spark-crowd Documentation

Release 0.2.1

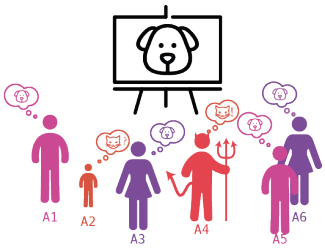
Enrique G. Rodrigo Juan A. Aledo Jose A. Gamez

Nov 07, 2018

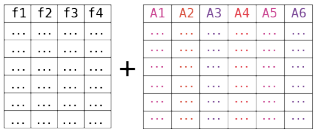
CONTENTS

Learning from crowdsourced Big Data

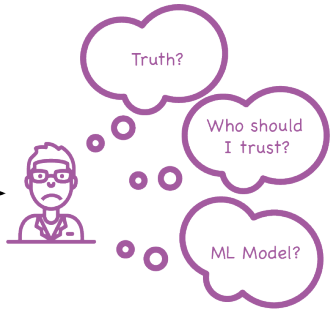
Learning from crowdsourced data imposes new challenges in the area of machine learning. This package, *spark-crowd*, helps practitioners when dealing with this kind of data at scale, using Apache Spark.



Each instance is labeled by several annotators, with different backgrounds and goals. This may influence the accuracy of the results obtained from the data.



The dataset obtained has several labels for each item, but not the ground truth, which is usually necessary for machine learning algorithms.



There are three main problems when learning from crowdsourced data: identifying which annotators to trust, estimating the ground truth label and learning a machine learning model.

The main features of *spark-crowd* are the following:

- It implements well-known methods for learning from crowdsourced labeled data.
- It is suitable for working with both large and small datasets.
- It uses Apache Spark, which allows the code to run in different environments, from a computer to a multi-node cluster.
- It is suitable both for research and production environments.
- It provides an easy to use API, allowing the practitioner to start using the library in minutes.
- It is licensed under the MIT license.

See the *Quick Start* to started using *spark-crowd*

CHAPTER

ONE

QUICK START

You can easily start using `spark-crowd` package through our docker image or through Spark Packages. See *Installation*, for all the installation alternatives (such as how to add the package as a dependency in your project).

1.1 Start with our docker image

The quickest way to try out the package is through the provided docker image with the latest version of the package, as you do not need to install any other software (apart from docker).

```
docker pull enriquegrodrigo/spark-crowd
```

Thanks to it, you can run the examples provided along with the package. For example, to run *DawidSkeneExample.scala* we can use:

```
docker run --rm -it -v $(pwd)/:/home/work/project enriquegrodrigo/spark-crowd␣
↪DawidSkeneExample.scala
```

You can also open a spark shell with the library preloaded.

```
docker run --rm -it -v $(pwd)/:/home/work/project enriquegrodrigo/spark-crowd
```

By doing that, you can test your code directly. You will not benefit from the distributed execution of Apache Spark, but you are still able to use the algorithms with medium-sized datasets (since docker can use several cores in your machine).

1.2 Start with Spark Packages

If you have an installation of Apache Spark, you can open a *spark-shell* with our package pre-loaded using:

```
spark-shell --packages com.enriquegrodrigo:spark-crowd_2.11:0.2.1
```

Likewise, you can submit an application to your cluster that uses *spark-crowd* using:

```
spark-submit --packages com.enriquegrodrigo:spark-crowd_2.11:0.2.1 application.scala
```

To use this option you do not need to have a cluster of computers, you may also execute the code from your local machine because Apache Spark can be installed locally. For more information on how to install Apache Spark, please refer to its homepage.

1.3 Basic usage

Once you have chosen a procedure to run the package, you have to import the method that you want to use as well as the types for your data, as you can see below:

```
import com.enriquegrodrigo.spark.crowd.methods.DawidSkene
import com.enriquegrodrigo.spark.crowd.types.MulticlassAnnotation

val exampleFile = "examples/data/multi-ann.parquet"

val exampleData = spark.read.parquet(exampleFile).as[MulticlassAnnotation]

//Applying the learning algorithm
val mode = DawidSkene(exampleData)

//Get MulticlassLabel with the class predictions
val pred = mode.getMu().as[MulticlassLabel]

//Annotator precision matrices
val annprec = mode.getAnnotatorPrecision()
```

You can find a description of the code below:

1. First the method and the type are imported, in this case `DawidSkene` and `MulticlassAnnotation`. The type is needed as the package API only accepts typed datasets for the annotations.
2. Then the data file (provided with the package) is loaded. It contains annotations for different examples. As you can see, the example uses the method `as` to convert the Spark DataFrame in a typed Spark Dataset (with type `MulticlassAnnotation`).
3. To execute the model and obtain the result you can use the model name directly. This function returns a `DawidSkeneModel`, which includes several methods to obtain results from the algorithm.
4. The method `getMu` returns the ground truth estimations made by the model.
5. We use `getAnnotatorPrecision` to obtain the annotator quality calculated by the model.

You can consult the models implemented in this package in *Methods*, where you can find a link to the original article for the algorithm.

CHAPTER

TWO

INSTALLATION

There are three alternatives to use the package in your own software:

- Using the package directly from Spark Packages
- Adding it as a dependency to your project through Maven central.
- Compiling the source code and using the `jar` file.

Alternatively, if you just want to execute simple scala scripts locally, you can use the provided docker image as explained in *Quick Start*

2.1 Using Spark Packages

The easiest way of using the package is through Spark Packages, as you only need to add the package in the command line when running your application:

```
spark-submit --packages com.enriquegrodrigo:spark-crowd_2.11:0.2.1 application.scala
```

You can also open a *spark-shell* using:

```
spark-shell --packages com.enriquegrodrigo:spark-crowd_2.11:0.2.1
```

2.2 Adding `spark-crowd` as a dependency

In addition to Spark Packages, the library is also in Maven Central, so you can add it as a dependency in your scala project. For example, in *sbt* you can add the dependency as shown below:

```
libraryDependencies += "com.enriquegrodrigo" %% "spark-crowd" % "0.2.1"
```

This allows you to use the methods inside your Apache Spark projects.

2.3 Compiling the source code

To build the package using *sbt* you can use the following command inside the `spark-crowd` folder:

```
sbt package
```

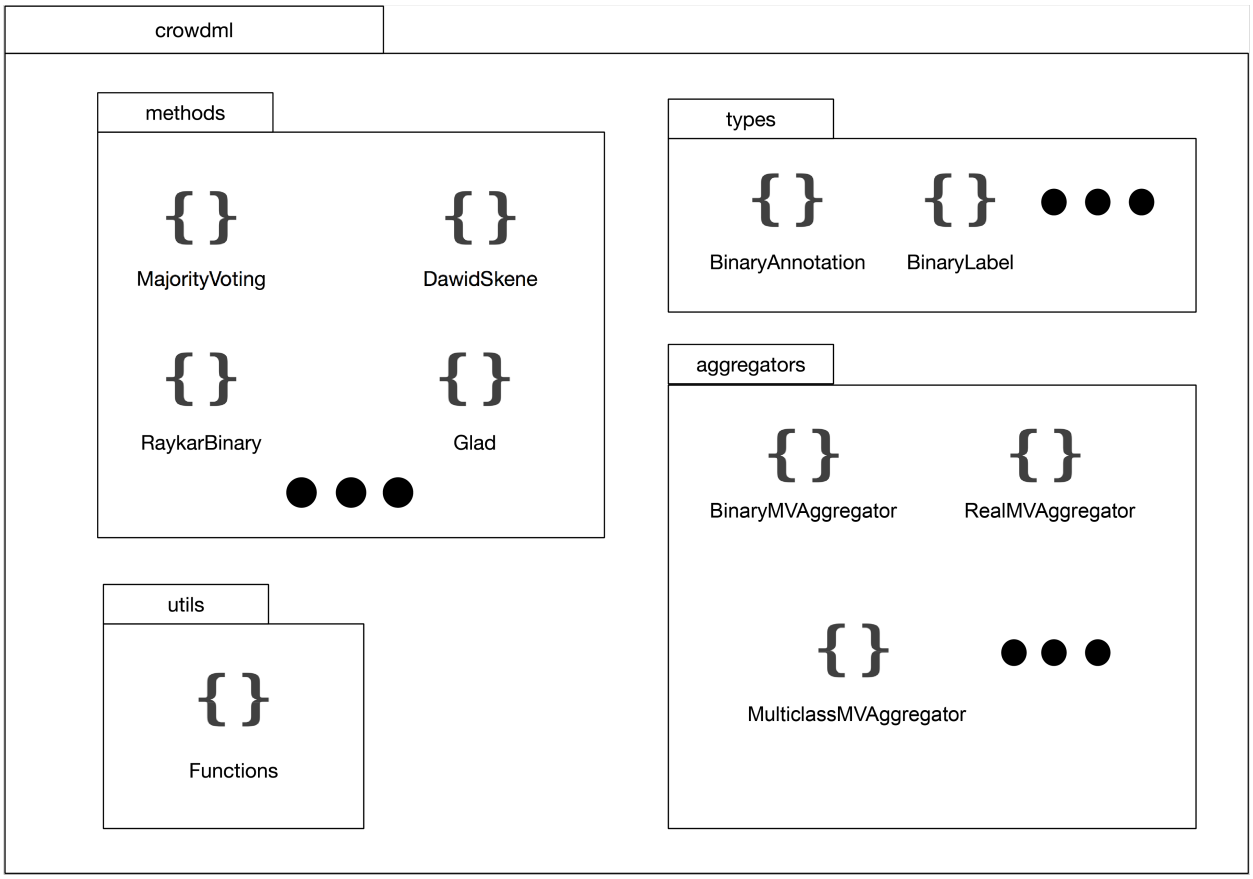
It generates a compiled `jar` file that you can add to your project.

CHAPTER

THREE

DESIGN AND ARCHITECHTURE

The package design can be found in the figure below.



Although the library contains several folders, the only important ones for the users are the `types` folder, and the `methods`. The other folders contain auxiliary functions for some of the methods. Specifically, it is interesting to explore the data types, as they are essential to understand how the package works, as well as the common interface of the methods.

3.1 Data types

The package provides types for annotation datasets and ground truth datasets, as they usually follow the same structure. These types are used in all the methods so you need to convert your annotations dataset to the correct format accepted by the algorithm.

There are three types of annotations that the package supports for which we provide Scala case classes, making it possible to detect errors at compile time when using the algorithms:

- `BinaryAnnotation`: a dataset of this type provides three columns:
  - The `example` column (i.e the example for which the annotation is made).
  - The `annotator` column (representing the annotator that made the annotation).
  - The `value` column, (with the value of the annotation, that can take value 0 or 1).
- `MulticlassAnnotation`: The difference from `BinaryAnnotation` is that the value column can take more than two values, in the range from 0 to the total number of values.
- `RealAnnotation`: In this case, the value column can take any numeric value.

You can convert an annotation dataframe with columns: `example`, `annotator` and `value` to a typed dataset with the following instruction:

```
val typedData = untypedData.as[RealAnnotation]
```

In the case of labels, we provide 5 types of labels, 2 of which are probabilistic. The three non probabilistic types are:

- `BinaryLabel`. A dataset with two columns: `example` and `value`. The column `value` is a binary number (0 or 1).
- `MulticlassLabel`. A dataset with the same structure as the previous one but where the column `value` can take more than two values.
- `RealLabel`. In this case, the column `value` can take any numeric value.

The probabilistic types are used by some algorithms to provide more information about the confidence of each class value for an specific example.

- `BinarySoftLabel`. A dataset with two columns: `example` and `prob`. The column `prob` represents the probability of the example being positive.
- `MultiSoftLabel`: A dataset with three columns: `example`, `class`, `prob`. This last column represents the probability of the example taking the class in the column `class`.

3.2 Methods

All the implemented methods are in the `methods` subpackage and are mostly independent of each other. The MajorityVoting algorithms are the the only exception, as most of the other methods use them in the initialization step. Apart from that, each algorithm is implemented in its specific file. This makes it easier to extend the package with new algorithms. Although independent, all the algorithms have a similar interface, which facilitates their use. To execute an algorithm, the user normally needs to use the `apply` method of the model (which in `scala`, is equivalent to applying the object itself), as shown below

```
...
val model = IBCC(annotations)
...
```

After the algorithm completes its execution, an object is returned, which has information about the ground truth estimations and other estimations that are dependent on the chosen algorithm.

The only algorithm that does not follow this pattern is `MajorityVoting`, which has methods for each of the class types and also to obtain probabilistic labels. See the API Docs for details.

CHAPTER

FOUR

METHODS

You can find the methods implemented in the library below. All of them contain a link to its API where you can find more information. Besides, in the table column, you can find a link to the reference.

Table 1: Methods implemented in spark-crowd

Method	Binary	Multiclass	Real	Reference
MajorityVoting	√	√	√	
DawidSkene	√	√		JRSS
IBCC	√	√		AISTATS
GLAD	√			NIPS
CGLAD	√			IDEAL
Raykar	√ RaykarBinary	√ RaykarMulti	√ RaykarCont	JMLR
CATD			√	VLDB
PM			√	SIGMOD
PMTI			√	VLDB2

Below, we provide a short summary of each method (see the references for the details).

4.1 MajorityVoting

With this, we refer to the mean for continuous target variables and the most frequent class (the mode) for the discrete case. Expressing these methods in terms of annotator accuracy, they suppose that all the annotators have the same experience. Therefore, their contributions are weighted equally. Apart from the classical mean and most frequent class, we also provide methods that return the proportion of each class value for each example. See the API Docs for more information.

4.2 DawidSkene

This method estimates the accuracy of the annotators from the annotations themselves. For this, it uses the EM algorithm, starting from the most frequent class and improving the estimations through several iterations. The algorithm returns both the estimation of the ground truth and the accuracy of the annotators (a confusion matrix for each). This algorithm is a good alternative when looking for a simple way of aggregating annotations without the assumption that all of the annotators are equally accurate.

4.3 IBCC

This method is similar to the previous one but uses probabilistic estimations for the classes. For each example, the model returns probabilities for each class, so they can be useful in problems where a probability is needed. It shows a good compromise between the complexity of the model and its performance, as can be seen in here or in our *own experimentation*.

4.4 GLAD

This method estimates both the accuracy of the annotators (one parameter per annotator) and the difficulty of each example (a parameter for each instance) through EM algorithm and gradient descent. This computational complexity comes at a cost of a higher execution time in general. However, GLAD, and its enhancement, CGLAD, also implemented here, are the only two algorithms capable of estimating the difficulty of the instances.

4.5 CGLAD

This method is an enhancement over the original GLAD algorithm to tackle bigger datasets more easily, using clustering techniques over the examples to reduce the number of parameters to be estimated. It follows a similar learning process to GLAD algorithm.

4.6 Raykar's algorithms

We implement the three methods proposed in the paper Learning from crowds (see the paper for the details) for learning from crowdsourced data when features are available. These methods use an annotations dataset and the features for each instance. The algorithms infers together a logistic model (or a regression model, for the continuous case), the ground truth and the quality of the annotators, which are all returned by the methods in our package.

4.7 CATD

This method estimates both the quality of the annotators (as a weight in the aggregation) and the ground truth for continuous target variables. From the annotations, it estimates which annotators provide better labels. Then, it assigns more weight to them for the aggregation. In the package, only the continuous version is implemented as other algorithms seem to work better in the discrete cases (see this paper for more information).

4.8 PM and PMTI

They are methods for continuous target variables. We implement two versions, one following the formulas appearing in the original paper, and the modification implemented in this package. The modification obtains better results in our experimentation (see *Comparison with other packages*).

CHAPTER

FIVE

EXAMPLES

In this page we provide examples for several of the algorithms in the library. You can find the data used for the examples in the Github repository.

5.1 MajorityVoting

The example below shows how to use the MajorityVoting algorithm for estimating the ground truth for a binary target variable.

```
import com.enriquegrodrigo.spark.crowd.methods.MajorityVoting
import com.enriquegrodrigo.spark.crowd.types.BinaryAnnotation

val exampleFile = "data/binary-ann.parquet"

val exampleDataBinary = spark.read.parquet(exampleFile).as[BinaryAnnotation]

val muBinary = MajorityVoting.transformBinary(exampleDataBinary)

muBinary.show()
```

The method returns a result similar to this one:

```
+-------+-----+
|example|value|
+-------+-----+
|     26|    0|
|     29|    1|
|    474|    0|
|    964|    1|
|     65|    0|
|    191|    0|
|    418|    1|
....
```

MajorityVoting algorithms assume that all the annotators are equally accurate, so they choose the most frequent annotation as the ground truth label. Because of this, they only return the ground truth for the problem.

The data file in this example follow the format from the `BinaryAnnotation` type:

```
example, annotator, value
      0,         0,     1
      0,         1,     0
```

(continues on next page)

(continued from previous page)

```
    0,          2,      1
    ...
```

In this example, we use a `.parquet` data file, which is usually a good option in terms of efficiency. However, we do not limit the types of files you can use, as long as they can be converted to typed datasets of `BinaryAnnotation`, `MulticlassAnnotation` or `RealAnnotation`. The algorithms do assume that there are no missing examples or annotators.

Specifically, MajorityVoting can make predictions both for discrete classes (`BinaryAnnotation` and `MulticlassAnnotation`) and continuous-valued target variables (`RealAnnotation`). You can find information about these methods in the API Docs.

5.2 DawidSkene

This algorithm is one of the most recommended both for its simplicity and its good results generally.

```
import com.enriquegrodrigo.spark.crowd.methods.DawidSkene
import com.enriquegrodrigo.spark.crowd.types.MulticlassAnnotation

val exampleFile = "examples/data/multi-ann.parquet"

val exampleData = spark.read.parquet(exampleFile).as[MulticlassAnnotation]

val mode = DawidSkene(exampleData, eMIters=10, emThreshold=0.001)

val pred = mode.getMu().as[MulticlassLabel]

val annprec = mode.getAnnotatorPrecision()
```

In the implementation, two parameters are used for controlling the algorithm execution, the maximum number of EM iterations and the threshold for the likelihood change. The execution stops if the number of iterations reaches the established maximum or if the change in likelihood is less than the threshold. You do not need to provide these parameters, as they have default values.

Once executed, the model provide an estimation of the ground truth and an estimation of the quality of each annotator, in the form of a confusion matrix. This information can be obtained as shown on the example.

5.3 GLAD

The GLAD algorithm is interesting as it provides both annotator accuracies and example difficulties obtained solely from the annotations. An example of how to use it can be found below.

```
import com.enriquegrodrigo.spark.crowd.methods.Glad
import com.enriquegrodrigo.spark.crowd.types.BinaryAnnotation

val annFile = "data/binary-ann.parquet"

val annData = spark.read.parquet(annFile).as[BinaryAnnotation]

val mode = Glad(annData,
                  eMIters=5, //Maximum number of iterations of EM algorithm
                  eMThreshold=0.1, //Threshold for likelihood changes
```

(continues on next page)

(continued from previous page)

```
                    gradIters=30, //Gradient descent max number of iterations
                    gradTreshold=0.5, //Gradient descent threshold
                    gradLearningRate=0.01, //Gradient descent learning rate
                    alphaPrior=1, //Alpha first value (GLAD specific)
                    betaPrior=1) //Beta first value (GLAD specific)

val pred = mode.getMu().as[BinarySoftLabel]

val annprec = mode.getAnnotatorPrecision()

val annprec = mode.getInstanceDifficulty()
```

This model as implemented in the library is only compatible with binary class problems. It has a higher number of free parameters in comparison with the previous algorithms, but we provided default values for all of them for convenience. The meaning of each of these parameters is commented in the example above, as it is in the API Docs. The annotator precision is given as a vector, with an entry for each annotator. The difficulty is given in the form of a DataFrame, returning a difficulty value for each example. For more information, you can consult the documentation and/or the paper.

5.4 RaykarBinary, RaykarMulti and RaykarCont

We implement the three variants of this algorithm, two for discrete target variables (`RaykarBinary` and `RaykarMulti`) and one for continuous variables (`RaykarCont`). These algorithms have in common that they are able to use features to estimate the ground truth and even learn a linear model. The model is also able to use prior information about annotators, which can be useful to add more confidence to certain annotators. In the next example we show how to use this model adding a prior which indicates that we trust a lot in the first annotator and that we know that the second annotator is not reliable.

```
import com.enriquegrodrigo.spark.crowd.methods.RaykarBinary
import com.enriquegrodrigo.spark.crowd.types.BinaryAnnotation

val exampleFile = "data/binary-data.parquet"
val annFile = "data/binary-ann.parquet"

val exampleData = spark.read.parquet(exampleFile)
val annData = spark.read.parquet(annFile).as[BinaryAnnotation]

//Preparing priors
val nAnn = annData.map(_.annotator).distinct.count().toInt

val a = Array.fill[Double](nAnn,2)(2.0) //Uniform prior
val b = Array.fill[Double](nAnn,2)(2.0) //Uniform prior

//Give first annotator more confidence
a(0)(0) += 1000
b(0)(0) += 1000

//Give second annotator less confidence
//Annotator 1
a(1)(1) += 1000
b(1)(1) += 1000
```

(continues on next page)

(continued from previous page)

```
//Applying the learning algorithm
val mode = RaykarBinary(exampleData, annData,
                        eMIters=5,
                        eMThreshold=0.001,
                        gradIters=100,
                        gradThreshold=0.1,
                        gradLearning=0.1
                        a_prior=Some(a), b_prior=Some(b))

//Get MulticlassLabel with the class predictions
val pred = mode.getMu().as[BinarySoftLabel]

//Annotator precision matrices
val annprec = mode.getAnnotatorPrecision()
```

Apart from the feature matrix and the priors, the meaning of the parameters is the same as in the previous examples. The priors are matrices of dimension $A \times 2$ where A is the number of annotators. In each row we have the hyperparameters of a Beta distribution for each annotator. The `a_prior` gives prior information about the ability of annotators to correctly classify a positive example. The `b_prior` does the same thing but for the negative examples. More information about this method as well as the methods for discrete and continuous target variables can be found in the API docs.

5.5 CATD

This method allows to estimate continuous-value target variables from annotations.

```
import com.enriquegrodrigo.spark.crowd.methods.CATD
import com.enriquegrodrigo.spark.crowd.types.RealAnnotation

sc.setCheckpointDir("checkpoint")

val annFile = "examples/data/cont-ann.parquet"

val annData = spark.read.parquet(annFile).as[RealAnnotation]

//Applying the learning algorithm
val mode = CATD(annData, iterations=5,
                         threshold=0.1,
                         alpha=0.05)

//Get MulticlassLabel with the class predictions
val pred = mode.mu

//Annotator precision matrices
val annprec = mode.weights
```

It returns a model from which you can get the ground truth estimation and also the annotator weight used (more weight means a better annotator). The algorithm uses parameters such as `iterations` and `threshold` for controlling the execution, and also `alpha`, which is a parameter of the model (check the API docs for more information).

CHAPTER

SIX

COMPARISON WITH OTHER PACKAGES

There exist other packages implementing similar methods in other languages, but with different aims in mind. To our knowledge, there are 2 software packages with the goal of learning from crowdsourced data:

- Ceka: it is a Java software package based on WEKA, with a great number of methods that can be used to learn from crowdsource data.
- Truth inference in Crowdsourcing makes available a collection of methods in Python to learn from crowdsourced data.

Both are useful packages when dealing with crowdsourced data, with a focus mainly on research. Differently, *spark-crowd* is useful not only in research, but also in production. It provides a clear usage interface as well as software tests for all of its methods with a high test coverage. Moreover, methods have been implemented with a focus on scalability, so it is useful in a wide variety of situations. A comparison of the methods over a set of datasets is provided in this section, taking into account both quality of the models and execution time.

6.1 Data

For this performance test we use simulated datasets of increasing size and a real multiclass dataset:

- **binary1-4**: simulated binary class datasets with 10K, 100K, 1M and 10M instances respectively. Each of them has 10 simulated annotations per instance, and the ground truth for each example is known (but not used in the learning process). The accuracy shown in the tables is obtained over this known ground truth.
- **cont1-4**: simulated continuous target variable datasets, with 10k, 100k, 1M and 10M instances respectively. Each of them has 10 simulated annotations per instance, and the ground truth for each example is known (but not used in the learning process). The Mean Absolute Error is obtained over this known ground truth.
- **crowdscale**. A real multiclass dataset from the *Crowdsourcing at Scale* challenge. The data is comprised of 98979 instances, evaluated by, at least, 5 annotators, for a total of 569375 answers. We only have ground truth for the 0.3% of the data, which is used for evaluation.

All the datasets are available through this link

6.2 CEKA

To compare our methods with Ceka, we used two of the main methods implemented in both packages, MajorityVoting and DawidSkene. Ceka and spark-crowd also implement GLAD and Raykar's algorithms. However, in Ceka, these algorithms are implemented using wrappers to other libraries. The library for the GLAD algorithm is not available on our platform, as it is given as an EXE Windows file, and the wrapper for Raykar's algorithms does not admit any configuration parameters.

We provide the results of the execution in terms of accuracy (Acc) and time (in seconds). For our package, we also include the execution time for a cluster (tc) with 3 executor nodes of 10 cores and 30Gb of memory each.

Table 1: Comparison with Ceka

	MajorityVoting					DawidSkene				
	Ceka		spark-crowd			Ceka		spark-crowd		
Method	Acc	t1	Acc	t1	tc	Acc	t1	Acc	t1	tc
binary1	0.931	21	0.931	11	7	0.994	57	0.994	31	32
binary2	0.936	15983	0.936	11	7	0.994	49259	0.994	60	51
binary3	X	X	0.936	21	8	X	X	0.994	111	69
binary4	X	X	0.936	57	37	X	X	0.994		
crowdscale	0.88	10458	0.9	13	7	0.89	30999	0.9033	447	86

Regarding accuracy, both packages achieve comparable results. However, regarding execution time, spark-crowd obtains significantly better results among all datasets, especially on the bigger ones, where it tackle problems that Ceka is not able to solve.

6.3 Truth inference in crowdsourcing

Now we compare `spark-crowd` with the methods implemented by the authors. Although they can certainly be used to compare and test algorithms, the integration of these methods into a large ecosystem might be difficult, as the authors do not provide a software package structure. Nevertheless, as it is an available package with a great number of methods, a comparison with them is advisable.

For the experimentation, the same datasets are used as well as the same environments. In this case, a higher number of models can be compared, as most of the methods are written in python. However, the methods can only be applied to binary or continuous target variables. As far as we know, the use of multiclass target variables is not possible as it is the case of feature information for Raykar's methods.

First, we compare the algorithms capable of learning from binary classes. In this category, MajorityVoting, DawidSkene, GLAD and IBCC are compared. For each dataset, the results in terms of Accuracy (Acc) and time (in seconds) are obtained. The table below shows the results for MajorityVoting and DawidSkene. Both packages obtain the same results in terms of accuracy. For the smallest datasets, the overhead imposed by parallelism makes Truth-Inf a better choice, at least in terms of execution time. However, as the size of the datasets increase, and especially, in the last two cases, the speedup obtained by our algorithm is notable. In the case of DawidSkene, the Truth-inf package is not able to complete the execution because of memory constraints in the largest dataset.

Table 2: MajorityVoting and DawidSkene comparison

	MajorityVoting					DawidSkene				
	Truth-inf		spark-crowd			Truth-inf		spark-crowd		
Method	Acc	t1	Acc	t1	tc	Acc	t1	Acc	t1	tc
binary1	0.931	1	0.931	11	7	0.994	12	0.994	31	32
binary2	0.936	8	0.936	11	7	0.994	161	0.994	60	51
binary3	0.936	112	0.936	21	8	0.994	1705	0.994	111	69
binary4	0.936	2908	0.936	57	37	M	M	0.994	703	426

Next we show the results for GLAD and IBCC. As can be seen, both packages obtain similar results in terms of accuracy. Regarding execution time, they obtain comparable results in the two smaller datasets (with a slight speedup in `binary2`) for the GLAD algorithm. However, for this algorithm, Truth-inf is not able to complete the execution in the two largest datasets. In the case of IBCC, the speedup obtained by our library starts to be noticeable from the second dataset on. It is also noticeable that `Truth-Inf` did not complete the execution for the last dataset.

Table 3: GLAD and IBCC comparison

	GLAD					IBCC				
	Truth-inf		spark-crowd			Truth-inf		spark-crowd		
Method	Acc	t1	Acc	t1	tc	Acc	t1	Acc	t1	tc
binary1	0.994	1185	0.994	1568	1547	0.994	22	0.994	74	67
binary2	0.994	4168	0.994	2959	2051	0.994	372	0.994	97	76
binary3	X	X	0.491	600	226	0.994	25764	0.994	203	129
binary4	X	X	0.974	2407	1158	X	X	X	1529	823

Note that the performance of GLAD algorithm seems to degrade in the bigger datasets. This may be due to the ammount of parameters the algorithm needs to estimate. A way to improve the estimation goes through decreasing the learning rate, which makes the algorithm slower, as it needs many more iterations to obtain a good solution. This makes the algorithm unsuitable for several big data contexts. To tackle this kind of problems, we developed an enhancement, CGLAD, which is included in this package (see the last section of this page for results of other methods in the package, as well as this enhancement). Next, we analize methods that are able to learn from continuous target variables: MajorityVoting (mean), CATD and PM (with mean initialization). We show the results in terms of MAE (mean absolute error) and time (in seconds). The results for MajorityVoting and CATD can be found in the table below.

Table 4: MajorityVoting (mean) and CATD comparison

	MajorityVoting (mean)					CATD				
	Truth-inf		spark-crowd			Truth-inf		spark-crowd		
Method	Acc	t1	Acc	t1	tc	Acc	t1	Acc	t1	tc
cont1	1.234	1	1.234	6	8	0.324	207	0.324	25	28
cont2	1.231	8	1.231	7	9	0.321	10429	0.321	26	24
cont3	1.231	74	1.231	12	13	X	X	0.322	42	38
cont4	1.231	581	1.231	56	23	X	X	0.322	247	176

As can be seen in the table, both packages obtain similar results regarding MAE. Regarding execution time, the implementation of MajorityVoting from the `Truth-inf` package obtains good results, especially in the smaller dataset. It is worth pointing out that, for the smallest datasets, the increase overhead imposed by parallelism makes the execution time of our package a little worse in comparison. However, as datasets increase in size, the speedup obtained by our package is notable, even in MajorityVoting, which is less complex computationally. Regarding CATD, Truth-inf seems not to be able to solve the two bigger problems in a reasonable time. However, they can be solved by our package in a small ammount of time. Even for the smallest sizes, our package obtains a high speedup in comparison to `Truth-inf` for CATD.

In the table below you can find the results for PM and PMTI algorithms.

Table 5: PM and PMTI comparison

	PM					PMTI				
	Truth-inf		spark-crowd			Truth-inf		spark-crowd		
Method	Acc	t1	Acc	t1	tc	Acc	t1	Acc	t1	tc
cont1	0.495	77	0.495	57	51	0.388	139	0.388	68	61
cont2	0.493	8079	0.495	76	57	0.386	14167	0.386	74	58
cont3	X	X	0.494	130	97	X	X	0.387	143	98
cont4	X	X	0.494	769	421	X	X	0.387	996	475

Although similar, the modification implemented in Truth-inf from the original algorithm seems to be more accurate. Even in the smallest sizes, our package obtains a slight speedup. However, as the datasets increase in size, our package is able to obtain a much higher speedup.

6.4 Other methods

To complete our experimentation, next we focus on other methods other methods implemented by our package that are not implemented by `Ceka` or `Truth-Inf`. These methods are the full implementation of the Raykar's algorithms (taking into account the features of the instances) and the enhancement over the GLAD algorithm. As a note, Truth-inf implements a version of Raykar's algorithms that does not use the features of the instances. First, we show the results obtained by the Raykar's methods for discrete target variables.

Table 6: Raykar's methods in spark-crowd for discrete targets .

	RaykarBinary			RaykarMulti		
	spark-crowd			spark-crowd		
Method	Acc	t1	tc	Acc	t1	tc
binary1	0.994	65	63	0.994	167	147
binary2	0.994	92	74	0.994	241	176
binary3	0.994	181	190	0.994	532	339
binary4	0.994	1149	560	0.994	4860	1196

Next we show the Raykar method for tackling continous target variables.

Table 7: Raykar method for continuous target variables.

	RaykarCont		
	spark-crowd		
Method	Acc	t1	tc
cont1	0.994	31	32
cont2	0.994	60	51
cont3	0.994	111	69
cont4	0.994	703	426

Finally, we show the results for the CGLAD algorithm. As you can see, it obtains similar results to the GLAD algorithm in the smallest instances but it performs much better in the larger ones. Regarding execution time, CGLAD obtains a high speedup in the cases where accuracy results for both algorithms are similar.

Table 8: CGLAD, an enhancement over the GLAD algorithm.

	CGLAD		
	spark-crowd		
Method	Acc	t1	tc
binary1	0.994	128	128
binary2	0.995	233	185
binary3	0.995	1429	607
binary4	0.995	17337	6190

CHAPTER

SEVEN

CONTRIBUTORS

We are open to contributions in the form of bugs reports, enhancements or even new algorithms.

7.1 Bug reports

Bugs are tracked using Github issues. When creating a bug report, please provide as much information as possible to help maintainers reproduce the problem

- Use a clear and descriptive title for the issue to identify the problem.
- Describe the exact steps which reproduce the problem in as many details as possible. For example, start by explaining how you prepared the data as well as how the package was installed and what version of the package are you using. When listing steps, do not just say what you did, but also explain how you did it.
- Provide specific examples to illustrate the steps. Include links to files or GitHub projects. If you are providing snippets in the issue, use Markdown code blocks.
- Describe the behavior you observed after following the steps and point out what exactly is the problem with that behavior. Explain which behavior you expected to see instead and why.
- If the problem is related to performance or memory, include a CPU profile capture with your report.

Provide more context by answering these questions:

- Did the problem start happening recently (e.g. after updating the version dependencies) or was this always a problem?
- If the problem started happening recently, can you reproduce the problem in an older version? What is the most recent version in which the problem does not happen?
- Can you reliably reproduce the issue? If not, provide details about how often the problem happens and under which conditions it normally happens.

Include details about your configuration and environment:

- Which version of spark-crowd are you using?
- What is the name and version of the OS you are using?
- Are you running the package in a virtual machine? If so, which VM software are you using and which operating systems and versions are used for the host and the guest?

7.2 Suggesting enhancements

We are open to suggestions of new features and improvements to existing functionalities. Please, follow the guidelines to help maintainers and the community understand your suggestions. When requesting an enhancement, please include as many details as possible.

Enhancement suggestions are tracked using Github Issues. To request an enhancement, create an issue and provide the following information:

- Use a clear and descriptive title for the issue to identify the suggestion.
- Provide a step-by-step description of the suggested enhancement.
- Provide specific examples to illustrate the steps. Include copy/pasteable snippets which you use in those examples, as Markdown code blocks.
- Describe the current behavior and explain which behavior you expected to see instead and why.
- Explain why this enhancement would be useful to the users.
- Specify which version of the package you are using. Specify the name and version of the OS you are using.

7.3 New algorithms

We are also grateful for contributions of new algorithms, as long as they improve the results or add new functionalities to the ones existing in the package. New algorithms must be published in peer-review publications to be considered. New algorithms must adhere to the architecture of this package and take into account the scalability of the learning process.

To contribute with an algorithm, first create a request using Github Issues, for the maintainers to review the suggestion. This request should provide the following information:

- Publication where the algorithm details can be reviewed.
- Explain why this algorithm would be useful to the users.

If the request is accepted, create a Github pull request with the new algorithm, as well as all the necessary types to use it, so that the maintainers can review the code and add it to the package.