Stuffed Version 2.0

R. J. Lyon [*]
School of Computer Science &
Jodrell Bank Centre for Astrophysics
University of Manchester
Kilburn Building
Oxford Road
Manchester M13 9PL

Wednesday 25th March, 2015

Abstract

The STUFFED framework is a collection of Java files, which wrap around the WEKA and MOA frameworks. The main goal of STUFFED is to make it easier to perform experiments with unlabelled and partially-labelled *imbalanced* data streams. Testing classifiers on such data streams is currently difficult to do within MOA and WEKA.

This API supplements MOA and WEKA functionality in a number of ways. It uses a custom sampling procedure to create data streams with varying proportions of labelled data, and varying levels of class imbalance. The API also enables the evaluation of classifiers on unlabelled data, thorough the use of custom meta data files, that allow true class labels to be retrieved during prediction evaluation. The API was developed specifically to enable the experiments described in [1, 2].

[*] Email: robert.lyon@postgrad.manchester.ac.uk , Web: `www.scienceguyrob.com` .

Contents

1 Acknowledgements

Thank you to the team responsible for developing MOA and WEKA.

Development work was supported by grant EP/I028099/1 for the University of Manchester Centre for Doctoral Training in Computer Science, from the UK Engineering and Physical Sciences Research Council (EPSRC).

2 System Requirements

The API has the following system requirements:

- Java 1.6 or later.
- MOA (`http://moa.cms.waikato.ac.nz/`)
- WEKA (`http://www.cs.waikato.ac.nz/ml/weka/`)

3 Key features

In brief the STUFFED API can be used to:

- Test MOA and WEKA classifiers on unlabelled data.
- Test custom classifiers on unlabelled data.
- Test in the standard 'static' supervised learning setting, or the non-stationary stream setting.
- Sample large data sets (many GB in size) in a reproducible way, in order to produce testing and training sets.
- Specify the precise configuration of training and test data sets (i.e. size, class distribution etc).
- Vary the proportion of labelled items in a test set.
- Vary the level of class imbalance in both testing and training sets.
- Evaluate classifier performance using many standard and imbalanced metrics.
- Test static and stream classifiers within external tools such as MatLab, since each classifier test outputs a confusion matrix describing classification results.

This functionality was designed in response to the requirements of my research. It is focused on building accurate binary classifiers for the unlabelled and heavily imbalanced data streams produced by modern radio telescopes. This work is described in more detail in [1, 2]. As such The API is unsuitable for multi-class problems, though with minor modifications multi-class problems could be supported.

4 Usage

STUFFED is an API as opposed to an executable application with a GUI. The onus is on the user to correctly use the API, and understand the general principles underlying both MOA [3] and WEKA [4]. Those who encounter problems with MOA, should look for solutions/advice in the MOA users group [1] or in the MOA manual [3, 5], likewise for WEKA[2].

That said, the following example illustrates how the API can be used. This assumes the Stuffed.jar file in the build path of an open development project.

Listing 1: Code which uses our sampling procedure to create a test stream for classification. A Hoeffding tree classifier is then used to classify it.

```
public static void main(String[] args)
{
  // Input variables.
  String log  = "Stuffed/Test.txt";
  String data = "Stuffed/data/MiniBoone.arff";

  // Destination training and test files.
  String testSet  = "Stuffed/data/test.arff";
  String trainSet = "Stuffed/data/train.arff";

  boolean v = true; // Logging flag.

  // Clean up earlier files.
  Common.fileDelete(log);
  Common.fileDelete(testSet);
  Common.fileDelete(trainSet);

  // Sampling variables:
  int pos = 36499; // Positives in the data file.
  int neg = 93565; // Negatives in the data file.

  // The desired training set distribution,
  // minimum of one example per class.
  int nTrnSamps = 1000; // - training samples
  int pTrnSamps = 1000; // + training samples

  // Desired training set balance.
  double trnSetBal = 1.0;// Equal +/- balance.

  // The proportion of the test set to label.
  double l = 0.1; // 10% of test data labelled.
  double tstSetBal = ((double)pos - (double)pTrnSamps)
                  / ((double)neg - (double)nTrnSamps);

  // Build sampler.
  Sampler s = new Sampler(log,false);
  s.load(data, -1); // Read file to be sampled.

  // Perform sampling, collect result information.
  Object[] out = s.sampleToARFF(trainSet,testSet,
                  nTrnSamps,pTrnSamps,trnSetBal,
                  tstSetBal,l);

  // Process outcome of sampling operation.
  boolean result = (Boolean) out[0];
  String outcome = (String)  out[1];
```

[1] `https://groups.google.com/forum/#!forum/moa-users`
[2] `http://wiki.pentaho.com/display/DATAMINING/Pentaho+Data+Mining+Community+Documentation`

```
  // If sampling fails...
  if(!result)
  {
        System.out.println("Error in sampling:");
        System.out.println(outcome);
        System.exit(0);
  }

  // Else sampling succeeded, continue to run test.
  StreamAlgorithmTester sat = new StreamAlgorithmTester(log,
                              "HTree",v,new HoeffdingTree());

  sat.train(trainSet); // Train the classifier.

  // Perform classification.
  int[][] confusionMatrix = sat.testStream(testSet,log);

  // Print confusion matrix for viewing.
  FormatCM.printConfusionMatrix(confusionMatrix);

  // Evaluate performance using confusion matrix.
  ClassifierStatistics stats = new ClassifierStatistics();
  stats.setConfusionMatrix(confusionMatrix);
  stats.calculate(); // Actually computes the stats.

  // Print out formatted results.
  System.out.println(stats.toString());
}
```

Note that if loading extremely large files for sampling, STUFFED will require more memory. In such cases use the following to execute your code, assuming it has been compiled into a jar file called 'yourCode.jar':

Listing 2: Executing a JAR file with more 256 MB of heap memory.

```
java -Xmx256m -jar yourCode.jar
```

A script has also been provided in the distribution showing how to evaluate classifier predictions within MatLab. Please look for the 'Example.m' script in the 'matlab' folder accompanying this report.

4.1 Testing & Evaluation

STUFFED is capable of performing three distinct types of algorithm test. These allow classifier performance to be determined in both static and streamed data scenarios. Each of these test modes is now described in detail.

4.1.1 Static Tests

In a traditional machine learning scenario, algorithms have access to all available data upfront. Typically this data is partitioned into two unique datasets. The first is a training set defined as,

$$T = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_m)\}.$$

Here each instance $x_i \in X$ where $i = 1, ...n$, is mapped to its correct label $y_j \in Y$ where $j = 1, ...m$. A traditional algorithm attempts to learn a function mapping instances to correct labels, with the goal of minimising error. The second data set is the test set. Classifiers are evaluated on the test set in order to determine how well they perform in practice.

STUFFED provides wrappers that allow static classifiers to be tested in the framework (on unlabelled data), whilst also providing the means to test stream classifiers as though they were static. A stream classifier can be treated as a static classifier if it is i) given the same training set to learn from as the static learner, and ii) if it is prevented from learning post classifying each x_i in the training set. Generally the main reason for allowing testing in a static scenario, is to establish a baseline level of performance using static classifiers applied to some dataset.

4.1.2 Stream Tests

The stream learning scenario is somewhat different from the standard supervised learning model. In a streamed scenario, classifiers do not have access to all training data upfront. Rather they are trained incrementally on each labelled example or labelled batch of examples they see.

Data stream tests in STUFFED can be performed in two slightly different ways. The first assumes an incremental model of data stream classification. Here a classifier is initialised without any prior training, and made to classify each instance from the stream x_i, one after the other. After a prediction has been made for x_i, the learner is then trained on x_i provided the instance label is available. This test strategy follows a test-then-train approach to data stream classification.

The second approach is similar, expect that we are able to 'pre-train' the classifier off-line on a small sample of data, before taking it on-line to classify the stream. The idea behind this approach is that if training data is readily available, then it may be advantageous to use it, if classifier predictions must be accurate upon the very first example from the stream. In essence this approach can simply be thought of as classifying a stream prefixed with a number of labelled examples. Only stream classifiers can be tested under both of these scenarios, neither are applicable for static classifiers.

5 Compatible Data

5.1 ARFF Format

ARFF is the standard file format used by the WEKA data mining tool. The STUFFED API can read ARFF files, and writes classifier predictions and miss-classifications to ARFF files. The ARFF format itself is simple. It includes a header that describes the data it contains, and requires that each row of features be associated with a class label. For example, imagine you wish to classify the following data set:

Listing 3: Simple data set.

```
8,5,2,1,1,2,3,1
6,5,3,1,1,1,4,2
6,2,3,1,1,2,3,1
8,5,2,1,1,2,3,1
5,3,3,1,1,1,5,3
```

This data describes 5 examples, by 8 individual features. To represent this data in ARFF format we must describe:

1. the file itself with the @relation tag.
2. each feature used with the @attribute tag.
3. the possible class labels, also via the @attribute tag.
4. where the data starts using the @data tag.

Thus the data would be represented in ARFF format as:

Listing 4: Feature data output in ARFF format.

```
@relation 5_candidates_described_by_8_features

@attribute Feature_1 numeric
@attribute Feature_2 numeric
@attribute Feature_3 numeric
@attribute Feature_4 numeric
@attribute Feature_5 numeric
@attribute Feature_6 numeric
@attribute Feature_7 numeric
@attribute Feature_8 numeric
@attribute class {0,1}

@data
8,5,2,1,1,2,3,1,1 % This is a comment.
6,5,3,1,1,1,4,2,?
6,2,3,1,1,2,3,1,0
8,5,2,1,1,2,3,1,?
5,3,3,1,1,1,5,3,1
```

Note that an extra class label has been added to each row of data. Here '?' indicates that the true class of a row is unknown. Though if an example was known to be positive then '?' would be replaced with 1. Likewise if an example was known to be negative then '?' would be replaced by 0.

5.2 CSV Format

The STUFFED API can also read comma separated value (CSV) files. The sampling procedures in the API are CSV compatible, however it is advisable to use the ARFF format when classifying data. The API contains methods for converting ARFF to CSV format, and vice versa.

5.3 Supplied Data

Three data sets are included with this API, which were obtained from the UCI data set repository [6]. These are described in Table 1.

Dataset	Instances	Attributes / Type	~Balance	Distribution	
				+	-
Skin [7]	245,057	3 / Discrete	+1 : -4	50,859	194,198
MiniBoone [8]	130,065	50 / Continuous	+1 : -3	36,499	93,565
Magic [9]	19,020	10 / Continuous	+1 : -2	6,688	12,332

Table 1: Characteristics of the data sets provided.

6 Design

6.1 Overview

STUFFED has been implemented as a simple wrapper around the MOA and WEKA APIs. STUFFED does not come with a graphical user interface, rather it is executed via user built applications. The API is split up in to nine principal component Java packages, each of which is responsible for a key piece of functionality. These are now briefly described in order to make the code more understandable.

- Package - *default*: Contains a simple example class illustrating how the code is to be used.
- Package - **cs.man.ac.uk.classifiers**: Contains classes which provide the basic functionality expected of a classifier. This package contains two important implementations which wrap the WEKA and MOA classifiers. The class *'StandardAlgorithmTester.java'* wraps WEKA classifiers, and is for use on standard data sets only. The class *'StreamAlgorithmTester.java'* wraps MOA classifiers, and can be used on both standard and streaming data. Note that in the static scenario,stream classifiers will attempt to learn from each example seen during classification - providing a label is available. In the streaming scenario, stream classifiers can only learn if a label is available.
- Package - **cs.man.ac.uk.common**: A package containing utility classes used throughout the application.
- Package - **cs.man.ac.uk.format**: A package containing utility classes for formatting classifier output.
- Package - **cs.man.ac.uk.io**: A package containing classes for dealing with file input and output.
- Package - **cs.man.ac.uk.log**: A package containing classes used for debug logging.

- Package - **cs.man.ac.uk.moawrappers**: Contains wrapper classes for MOA and WEKA classifiers.
- Package - **mvc**: This package contains controllers through which the API is most easily accessed. Thus this package provides the interface through which users should normally interact with the API.
- Package - **cs.man.ac.uk.sample**: Contains classes used for sampling large datasets. Can be used to vary the proportion of labelled data, or the class distribution.
- Package - **cs.man.ac.uk.stats**: Contains classes used for computing classification performance metrics, helps makes evaluation easier.

6.2 Reason for Developing the API

The MOA framework provides an excellent tool set which allows benchmark experiments to be undertaken in a streamed data environment. The framework includes multiple evaluation techniques, a collection of pre-implemented machine learning algorithms, and a number of configurable data stream generators. There a a number of advantages to using MOA:

1. MOA makes i simple to develop, execute and evaluate new algorithms within the framework.
2. MOA allows experimental results can be easily reproduced.
3. The MOA code base is open source, extensible and stable (built upon the success of the existing WEKA [4] data mining tool).
4. MOA is becoming recognised as the standard tool-kit for the stream mining community.

Despite these advantages, MOA is lacking two pieces of functionality required if working with imbalanced and mostly or completely unlabelled data streams[3]. These problems are summarised below.

- **Unlabelled data:** MOA cannot evaluate classifiers operating on unlabelled data. MOA requires that class labels be known in advance, in order to evaluate a class prediction. If the true class labels are unavailable, the best MOA can do is output a file containing classifier predictions. To determine how well a classifier has performed, the predictions in this file must be compared to true class labels, outside of the evaluation framework. This gives rise to the following difficulties:
    1. For data stream experiments which may require the classification of millions of instances, this output file can become extremely large. Thus the predictions file may be too large to process in a reasonable amount of time.
    2. Without a standard tool to process these predictions, researchers will have to take time to write their own evaluation tools. This could lead to different evaluation strategies being implemented, making a true comparison between results obtained on unlabelled data difficult.

[3] True at the time of writing.

- **Varied class balance & labelling:** MOA cannot vary i) the class balance and ii) the proportion of labelled instances in very large datasets. However when experimenting with unlabelled and imbalanced datasets, there are principally three modifications that one may wish/need to make. Either,

    1. the labelling must be modified leaving the class imbalance intact.
    2. the imbalance must be modified leaving the labelling intact.
    3. both must be varied at the same time.

  If using large real world datasets as streams for experimentation (rather than a data generator), it is often desirable to modify datasets in this way. Although WEKA (upon which MOA is based) contains tools that can be used to re-sample data, these are unsuitable for use on very large data sets. Primarily because these datasets do not fit into the memory of modern machines.

6.3 API Requirements

Given these minor shortcomings of MOA, it became necessary to consider ways in which the framework could be augmented to achieve the goals of testing upon imbalanced and unlabelled data streams. The first step in this process involved the development of 'modification' requirements, given the needs of our specific problem domain [1, 2]. The requirements which drove the design of the wrapper are presented below.

1. **Data requirements:** Real-world data sets are often very large, comprising millions of individual instances. When stored in a single file these often have a collective disk footprint of many gigabytes. Dealing with each of these instances individually (i.e. smaller separate files) is difficult and time consuming. Therefore to save time and effort, it is desirable to have a single 'master' file containing all instances, that can be easily use during classifier tests. Thus we have the following data requirements:

    (a) To load data from a single attribute relation file format file (ARFF) in to the MOA framework.

    (b) To treat the data as either a static data set, or a data stream as necessary.

    (c) To be able to randomly shuffle data, producing a new shuffled ARFF dataset file.

    (d) To be able to extract meta information from the labelled ARFF data file. Considering only the two class case, it should be possible to:

        i. Extract the positive instances in the dataset to a separate ARFF file. Instance labels should remain intact, and an index indicating their position in the original file should also be retained.

        ii. Extract the negative instances in the dataset to a separate ARFF file. Instance labels should remain intact, and an index indicating their position in the original file should also be retained.

Meeting this particular requirement allows classifier evaluation on unlabelled data. By keeping a count of the classifications made at a given time step, it is possible to cross reference an instance with the indexes in the meta data file, in order to obtain its correct label. This will be explained in more detail later.

(e) To determine the class distribution automatically from a ARFF dataset file.

2. **Sampling requirements:** To test algorithms under a specific level of class imbalance b, and a particular ratio of labelled data l, a dataset must be re-sampled to produce a new file describable by b and l. As many real-world datasets are extremely large, sampling in this way is time consuming and requires a great deal of disk space. These considerations give rise to the following requirements:

   (a) To provide the means to easily repeat a sampling procedure, so that data sets do not have to be retained in order to repeat a test.

   (b) To be able to randomly sample a two class static dataset D. This should produce a test, and an optional training set in ARFF format, given the following parameters:

      i. The desired balance between the positive and negative class in the test set $b_{test} \in \mathbb{R}$ where $0 \leq b_{test} \leq 1$.

      ii. The desired balance between the positive and negative class in the training set $b_{train} \in \mathbb{R}$ where $0 \leq b_{train} \leq 1$.

      iii. The proportion of data that should be labelled in the test set $l \in \mathbb{R}$ where $0 \leq l \leq 1$.

      iv. The number of positive samples to include in the training set $p_{train} \in \mathbb{Z}^{+}$.

3. **Test requirements:** The MOA API allows classifiers to be trained and tested on both static datasets and data streams. To use the API in conjunction with unlabelled data, the following must be adhered to:

   (a) All test/train dataset files must be in the ARFF format to be used by MOA API.

   (b) It should be possible to train a classifier on a specific ARFF dataset.

   (c) It should be possible to test a classifier on a specific ARFF dataset.

   (d) It should be possible to set the options for the classifier using the MOA API.

   (e) It should be possible to record test statistics after a predefined number of instances have been classified. Using the parameter $I_{samp} \in \mathbb{Z}$ where $I_{samp} \geq 0$, it should be possible to gather test statistics for every I_{samp} instances classified.

   (f) It should be possible to record each misclassification made during a test.

(g) At the end of a test, all assessment metrics should be automatically calculated and written to a user specified result file.

(h) It must be possible to define the execution of multiple tests, for a range of imbalances and labellings.

(i) The execution of a single test or multiple tests must be automatable. All test results should be automatically collected and written to an output file.

4. **Evaluation requirements:**

   (a) The class prediction for each instance must be evaluated after the prediction is made.

   (b) Summary statistics should be updated, after the prediction for each instance is evaluated.

   (c) Summary statistics should be written to a CSV file following the completion of a test, allowing for analysis in external tools.

   (d) A number of statistics should be recorded and updated after each instance has been classified:

      i. Multiple classifier assessment metrics must be available.

      ii. The following information should be accessible: total true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

      iii. Runtime in nanoseconds and seconds must be recorded.

      iv. Memory used in bytes and megabytes but be obtainable.

6.4 Data Pre-processing & Meta data Creation

The majority of data manipulation required by STUFFED is done by the MOA API. However in order to evaluate classifier performance on unlabelled data, true class labels must be accessible in some way. STUFFED uses meta data files to achieve this, and to speed up the sampling of large data set.

6.4.1 Meta data for Evaluation

When an ARFF file containing a data set D is loaded into the STUFFED sampling classes, it is thoroughly analysed. A first pass over the data set determines i) the total number of instances in the file, ii) the number of attributes describing the data, and iii) the class distribution.

Each instance in D, can by uniquely identified by the line number i on which it occurs. Thus each instance in D can be described as x_i for $i = 1, ..., n$ where n is the total number of instances in D, and i the line number. Each instance x_i is also implicitly associated with a class label. Thus each line of the file D is also linked to a true class label. Using these observations, we generate a meta data file per class, that contains details of the mapping from line numbers to correct class labels. In particular for the binary case, two meta data files are created, one for the positive class, and one for the negative class. The positive meta data file contains only positive examples, likewise the negative file contains only negative examples. The new files have almost the same structure as the original file D, except that they posses an additional feature. This feature describes the line number where each instance originally occurs in D. Consider the following ARFF file representing D.

Listing 5: The original ARFF data set file D.

```
@relation D
@attribute a    numeric
@attribute b    numeric
@attribute c    numeric
@attribute class        {0,1}
@data
13  ,  10  ,  2  ,  1
24  ,  12  ,  2  ,  0
61  ,  21  ,  3  ,  1
18  ,  12  ,  2  ,  0
26  ,  11  ,  2  ,  1
```

Each line following the '@data' declaration defines an individual instance. Each instance has three numeric features a, b and c, followed by a class label. The positive ARFF meta data file created from D would be as follows:

Listing 6: The positive ARFF meta data file.

```
@relation PositiveMetaData
@attribute lineIndex    numeric
@attribute a                    numeric
@attribute b                    numeric
@attribute c                    numeric
@attribute class        {0,1}
@data
1  ,  13  ,  10  ,  2  ,  1
3  ,  61  ,  21  ,  3  ,  1
5  ,  26  ,  11  ,  2  ,  1
```

The corresponding negative meta data file is as follows:

Listing 7: The negative ARFF meta data file.

```
@relation NegativeMetaData
@attribute lineIndex    numeric
@attribute a                    numeric
@attribute b                    numeric
@attribute c                    numeric
@attribute class        {0,1}
@data
2  ,  24  ,  12  ,  2  ,  0
4  ,  18  ,  12  ,  2  ,  0
```

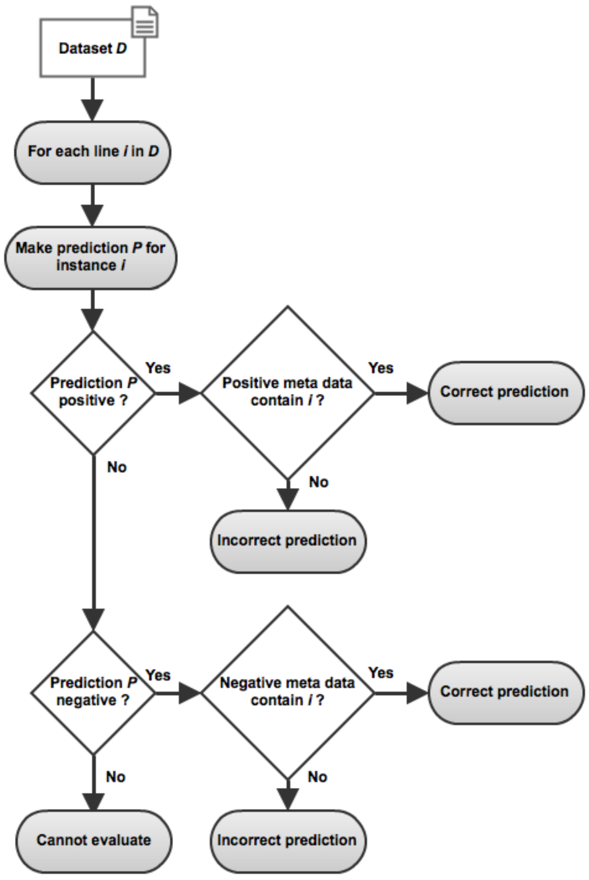
A simple hash based data structure (key-value pairs) can be used to make use of the information stored in these meta data files. Consider the positive meta data file presented in Listing 6. The information it contains can be easily stored in a hash map as shown in Table 2. Here the key is simply a count of the instances in the data structure, and the value is the line number of the instance in the data set D. STUFFED creates both a positive and negative hash map to hold this information. Using the line index as the key and the instance/class as the value, a classification prediction can then be easily evaluated by STUFFED, via checking if the current line number i, occurs in either the positive or negative hash maps.

As most hash based data structures can perform lookup operations in constant time, such cross checking can be performed very quickly. In addition these meta data files need only be created once, meaning that there is reduced computational overhead when re-running a test.

Key	Value
1	1
2	3
3	5

Table 2: A example of the hash data structure for the positive instances in Listing 5.2.

Figure 1 illustrates the role that meta data plays in the classifier evaluation process. It is key for enabling the evaluation of unlabelled data sets.

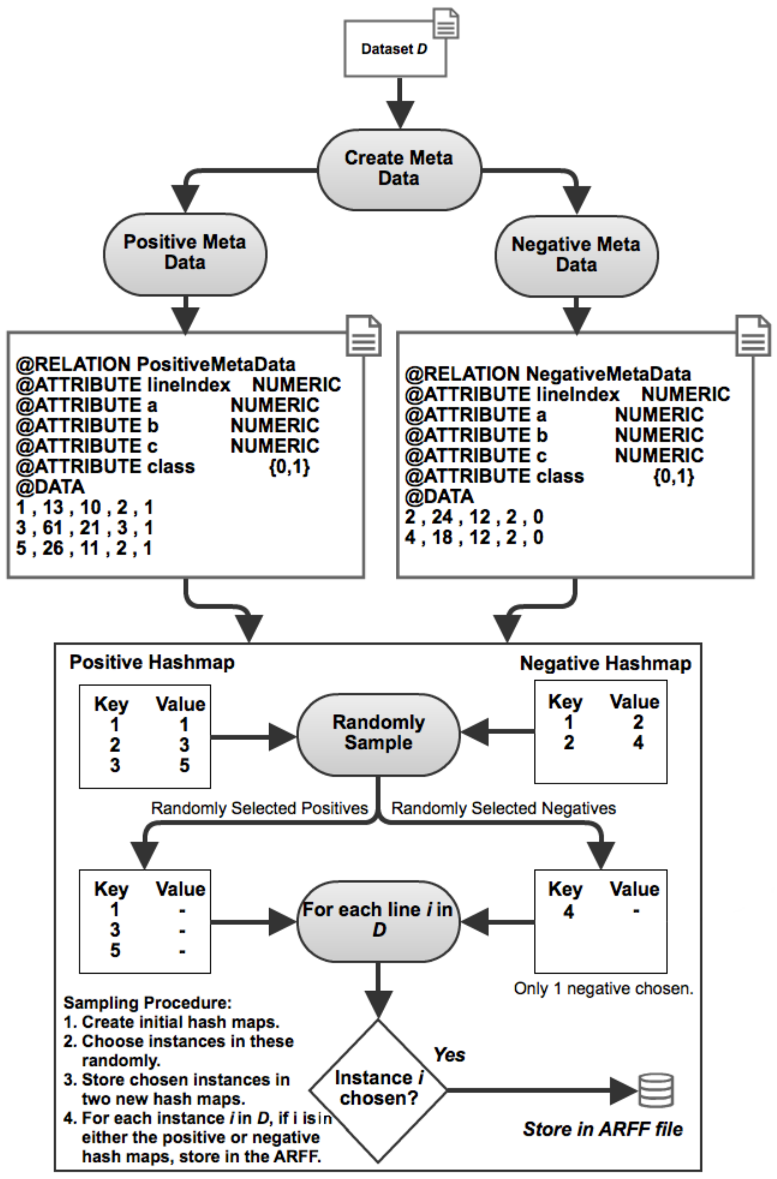


Dataset *D*
For each line *i* in *D*
Make prediction *P* for instance *i*
Prediction *P* positive ?
Yes
Positive meta data contain *i* ?
Yes
Correct prediction
No
No
Incorrect prediction
Prediction *P* negative ?
Yes
Negative meta data contain *i* ?
Yes
Correct prediction
No
No
Cannot evaluate
Incorrect prediction

Figure 1: An overview of how meta data is used during evaluation.

6.4.2 Meta data for Sampling

Meta data plays an important role in the sampling of data sets. During the sampling of two class binary data set D, the positive and negative ARFF meta data files (described above), are used to create a hash map based data structure similar to that used during evaluation. The hash maps for the positive and negative classes respectively, can be used to randomly select instances in D. This actually corresponds to selecting line numbers in D. Therefore to randomly select instances to include in a sampling, all we need to do is randomly select line numbers from these hash maps. This allows a sampling to be created even when the data set has only been read once. Lets extending the example used in the previous section. To sample positive examples, all we need to do is simply choose a random number between 1-3 (the key value).



```
@RELATION PositiveMetaData
@ATTRIBUTE lineIndex   NUMERIC
@ATTRIBUTE a           NUMERIC
@ATTRIBUTE b           NUMERIC
@ATTRIBUTE c           NUMERIC
@ATTRIBUTE class         {0,1}
@DATA
1 , 13 , 10 , 2 , 1
3 , 61 , 21 , 3 , 1
5 , 26 , 11 , 2 , 1
```

```
@RELATION NegativeMetaData
@ATTRIBUTE lineIndex   NUMERIC
@ATTRIBUTE a           NUMERIC
@ATTRIBUTE b           NUMERIC
@ATTRIBUTE c           NUMERIC
@ATTRIBUTE class         {0,1}
@DATA
2 , 24 , 12 , 2 , 0
4 , 18 , 12 , 2 , 0
```

Sampling Procedure:
1. Create initial hash maps.
2. Choose instances in these randomly.
3. Store chosen instances in two new hash maps.
4. For each instance *i* in *D*, if i is in either the positive or negative hash maps, store in the ARFF.

Figure 2: An overview of the meta data creation procedure, and the role of meta data in sampling data sets.

Training and test sets are built in this manner, by randomly sampling from these hash maps. In other words, given a specified training set size and class distribution, the required number of random samplings can be made from these hash maps. To then create a training set, the original file D is read for a second time. On this second pass, when a line number corresponding to a randomly chosen positive or negative is encountered, it is written to a separate training set file. Test sets are constructed similarly, except that examples chosen for the training set, cannot occur in the test set. As the sampling procedure used is random, training and test sets vary with each sampling run.

This approach is by no means optimal, although it appears to work well in practice. However there are some clear drawbacks. Firstly, the larger the dataset, the longer it will take to produce a larger test/train sample. Whilst this may seem obvious, it is certainly worth emphasising before embarking upon sampling a file with millions of instances. In practice, the amount of time taken to sample a dataset containing 10 million instances, will take anywhere between 10-20 minutes - assuming a standard disk. The precise time depends entirely on the I/O speed of the hard disk used, and the operating system load at sample time[4]. In general the amount of time it takes to sample a dataset D, scales with the number of instances to be included in the sample. Secondly, the completion time of this approach is effected by how many times a random number generator chooses a line index, previously selected. As a uniform random number generator is used, the same index is likely to be chosen multiple times, thus increasing the runtime, as a new random index must then be generated.

6.5 Sampling

STUFFED may be required to create new training/test data streams from very large binary datasets. Here we consider a very large dataset to be one infeasible to load into memory. In practice any data set which is more than a few gigabytes in size, is difficult to load into memory, especially given that an internal representation of the file usually requires more memory than the file footprint. Thus, any sampling procedure must be able to sample these files without loading them into memory, and preferably in a single pass. This is not a hugely complex task, but it is comprised of many steps. We present the sampling algorithm here in full to facilitate an understanding of the code, and explain precisely how this particular procedure works. We begin by explaining the mathematical foundation of the sampling procedure, before presenting the sampling algorithm.

6.5.1 Sampling Foundation

To sample a data set D, we need to know what the desired test/training sets should look like. There are many possible configurations for these datasets. Let us assume that we firstly wish to alter the class balance, in both the training and test sets. Each dataset D has a natural balance b, which we may wish to alter. The class balance for the training set D_{train} is given by,

$$D^b_{train} = \frac{\text{positives in training set}}{\text{negatives in training set}},$$

[4] On a solid state drive, this takes only 1 or 2 minutes.

and is defined as the ratio between the positive and negative classes. Likewise for the test set D_{test}, the balance D^b_{test} is defined similarly. To change these balances we must specify a new class balance ratio. We must then populate D_{train} and D_{test} with instances from D, ensuring that the ratios D^b_{train} and D^b_{test} are adhered to.

However it may not be possible to satisfy a desired training/test set balance, for a given data set D. For instance if D contains 1000 instances, of which there are +500 : -500, then a test data set with a +1 : -1000 balance simply cannot be created. This raises the question of how many examples would be required to achieve that level of imbalance? We can answer this question with some basic mathematics. To make such a test set, we require a number of negative test set instances given by,

$$D^-_{test} = \frac{D^+_{test}}{D^b_{test}},$$

where D^-_{test} is the number of negatives in the test set, D^+_{test} the number of positives in the test set, and D^b_{test} the desired balance. Thus we require in this example,

$$\frac{1}{0.001} = 1000,$$

negative instances. Clearly there are not enough negatives to satisfy the desired balance, so this sampling cannot be achieved.

Variable	Description
D	The input data set.
D^+	The number of positives in the data set D.
D^-	The number of negatives in the data set D.
D_{tot}	The number of positives and negatives in the data set D.
D^+_{train}	Positives in the training set.
D^-_{train}	Negatives in the training set.
D^{max}_{train}	Maximum number of instances allowed in the training set.
D^b_{train}	Training set class balance.
D^+_{test}	Positives in the test set.
D^-_{test}	Negatives in the test set.
D^{max}_{test}	Maximum number of instances allowed in the test set.
D^b_{test}	Test set class balance.
$skipablePositives$	The number of positives that can be left out.
$skipableNegatives$	The number of negatives that can be left out.
D^l_{test}	The ratio of labelled instances in the test set.
F	The features to use.

Table 3: The variables used for data set sampling.

The previous example does not consider what happens when a training set must also be created. Creating a training set only slightly complicates matters.

Let us imagine that we have a data set D with 1500 instances, of which there are $D^- = 1000$ negatives and $D^+ = 500$ positives. The goal is to sample this data set to produce a training set with 50 positives, $D^+_{train} = 50$, with a balance $D^b_{train} = \frac{1}{2} = 0.5$, and a test set with a balance of $D^b_{test} = \frac{2}{5} = 0.4$.

Firstly we must find the parameters of the training set. The number of negatives in the training set is given by,

$$D^-_{train} = D^+_{train} \div D^b_{train}.$$

Upon substituting in our values gives,

$$D^-_{train} = 50 \div 0.5 = 100.$$

Now we try to find the parameters for the test set. The number of positives in the test set is given by,

$$D_{test}+ = D^+ - D^+_{train}.$$

This gives us $D_{test}+ = 450$. The number of negatives required in the test set then is given by,

$$D^-_{test} = D^+_{test} \div D^b_{test},$$

which is $D^-_{test} = 450 \div 0.4 = 1125$. As $D^-_{test} < (D^- - D^-_{train})$ (enough negatives left over after creating the training set) then clearly this is an ideal situation in which everything works well. In situations where,

$$D^-_{test} > (D^- - D^-_{train}),$$

then we must modify the parameters. In particular we reset,

$$D^-_{test} = D^- - D^-_{train},$$

and then obtain,

$$D^+_{test} = D^-_{test} \times D^b_{test}.$$

The variables introduced are listed in Table 3. Those we have not described so far are $skipablePositives$ and $skipableNegatives$, which are simply counts of the number of positive and negative instances respectively that will be not used in the sampling (i.e. left over instances). Whilst $F = \{(f_i), ..., (f_n)\}$, $i = 1, ..., n$ where $f_i \in \mathbb{Z}$ is the array of features to use.

Aside from the ability to rebalance a dataset, the requirements for the sampling procedure set out in section 6.3, also describe the need to vary the labelling of the test data D_{test}. The variable D^l_{test} introduced in Table 3 helps to achieve this, irrespective of the balance of D_{test}. The variable D^l_{test} is defined as,

$$D^l_{test} = \frac{\text{Labelled instances in } D_{test}}{\text{total instances in } D_{test}}.$$

Thus by determining a value for D^l_{test}, we can ensure a test data set contains the desired ratio of labelled to unlabelled instances, by simply keeping track of those instances labelled.

6.5.2 Sampling Algorithm

With the principal variables defined we now begin to present the psuedocode for the algorithmic sampling procedure.

Algorithm 1 Part 1: Variable Initialisation

```
Require: An input dataset $D$, a destination training set file path $trainPath$,
    a destination test set file path $testPath$, the number of positive training
    samples $D^+_{train}$, the desired training set balance $D^b_{train}$, the desired test set
    balance $D^b_{test}$, the labelling ratio $D^l_{test}$, and an array of features to use $F$.
 1: procedure SAMPLE($D$, $out$, $D^+_{train}$, $D^b_{train}$,$D^b_{test}$,$l$,$F$)
 2:     $classDist[] \leftarrow$ getClassDistribution()
 3:     $D^- \leftarrow classDist[0]$
 4:     $D^+ \leftarrow classDist[1]$
 5:     $D_{tot} \leftarrow D^+ + D^-$
 6:     $D^-_{train} \leftarrow D^+_{train} \div D^b_{train}$
 7:     $D^{max}_{train} \leftarrow D^-_{train} + D^+_{train}$
 8:     $D^+_{test} \leftarrow D^+ - D^+_{train}$
 9:     $D^-_{test} \leftarrow D^+_{test} \div D^b_{test}$
10:     if $D^-_{test} > (D^- - D^-_{train})$ then
11:         $D^-_{test} \leftarrow D^- - D^-_{train}$
12:         $D^+_{test} \leftarrow D^-_{test} \times D^b_{test}$
13:     $D^{max}_{test} \leftarrow D^+_{test} + D^-_{test}$
14:     $skipablePositives \leftarrow (D^+ - D^+_{train}) - D^+_{test}$
15:     $skipableNegatives \leftarrow (D^- - D^-_{train}) - D^-_{test}$
16:                                                 ▷ continued in Part 2...
```

The code in part 1 sets up the key parameter variables necessary to perform the sampling. The call to $getClassDistribution()$ in step 2 simply returns an array with two components, where component zero contains the number of negative instances in D, while component one contains the number of positives. At this stage we now know exactly how many positive and negative instances we require in the training/test data sets, and what proportion of the data should be labelled. To actually perform the sampling then we must read in each instance x_i from D one at a time, and decide what to do with it. There are three distinct possibilities,

1. Insert instance x_i in to the training set.
2. Insert instance x_i in to the test set and label it.
3. Insert instance x_i in to the test set and remove its label.

In order to decide what to do, we randomly select instances to insert in to the test and training sets, using meta data described in Section 6.4. There are some further variables which must be initialised before this can be achieved. This is because in order to sample the data set in a single pass [5], we must keep a count of,

1. those instances which have been inserted in to the positive and negative training sets.
2. how many of those were labelled.
3. which particular instance x_i in D we are up to.

[5] Strictly speaking the sampling is not single pass, as meta data must be obtained before hand.

These extra variables are summarised in Table 4 below.

Variable	Description
toLabel	The number of instances in D_{test} to label.
notToLabel	The number of instances in D_{test} not to label.
labelled	A count of the instances labelled in D_{test}.
unlabelled	A count of the instances unlabelled in D_{test}.
negInTrainingSet	A count of the negatives put in the training set.
negInTestSet	A count of the negatives put in the test set.
posInTrainingSet	A count of the positives put in the training set.
posInTestSet	A count of the positives put in the test set.
trainingSetCount	A count of the instances put in the training set.
testSetCount	A count of the instances put in the test set.
instanceIndex	The current index i of the instance x.

Table 4: The secondary variables used for data set sampling.

With these additional variables described, we move on to consider the remainder of the algorithm. There are two components worthy of note. The first randomly selects candidates to include in the sample, using the meta data described in Section 6.4. Meta data are stored in a hash map based data structure on lines 31-32 of algorithm 2, one hash map per class. The key for these data structures is simply an integer representing the order an instance was inserted into the map, whilst the value contains the line each instance can be found at in D. For the positive class, this data structure can be described abstractly by the following table.

Key	Value
<Instance 1>	<Instance line number in D >
<Instance 2>	<Instance line number in D >
...	...
<Instance $D^+ - 1$ >	<Instance line number in D >
<Instance D^+ >	<Instance line number in D >

Table 5: An example of the positive hash data structure.

A random number generator is used to choose instances to use. The generator chooses values in the range $[1, N]$, where N is the total number of instances in the hash map. In the case of the positive hash map described in Table 5, the range would be $[1, D^+]$.

Algorithm 2 Part 2

```
17:   toLabel ← $(D^+_{test} + D^-_{test}) \times D^l_{test}$
18:   notToLabel ← $(D^+_{test} + D^-_{test}) - toLabel$
19:   labelled ← 1
20:   unlabelled ← 1
21:   negInTrainingSet← 0
22:   negInTestSet← 0
23:   posInTrainingSet← 0
24:   posInTestSet← 0
25:   testSetCount← 0
26:   trainingSetCount← 0
27:   instanceIndex← 0
28:   writeCounter ← 0
29:   testSetString← null
30:   HashMap<int,int> $positives \leftarrow getMetaData(pathToPositiveMetaData)$
31:   HashMap<int,int> $negatives \leftarrow getMetaData(pathToNegativeMetaData)$
32:   posTrainingSet ← $HashMap < int, bool >$
33:   posTestSet ← $HashMap < int, bool >$
34:   negTrainingSet ← $HashMap < int, bool >$
35:   negTestSet ← $HashMap < int, bool >$
36:   $r \leftarrow new\ Random()$
37:                                              ▷ continued in Part 3...
```

It is then simply a case of populating the hash maps declared in lines 33-36 of algorithm 2, with those instances randomly selected. The pseudocode for how this is achieved is presented below. For brevity we only show the code applicable for selecting positive instances for the training set.

Algorithm 3 Part 3

```
38:   while $(posTrainingSet.size() < D^+_{train})$ do
39:       $randIndex \leftarrow r.nextInt(0, D^+)$
40:       if $(positives.get(randIndex) \neq null)$ then
41:           while $(posTrainingSet.hasKey(positives.get(randIndex)))$ do
42:               $randIndex \leftarrow r.nextInt(0, D^+)$
43:       $posTrainingSet.put(positives.get(randIndex), true)$;
44:                                              ▷ continued in Part 4...
```

In the code above $nextInt()$ is a method that returns a random number uniformly distributed in a range specified, using minimum and maximum values. The $get(int)$ method on the other hand, simply returns a value from the hash map specified by the supplied integer key.

Once the instances to be used in the sampling have been selected, the next crucial component reads the data file D, and writes each candidate to either i) a destination training set, ii) a destination test set, or iii) neither if not selected. The final part of the sampling procedure that makes these determinations, is presented overleaf.

The while loop on line 50 simply iterates over the dataset D, until both the training and test data are filled with instances. The calls to $Write()$ simply write out the current instance to the specified file. On line 64 and 74, the $label()$ function is used to decided whether or note a instance should be labelled. The pseudocode for this procedure is also presented for completeness overleaf.

Algorithm 4 Part 4

```
45:   $i \leftarrow 0$
46:   while $(i < D_{tot} \wedge (trainingSetCount < D^{max}_{train}) \wedge (testSetCount < D^{max}_{test}))$ do
47:       $i \leftarrow i + 1$
48:       $Instance \leftarrow D.readLine()$
49:       if $(posTrainingSet.containsKey(i))$ then
50:           $posInTrainingSet \leftarrow posInTrainingSet + 1$
51:           $trainingSetCount \leftarrow trainingSetCount + 1$
52:           $Write(trainPath, Instance)$
53:       else if $(negTrainingSet.containsKey(i))$ then
54:           $negInTrainingSet \leftarrow negInTrainingSet + 1$
55:           $trainingSetCount \leftarrow trainingSetCount + 1$
56:           $Write(trainPath, Instance)$
57:       else if $(posTestSet.containsKey(i))$ then
58:           $posInTestSet \leftarrow posInTestSet + 1$
59:           $testSetCount \leftarrow testSetCount + 1$
60:           if $label(labelled, unlabelled, toLabel, notToLabel, D^{l}_{test})$ then
61:               $labelled \leftarrow labelled + 1$
62:               $Write(testPath, Instance)$
63:           else
64:               $unlabelled \leftarrow unlabelled + 1$
65:               $Write(testPath, RemoveLabel(Instance))$
66:       else$(negTestSet.containsKey(i))$
67:           $negInTestSet \leftarrow negInTestSet + 1$
68:           $testSetCount \leftarrow testSetCount + 1$
69:           if $label(labelled, unlabelled, toLabel, notToLabel, D^{l}_{test})$ then
70:               $labelled \leftarrow labelled + 1$
71:               $Write(testPath, Instance)$
72:           else
73:               $unlabelled \leftarrow unlabelled + 1$
74:               $Write(testPath, RemoveLabel(Instance))$
```

Here is the labelling procedure:

Algorithm 5 Labelling Procedure

Require: The number of instances currently with labels *labelled*, the number of instances not labelled *unlabelled*, the number of instances that must be labelled *toLabel*, the number of instances that must not be labelled *notToLabel*, and the labelling ratio D^l_{test}.

Output: True if an item should be labelled, else false.

```
procedure LABEL($labelled$, $unlabelled$, $toLabel$, $notToLabel$,$D^l_{test}$)
    $r \leftarrow new\ Random()$
    $result \leftarrow 0$
    while ($result = 0$) do
        $rand \leftarrow r.nextDouble()$
        $result \leftarrow compare(rand, D^l_{test})$
    if ($compare(D^l_{test}, 1) = 0$) then        ▷ $D^l_{test} = 1$ label every instance.
        return true
    else if ($compare(D^l_{test}, 0.0) = 0$) then        ▷ $D^l_{test} = 0$ label none.
        return false
    else
        if ($result > 0 \wedge (unlabelled < notToLabel)$) then
            return false
        else if ($result < 0 \wedge (labelled < toLabel)$) then
            return true
        else if ($result > 0 \wedge (unlabelled < notToLabel)$) then
            return true
        else
            return false
```

This procedure decides if an instance should be labelled, by first choosing a random number *rand*, in the interval $[0, 1]$. If $rand < D^l_{test}$, then the instance will be labelled. Else it will not be labelled. For instance, if $D^l_{test} = 0.75$ (i.e. 75% of the instances should be labelled), then $0 < rand < 0.75$ will result in the instance being labelled, and if $0.75 < rand < 1$ it will not be labelled. Here the call to *nextDouble*(), returns a random double value generated uniformly in the interval $[0, 1]$. Whilst $compare(a, b)$, returns zero if $a = b$, a value less than zero when $a < b$, else a value greater than 0 is returned.

6.6 Evaluation

The procedure which STUFFED uses to execute and evaluate tests, is summarised in Figure 3. A discrete time model is adopted for each classifier test, thus per instance x_i arriving at time step t, a class prediction is made at time step $t+1$. The classifier model is then updated via training at time step $t+2$, but only if the instance was labelled in the stream. This corresponds to an incremental, rather than a batch processing approach to learning [6].

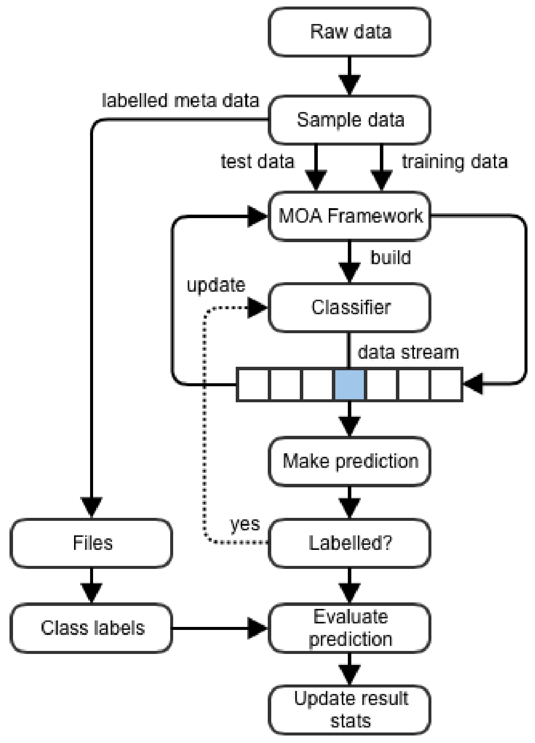


Figure 3: An overview of the test test and evaluation procedure.

Following each prediction a count of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) is updated, by checking the prediction against the correct label of x_i. If x_i was unlabelled in the stream, then the correct label is obtained from a meta data file, and compared to the prediction as before. Thus each prediction is evaluated during testing. However, overall evaluation statistics are not computed until the end of a test run.

This incremental evaluation differs slightly, from the two most common approaches used for evaluating classifier performance under the batch paradigm. The first of these evaluates classifier performance on a single batch of data, selected from among the many batches in the stream. The batch selected is usually the last one received from the stream, on the assumption that performance on the last batch is representative of performance on the stream as a whole.

The second evaluates performance over all the batches in the stream using some measure of average performance. The former approach is unsuitable for evaluating classifier performance in our problem domain, as evaluating

[6] In the batch processing scenario, at time t a batch b of n unlabelled instances would be received and classified using a model trained on batches b_1 to b_{t-1}. At time $t+1$ labels arrive for batch b_t, along with a new batch of unlabelled instances b_{t+1}.

a single batch from an imbalanced stream, can be extremely misleading. Consider the following illustrative example. Suppose we have a stream containing 1 million instances, imbalanced such that there are 10,000 negative examples for each positive. Even with a batch size as large as 10,000 instances, given that positives occur randomly and infrequently in the stream, it is possible that any arbitrary batch chosen for evaluation will contain zero positive instances. Such a batch says little about classifier performance on the minority positive class. In terms of evaluating across all batches using some averaging metric, whilst this approach is preferable to evaluating on a single batch, it is not without problems. For imbalanced streams such averages will also provide a misleading impression of performance, given that most measures of accuracy will tend towards one, as more batches from the stream are seen (see discussion below). In practice the incremental evaluation approach employed by STUFFED, has greater precision given that absolute statistics rather than averages, are maintained throughout.

Any evaluation performed on the streams described thus far, must account for class imbalance. However evaluating classifier performance on imbalanced data is known to be difficult, particularly as many metrics are sensitive to the underlying class distribution [10]. Using these metrics can provide misleading, often positive impressions of performance. One such metric to which this happens is classifier accuracy (see Eq. **??**). Accuracy is acutely effected by class imbalance, and this can be illustrated using a simple example. Suppose we have a stream of data imbalanced in favour of the negative class. A hypothetical classifier which is 100% accurate on the negative class and 0% accurate on the positive class, has an accuracy that tends towards 1, as the number of instances classified $n = TN + TP + FP + FN$ becomes larger,

$$\lim_{TN \to \infty} \frac{TP + TN}{TP + TN + FP + FN} = 1 \ , \tag{1}$$

where TP is the number of instances correctly classified as positive (in this case zero), and TN the number of instances correctly classified as negative. If actively monitoring classifier accuracy, we must then be prepared for misleading results. In any case, a single metric will tell us little with regards to classifier performance. Therefore STUFFED keeps track of multiple assessment metrics. Some of these are described in Table 6.

Statistic	Description	Definition
Accuracy	Overall accuracy.	$\frac{(TP+TN)}{(TP+FP+FN+TN)}$
False positive rate (FPR)	Fraction of negative instances incorrectly labelled positive.	$\frac{FP}{(FP+TN)}$
F-Score	Measure of accuracy that considers both precision and recall.	$2 \times \frac{precision \times recall}{precision + recall}$
G-Mean	Imbalanced data metric describing the ratio between positive and negative accuracy.	$\sqrt{\frac{TP}{TP+FN} \times \frac{TN}{TN+FP}}$
Precision	Fraction of retrieved instances that are positive.	$\frac{TP}{(TP+FP)}$
Recall	Fraction of positive instances that are retrieved.	$\frac{TP}{(TP+FN)}$
Specificity	Fraction of negatives correctly identified as such.	$\frac{TN}{(FP+TN)}$

Table 6: Standard evaluation metrics for classifier performance. True Positives (TP), are those candidates correctly classified as pulsars. True Negatives (TN) are those correctly classified as *not* pulsars. False Positives (FP) are those incorrectly classified as pulsars, finally False Negatives (FN) are those incorrectly classified as *not* pulsars. All metrics produce values in the range $[0, 1]$.

These include amongst others the F-Score and the G-Mean [10]. The F-Score is monitored due to its widespread use by the machine learning community, however we note that the F-Score is sensitive to changes in the class distribution, given its dependence on precision. Thus when a data set is large and imbalanced, the numerator of the F-Score equation remains small, while the denominator becomes increasingly large. Thus low precision can cause the F-Score to obtain a small value, even given a high level of recall. The G-Mean however is not sensitive to the class distribution in the same way. It evaluates the inductive bias in terms of the ratio between positive and negative accuracy. Together then these two metrics give a representative view of classifier performance which should allow a comparison between algorithms tested within STUFFED and other data mining tools.

References

[1] R. J. Lyon, J. D. Knowles, J. M. Brooke, and B. W. Stappers, “A Study on Classification in Imbalanced and Partially-Labelled Data Streams”, in *Simple and Effective Machine Learning for Big Data, Special Session, IEEE International Conference on Systems, Man, and Cybernetics*, SMC ’13, pp. 1506–1511, IEEE, Manchester, United Kingdom, October 2013.

[2] R. J. Lyon, J. D. Knowles, J. M. Brooke, and B. W. Stappers, “Hellinger Distance Trees for Imbalanced Streams”, in *22nd IEEE International Conference on Pattern Recognition*, ICPR ’14, pp. 1969–1974, IEEE, Stockholm, Sweden, August 2014.

[3] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, “Data stream mining: a practical approach”, tech. rep., The University of Waikato, May 2011.

[4] The University of Waikato, “Weka 3: Data mining software in java.” World Wide Web Accessed (12/10/2012), `http://www.cs.waikato.ac.nz/ml/weka/`, 2013.

[5] The University of Waikato, “Massive Online Analysis Manual.” World Wide Web Accessed (12/10/2012), `http://www.cs.waikato.ac.nz/~abifet/MOA/Manual.pdf`, August 2009.

[6] Lichman M., “UCI machine learning repository”, 2013.

[7] Bhatt R. and Dhall A., “Skin segmentation dataset”, 2013.

[8] Roe B., “Miniboone particle identification data set”, 2013.

[9] Savicky P. and Bock R. K., “Magic gamma telescope data set”, 2013.

[10] H. He and E. Garcia, “Learning from Imbalanced Data”, *Knowledge and Data Engineering, IEEE Transactions on*, vol. 21, no. 9, pp. 1263–1284, 2009.