions. By contrast, CEKA conceives a more ambitious blueprint. It attempts to support the entire knowledge discovery procedure including analysis, inference and model learning. In the data layer, CEKA is able to read an arff(x) file defined by WEKA, which contains features of instances for subsequent model building. In the inference and learning layer, it provide a large number of inference algorithms. Our on-going studies find that mislabeled instances after inference can be effectively detected and corrected, if a noise (mislabeled instance) handling algorithm can take advantage of the information generated in the previous inference procedure. Thus, CEKA provides a batch of noise handling algorithms. The core classes in this layer are derived from related classes in WEKA. In the application layer, CEKA provides a lot of utilities such as calculating performance evaluation metrics (i.e., accuracy, recall, precision, F source, AUC, M-AUC), manipulating data (i.e., shuffling, splitting and combining data), etc.


Application Layer
Analyzer
Simulator
Explorer
Utilities
APIs
Inference and Learning Layer
Ground Truth Inference Algorithms
MV, GLAD, DS, RY, ZC, PLAT ...
Info.
Noise Handling Algorithms
Noise Filtering
Noise Correction
Learning for Crowdsourcing
Class Wrappers for Learning with WEKA
Active Learning Algorithms
Data
Data Layer
Data Converter, Laoder and Saver
response file
gold file
WEKA arff(x) file
implemented (also by SQUARE and BATC)
implemented (only by CEKA)
to implement in the future

2 Deploy with Eclipse

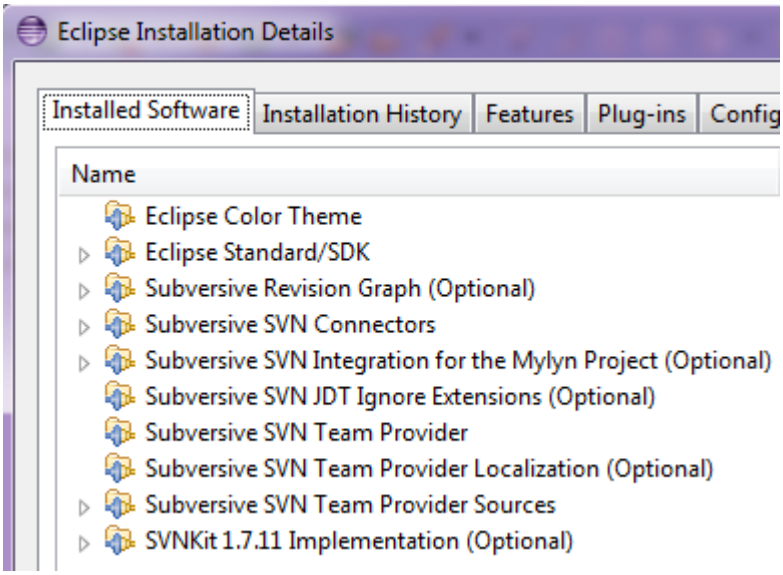
2.1 Install Eclipse

Download and Install Eclipse

This guide demonstrates the installation on the Windows system. Since CEKA and Eclipse are both written in Java, the installation on the Linux system is almost the same.

(1) Download Eclipse from: http://www.eclipse.org. Install it on the Windows system.

(2) For convenience in accessing the source code at sourceforge.net, we suggest that you install SVN plugins in Eclipse. Run Eclipse, click menu “Help” -> “Eclipse Marketplace.” Please install “Subversive –SVN Team Provider 2.0.” After that, the following SVN related components will be installed in Eclipse. (Note that the SVN plugins are dependent on the SVN client, which can be downloaded from http://tortoisesvn.net/ and should have been installed before you install this plugin.)



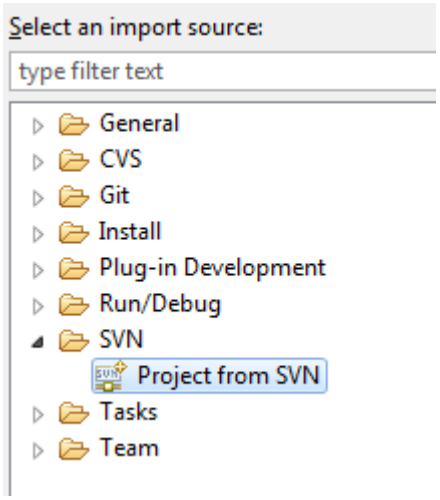
2.2 Install and Configure CEKA

Check out CEKA

You can create a new Eclipse workspace to accommodate CEKA and your application based on CEKA. In this guide, we create the workspace in the path `E:\CekaSpace`. We also use a variable CEKASPACE to represent this path in this guide. You can download and Install CEKA through the following steps.

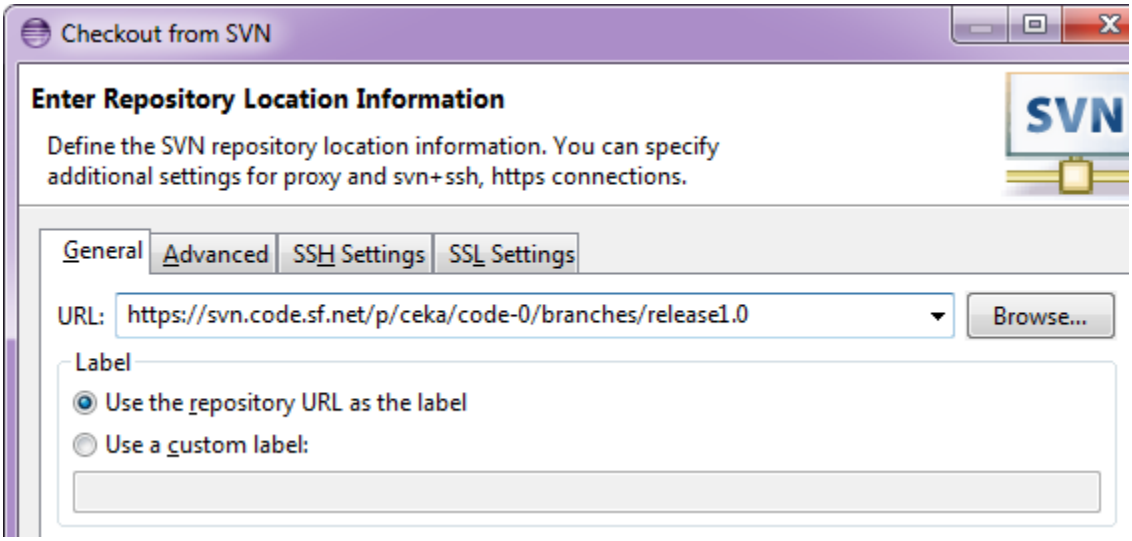
(1) Open Eclipse, click menu “File” -> “Switch Workspace” to switch workspace to CEKASPACE.

(2) Click menu “File” -> “Import.” In “Import” dialog box, select “Project from SVN.”



(3) The release version 1.0 of the CEKA is at:

https://svn.code.sf.net/p/ceka/code-0/branches/release1.0



If you are not the developer of CEKA, you need not fill the Authentication information on this page. Then, click “Next.” On the page of “Check Out As,” you can specify a new name to this project. For convenience, this guide use “Ceka.”

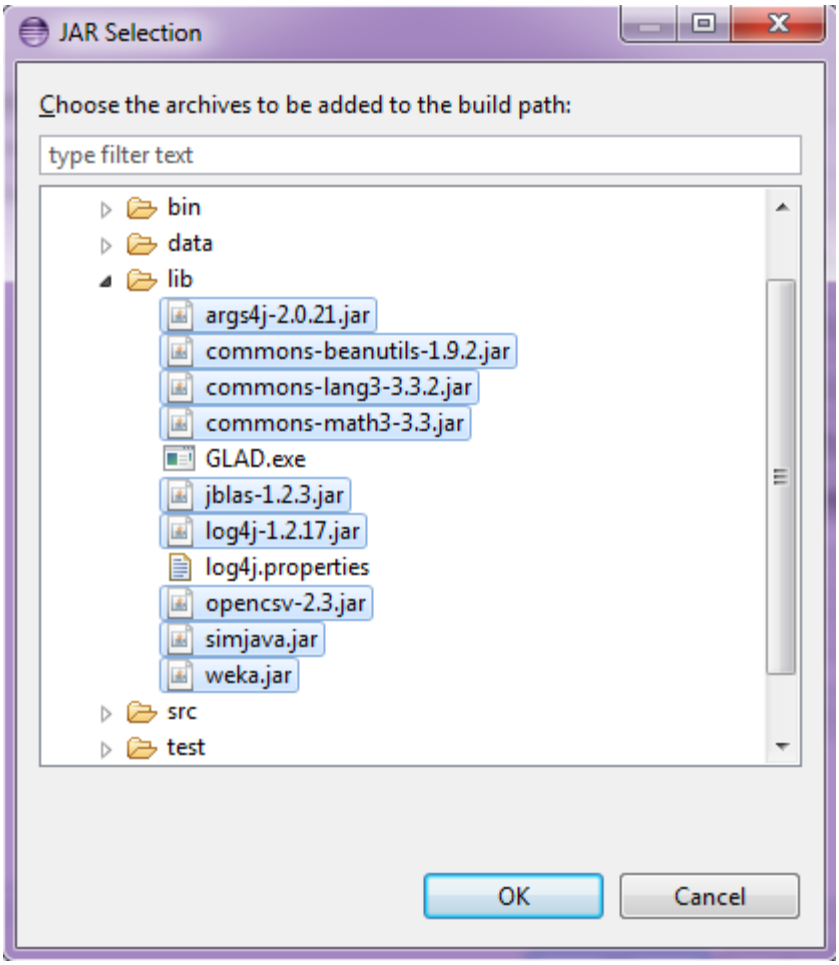
Configure CEKA

After CEKA is checked out, you should correctly set up the project by configuring its properties.

(1) Right-Click project Ceka in “Package Explorer” view. Select “Properties” then select “Java Build Path.”

(2) “Source” Tab. Besides `Ceka\src`, we add `Ceka\test` as another source fold.

(3) “Libraries” Tab. Click “add JARs.” Select all JARs in “lib” directory. (Warning: DO NOT select “GLAD.exe” and “log4j.properties.”).



Now Ceka has been correctly setup.

2.3 Cooperation with the Source Code of WEKA

In Section 2.2, CEKA is set up with the dependence of WEKA's jar package which has no source code provided. To encourage the users to cooperate with the source code of WEKA, we also have an image of the source code of WEKA-3.6.10 at:

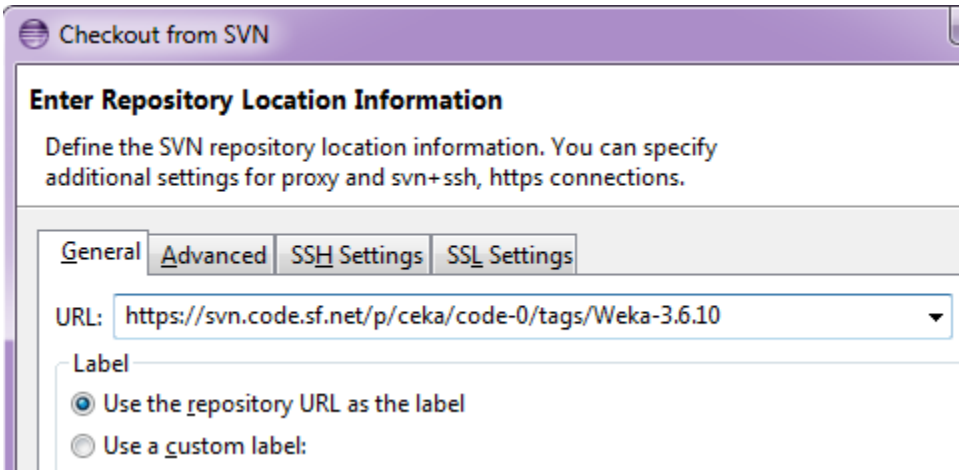
https://svn.code.sf.net/p/ceka/code-0/tags/Weka-3.6.10

You can set up your project with the source code of WEKA as follows.

(1) Click menu "File" -> "Import." In the "Import" dialog box, select "Project from SVN."

(2) Import WEKA-3.6.10 at:

https://svn.code.sf.net/p/ceka/code-0/tags/Weka-3.6.10



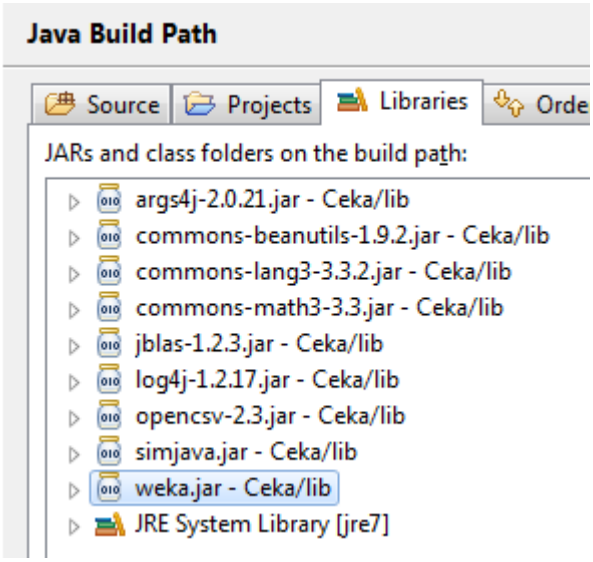
If you are not the developer of CEKA, you need not fill out the Authentication information on this page. Then, click “Next.” On the page of “Check Out As,” you can specify a new name to this project. For convenience, this guide uses “Weka.”

Configure CEKA

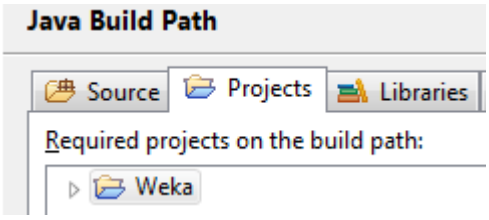
After Weka is checked out, you should re-configure CEKA with the proper setups.

(1) Right-Click project Ceka in “Package Explorer” view. Select “Properties” then select “Java Build Path.”

(2) “Libraries” Tab. Remove “weka.jar.”.



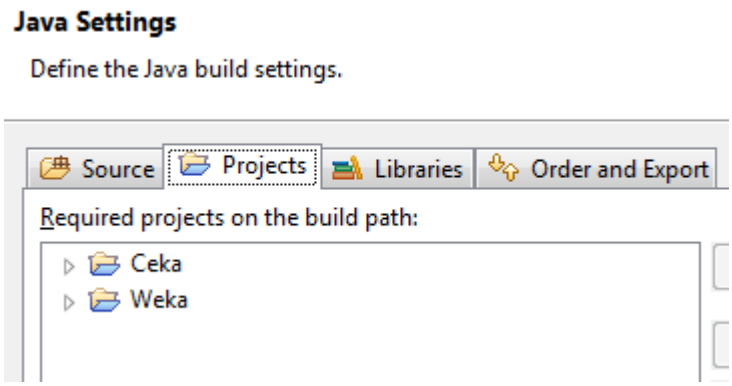
(3) “Projects” Tab. Add “Weka” project.



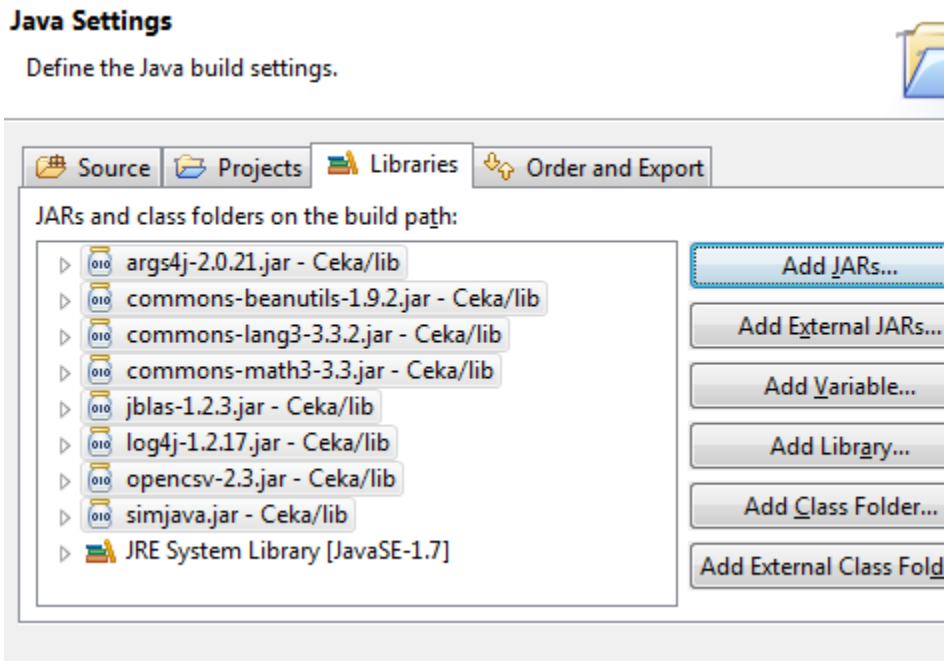
2.4 Set up Your Own Project Based on CEKA

It is very easy to create your own project that is willing to integrate CEKA. Click menu “File” -> “New” -> “Java Project.” Create a new Java project (suppose it’s named MyCekaApp). Click “Next.” On the page “Java Settings”:

(1) Add Projects “CEKA” and “Weka” into the dependent build path via “Project” tab page.



(2) Add all JARs in “Ceka\lib” except “weka.jar” (if you don’t include WEKA source code as Section 2.3 describes, you must add weka.jar as well.) into the dependent build path via “Libraries” tab page.



2.5 Overview of the Packages in CEKA

The following table describes the overview of the overview of the packages and directories in

CEKA. CEKA has a hierarchical package structure. In this table, we only provide main packages and classify these packages (classes and directories included) according to their logical functions.

Components	packages/classes	description
ceka.consensus (the ground truth inference algorithms)	ds	the Dawid & Skene's algorithm (Dawid and Skene, 1979), including an implementation by (Ipeirotis et al., 2010)
	glad	GLAD (Whitehill et al., 2009)
	gtic	GTIC (unpublished), an algorithm for multi-class inference.
	kos	KOS (Karger et al., 2011)
	plat	PLAT (Zhang et al., preprint)
	square	including RY (Raykar et al., 2010) and ZenCrowd (Demartini et al., 2012) implemented by SQUARE (Sheshadri and Lease, 2013)
	MajorityVote.java	Majority Voting
	WeightedVote.java	Adaptive Weighted Voting for binary imbalanced labeling (unpublished)
ceka.converters	FileLoader.java	a class for loading files to form a data set
	FileSaver.java	a class for saving a data set to files
ceka.core	Dataset.java, Example.java, Category.java, ….	This package contains the core classes of CEKA. For details, refer to Section 4.
ceka.noise (noise handling techniques for crowdsourcing)	avnc	Adaptive Voting Noise Correction for crowdsourcing (unpublished)
	ClassificationFilter.java	ClassificationFiltering algorithm (Gamberger et al., 1999)
	IterativePartitionFilter.java	iterative partition filtering (IPF) (Khoshgoftaar and Rebours, 2007)
	MajorityFilter.java	voting filtering (Brodley and Friedl, 1999)
	MultiplePartitioningFilter.java	multiple partitioning filtering (Khoshgoftaar and Rebours, 2007)
	SelfTrainCorrection.java	Self-training correction (STC) (Triguero et al., 2014).
	STCConfidence.java	Self-training correction (STC) (Triguero et al., 2014) with confidence parameter (unpublished)

ceka.simulation	ExampleMask.java ExampleWorkersMask.java GaussianLabelingStrategy.java SingleQualLabelingStrategy.java …	classes used for simulating labeling behaviors of workers.
ceka.utils (utiltity functions)	PerformanceStatistic.java	a class used for calculation of a set of performance measures, such as accuracy, precision, recall, AUC,…
	DatasetManipulator.java	a class for data set manipulation
	Misc.java	a lot of useful functions for programming.
com.ipeirotis.gal	----	Dawid & Skene's algorithm implementation by (Ipeirotis et al., 2010)
mloss.roc	----	code to compute ROC and AUC
org.square.qa	----	code implemented by SQUARE (Sheshadri and Lease, 2013)
Ceka/lib (directory)	----	all dependent JARs for CEKA GLAD.exe log4j.proporties
Ceka/data (dirrectory)	real-world	the real-world data sets
	synthetic	the synthetic data sets

3 Input File Formats

3.1 Input Files

The current version (1.0) of CEKA accepts four kinds of text files as its input files. The extension names of these four kinds of files are ".gold.txt," ".response.txt," ".arff," and ".arffx." The common name of these files defines a name of a data set. It means that a data set exists in the disk as several related files with the same common name. For example, a data set named "leaves" may include three input files "leaves.gold.txt," "leaves.response.txt," and "leaves.arff."

3.1.1 File ".gold.txt"

A file with the extension name ".gold.txt" defines the ground truth of all instances in a data set, which is used for evaluation of an algorithm performance. Each line in this file defines an instance and the true label of this instance with the format as:

INSTANCE-ID**0x09**TRUELABEL**0x0D0x0A**

INSTANCE-ID is a string that can be treated as a unique name of an instance.

TRUELABEL is the true label of this instance, which is an integer string that ranges from "0" to the maximum number of classes. This class identification MUST be consecutive integer starting from "0."

0x09 is a delimiter whose ASCII code is 0x09, known as "Tab" (\t).

0x0D0x0A or **0x0A** is a line feed, known as "\r\n" or "\n."

Example: a segment of a binary labeling ".gold.txt" file

```
0   0
1   0
2   1
3   0
4   1
5   0
6   1
7   0
8   0
9   0
```

3.1.2 File ".response.txt"

A file with the extension name ".reponse.txt" defines all labels obtained from the annotators in a data set, which is the labeling information of crowdsourced data. Each line in this file defines a label assigned to an instance provided by an annotator with the following format:

WORKER-ID**0x09**INSTANCE-ID**0x09**LABEL**0x0D0x0A**

WORKER-ID is a string that can be treated as a unique name of a worker.

INSTANCE-ID is a string that can be treated as a unique name of an instance.

LABEL is the label of this instance provided by this annotator, which is an integer string that ranges from "0" to the maximum number of classes. This class identification MUST be a consecutive integer starting from "0."

0x09 is a delimiter whose ASCII code is 0x09, known as "Tab" (\t).

0x0D0x0A or **0x0A** is a line feed, known as "\r\n" or "\n".

Example: a segment of a binary labeling ".response.txt" file

```
12  164 1
13  164 0
16  164 0
19  164 1
24  164 0
0   130 1
2   130 1
3   130 1
```

3.1.3 File ".arff"

A file with the extension name ".arff" is a data set file defined by WEKA, which provides both features and true labels of each instance in the data set. More details about ".arff" file can be found at http://weka.wikispaces.com/ARFF.

3.1.4 File ".arffx"

A file with the extension name ".arffx" is a data set file defined by CEKA to extend the ".arff" file in WEKA. In ".arff" file, all instances in a data set listed in this file only have their features and true labels but no identities. However, in crowdsourcing, we sometime need to know which instance is labeled by which annotators. Thus, we extend the ".arff" file type by adding a list at the end of the file which specifies the ID of each instance. In this file, the section "@DATA" of a ".arrfx" file and the

section “@ID-MAP” contains the same number of lines. For each line (instance) in the section “@DATA,” we will specify its ID in the corresponding line in the section “@ID-MAP.”

Example: a segment of a “.arff” file

```
[the headers of arff file are omited]

@DATA
Some-college,10,Divorced,Tech-support,Own-child,White,40,US,0
Some-college,10,Divorced,Handlers-cleaners,Not-in-family,Amer-Indian-Eskimo,84,US,0
HS-grad,9,Married-civ-spouse,Farming-fishing,Husband,Asian-Pac-Islander,40,Cambodia,1
HS-grad,9,Never-married,Exec-managerial,Not-in-family,White,40,US,0
Masters,14,Married-civ-spouse,Exec-managerial,Wife,White,50,U-S,1

@ID-MAP
50
51
52
60
61
```

In this example, five instances listed in the section “@DATA” will have the IDs “50,” “51,” “52,”“60” and “61,” respectively.

3.2 File Loading

The class `FileLoader` in the package `ceka.converters` is responsible for loading the data from different kinds of input files to create the class `Dataset`. (Note: the details of the class `Dataset` are in Section 4.)

The main public functions of `FileLoader` are listed below.

Function	`Dataset loadFile(String responsePath, String goldPath) throws Exception`	
	comments: the returned Dataset doesn’t contain any features	
Parameters	`String responsePath`	the path of the “.response.txt” file
	`String goldPath`	the path of the “.gold.txt” file
Return	`Dataset`	the Dataset created form two files

Function	`Dataset loadFile(String responsePath, String goldPath, String arffxPath) throws Exception`	
	comments: the returned Dataset that contains features	
Parameters	`String responsePath`	the path of the “.response.txt” file

	`String goldPath`	the path of the ".gold.txt" file
	`String arffxPath`	the path of the ".arffx" file
Return	`Dataset`	the Dataset created form two files

Function	`Dataset loadFileX(String responsePath, String goldPath, String arffxPath) throws Exception`	
	comments: the returned Dataset that contains features NOTE: if `goldPath` is null, then the true labels of instances will be set to the values in `arffxPath`, otherwise, they will be set to the values in `goldPath`.	
Parameters	`String responsePath`	the path of the ".response.txt" file
	`String goldPath`	the path of the ".gold.txt" file
	`String arffxPath`	the path of the ".arffx" file
Return	`Dataset`	the Dataset created form three files

Function	`Dataset loadFile(String responsePath, String goldPath, String arffPath) throws Exception`	
	comments: the returned Dataset that contains features NOTE: (1) If `goldPath` is null, then the true labels of instances will be set to the values in `arffPath`, otherwise, they will be set to the values in `goldPath`. (2) The IDs of all instances automatically start from 0 to the maximum number of instances.	
Parameters	`String responsePath`	the path of the ".response.txt" file
	`String goldPath`	the path of the ".gold.txt" file
	`String arffPath`	the path of the ".arff" file
Return	`Dataset`	the Dataset created form three files

Example: read files to form a `Dataset` object

```
String responsePath = "E:/CekaSpace/Ceka/data/income94.response.txt";
String goldPath = "E:/CekaSpace/Ceka/data/income94.gold.txt";
String arffPath = "D:/CekaSpace/Ceka/data/Income94.arff";
Dataset dataset = FileLoader.loadFile(responsePath, goldPath,
arffPath);
```

3.3 File Saving

The class `FileSaver` in the package `ceka.converters` is responsible for saving a `Dataset` object to several kinds of files mentioned above.

The main public functions of `FileSaver` are listed below.

Function	`void saveDataset(Dataset dataset, String responsePath, String goldPath) throws IOException`	
	comments: save a data set into ".response.txt" and ".gold.txt" files	
Parameters	`Dataset dataset`	the dataset to be saved
	`String responsePath`	the path of the ".response.txt" file
	`String goldPath`	the path of the ".gold.txt" file
Return	`void`	

Function	`void saveDatasetArff(Dataset dataset, String arffPath) throws Exception`	
	comments: save a data set to an arff file	
Parameters	`Dataset dataset`	the dataset to be saved
	`String arffPath`	the path of the ".arff" file
Return	`void`	

Function	`void saveDatasetArffx(Dataset dataset, String arffxPath) throws Exception`	
	comments: save a data set to an arffx file	
Parameters	`Dataset dataset`	the dataset to be saved
	`String arffxPath`	the path of the ".arffx" file
Return	`void`	

Function	`void saveDatasetResponseArffx(Dataset dataset, String responsePath, String goldPath, String arffxPath, ExampleWorkersMask mask) throws Exception`
	Comments: save a `Dataset` object into ".response.txt," ".gold.txt," and "arffx" files. When saving the data set, an object of `ExampleWorkersMask` is used to determine whether a labeler that assigns label to an instance needs to be stored into the ".response.txt" file. That is, the `ExampleWorkersMask` object

	acts as a filter. For more information about the class `ExampleWorkersMask`, refer to Section 7. NOTE: this function is used to create different data sets from a basic one.	
Parameters	`Dataset dataset`	the dataset to be saved
	`String responsePath`	the path of the ".response.txt" file
	`String goldPath`	the path of the ".gold.txt" file
	`String arffxPath`	the path of the ".arffx" file
	`ExampleWorkersMask mask`	a mask for filtering workers according to some rules. See Section 7.
Return	`void`	

Function	`saveDatasetResponseArffx(Dataset dataset, String responsePath, String goldPath, String arffxPath, ExampleMask mask)`	
	Comments: save a `Dataset` object into ".response.txt," ".gold.txt," and "arffx" files. When saving the data set, an object of `ExampleMask` is used to determine whether an instance needs to be stored into the ".response.txt," ".gold" and ".arffx" files. That is, `ExampleMask` object is served as a filter to instances. For more information about the class `ExampleMask`, refer to Section 7. NOTE: this function is used to create different data sets from a basic one.	
Parameters	`Dataset dataset`	the dataset to be saved
	`String responsePath`	the path of the ".response.txt" file
	`String goldPath`	the path of the ".gold.txt" file
	`String arffxPath`	the path of the ".arffx" file
	`ExampleMask mask`	a mask for filtering instances according to some rules. See Section 7.
Return	`void`	

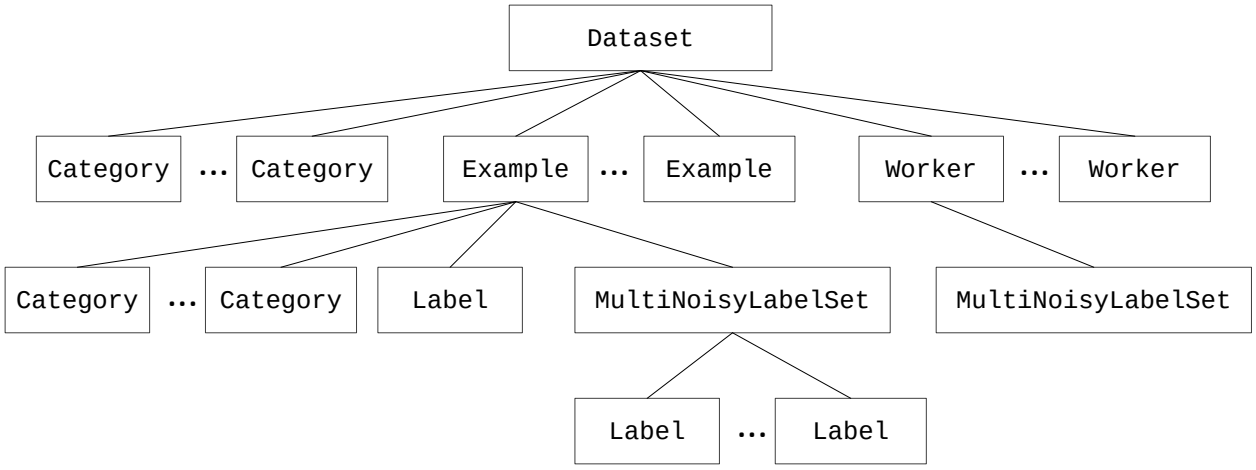
4 Core Classes

4.1 Overview

This section first describes the hierarchical structure of core classes. Then, the details of every class are provided with some example code.

4.1.1 Hierarchical structure of core classes

The core classes of CEKA are directly derived from the real-world objects in crowdsourcing. The following figure describes the hierarchical structure of these core classes.



4.1.2 Brief descriptions of core classes

- **`Dataset`**

  `Dataset` is a class that contains the crowdsourced data read from the files. `Dataset` is a subclass of class `weka.Instances`.

  Class `Dataset` consists of multiple objects of classes `Category`, `Example` and `Worker`.

- **`Category`**

  `Category` is a class that describes the concept of class in machine learning.

  Class `Category` consists of a name, an integer that starts from 0 and identifies a class, and a probability of this class.

  For example, for a binary data set, the object of class `Dataset` contains two objects of class `Category`. One stands for class 0 and the other stands for class1. Both of them have the same

name, `Label.DEFAULT_LABEL_NAME`. The name of a `Category` or `Label` is for multi-label extension, which is used for identifying different class labels. For single-label, the name is always `Label.DEFAULT_LABEL_NAME`.

- **`Example`**

  `Example` is a class that stands for an instance in a data set. `Example` is a subclass of class `weka.Instance`. The features of an instance are held by its superclass `weka.Instance`.

  Class `Example` consists of multiple objects of classes `Category`, an object of class `Label` which represents the true label of this example, and an object of class `MultiNoisyLabelSet`.

- **`Worker`**

  `Worker` is a class that stands for an annotator in a crowdsourcing system.

  Class `Worker` consists of an object of class `MultiNoisyLabelSet` that represents all labels given by this worker.

- **`Label`**

  `Label` is a class that describes the concept of class label in machine learning.

  Class `Label` consists of a name and an integer that starts from 0 and identifies a class. A label is associated with a worker ID and an example ID, which stands for the worker (ID) giving this label to the example (ID).

  The name of a `Label` is for multi-label extension, which is used for identifying different class labels. For single-label, the name is always `Label.DEFAULT_LABEL_NAME`.

- **`MultiNoisyLabelSet`**

  `MultiNoisyLabelSet` is a class that describes a set of multiple objects of the class `Label`.

  Class `MultiNoisyLabelSet` consists of a list of objects of the class `Label` and a single object of the class `Label` which is called `intergatedLabel`. An `intergatedLabel` is the label that is inferred from the current data set.

  The name of a `Label` is for multi-label extension, which is used for identifying different class labels. For single-label, the name is always `Label.DEFAULT_LABEL_NAME`.

4.2 Class `Dataset`

4.2.1 Create an empty data set

Besides creating an object of the class dataset from the ".response.txt," ".gold.txt," ".arff" and ".arffx" files as described in Section 3, sometimes we may want to create an empty data set that contains no instances. We can use the following code to do it.

```
String datasetName = "…";
Dataset dataset = new Dataset (datasetName, null, 0);
```

If we want to create an empty data set based on our current data set, which means the meta data (descriptions) of the attributes of our current data set will be inherited by the newly created one, we can use the following code.

```
/*baseDataset is an existing data set*/
Dataset dataset = new Dataset (baseDataset, 0);
```

4.2.2 Manipulation of the instances in a data set.

After an empty data set is created, we can add instances into this data set by using the following function.

Function	`void addExample(Example e)`	
	Note: when adding an instance into a data set, always call this function, DO NOT call `weka.add(Instance)` function.	
Parameter	`Example e`	an instance to be add into this data set
Return	`void`	

Class `Dataset` provides the following functions for users to query instances in it.

Function	`Example getExampleByIndex(int index)`	
	Comments: the instances in a data set are arranged as a list. Using the index of a position can retrieve that instance.	
Parameter	`int index`	the index of the position that holds the instance
Return	`Example`	Queried example

Function	`Example getExampleById(String id)`	
	comments: retrieve an instance by using an ID	
Parameters	`String id`	the ID of the instance to be retrieved
Return	`Example`	Queried example

Example: traversing all instances in a data set

```
/*dataset is an object of class Dataset*/
for (int i = 0; i < dataset.getExampleSize(); i++) {
   Example e = dataset.getExampleByIndex(i);}
```

4.2.3 Other functions

Example: traversing all workers in a data set

```
/*dataset is an object of class Dataset*/
for (int i = 0; i < dataset.getWorkerSize(); i++) {
      Worker w = dataset.getWorkerByIndex(i);
}
```

An object of class worker in a data set can also be retrieved by its ID using function `getWokersById(String id)`.

Example: traversing all categories in a data set

```
/*dataset is an object of class Dataset*/
for (int i = 0; i < dataset.getCategorySize(); i++) {
      Category c = dataset. getCategory(i);
}
```

In some machine learning algorithms, randomization operations should be applied to a data set. We can use the following function to shuffle all instances in a data set.

Function	`void randomize(Random random)`	
	Comments: randomly shuffle all instances in a data set	
Parameter	`Random random`	the object of the class `Random` (in Java)
Return	`void`	

4.2.4 Compatible with WEKA

Since class `Dataset` is a subclass of `weka.Instances`, it can be directly passed as a parameter to the related classes in Weka. The following code is an example of training a model.

```
/*dataset is an object of class Dataset*/
Classifier smo = new weka.classifiers.functions.SMO();
smo. buildClassifier(dataset);
```

4.3 Class `Example`

4.3.1 Create examples

In this document, the terms example and instance are equivalent. Since the class `Example` is a subclass of `weka.Instance`, many constructors of `Example` must be compatible with the constructors of `weka.Instance`, such as `Example(weka.core.Instance instance)` and `Example(weka.core.Instance instance, String idStr)`, which accept another base object of `weka.Instance` to create a new one. The meta information of attributes will be inherited by the newly created object. However, sometimes we only want to do ground truth inference instead of model learning. Under this circumstance, no features of an example will be provided. We can use the following function to create an example without features by setting the first parameter to 1.

Function	`Example(int numAttributes, String id)`	
	Comments: create an example by setting its number of attributes and id. If numAttributes is set to 1, the example will be created with only one attribute that is the true label.	
Parameters	`int numAttributes`	the number of attributes, must great than 0
	`String id`	the ID of this example

We also can create a new example that contains the same content as the other does by calling the function `copy`.

Function	`Example copy()`	
	Comments: the newly created example has the same set of categories, true label and multiple noisy label set as the original one has.	
Return	`Example`	the new example

Compared with the `weka.Instance`, the object of class `Example` has a unique identity which is set during the creation of the object and can be retrieved by the function `getId()`.

4.3.2 Manipulation of different kinds of labels

Each example has a true label that is provided by the oracle (expert with a perfect correct rate) used for evaluating the performance of algorithms. The true label can be retrieved and set by the functions `Label getTrueLabel()` and `void setTrueLabel(Label l)`. (Note: the current version of CEKA is only for single-label application; for multi-label application, there will be multiple true labels. In this document, we only focus on the single-label issue.)

Each example has an integrated label which is the label inferred by an inference algorithm. Class `Example` provides the following functions to get and set the integrated label.

Function	`Label getIntegratedLabel()`	
	Note: the integrated label is set automatically by an inference algorithm described in Section 5.	
Return	`Label`	integrated label

Function	`void setIntegratedLabel(Label label)`	
	Comments: if the example already has an integrated label, calling this function will overwrite the original one.	
Parameter	`Label`	the new integrated label
Return	`Example`	Queried example

Example: retrieve all noisy labels assigned to an example

```
/*example is an object of class Example*/
MultiNoisyLabelSet mnls = example.getMultipleNoisyLabelSet(0);
for (int i = 0; i < mnls.getLabelSetSize(); i++) {
      Label label = mnls.getLabel(i);
}
```

Sometimes we need to retrieve a specific noisy label provided by the identity of an annotator. We can use the following function.

Function	`Label getNoisyLabelByWorkerId(String wId)`	
Parameter	`String wId`	identity of a worker
Return	`Label`	the label assigned by this worker

4.3.3 Cooperation with WEKA

Suppose a new inference algorithm is designed by a user of CEKA. After inference, each example will be assigned an integrated label by calling function `setIntegratedLabel(Label label)`. Then the data set will be used to train a model by WEKA. Since WEKA does not call function getIntegratedLabel(), we must call function `void assignIntegeratedLabel2WekaInstanceClassValue()` before we train model. The value of the label used in train also can be retrieved by the function `getTrainingLabel()`.

Since class `Example` is a subclass of `weka.Instance`, it can be directly passed as a parameter to the related classes in Weka. The following code is an example of training a model and predicting the class of an instance.

```
/*dataset is an object of class Dataset*/
Classifier smo = new weka.classifiers.functions.SMO();
smo. buildClassifier(dataset);
/*testExample is an object of class Example*/
double predict = smo.classifyInstance(testExample);
```

4.4 Classes MultiNoisyLabelSet and Label

4.4.1 Class MultiNoisyLabelSet

Class `MultiNoisyLabelSet` is a simple container that holds a set of labels. Actually, each integrated label is associated with a `MultiNoisyLabelSet` because we deem that the integrated label is derived from a `MultiNoisyLabelSet`. The integrated label can be obtained by calling the following function.

Function	`Label getIntegratedLabel()`	
	Comments: get the integrated label associated with this multiple noisy label set.	
Return	`Label`	the integrated label

The following sample code shows how to retrieve all labels in a multiple noisy label set.

```
/* mnls is an object of class MultiNoisyLabelSet */
for (int i = 0; i < mnls.getLabelSetSize(); i++) {
       Label label = mnls.getLabel(i);
}
```

4.4.2 Class `Label`

Class `Label` is associated with an integer which is in the range from 0 to the maximum number of categories in the crowdsourced data. For example, for binary labeling, the maximum number of categories is 2. One can use the function `getValue()` to get this value and use the function `setValue()` to set this value.

An object of the class `Label` can be created by calling the following constructor.

Function	`Label(String name, String value, String exampleId, String workerId)`	
	Comments: create an object of the class `Label` for the single-label issue, the name of a label is always `Label.DEFAULT_LABEL_NAME`	
Parameters	`String name`	the name of the label
	`String value`	the class value of this label
	`String exampleId`	the example ID that this label is associated with
	`String workerId`	the worker ID that this label is associated with

4.5 Class `Worker`

Class `Worker` represents an annotator in the crowdsourcing system which has a unique ID. The ID of a worker is set when the object is created and can be retrieved by calling the function `getId()`. The class `Worker` provides the following functions.

Function	`void addNoisyLabel(Label label)`	
	Comment: add a noisy label to the multiple noisy label set of this worker	
Parameter	`Label label`	the noisy label to add
Return	`void`	

Function	`MultiNoisyLabelSet getMultipleNoisyLabelSet(int index)`	
	Comments: get the multiple noisy label set of this index Currently, the parameter index must be set to 0.	
Parameter	`int index`	the index of a multiple noisy label set, must be 0
Return	`MultiNoisyLabelSet`	the multiple noisy label set retrieved

5 Inference Algorithms

5.1 Common Function

CEKA focuses on the agnostic inference algorithms, which require no additional prior knowledge expect for the labels provided by annotators. To simplify the usage of these inference algorithms, CEKA provides a very simple uniform function to conduct inference as follows.

Function	`void doInference(Dataset data)`	
	Comments: after calling the function, the integrated labels of all instances in the data set are set.	
Parameter	`Dataset data`	the data set to be inferred
Return	`void`	

Every inference algorithm has a name. The following sample code shows how to create different inference algorithms and conduct an inference task.

```
String consensusName = "…";
if (consensusName.equals(MajorityVote.NAME)) {
   MajorityVote mv = new MajorityVote();
   mv.doInference(dataset);}
if (consensusName.equals(GLADWraper.NAME)) {
   GLADWraper glad = new GLADWraper(tempDir, gladExePath);
   glad.doInference(trainSet);}
if (consensusName.equals(SquareIntegration.methodZenCrowd)) {
   SquareIntegration si = new SquareIntegration(tempDir);
   si.doInference(trainSet, SquareIntegration.methodZenCrowd);}
if (consensusName.equals(SquareIntegration.methodRYBinary)) {
   SquareIntegration si = new SquareIntegration(tempDir);
   si.doInference(trainSet, SquareIntegration.methodRYBinary);}
if (consensusName.equals(DawidSkene.NAME)) {
   DawidSkene ds = new DawidSkene(50);
   ds.doInference(trainSet);}
if (consensusName.equals(GalDawidSkene.NAME)) {
   GalDawidSkene ds = new GalDawidSkene(tempDir);
   ds.doInference(trainSet);}
if (consensusName.equals(KOS.NAME)) {
   KOS kos = new KOS(10);
   kos.doInference(trainSet);}
```

5.2 Details of the Inference Algorithms

5.2.1 Definitions

In a crowdsourcing system, the set of examples is denoted by $E = \{e_i\}_{i=1}^{I}$, where $e_i =< x_i, y_i >$, x_i is the *feature* portion and y_i is the true label of the example. The set of annotators is denoted by $U = \{u_j\}_{j=1}^{J}$. Each label belongs to a set of classes $C = \{c_k\}_{k=1}^{K}$. For convenience, we can use the index as the identity of an annotator, an example and a class, saying an annotator *j,* an example *i* and a class *k.* Since we focus on binary labeling problems, we map c_1 *(k=1)* and c_2 *(k=2)* to the negative (-) and the positive (+) classes, respectively.

Each example *i* is associates with a multiple noisy label set $\bar{l}_i = \{l_{ij}\}_{j=1}^{J}$, where every element l_{ij} comes from the annotator *j*. All labels of the examples in the dataset form a matrix $L = \{\bar{l}_i\}_{i=i}^{I}$, $l_{ij} \in \{c_1, 0, c_2\}$, where 0 means that the annotator does not provide any label for that example. Each annotator *j* is associated with a matrix $n^{(j)} = \{n_{ik}^{(j)}\}, 1 \le i \le I$ and $1 \le k \le K$. Every element inside the matrix presents the number of times that the annotator *j* labels the example *i* as class *k*. In practice, each annotator labels an object at most once, that is, $n_{ik}^{(j)} \in \{0,1\}$. We also define the priori probabilities of negative and positive classes as p^- and p^+.

The goal is to estimate the gold standards $\widehat{y_i}$s that maximizes $\sum_{i=1}^{I} \mathbf{I}\left(\widehat{y_i} = y_i\right)$, given L, $\widehat{y_i}$, where $y_i \in (c_1, c_2)$ and $\mathbf{I}$ is the indicator function which outputs 1 when the test condition is satisfied; otherwise it outputs 0.

5.2.2 Majority Voting

Majority Voting is the simplest inference algorithm. For each example, its integrated label has the class with the maximum labels belonging to it. That is,

$$c^{(i)} = \arg\max_k \{\sum_{j=1}^{J} \mathbf{I}(l_{ij} = c_k), 1 \le k \le K\}$$

If multiple classes have the same number of members, the class of the integrated label will be randomly chosen from them.

5.2.3 Dawid & Skene's

Dawid & Skene's algorithm (DS) (Dawid and Skene, 1979) was proposed by Dawid and Skene in 1979. Itmodels annotators by using confusion matrices in multi-class medical diagnoses. One confusion matrix presents one annotator. The element $\pi_{kl}^{(j)}$ of the confusion matrix of the annotator *j* is the probability of labeling examples with true class *k* to class *l*.

E-step: DS estimates the probabilities of each example *i* belonging to class *k* by using the following equation:

$$P(\hat{y}_i = c_k \mid L) = \frac{\prod_{J=1}^{J}\prod_{l=1}^{K}(\pi_{kl}^{(j)})^{n_{il}^j} P(c_k)}{\sum_{q=1}^{K}\prod_{j=1}^{J}\prod_{l=1}^{K}(\pi_{ql}^{(j)})^{n_{il}^j} P(c_q)}$$

M-step: DS updates the confusion matrix of every annotator and the priori probabilities of all classes.

$$\hat{\pi}_{kl}^{(j)} = \sum_{i=1}^{I}\mathbf{I}(\hat{y}_i = c_k)n_{il}^{(j)} / \sum_{l=1}^{K}\sum_{i=1}^{I}\mathbf{I}(\hat{y}_i = c_k)n_{il}^{(j)}$$

$$\hat{P}(c_k) = \sum_{i=1}^{I}\mathbf{I}(\hat{y}_i = c_k) / I$$

Because DS is an EM algorithm, when calling the constructor of the object of class `DawidSkene` we must specify an integer designating the maximum iterations of EM procedures.

5.2.4 GLAD

GLAD (Whitehill et al., 2009) was proposed to model the expertise of each annotator ($\alpha_j \in (-\infty, +\infty)$) and the difficulty level of each example ($1/\beta_i \in [0, +\infty)$). To estimate the labeling probability of an annotator *j* on an example *i*, GLAD uses the following logistic model.

$$P(l_{ij} = y_i \mid \alpha_j, \beta_i) = 1 / (1 + e^{-\alpha_j \beta_i})$$

E-step: GLAD computes the posterior probabilities of both negative and positive classes of all examples given the values of two parameters ($\boldsymbol{\alpha}, \boldsymbol{\beta}$) from the last M-Step and the observed labels.

$$P(y_i = + \mid L, \boldsymbol{\alpha}, \boldsymbol{\beta}) = P(y_i = + \mid \bar{l}_i, \boldsymbol{\alpha}, \beta_i) \quad \propto \quad P(y_i = +)\prod_{j=1}^{J} p(l_{ij} \mid y_i = +, \alpha_j, \beta_i)$$

M-step: GLAD maximizes the standard auxiliary function *Q* and updates the values of two parameters ($\boldsymbol{\alpha}, \boldsymbol{\beta}$) by using a gradient descent algorithm as follows.

$$Q(\boldsymbol{\alpha}, \boldsymbol{\beta}) = E[\ln P(L, Y \mid \boldsymbol{\alpha}, \boldsymbol{\beta})] = E[\ln \prod_{i=1}^{I}(P(y_i = +)\prod_{j=1}^{J} P(l_{ij} \mid y_i = +, \alpha_j, \beta_i))]$$

The original implementation of GLAD only runs on Linux systems, so we have transplanted it on the Windows system. It can be found in directory Ceka\lib.

5.2.5 Raykar, Yu, et al. (RY)

RY (Raykar et al., 2010) was proposed to model the *sensitivity* (α_j) and the *specificity* (β_j) of an annotator *j*. In the case of binary labeling, the *sensitivity* defines the bias toward the positive class and the *specificity* defines the bias toward the negative class. RY uses a classifier to predict labels, but this classifier requires a feature representation of examples. In our study, we ignore this classifier.

RY uses a Bayesian approach to estimate the prior probabilities of the parameters (α_j, β_j) and the positive class:

$$P(\alpha_j \mid a_j^+, a_j^-) = Beta(\alpha_j \mid a_j^+, a_j^-)$$
$$P(\beta_j \mid b_j^+, b_j^-) = Beta(\beta_j \mid b_j^+, b_j^-)$$
$$P(p^+ \mid n^+, n^-) = Beta(p^+ \mid n^+, n^-)$$

where a_j^+ and a_j^- are the number of positive labels and negative labels, respectively, provided by the annotator *j* to the positive class inferred at this moment, b_j^+ and b_j^- are the number of positive labels and negative labels, respectively, provided by the annotator *j* to the negative class inferred at this moment, n^+ and n^- are the total number of positive and negative labels, respectively, provided by all annotators to all examples. *Beta* is a beta probability distribution function.

E-step: RY computes the probability of each example *i* belonging to the positive class as follows.

$$\mu_i = P(y_i = + \mid x_i, L, \alpha, \beta, p^+) \quad \propto \quad \frac{p^+ a_i}{p^+ a_i + (1 - p^+) b_i}$$

where,

$$a_i = \prod_{j=1}^{J} (\alpha_j)^{\mathbf{I}(L_{ij}=+)} (1-\alpha_j)^{\mathbf{I}(L_{ij}=-)}, \quad b_i = \prod_{j=1}^{J} (\beta_j)^{\mathbf{I}(L_{ij}=-)} (1-\beta_j)^{\mathbf{I}(L_{ij}=+)}$$

M-step: RY updates the parameters and the prior probability of the positive class as follows:

$$\alpha_j = \frac{a_j^+ - 1 + \sum_{i=1}^{I} \mu_i L_{ij}}{a_j^+ + a_j^- - 2 - \sum_{i=1}^{I} \mu_i}$$

$$\beta_j = \frac{b_j^+ - 1 + \sum_{i=1}^{I} (1-\mu_i)(1-L_{ij})}{b_j^+ - b_j^- - 2 + \sum_{i=1}^{I} (1-\mu_i)}$$

$$p^+ = n^+ - 1 + \sum_{i=1}^{I} \mu_i / (n^+ - n^- - 2 + I)$$

The RY algorithm is implemented by SQUARE (Sheshadri and Lease, 2013); we have integrated its code in a class called `SquareIntegration`. When inferring the ground truth, we must specify the name of an algorithm `SquareIntegration.methodRYBinary` if we want to use

it.

```
String consensusName = "…";
if (consensusName.equals(SquareIntegration.methodZenCrowd)) {
   SquareIntegration si = new SquareIntegration(tempDir);
   si.doInference(trainSet, SquareIntegration.methodZenCrowd);}
if (consensusName.equals(SquareIntegration.methodRYBinary)) {
   SquareIntegration si = new SquareIntegration(tempDir);
   si.doInference(trainSet, SquareIntegration.methodRYBinary);}
```

5.2.6 ZenCrowd

ZenCrowd (Demartini et al., 2012) only uses a binary parameter {*good*, *bad*} to model the *reliability* of an annotator. It is a little more complex than MV, but simpler than the other methods above. It is used for tackling the problem of entity linking for large collections of online pages.

E-step: ZenCrowd calculates the *reliability* of each annotator which is defined as:

$$P(u_j = reliable) = \sum_{i=1}^{I} \mathbf{I}(L_{ij} = \hat{y}_i) / \sum_{k=1}^{K} \sum_{i=1}^{I} n_{ik}$$

M-step: ZenCrowd uses *reliabilities* of annotators to update the probability of an example belonging to a specific class.

$$P(y_i = c_k) = \frac{\prod_{j=1}^{J} [P(u_j = reliable)]^{\mathbf{I}(\hat{y}_i = c_k)}}{\sum_{k=1}^{K} \prod_{k=1}^{w} [P(u_j = reliable)]^{\mathbf{I}(\hat{y}_i = c_k)}}$$

The ZenCrowd algorithm is implemented by SQUARE (Sheshadri and Lease, 2013); we have integrated its code in a class called `SquareIntegration`. When inferring the ground truth, we must specify the name of an algorithm `SquareIntegration.methodZenCrowd` if we want to use it.

5.2.7 KOS

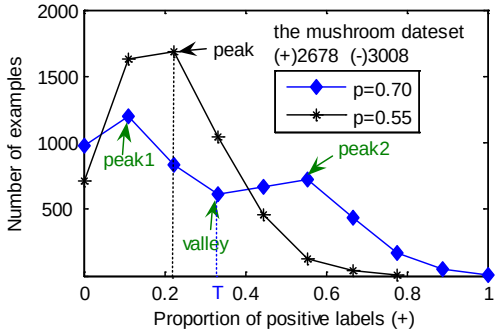
KOS (Karger, Oh, and Shah, 2011) was motivated by the reality of differences in labeling quality among different labelers. The authors sought to infer what they call the "correct answers" to "tasks," analogous to the ground-truth labels of examples. Their algorithm operates under the paradigm of two types of messages—task messages and worker messages—that dependently update themselves iteratively. It represents workers and the tasks they are assigned in the form of a graph of worker nodes and task nodes where, say, node w is connected to node t if worker w is assigned task t. The messages are essentially quantitative commentaries on the quality of each worker and

each task. Using the information from the graph, the messages self-update k_{max} times, and then use the final task messages to infer the correct values for each task.

5.2.8 PLAT

PLAT (Zhang et al., preprint) was proposed to handle the imbalanced labeling issue. The basic principle of the PLAT algorithm is as follows. When using MV for label integration, as long as a half of the labels above in the multiple label set of an example are negative, we make the decision that the example is negative; otherwise it is positive, which means the decision boundary is 0.5 implicitly. However, when labeling is imbalanced, 0.5 shouldn't be treated as the appropriate decision boundary any more. Supposing that labelers have the tendency to provided negative labels, the decision boundary should move towards 0 (< 0.5). Therefore, PLAT achieves the goal of increasing the number of positive examples in the training set by dynamically estimating a deterministic threshold *T* for label integration.

In PLAT, the threshold *T* is concretized as a certain frequency of positive labels (denoted by f^+) in a multiple label set. PLAT first calculates the f^+ value of each multiple label set, and then groups the examples with (almost) the same f_+ values together. Under the imbalanced labeling, the labeling qualities on two classes (p_P and p_N) are different. Given an example where the true label is positive, the number of positive labels (*k*) obeys a binomial distribution $b(k; R, p_P)$, where *R* is the size of the multiple noisy label set. Similarly, for a true negative example, the number of positive labels (*k*) also obeys a binomial distribution $b(k; R, 1-p_N)$. PLAT introduces a heuristic procedure *EstimateThresholdPosition* to estimate the optimal threshold *T*. *EstimateThresholdPosition* analyzes the distribution of the positive labels of all samples and obtains the estimation (denoted by *t*) of the threshold *T*. This procedure is illustrated in the following figure, which shows the *Positive Frequency Distribution (PFD) curve* of examples. If there exist two peaks in the *PFD* curve, *T* is the x-axis value of the valley between the two peaks (as diamond-marked line). Otherwise, there only exists one peak (as asterisk-marked line) in the *PFD* curve, and then *T* is the x-axis value of this unique peak.



After *T* is estimated as *t*, PLAT induces the integrated label from the multiple label set of each instance in the training set based on the threshold *T.* The examples with $f^+ > t$ are assigned an integrated positive label. For those examples with $f^+ \le t$, they may be assigned integrated negative labels at a very high probability. During this procedure, PLAT tries to keep the ratio of the numbers of *integrated* positive and negative examples close to the true underlying class distribution of the training set. Note that the true underlying class distribution is unknown; it is estimated in the former procedure *EstimateThresholdPosition.*

5.2.9 Adaptive Weighted Majority Voting (unpublished)

Adaptive Weighted Majority Voting (AWMV) is proposed to handle the imbalanced labeling issue. It is very similar to the PLAT algorithm (Zhang et al. preprint).

In MV, we implicitly assume that a negative label has the same weight as a positive label. Here we state that each weight is 0.5, so that the summation of the two weights is 1. Under biased labeling circumstances, we can set a *Bias Rate r* to adjust the weights for two classes as follows.

$$\begin{cases} w_N = (1-r)*0.5 \\ w_P = 1-w_N \end{cases}, \ 0 \le \mathrm{r} \le 1$$

In AWMV algorithm, the bias rate *r* is defined as:

$$r = \begin{cases} 8 {f_{+T}}^2 - 4 f_{+T} + 1, & f_{+T} \in [0,\ 0.25] \\ -8 {f_{+T}}^2 + 4 f_{+T}, & f_{+T} \in (0.25,\ 0.5] \end{cases}$$

Where f_{+T} is the Threshold of Positive Label Frequency ( ) which describes the bias. This value can be estimated by using the same procedure *EstimateThresholdPosition.*

After we obtain these two weights, the probability of an example e_i to be negative and positive can be calculated as:

$$\begin{cases} \Pr(\hat{y}_i = '+') = n_+^{(i)} w_P \,/\, (n_+^{(i)} w_P + n_-^{(i)} w_N) \\ \Pr(\hat{y}_i = '-') = 1 - \Pr(\hat{y}_i = '+') \end{cases}$$

The estimated class of this example is the one with the larger probability.

5.2.10 GTIC (unpublished)

GTIC is proposed by the authors of CEKA. Consider e_i with a noisy label set $\bar{l}_i$. $\bar{l}_i$ consists of labels belonging to classes c_1 to c_K, in which c_k occurs N_k times (i.e. $N_k = \sum_{j=1}^{J} \mathbf{I}(l_{ij} = c_k)$). We denote the probability of this example being a member of class *k* by a parameter θ_k. Then we have:

$$\boldsymbol{\theta} = [\theta_1, \theta_2, ..., \theta_K], \text{where } 0 \le \theta_k \le 1, \sum_{k=1}^{K} \theta_k = 1$$

According to the MAP estimation:

$$\hat{\theta}_k = \frac{N_k + \alpha_k - 1}{N + \sum_{j=1}^{K} \alpha_j - K}$$

The principle of our proposed method is based on the similarity measurement of probability vectors $\boldsymbol{\theta}$s of examples. For each example, we treat θ_k as its *k*th feature vector, and then we use a clustering algorithm to cluster similar examples together. Examples in the same cluster belong to the same class. In addition, we also generate the *(K+1)*th feature (denoted as θ_z) calculated by

$$\theta_z = \frac{1}{K} \sum_{k=1}^{K-1} (\theta_{k+1} - \theta_k)$$

After *K+1* features are generated, an example *i* is denoted by e_i=<(θ_1,…, θ_k, θ_z,), *y*>, where (θ_1,…, θ_k, θ_z) is its feature portion used for clustering, and *y* is its unknown true label.

Algorithm Ground Truth Inference using Clustering (GTIC)

Input: A sample set *E* in which each e_i has a multiple noisy label set and has no true label, the number of class *K*

Output: A sample set *E* in which each e_i has an estimated label

1. For each e_i in *E*, use Equations 6 and 7 to generate its *K+1* features, i.e., $\boldsymbol{\theta}^{(i)} = (\theta_1^{(i)}, ..., \theta_k^{(i)}, \theta_z^{(i)})$.
2. Select a *K*-centroid set Φ based on the $\boldsymbol{\theta}$s of the examples.
3. Run the K-Means clustering algorithm with Euclidean distance by setting Φ as the initial centroids.
4. For each cluster *s* sized $M^{(s)}$ obtained from K-Means, create a vector $\boldsymbol{\tau}^{(s)}$ whose element $\tau_k^{(s)}$ is calculated using $\tau_k^{(s)} = \sum_{i=1}^{M^{(s)}} \theta_k^{(i)}$, where $1 \le s \le K$.
5. For each cluster *s*, based on its vector $\boldsymbol{\tau}^{(s)}$, assign this cluster with the class $\kappa^{(s)} = \arg\max_k \{\tau_k^{(s)}\}$ under the constraint that a cluster is mapped to one and only one class.
6. Assign each e_i an inferred label according to the label of each cluster and return *E*.

6 Noise Handling Algorithms

6.1 Introduction

Obviously, crowdsourced data after inference still has mislabeled instances, i.e., label noise. Therefore, it would be natural to use noise handling techniques to identify and correct this noise. The principle behind this is that human intelligence is not always superior to machine intelligence, especially when the labelers are not experts. For example, in the text classification tasks conducted by Shinsel et al (2011), although each instance obtained six labels from different labelers, the overall integrated accuracy is still less than 94%, which is significantly inferior to the performance gained by a machine learning algorithm SVM, whose accuracy is greater than 99% (Frank and Bouckaert 2006). Thus, a potential high quality learning model may have a great opportunity to improve label accuracy.

6.2 Noise Filtering

When handing noise, the first step is to identify noises and separate them from the original data set, which is called noise filtering. CEKA provides an abstract class `Filter` to define some uniform functions for different specific filtering algorithms.

6.2.1 Class `Filter`

The abstract class `Filter` provides the following function to filter noise.

Function	`abstract void filterNoise(Dataset dataset, Classifier[] classifier) throws Exception`	
	Comments: (1) This function identifies the potential mislabeled instances. (2) If this function successes, it will create two data sets called noise data set and cleansed data set. The noise data set contains the instances that have high probability of being mislabeled and the cleansed data set contains the instances that probably have the correct class labels. (3) The parameter `dataset` doesn't change after the function is called. (4) Many algorithms use one or multiple classifiers to build models for the subsequent noise identification procedure.	
Parameters	`Dataset dataset`	the data set that the function to process

	`Classifier[] classifier`	a group of classifiers that will be used in the noise filtering procedure
Return	`void`	

After the above function has been called, we can use the following functions to get the noise data set and the cleansed data set.

Function	`Dataset getCleansedDataset()`	
	Comments: get the cleansed data set	
Return	`Dataset`	the cleansed data set

Function	`Dataset getNoiseDataset()`	
	Comments: get the noise data set	
Return	`Dataset`	the noise data set

The filtering algorithms described below are all subclasses of `Filter`, so these three functions work for them.

6.2.2 Classification Filtering

The class `ClassificationFilter` implements the classification filtering algorithm (Gamberger et al., 1999). The main steps of the algorithm are as follows.

Algorithm Classification Filter

1. Split the data set *TR* using an k-fold cross validation scheme
2. For each of these *k* parts, a learning algorithm is trained on the other *n-1* parts, resulting in n different classifiers.
3. These *n* resulting classifiers are used to tag each instance in the excluded part as either correct or mislabeled, by comparing the train label with that assigned by the classifer.
4. The misclassified examples are added to noise data set
5. Remove noisy examples from *TR*, the remaining examples in *TR* is the cleansed data set.

When creating the object of the class `ClassificationFilter`, the number of the fold should be specified through a parameter.

Function	`ClassificationFilter (int nFold){`	
	Comments: create an object of the class `ClassificationFilter`	
parameter	`int nFold`	the number of fold

6.2.3 Majority Voting Filtering

The class `MajorityFilter` implements the classification filtering algorithm (Brodley and Friedl, 1999). The main steps of the algorithm are as follows.

Algorithm Majority Filter

1. Split the data set *TR* using an k-fold cross validation scheme
2. For each of these *k* parts, *m* learning algorithms are trained on the other *n-1* parts, and the resulting *m* classifiers are used to tag each instance from the *k*th part as labeled or mislabeled.
3. The examples that more than half of the *m* classifiers tagged as mislabeled are added to the noise data set.
4. Remove noisy examples from *TR*, the remaining examples in *TR* is the cleansed data set.

When creating an object of class `MajorityFilter`, the number of folds should be specified through the parameter `nFold`.

Function	`MajorityFilter (int nFold){`	
	Comments: create an object of the class `MajorityFilter`	
parameter	`int nFold`	the number of folds

6.2.4 Iterative Partitioning Filtering

The class `IterativePartitionFilter` implements the iterative partitioning filtering algorithm (Khoshgoftaar and Rebours, 2007). The main steps of the algorithm are as follows.

Algorithm Iterative Partitioning Filter

1. Partition the data set *TR* using a k-fold cross validation scheme.
2. Train a learning algorithm on each of these k folds, resulting in *k* classifiers.
3. Depending upon voting scheme (majority or consensus), add each example to noise data set for which less than half of the classifiers or less than all of the classifiers, respectively, tag it as mislabeled.
4. Record the number of examples added to noise data set each iteration.
5. Repeat steps 1-4 until each of the last three iterations fail to add at least a specified number of examples to the noise data set, specified by a percentage parameter.

When creating an object of class `IterativePartitionFilter`, the number of folds, the voting scheme, and the percentage parameters must all be specified.

Function	`IterativePartitionFilter (int nFold, String votingScheme, double percentage){`	
	Comments: create an object of the class `IterativePartitionFilter`	
parameters	`int nFold`	the number of folds
	`String votingScheme`	The voting scheme used
	`double percentage`	proportion of examples (see step 5)

6.2.5 Multiple Partitioning Filtering

The class `MultiplePartitioningFilter` implements the multiple partitioning filtering algorithm (Khoshgoftaar and Rebours, 2007). The main steps of the algorithm are as follows.

Algorithm Multiple Partition Filter

1. Partition the data set *TR* using a *k*-fold cross validation scheme.
2. Train *m* learning algorithms on each of these *k* folds, resulting in *mk* classifiers.
3. Tag each example in *TR* as correctly labeled or mislabeled by each of the *mk* classifers.
4. For a specified filtering level *fl*, add all examples that were tagged as mislabeled at least *fl* times to the noise data set.

When creating an object of class `MultiplePartitioningFilter`, the number of folds and the filtering level must be specified.

Function	`MultiplePartitioningFilter (int nFold, int filteringLevel){`	
	Comments: create an object of the class `MultiplePartitioningFilter`	
parameters	`int nFold`	the number of folds
	`int filteringLevel`	the filtering level

6.3 Noise Correction

6.3.1 Self-Training Correction Algorithm

The class `SelfTrainCorrection` implements the self-training noise correction algorithm (Triguero et al., 2014). The main steps of the algorithm are as follows.

Algorithm Self-Training Correction

1. Train a learning algorithm on the clean data set.
2. Use the resulting classifier to obtain a probability distribution for each noisy example, representing its likelihood of belonging to each class.

3. For each class, find a number of examples from the noisy data set which have the greatest likelihood of belonging to that class, set their class value to that class, and move it to the clean data set (the correction step).
4. Repeat steps 1-3 while there are still examples to correct.
5. Return new noisy data set and clean data set.

When creating an object of class `SelfTrainCorrection`, the clean data set, noisy data set, and proportion of instances to correct must all be specified.

Function	`SelfTrainCorrection (Dataset cleanExamples, Dataset noisyExamples, double proportion){`	
	Comments: create an object of the class `SelfTrainCorrection`	
parameters	`Dataset cleanExamples`	the clean data set, determined from a prior noise filter execution
	`Dataset noisyExamples`	the noisy data set, determined from a prior noise filter execution
	`double proportion`	the proportion of instances to correct

The class `STCConfidence` implements another version self-training noise correction algorithm (Triguero et al., 2014) with a confidence level. The main steps of the algorithm are as follows.

Algorithm STC Confidence

1. Train a learning algorithm on the clean data set.
2. Use the resulting classifier to obtain a probability distribution for each noisy example, representing its likelihood of belonging to each class.
3. For each example that has a likelihood of belonging to a certain class that is greater than the predefined confidence threshold value, set its class value to the likely value, and move it to the clean data set.
4. Repeat steps 1-3 as long as there is at least one example moved to the clean data set.
5. Return new noisy data set and clean data set.

When creating an object of class `STCConfidence`, the clean data set, noisy data set, and confidence threshold must all be specified.

Function	`STCConfidence (Dataset cleanExamples, Dataset noisyExamples, double threshold){`	
	Comments: create an object of the class `STCConfidence`	
parameters	`Dataset`	the clean data set, determined from a prior

	`cleanExamples`	noise filter execution
	`Dataset noisyExamples`	the noisy data set, determined from a prior noise filter execution
	`double threshold`	the confidence threshold

6.3.2 Polishing Labels Algorithm

The class `PolishingLabels` implements the label-polishing noise correction algorithm (Nicholson, Zhang, and Sheng, preprint). The main steps of the algorithm are as follows.

Algorithm Polishing Labels

1. Split the data set TR into 10 folds.
2. Train a learning algorithm on each fold, resulting in 10 classifiers.
3. Have the 10 classifiers label each example in *TR*.
4. For each example in *TR*, set its class equal to the class with the most votes from the 10 classifiers. Break ties randomly, unless the example's original class is involved in the tie, then do nothing.

When creating an object of class `PolishingLabels`, the learning algorithm to use must be specified.

Function	`PolishingLabels (Classifier classifier){`	
	Comments: create an object of the class `PolishingLabels`	
parameter	`Classifier classifier`	the learning algorithm

6.3.3 Cluster Correction Algorithm (unpublished)

The class `ClusterCorrection` implements its namesake noise-correction algorithm (Nicholson, Zhang, and Sheng, unpublished). The main steps of the algorithm are as follows.

Algorithm Cluster Correction

1. Perform *a* clusterings of the data set *TR*.
2. For each cluster in each clustering, add to each example in that cluster a set of weights that signify its likely class value given the other examples in that cluster.
3. For each example, take the class corresponding to its maximum weight value, and assign that class to the example.

When creating an object of class `ClusterCorrection`, the clean data set, noisy data set, and proportion of instances to correct must all be specified.

Function	`ClusterCorrection (Dataset dataset, String datasetPath, Clusterer[] clusterers){`	
	Comments: create an object of the class `ClusterCorrection`	
parameters	`Dataset dataset`	the training set
	`String datasetPath`	the path to the .arff file of data set
	`Clusterer[] clusterers`	the set of clusterers with which to cluster the training set

6.3.4 Adaptive Voting Noise Correction (unpublished)

The package ceka.noise.avnc implements adaptive voting noise correction algorithm (Zhang et al., unpublished). AVNC is proposed to identify and correct the most likely noisy examples with the help of the estimated quality information of labelers provided by inference algorithms.

The following sample code illustrates how to use AVNC to correct noise after the ground truth inference.

```
/*dataset is an object of class Dataset that has been inferred by an
inference algorithm*/
/*statistical info of workers*/
WorkerStat workerStat = new WorkerStat();
double estimatedMeanProb =
   workerStat.calculateEstimatedMeanAcc(dataset);
double integratedCorrectProb =
   Stochastics.binomialIntegration(numberOfLabeler,
   estimatedMeanProb);
int nFold = 10;
int nModel = 5;
AdaptiveClassificationFilter acf
   = new AdaptiveClassificationFilter(nFold, nModel);
acf.setMinEstimatedNoiseProportion(1 - integratedCorrectProb);
acf.setMaxEstimatedNoiseProportion(1 - estimatedMeanProb);
/*filtering noise*/
acf.filterNoise(dataset, classifier);
Dataset cleansedSet = noiseFilter.getCleansedDataset();
Dataset noiseData = noiseFilter.getNoiseDataset();
Dataset [] highDatasets = acf.getHighQualityDatasets();
```

```
Classifier [] classifiers = new Classifier[highDatasets.length];
/* use SMO classifier */
for (int i = 0; i < classifiers.length; i++) {
   Class<?> m_class = Class.forName("weka.classifiers.functions.SMO");
   classifiers[i] = (Classifier) m_class.newInstance();
   ((SMO)classifiers[i]).setKernel(new NormalizedPolyKernel());
}
/*correction*/
VoteCorrection corrector = new VoteCorrection();
corrector.correct(noiseData, highDatasets, classifiers, (int)
(highDatasets.length * 0.5));
/*performance evaluation*/
for (int i = 0; i < noiseData.getExampleSize(); i++)
   cleansedSet.addExample(noiseData.getExampleByIndex(i));
PerformanceStatistic perfStat = new PerformanceStatistic();
perfStat.stat(cleansedSet);
```

7 Evaluation and Simulation

7.1 Performance Measures

The class `PerformanceStatistic` can be applied to a data set to get the performance measures. The statistics of the performance can be processed by the following function.

Function	`void stat(Dataset dataset)`	
	Comments: after a dataset has been processed by an algorithm, this function is used for calculating the performance statistic information	
Parameter	`Dataset dataset`	the data set that the function to process
Return	`void`	

After above function is called, we can get the metrics such as accuracy, recall, precision, F score, AUC and multi-class AUC through their corresponding “getXXX()” functions.

Example: the following sample code shows an entire inference procedure with the performance evaluation.

```
/*dataset is an object of class Dataset*/
/*use MV to infer the integrated label*/
MajorityVote mv = new MajorityVote();
mv.doInference(dataset);}
/*get the performance */
PerformanceStatistic reporter = new PerformanceStatistic();
reporter.stat(dataset);
System.out.println("PLAT accuracy: " + reporter.getAccuracy() + " Roc
Area: " + reporter.getAUC() + " Recall: " + reporter.getRecallBinary()
+ " Precision: " + reporter.getPresicionBinary()+ " F1:" +
reporter.getF1MeasureBinary());
```

7.2 Package ceka.simulation

Research in crowdsourcing is not always conducted on real-world datasets. Sometimes, we need to simulate the behaviors of labelers or generate a sub data set from an entire one. The package ceka.simulation provides some simple classes to do these tasks.

7.2.1 Class ExampleMask

The class `ExampleMask` is used for selecting the examples to be saved to files. It provides the following functions.

Function	`void intialize(Dataset data)`	
	Comments: initialize the mask with a data set. All examples in the data set will be selected (active).	
Parameter	`Dataset dataset`	the data set that the function processes
Return	`void`	

Function	`void disableExample (String exampleId)`	
	Comments: disable an example through ID. The example will not be selected (inactive).	
Parameter	`String exampleId`	the ID of the example to be inactive
Return	`void`	

Function	`void enableExample (String exampleId)`	
	Comments: enable an example through ID. The example will be selected (active).	
Parameter	`String exampleId`	the ID of the example to be active
Return	`void`	

Function	`boolean isActiveExample(String exampleId)`	
	Comments: query the status of an example	
Parameter	`String exampleId`	the ID of the example to query
Return	`boolean`	status

7.2.2 Class ExampleWorkersMask

The class `ExampleWorkersMask` is used for selecting noisy labels of each example to be saved to files (supposing each worker at most provides one label for each instance). It provides the following functions.

Function	`void intialize(Dataset data)`	
	Comments: initialize the mask with a data set. All workers of each example in the data set will be selected (active).	
Parameter	`Dataset dataset`	the data set that the function processes
Return	`void`	

Function	`ArrayList<String> getWorkerMask(String exampleId)`	
	Comments: get the worker mask of an example.	
Parameter	`String exampleId`	the ID of the example to query
Return	`ArrayList<String>`	the IDs of the active workers of this example

Function	`void sequentialSelect(int numberOfLabels)`	
	Comments: consecutively select N labels. For example, if N=3, then select the first 3 labels from the multiple noisy label set of this example.	
Parameter	`int numberOfLabels`	the number of consecutive labels to select
Return	`void`	

Function	`void randSelect(int numberOfLabels)`	
	Comments: randomly select N labels. For example, if N=3, then randomly select the first 3 labels from the multiple noisy label set of this example	
Parameter	`int numberOfLabels`	the number of labels to be selected
Return	`boolean`	status

Example: the following sample code shows how to use a basic data set to generate a new data set with the help of mask.

```
/*dataset is an object of class Dataset, the original data set*/
/*suppose the number of labels of each instance is greater than 10*/
/*randomly select 1-10 labels for each instance to create
  new sub data set*/
String prefix = “E:\\Ceka\\dataset\myDataName”;
for (int numLabels = 1; numLabels < 10 numLabels++) {
   ExampleWorkersMask mask = new ExampleWorkersMask();
   mask.initialize(dataset);
   mask.randSelect(numLabels);
   String numStr = new Integer(numLabels).toString();
   String subResponsePath = prefix + numStr + ".response.txt";
   String subArffxPath = prefix + numStr + ".arffx";
   FileSaver.saveDatasetResponseArffx(dataset, subResponsePath, null,
subArffxPath, mask);
   /* load new sub data set*/
   Dataset subDataset =  FileLoader.loadFileX(subResponsePath, null,
subArffxPath);
}
```

Thus, we can use classes `ExampleMask` and `ExampleWorkersMask` to create new sub data sets with the help of the disk files.

7.2.3 Simulation of workers

CEKA provides simple functions to simulate workers. The simulation can be conducted via classes `GaussianLabelingStrategy`, `SingleQualLabelingStrategy` and `MockWorker`. Class `GaussianLabelingStrategy` simulates the labeling quality of each worker that is similar to a Guassian distribution with the parameters (mean, std. deviation). The labeling quality provided by this class is in the range of [mean - std. devation, mean + std.deviation]. `SingleQualLabelingStrategy` simulates the labeling quality of each worker with a uniform probability. `MockWorker` is a subclass of the class `Worker`. Classes `GaussianLabelingStrategy` and `SingleQualLabelingStrategy` apply to the class `MockWorker` to generate the object with desired labeling quality. `GaussianLabelingStrategy` and `SingleQualLabelingStrategy` are inherited from the abstract class `LabelingStrategy` which provides the following functions for generating simulated workers and labeling instances.

Function	`void assignWorkerQuality(MockWorker [] workers)`	
	Comments: assign labeling quality to a set of workers	
Parameter	`MockWorker [] workers`	a set of workers to whom to assign labeling qualities

Return	`void`	

Function	`void labelDataset(Dataset dataset, MockWorker worker)`	
	Comments: simulate a worker to label the entire dataset, which means each instance will get a label from this worker.	
Parameter	`Dataset dataset`	the data set to be labeled
	`MockWorker worker`	the worker who labels the data set
Return	`void`	

Example: the following sample code shows how to simulate multiple workers labeling a data set.

```
/*dataset is an object of class Dataset*/
/*MAX_WKR is the maximum number of workers*/
double meanQ = 0.7;
double stdQ = 0.1;
MockWorker [] mockWorkers = new MockWorker[MAX_WKR];
GaussianLabelingStrategy strategy
   = new GaussianLabelingStrategy(meanQ, stdQ);
for (int j = 0; j < MAX_WKR; j++) {
   mockWorkers[j] = new MockWorker(new Integer(j).toString());
}
strategy.assignWorkerQuality(mockWorkers);
for (int j = 0 ; j < MAX_WKR; j++) {
   /*do labeling*/
   mockWorkers[j].labeling(trainSet, strategy);
}
```

8 References

Carla E Brodley and Mark A Friedl. Identifying mislabeled training data. Journal of Artificial Intelligence Research, 11:131–161, 1999.

Alexander Philip Dawid and Allan M Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. Applied statistics, pages 20–28, 1979.

Gianluca Demartini, Djellel Eddine Difallah, and Philippe Cudr´e-Mauroux. Zencrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity link-ing. In World Wide Web, pages 469–478. ACM, 2012.

Eibe Frank and Remco R Bouckaert. Naive bayes for text classication with unbalanced classes. In Knowledge Discovery in Databases: PKDD 2006, pages 503–510. Springer, 2006.

Dragan Gamberger, Nada Lavrac, and Ciril Groselj. Experiments with noise .filtering in a medical domain. In ICML, pages 143–151, 1999.

Mark Hall, Eibe Frank, Geo.rey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. ACM SIGKDD explorations newsletter, 11(1):10–18, 2009.

Je. Howe. The rise of crowdsourcing. Wired magazine, 14(6):1–4, 2006.

David R Karger, Sewoong Oh, and Devavrat Shah. Iterative learning for reliable crowd-sourcing systems. In NIPS, pages 1953–1961, 2011.

Taghi M Khoshgoftaar and Pierre Rebours. Improving software quality prediction by noise .filtering techniques. Journal of Computer Science and Technology, 22(3):387–396, 2007.

Quoc Viet Hung Nguyen, Thanh Tam Nguyen, Ngoc Tran Lam, and Karl Aberer. Batc: a benchmark for aggregation techniques in crowdsourcing. In ACM SIGIR, pages 1079– 1080. ACM, 2013.

Vikas C Raykar, Shipeng Yu, Linda H Zhao, Gerardo Hermosillo Valadez, Charles Florin, Luca Bogoni, and Linda Moy. Learning from crowds. The Journal of Machine Learning Research, 11:1297–1322, 2010.

Aashish Sheshadri and Matthew Lease. Square: A benchmark for research on computing crowd consensus. In First AAAI Conference on Human Computation and Crowdsourcing, 2013.

Amber Shinsel, Todd Kulesza, Margaret Burnett, William Curran, Alex Groce, Simone Stumpf, and Weng-Keen Wong. Mini-crowdsourcing end-user assessment of intelligent assistants: A cost-benet study. In Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on, pages 47–54. IEEE, 2011.

Isaac Triguero, Jos´e A S´aez, Juli´an Luengo, Salvador Garc´ia, and Francisco Herrera. On the characterization of noise .lters for self-training semi-supervised in nearest neighbor classification. Neurocomputing, 132:30–41, 2014.

Jacob Whitehill, Ting-fan Wu, Jacob Bergsma, Javier R Movellan, and Paul L Ruvolo. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In NIPS, pages 2035–2043, 2009.

Jing Zhang, Xindong Wu, and Victor S Sheng. Imbalanced multiple noisy labeling. IEEE Transaction on Knowledge and Data Engineering, preprint.