

DPASF: A Flink Library for Streaming Data preprocessing

Alejandro Alcalde-Barros^{a,*}, Diego García-Gil^{a,**}, Salvador García^a, Francisco Herrera^a

^a*Department of Computer Science and Artificial Intelligence, CITIC-UGR (Research Center on Information and Communications Technology), University of Granada, Calle Periodista Daniel Saucedo Aranda, 18071, Granada, Spain*

Abstract

Data preprocessing techniques are devoted to correct or alleviate errors in data. Discretization and feature selection are two of the most extended data preprocessing techniques. Although we can find many proposals for static Big Data preprocessing, there is little research devoted to the continuous Big Data problem. Apache Flink is a recent and novel Big Data framework, following the MapReduce paradigm, focused on distributed stream and batch data processing.

In this paper we propose a data stream library for Big Data preprocessing, named DPASF, under Apache Flink. We have implemented six of the most popular data preprocessing algorithms, three for discretization and the rest for feature selection.

The algorithms have been tested using two Big Data datasets. Experimental results show that preprocessing can not only reduce the size of the data, but to maintain or even improve the original accuracy in a short time.

DPASF contains useful algorithms when dealing with Big Data data streams. The preprocessing algorithms included in the library are able to tackle Big Datasets efficiently and to correct imperfections in the data.

Keywords: Flink, Big Data, Machine Learning, Data preprocessing

1. Background

In recent years, the amount of data generated can no longer be treated directly by humans or manual applications, there is a need to analyze this data automatically and at large scale [1]. This is what is known to be Big Data, data generated at high volume, velocity and variety. This kind of data require a new high-performance processing and it can be found in many fields in these times [2].

In order to extract quality information from data, a previous steps to learning must be performed. This is known as data preprocessing [3], this step is almost mandatory in order to obtain a good model. Data preprocessing [4] deals with missing values, noise, and redundant features [1] among others.

*Corresponding author

**Correspondence

Email addresses: algui91@gmail.com (Alejandro Alcalde-Barros), djgarcia@decsai.ugr.es (Diego García-Gil)

Preprint submitted to Elsevier

October 16, 2018

Data streams [5] are sequences of unbounded and ordered data that arrives one at a time. This imposes restrictions on the learning algorithms which do not appear on static data. Therefore, new algorithms that can deal with this kind of data must be developed.

To address the problem of dealing with such quantity of data in the Big Data era, distributed frameworks such as Apache Spark [6] and Apache Flink [7, 8] has been developed. Apache Spark is well known, designed as a fast and general engine for large-scale in-memory data processing. Apache Flink focus, on the other hand, resides on distributed streams and batch data processing [9]. On top of that, Flink is the only system incorporating a distributed dataflow runtime exploiting pipelined streaming execution for both stream and batch workloads, exactly once state consistency using a lightweight checkpoint system, iterative processing built-in and window semantics supporting out-of-order processing [10].

This paper present the data stream library for Big Data preprocessing named DPASF, where six data preprocessing algorithms are implemented for Apache Flink, focusing on the discretization and feature selection problems. The selected algorithms are InfoGain [11], FCBF [12] and OFS [13] for feature selection and IDA [14], LOFD [15] and PiD [16] for discretization.

The rest of the paper describes the theoretical background of the algorithms, experimental results and a brief tutorial on how to use them. Source code is available on GitHub¹ [17]. The algorithms chosen are the most representative in data streaming preprocessing, in addition, have shown positive results.

1.1. Big Data

In general, Big Data [18, 19, 20] is known to be data that is too big or too complex to handle by conventional tools and on a single machine. For this reason, there is a increasing need in developing new tools that can handle all this amount of data efficiently. In order to accomplish it, distributed frameworks like Hadoop [21], Spark [6] and Flink [7] were developed. This type of frameworks allow to process larges amount of data in a scalable way.

One common way of defining Big Data is to describe it in terms of three dimensions, also known as the 3 V's [22] (Volume, Velocity and Variety). Volume just refers to how much data there is. Velocity refers to the speed at which data is processed and analyzed. Lastly, Variety refers to how many different data formats there are to be analyzed.

1.2. Data Streaming

The main characteristics of streaming data [23] are the following. In streaming data, instances are not available beforehand, it becomes available in a sequence fashion, one by one, or in batches. Instances can arrive quickly and at irregular time intervals. Due to streaming data is unbounded, it may be infinite and can not be stored in memory. Each instance is accessed only one time (most of the time) and then is discarded. In order to provide real-time processing, instances are processed within a limited amount of time. The intrinsic characteristics of data are subject to change over time, this is what is know as concept drift [24].

¹<https://github.com/elbaulp/dpasf>

Concept drift [5] is the main problem within streaming data, as it is important for the algorithm to detect it and update the learned model to reflect the changes underlying to the data.

1.3. Data preprocessing

Before applying any data mining process, it is necessary to adapt the data into the requirements imposed by each learning algorithm and clean the data properly.

Although data preprocessing is a critical step, often it is time consuming. There are two types of data preprocessing, those designed to reduce the complexity in the data, and those designed to prepare the data, this means data transformation, cleaning, normalizing etc. The former is called data reduction, the latter data preparation. When these techniques are applied, the data is in its final stage to be fed to the data mining algorithm.

Among data transformation, feature selection [1, 25] takes care of selecting only relevant and non-redundant attributes. The aim of this type of data preprocessing is to obtain a subset of the original data that stills maintains the ability to describe the inherent concept. As a side effect, reducing the complexity of the data also results in better efficiency in terms of the amounts of time to learn a model, as well as preventing over-fitting.

Discretization [14] is a technique for reducing the complexity of the data by dividing the domain of the variables into bins defined by cut points. This process transform quantitative values into qualitative, as cut points define a set of non-overlapping intervals. Once the algorithm has computed cut points for each attribute, data is then mapped to its corresponding interval [1].

1.4. Apache Flink

Although it may seems Apache Spark and Apache Flink are similar, they are designed to address different problems. Apache Spark process all data using a batch approach, it lacks a true streaming processing. Apache flink fills this gap. Flink provides both kind of processing, batch and streaming, but Flink process streaming data as it happens, in an online fashion. In other words, Spark “emulates” streaming by processing streaming data in mini-batches, whereas Flink process them online. This makes Flink more efficient in terms of low latency.

Apache Flink has a fault tolerance system in order to recover from exceptions that may occur. It is designed to work at low latency even with large amounts of data.

2. Theoretical description of the algorithms presented

This section presents a theoretical description of the implemented algorithms, as well as an introduction to feature selection (InfoGain [11], FCBF [12] and OFS [13]) and discretization (IDA [14], LOFD [15] and PiD [16]).

2.1. Feature Selection

Feature selection [26] is meant to reduce the dimensionality of a dataset by removing irrelevant and redundant features. By doing this, a subset of the original features that still describes the inherent concept behind the data is returned. FS methods can be divided in the following categories:

- *Wrapper Methods* It uses an external evaluator, which depends on a learning algorithm.
- *Filtering Methods* It uses selection techniques based on separability measures or statistical dependencies.
- *Embedded Methods* It uses a search procedure implicitly embedded on the classifier or regressor.

In general, filtering methods tend to achieve better results when generalizing due to learning independence. In addition, filter methods are more efficient than wrapper methods, since the latter need to learn a model first. Therefore, in the context of big data, filtering methods are more widely used. Information Gain [11], OFS [13] and FCBF [12] are the most popular preprocessing algorithms in this area.

2.1.1. Information Gain

This feature selection scheme, described in [?] is formed by two steps: An incremental feature ranking method, and an incremental learning algorithm that can consider a subset of the features during prediction.

For this algorithm, the conditional entropy with respect to the class is computed with

$$H(X|Y) = - \sum_j P(y_j) \sum_i P(x_i|y_j) \log_2(P(x_i|y_j)) \quad (1)$$

then, the Information Gain (IG) is computed for each attribute with

$$IG(X|Y) = H(X) - H(X|Y) \quad (2)$$

Once the algorithm has all Information Gain values for each attribute, the top N are selected as best features.

2.1.2. Online Feature Selection (OFS)

OFS [13] proposes an ϵ -greedy online feature selection method based on weights generated by an online classifier (neural networks) which makes a trade-off between exploration and exploitation of features.

The main idea behind this algorithm is that when a vector \mathbf{x} falls within a $L1$ ball, most of its numerical values are concentrated in its largest elements, therefore, removing the smallest values will result in a small change in the original vector x as measured by the L_q norm. This way, the classifier is restricted to a $L1$ ball:

$$\Delta_R = \left\{ \mathbf{w} \in \mathbb{R}^d : \|\mathbf{w}\|_1 \leq R \right\} \quad (3)$$

OFS maintains an online classifier \mathbf{w}_t with at most B nonzero elements. When an instance (\mathbf{x}_t, y_t) is incorrectly classified, the classifier gets updated through online gradient descent and then it is projected to a $L2$ ball to delimit the classifier norm. If the resulting classifier $\hat{\mathbf{w}}_{t+1}$ has more than B nonzero elements, the elements with the largest absolute value will be kept in $\hat{\mathbf{w}}_{t+1}$.

The above approach presents an inefficiency, even although the classifier consists in B nonzero elements, full knowledge of the instances is required, that is, each attribute \mathbf{x}_t must be measured and computed. As a solution, OFS limit online feature selection to no more than B attributes of \mathbf{x}_t

2.1.3. Fast Correlation-Based Filter (FCBF)

FCBF [12] is a multivariate feature selection method where the class relevance and the dependency between each feature pair are taken into account. Based on information theory, FCBF uses symmetrical uncertainty to calculate dependencies of features and the class relevance. Starting with the full feature set, FCBF heuristically applies a backward selection technique with a sequential search strategy to remove irrelevant and redundant features. The algorithm stops when there are no features left to eliminate.

The algorithm chooses as a correlation measure the entropy of a variable X , which is defined as

$$H(X) = - \sum_i P(x_i) \log P(x_i) \quad (4)$$

and the entropy of X after observing values of another variable Y is defined as

$$H(X|Y) = - \sum_j P(y_j) \sum_i P(x_i|y_j) \log_2(P(x_i|y_j)) \quad (5)$$

where $P(x_i)$ is the prior probability for all values of X and $P(x_i|y_j)$ is the posterior probability of X given the values of Y . With this information, a measure called *Information Gain* can be defined:

$$IG(X|Y) = H(X) - H(X|Y) \quad (6)$$

According to IG, a feature Y is more correlated to X than to a feature Z if $IG(X|Y) > IG(Z|Y)$.

Now is all set to define the main measure for FCBF, *symmetrical uncertainty* [27]. As a pre-requisite, data must be normalized in order to be comparable.

$$SU(X, Y) = 2 \left[\frac{IG(X|Y)}{H(X) + H(Y)} \right] \quad (7)$$

SU compensate the bias in IG toward features with more values and normalizes its values to the range $[0, 1]$. A SU value of 1 indicates total correlation whereas a value of 0 indicates independence.

The algorithm follows a two step approach, first, it has to decide if a feature is *relevant* to the class and two, decide if those features are *redundant* with respect to each other.

To solve the first step, a user defined SU threshold can be defined. If $SU_{i,c}$ is the SU value for feature F_i with the class c , the subset S' of relevant features can be defined with a threshold δ such that $\forall F_i \in S', 1 \leq i \leq N, SU_{i,c} \geq \delta$.

For the second step, in order to avoid analysis of pairwise correlations between all features, a method to decide whether the level of correlation between two features in S' is high enough to produce redundancy is needed in order to remove one of them. Examining the value $SU_{j,i} \forall F_j \in S' (j \neq i)$ allow to estimate the level to what F_j is correlated by the rest of features in S' .

The last piece of the algorithm comprehend two definitions:

Definition 1 (Predominant Correlation). The correlation between a feature F_i and the class C is predominant iff $SU_{i,c} \geq \delta$ and $\forall F_j \in S' (j \neq i) \nexists F_j$ such that $SU_{j,i} \geq SU_{i,c}$

If such feature F_j exists to a feature F_i , it is called a redundant peer to F_i and its added to a set S_{P_i} identifying all the redundant peers for F_i . S_{P_i} is divided in two parts: $S_{P_i}^+$ and $S_{P_i}^-$, where $S_{P_i}^+ = \{F_j | F_j \in S_{P_i}, SU_{j,c} > SU_{i,c}\}$ and $S_{P_i}^- = \{F_j | F_j \in S_{P_i}, SU_{j,c} \leq SU_{i,c}\}$

Definition 2 (Predominant Feature). A feature is predominant to the class iff its correlation to the class is predominant or can become predominant after removing all its redundant peers.

With the definitions above, a feature will be a *good feature* if it is *predominant* in predicting the class. This two definitions along with the next heuristics can effectively identify predominant features and remove the need of pairwise comparisons.

Heuristic 1 (when $S_{P_i}^+ = \emptyset$). F_i is a predominant feature, delete all features in $S_{P_i}^-$ and stop searching for redundant peers for those features.

Heuristic 2 (when $S_{P_i}^+ \neq \emptyset$). All features in $S_{P_i}^+$ are processed before making decisions on F_i . If none of them become predominant go to Heuristic 1, else remove F_i and decide if features in $S_{P_i}^-$ need to be removed based on other features in S' .

Heuristic 3 (Start point). The algorithm begins examining the feature with the largest $SU_{i,c}$, as this feature is always predominant and acts as a starting point to remove redundant features.

2.2. Discretization

Broadly speaking, discretization [28] translates quantitative data into qualitative data, trying to avoid an overlap between the continuous domain of the variable. This process results in a mapping of a value to a given interval. For this reason, discretization can be considered as a data reduction process, since it reduces data from a numerical domain to a subset of categorical values.

More formally, a discretization δ of a numeric attribute X_i is a set of m intervals called *bins*. The bins are defined by cut points $\{k_1, \dots, k_{m-1}\}$ that divide the domain of X_i into m bins where $b_1 = [-\infty, k_1]$, $b_m = [k_{m-1}, \infty]$ and for $1 < i < m$, $b_i = (k_{i-1}, k_i]$. Therefore, a discretization for an attribute X_i is a mapping between values v of X_i and its bin indexes $\delta v = z$ such that $v \in b_z$.

The most popular discretization algorithms are IDA [14], PiD [16] and LOFD [15].

2.2.1. Incremental Discretization Algorithm (IDA)

IDA [14] approximates quantile-based discretization on the entire data stream encountered to date by maintaining a random sample of the data which is used to calculate the cut points. IDA uses the reservoir sampling algorithm to maintain a sample drawn uniformly at random from the entire stream up until the current time.

In IDA, a random sample is used because it's not feasible for high-throughput streams to maintain a complete record of all the values seen so far. The sample method used

is called reservoir sampling [29], and maintains a random sample of s values V_i for each attribute X_i . The first s values that arrives for each X_i are added to its corresponding V_i . Thereafter, every time a new instance $\langle \mathbf{x}_n, y_n \rangle$ arrives, each of its values \mathbf{x}_n^i replaces a randomly selected value of the corresponding V_i with probability s/n .

Each value of each attribute is stored in a vector of interval heaps [30]. V_i^j stores the values for the j^{th} bin of X_i . The reason to use a Interval Heap is that it provides efficient access to minimum and maximum values in the heap and direct access to random elements within the heap.

2.2.2. Partition Incremental Discretization algorithm (PiD)

PiD [16] performs incremental discretization. The basic idea is to perform the task in two layers. The first layer receives the sequence of input data and keeps some statistics on the data using many more intervals than required. Based on the statistics stored by the first layer, the second layer creates the final discretization. The proposed architecture processes streaming examples in a single scan, in constant time and space even for infinite sequences of examples.

PiD [16] performs incremental discretization. The basic idea is to perform the task in two layers.

The first layer receives the sequence of input data and the range of the variable and keeps some statistics on the data using many more intervals than required. The range of the variable is used to initialize the cut points with the same width. Each time a new value arrives, this layer is updated in order to compute the corresponding interval for the value. Each interval has a internal count of the values it has seen so far. When a counter for an interval reach a threshold a split process is triggered to generate two new intervals. If the interval triggering the split process is the last or the first, a new interval with the same step is created. Otherwise the interval is splitted in two. In summary, the first layer simplifies and summarizes the data.

Based on the statistics stored by the first layer, the second layer creates the final discretization. The proposed architecture processes streaming examples in a single scan, in constant time and space even for infinite sequences of examples. To accomplish it, this layer merges the set of intervals computed in the previous layer.

PiD stores the information about the number of examples per class in each interval in a matrix. In this matrix, columns corresponds with the number of intervals and rows with the number of classes. With this information, the conditional probability of an attribute belonging to an interval given that the corresponding example belongs to a class can be computed as $P(b_i < x \leq b_{i+1} | Class_j)$.

To perform the actual discretization *Recursive entropy discretization* [31] is used. This algorithm was developed by Fayyad and Irani [32]. It uses the class information entropy of two candidate partitions to select the boundaries for discretization. It begins searching for a single threshold that minimizes the entropy over all possible cut points, then, it is applied recursively to both partitions. It uses the *minimum description length* [33] principle as stop criteria. The algorithm works as follow:

First, the entropy before and after the split is computed as well as its information gain. Then, the entropy for both left and right splits is computed and finally the algorithm

check if the split is accepted with the following formula

$$Gain(A, T; S) < \frac{\log_2(N - 1)}{N} + \frac{\Delta(A, T; S)}{N} \quad (8)$$

where N is the number of instances in the set S ,

$$Gain(A, T; S) = H(S) - H(A, T; S) \quad (9)$$

and

$$\Delta(A, T; S) = \log_2(3^k - 2) - [k \cdot H(S) - k_1 \cdot H(S_1) - k_2 \cdot H(S_2)] \quad (10)$$

where k_i is the number of class labels represented in the set S_i .

2.2.3. Local Online Fusion Discretizer (LOFD)

LOFD [15] is an online, self-adaptive discretizer for streaming classification. It smoothly adapts its interval limits reducing the negative impact of shifts and analyze interval labeling and interaction problems in data streaming. Interaction discretizer-learner is addressed by providing 2 alike solutions. The algorithm generates an online and self-adaptive discretization solution for streaming classification which aims at reducing the negative impact of fluctuations in evolving intervals.

The algorithm is constituted by two phases, the main process, at instance level, and the merge/split process, at interval level. The main process works as follows. First, discrete interval are initialized following the static process defined in [34]. The discretization is then performed on the first *initTh* instances. From that moment on, LOFD updates the scheme of intervals in each iteration and for each attribute. For each new instance, it retrieves its ceiling interval (implemented as a red-black tree). If the point is above the upper limit a new interval is generated at that point, being that point the new maximum for the current attribute. A merge between the old last interval and the new is evaluated by computing the quadratic entropy, if the result is lower than the sum of both parts, the merge is accepted.

Finally, each point is added to a queue with timestamp to control future removals in the case the histogram overflows. If necessary, LOFD recovers points from the queue in ascending order and remove them until there is space left in the histogram.

The split/merge phase is triggered each time a boundary point is processed. The new boundary point splits an interval in two, one interval contains the points in the histogram with values less than or equal to the new point and keeps the same label. Each time a new interval is generated, the merge process is triggered for the intervals being divided and its neighbors.

3. Implementation

This section presents the pseudocode for the implemented algorithms. The following Flink primitives have been used:

- **map** The Map transformation applies a user-defined map function on each element of a DataSet
- **reduce** A Reduce transformation reduces the dataset to a single element using a user-defined reduce function.

Algorithm 1 FCBF Algorithm

```
1: Input: data a DataSet LabeledVector (label, features)
2: Input: thr threshold
3: Output: DataSet with the most important features
4: su  $\leftarrow$ 
5: for i  $\leftarrow$  0 until nAttrs do
6:   attr  $\leftarrow$ 
7:   map instance  $\in$  data
8:     (label, featurei)
9:   end map
10:  yield SU(attr)
11: end for
12: suSorted  $\leftarrow$  FILTER(su  $\geq$  thr).SORTDESC
13: sBest  $\leftarrow$  FCBF(suSorted)
14: return sBest
```

- **mapPartition** MapPartition transforms a parallel partition in a single function call.
- **reduceGroup** A GroupReduce transformation that is applied on a grouped DataSet calls a user-defined group-reduce function for each group. The difference between this and Reduce is that the user defined function gets the whole group at once.

Algorithm 1 shows pseudocode for FCBF, the SU value is computed for each attribute in parallel. All SU values are then filtered according to the threshold parameter and then sorted descendingly. With this final sorted values, FCBF algorithm is applied like originally described in [12]. Algorithm 2 shows how Symmetrical Uncertainty is computed in a distributed fashion. First, each parallel partition compute the partial counts of each value, then this partial counts are aggregated using a reduce function in order to compute the total counts. With this information, probabilities for each value are computed and its entropy and mutual information are calculated. Finally, it returns the corresponding SU value for that attribute.

Algorithm 3 shows the implementation for Information Gain [?]. First the frequencies of each value with respect to the class label are computed. With this information, the total entropy of the dataset is computed. Next, for each attribute, its frequency, probability, entropy and conditional entropy are computed. Finally, the information gain for the *i*-th attribute its computed and stored into *gains*. Algorithm 4 shows how frequencies are computed.

Algorithm 5 shows the pseudocode for OFS [13], this algorithm maps each label and feature with its corresponding value for the original OFS algorithm.

Algorithm 6 shows pseudocode for IDA [14]. This algorithm first compute the cut points for the dataset with the desired number of bins. In order to compute the cut points, each instance is mapped to the result of IDA, which returns the computed cut points. To achieve it, each feature is zipped with its index, and then folded with its corresponding class label and a zero feature vector that will be filled in each iteration of the fold operation, with the returned value of IDA algorithm. Once cut points are stored, line 5 in Algorithm 6 discretizes the data according to those cut points.

Algorithm 2 Symmetrical Uncertainty function (SU)

```
1: Input: attr Attribute to compute SU to
2: Output: SU value for attr
3: xypartialCounts  $\leftarrow$ 
4: mapPartition (y, x)  $\in$  attr
5:   xPartialCounts  $\leftarrow$  COMPUTECOUNTS(x)
6:   yPartialCounts  $\leftarrow$  COMPUTECOUNTS(y)
7:   (xPartialCounts, yPartialCounts)
8: end mapPartition
9: totalCounts  $\leftarrow$  REDUCE(xypartialCounts)
10: su  $\leftarrow$ 
11: map (xcounts, ycounts, x, y)  $\in$  totalCounts
12:   px  $\leftarrow$  PROB(x)
13:   py  $\leftarrow$  PROB(y)
14:   hx  $\leftarrow$  ENTROPY(x)
15:   hy  $\leftarrow$  ENTROPY(y)
16:   mu  $\leftarrow$  MUTUALINFORMATION(x,y)
17:    $\frac{2mu}{hx+hy}$ 
18: end map
```

Algorithm 3 InfoGain Algorithm

```
1: Input: data a DataSet LabeledVector (label, features)
2: Input: selectNF Number of features to select
3: Output: DataSet with the most selectNF important features
4: freqs  $\leftarrow$  FREQUENCIES(data, groupBy label)
5: H  $\leftarrow$  ENTROPY(freqs)
6: gains  $\leftarrow$ 
7: map i  $\in$  0 until nFeatures
8:   freqs  $\leftarrow$  FREQUENCIES(data, featurei)
9:   px  $\leftarrow$  PROBS(freqs)
10:  H  $\leftarrow$  ENTROPY(freqs)
11:   $H(Y|Feature_i) \leftarrow$  CONDITIONALENTROPY(freqs)
12:   $H - H(Y|Feature_i)$ 
13: end map
14: return SELECTFEATURES(selectNF, gains)
```

Algorithm 4 Frequencies function

```
1: Input: attr attribute to compute frequencies to
2: Input: f function to group by
3: Output: Frequencies for attr using f
4: grouped  $\leftarrow$  groupBy(data, f)
5: freqs  $\leftarrow$  reduceGroup(grouped)
6: return freqs
```

Algorithm 5 OFS Algorithm

1: **Input:** *data* a DataSet LabeledVector (label, features)
2: **Input:** η parameter
3: **Input:** λ parameter
4: **Input:** *selectNF* Number of features to select
5: **Output:** DataSet with the most *selectNF* important features
6: *finalweights* \leftarrow
7: **map** (*label, features*) \in *data*
8: OFS(*label, features*)
9: **end map**

Algorithm 6 IDA Algorithm

1: **Input:** *data* a DataSet LabeledVector (label, features)
2: **Input:** *bins* number of bins
3: **Output:** Discretized dataset with desired number of *bins*
4: *cuts* \leftarrow
5: **map** ($(y, x) \in$ *data*)
6: *zipped* \leftarrow ZIPWITHINDEX(*x*)
7: *FoldLeft*($(y, \text{emptyfeature})$)(*IDA*())
8: **end map**
9: **return** DISCRETIZE(*data, cuts*)

Algorithm 7 PiD Algorithm

1: **Input:** *data* a DataSet LabeledVector (label, features)
2: **Input:** α parameter
3: **Input:** *step* parameter
4: **Output:** Discretized dataset
5: *cuts* \leftarrow
6: **map** *instance* \in *data*
7: (*instance, Histogram, 1*)
8: **end map**
9: **reduce** (*m1, m2*) \in *cuts*
10: UPDATELAYER1(*m1, m2*)
11: UPDATELAYER2(*m1, m2*)
12: **end reduce**
13: **return** DISCRETIZE(*data, cuts*)

Algorithm 7 shows pseudocode for PiD [16], this algorithm first initialize the required data structures using a map function, this map function expands the dataset and adds to it a histogram and a counter of total instances seen so far. Then this data is reduced computing in each reduce step the layers one and two as described in the original algorithm [16]. Once this reduce stage has been completed, it returns the discretized data using the previously computed cut points.

Algorithm 8 shows pseudocode for LOFD [15]. This algorithm first instantiate a

Algorithm 8 LOFD Algorithm

```
1: Input: data a DataSet LabeledVector (label, features)
2: Output: Discretized dataset
3: lofd ← LOFDInstance
4: cuts ←
5: map  $x \in data$ 
6:   discretized ← lofd.applyDiscretization(x)
7:   for  $s$  in 0 until discretized.size do
8:     lofd.getCutpoints(s)
9:   end for
10: end map
11: reduce  $(-, b) \in cuts$ 
12:   b
13: end reduce
14: return DISCRETIZE(data, cuts)
```

LOFD helper, and maps the data to the computed cut points this helper returns. Once all cutpoints has been collected, the reduce function extract only the most recently computed cut points and apply the discretization based on them.

4. Results

This section present in detail the six algorithms implemented in Apache Flink, there are three algorithms for feature selection and three for discretization.

4.1. Examples

The software has been implemented in the Scala programming language² language. As mentioned above, DPASF consist of six algorithms for data streams, three discretization methods and three feature selection methods. The software can be found on GitHub³. The next section presents how to use each algorithm within Apache Flink.

4.2. Usage

4.2.1. Feature Selection

FCBF. In order to benefit from the Apache Flink framework, symmetrical uncertainty computation for each pair of attributes are distributed across each node in order to speed the process.

Suppose the data set to be used is the Abalone DataSet⁴, to load it in Apache Flink:

```
val abaloneDat = env.readCsvFile[(Int, Double, ..., Int)]
(getClass.getResource("/abalone.csv").getPath)
```

²<https://scala-lang.org/>

³<https://github.com/elbault/DPASF>

⁴<https://archive.ics.uci.edu/ml/datasets/Abalone>

```

.name("Reading Abalone DS")
val abaloneDS = abaloneDat
.map { tuple =>
  val list = tuple.productIterator.toList
  val numList = list map { x =>
    x match {
      case d: Double => d
      case i: Int => i
    }
  }
  LabeledVector(numList(8), DenseVector(numList.take(8).toArray))
}.name("Abalone DS")

```

Then, a `FCBFTransformer` must be instantiated, configure its parameters and finally define a pipeline:

```

val fcbf = FCBFTransformer()
.fsetThreshold(.05)

fcbf fit abaloneDS
val bestFeatures = fcbf transform abaloneDS

```

After fitting the algorithm, calling `transform` on `fcbf` will return the Abalone data set with the most important features.

InfoGain. For this algorithm, each attribute's Information Gain value is computed in parallel.

The use of `InfoGainTransformer` is similar:

```

val data = Vector(
  Vector("1", "0", "10"),
  Vector("0", "0", "10"),
  Vector("1", "0", "10"),
  Vector("0", "1", "20"),
  Vector("0", "0", "10"),
  Vector("1", "1", "20"),
  Vector("1", "0", "10"))

val gain = InfoGainTransformer()
.setNFeatures(2)
.setSelectNF(1)

gain fit dataSet

val result = gain transform dataSet

```

OFS. One difference of OFS with respect to the previous algorithms is that it does not require a fitting phase:

```
val ofs = OFSGDTransformer()
  .setNFeature(5)
val result = ofs transform data
```

4.2.2. Discretization

In this section the usage of discretization method is presented, all of them use the Iris⁵ DataSet, loaded as:

```
// Iris POJO
case class Iris(
  SepalLength: Double,
  SepalWidth: Double,
  PetalLength: Double,
  PetalWidth: Double,
  Class: Double)

val data = env.readCsvFile[Iris](getClass.getResource("/iris.dat").getPath)
val dataSet = data map { tuple =>
  val list = tuple.productIterator.toList
  val numList = list map (_.asInstanceOf[Double])
  LabeledVector(numList(4), DenseVector(numList.take(4).toArray))
}
```

IDA. For IDA, cut points are computed in parallel, in order to get the most recent computed cut point, data is reduced to get the latest set of cuts.

```
val ida = IDADiscretizerTransformer()
  .setBins(5)
val discretizedIris = ida transform dataSet
```

PiD. For PiD, the needed histogram is shared across all nodes. After histogram is initialized, data is reduced in order to produce the final histogram, where the cut points to perform the discretization are.

In PiD, data must be normalized as a previous step, so a `ChainTransformer` is used in the pipeline.

```
val pid = PIDiscretizerTransformer()
  .setAlpha(.10)
```

⁵<https://archive.ics.uci.edu/ml/datasets/Iris/>

```

.setUpdateExamples(50)
.setL1Bins(5)

val scaler = MinMaxScaler()
val pipeline = scaler.chainTransformer(pid)

pipeline fit dataSet
val result = pipeline transform dataSet

```

LOFD. For LOFD, PiD-like approach is used. First, all features are mapped in order to extract the necessary information from them, then, data is reduced to extract the final cut points to perform discretization.

```

val lofd = LOFDiscretizerTransformer()
.setInitTh(1)
val discretized = ofs transform (dataSet)

```

4.3. Results

The experimental set up has used two datasets, *ht_Sensor* and *skin_nonskin*, Table 1 describe them.

Table 1: Information about DataSets for experiments

DataSet	Instances	Attributes	Classes
ht_sensor	929000	11	3
skin_nonskin	245000	3	2

All algorithms has been tested with KNN and Decision Trees using 5-fold cross validation. A baseline is fitted without any preprocessing step, and other with the corresponding preprocessing algorithm. In addition KNN has been fitted with $k = 3$ and $k = 5$.

For feature selection methods, all of them have been set up to select 50% of features.

Table 2 shows the amount of time it took to preprocess the data. The worst algorithm by far is IDA, which took about 5 hours to finish for *ht_sensor*. On the contrary, the fastest was InfoGain. OFS could not be measured as it only accepts binary datasets. It is worth mentioning that these experiments could not have been possible in normal environments due to the amount of time they would have taken.

For all experiments we have used a cluster composed of 14 computing nodes. The nodes hold the following characteristics: 2 x Intel Core i7-4930K, 6 cores each, 3.40 GHz, 12 MB cache, 4 TB HDD, 64 GB RAM. Regarding software, we have used the following configuration: Apache Flink 1.6.0, 238 TaskManagers (17 TaskManagers/core), 49 GB RAM/node.

Tables 3 and 4 show the accuracy obtained by the algorithms, as well as the accuracy without any preprocessing. The three feature selection methods obtain considerable results, even when they are configured to remove half the features on the datasets. FCBF is not among the bests, but it is among the fastest, this may be due to the fact that it

Table 2: Times in seconds

Preprocessing algorithm	ht_sensor	skin_nonskin
FCBF	19	1
OFS	-	1
InfoGain	16	2
IDA	20854	93
LOFD	28	3
PiD	118	7

avoids computing pairwise comparisons when selecting features. Also, FCBF can not be set to select a fixed number of features, it uses a threshold to select features based on its SU value, so in some cases it will select less than half the features. InfoGain also gives excellent results, close to baseline and even improve when k=3. Among the discretizers, PiD outperforms baseline, in all cases but skin-nonsking with k=3.

Table 3: Accuracy for KNN with k = 3

k = 3	ht_sensor	skin_nonskin
No-PP	0.9998	0.9995
FCBF	0.8965	0.8642
OFS	-	0.8985
InfoGain	0.9999	0.9825
IDA	0.8845	0.6591
LOFD	0.9662	0.9755
PiD	0.9999	0.9966

Table 4: Accuracy for KNN with k = 5

k = 5	ht_sensor	skin_nonskin
No-PP	0.9999	0.9994
FCBF	0.8037	0.8684
OFS	-	0.9006
InfoGain	0.9991	0.9838
IDA	0.8850	0.7092
LOFD	0.9665	0.9766
PiD	0.9999	0.9994

Table 5 shows accuracy for a Decision Tree model, results are consistent with the previous model. In general, feature selection methods result in a decrease on accuracy whereas discretization methods are consistent with baseline, albeit PiD improves accuracy.

5. Conclusions

In this paper we have tackled the Big Data streaming preprocessing problem. We have proposed a library for Big Data data stream preprocessing, named DPASF, implemented

Table 5: Accuracy for Decision Trees

DT	ht_sensor	skin_nonskin
No-PP	70.13	98.45
FCBF	57.50	88.00
OFS	-	88.15
InfoGain	67.51	97.10
IDA	68.35	94.24
LOFD	69.85	94.18
PiD	71.06	98.74

in the Big Data streaming framework Apache Flink. This library includes six classic data preprocessing algorithms, three for performing discretization, and another three for the feature selection task. All the algorithms have been redesigned in order to make them able to cope with big datasets.

The performance of the six algorithms in Big Data scenarios has been analyzed using two Big Data datasets. Experimental results have shown that preprocessing can improve the original accuracy in a short amount of time. We have also observed that choosing the right technique is crucial depending on the problem and the classifier used.

References

- [1] S. García, S. Ramírez-Gallego, J. Luengo, J. M. Benítez, F. Herrera, Big data preprocessing: methods and prospects, *Big Data Analytics* 1 (1) (2016) 9.
URL <https://doi.org/10.1186/s41044-016-0014-0>
- [2] F. Saeed, Towards quantifying psychiatric diagnosis using machine learning algorithms and big fmri data, *Big Data Analytics* 3 (1) (2018) 7.
URL <https://doi.org/10.1186/s41044-018-0033-0>
- [3] S. García, J. Luengo, F. Herrera, Tutorial on practical tips of the most influential data preprocessing algorithms in data mining, *Knowledge-Based Systems* 98 (2016) 1 – 29.
URL <http://www.sciencedirect.com/science/article/pii/S0950705115004785>
- [4] S. García, J. Luengo, F. Herrera, *Data Preprocessing in Data Mining*, Springer International Publishing, University of Granada, 2015.
- [5] S. Ramírez-Gallego, B. Krawczyk, S. García, M. Woźniak, F. Herrera, A survey on data preprocessing for data stream mining: Current status and future directions, *Neurocomputing* 239 (2017) 39 – 57.
URL <http://www.sciencedirect.com/science/article/pii/S0925231217302631>
- [6] Spark a. apache spark: lightning-fast cluster computing.
URL <http://spark.apache.org>
- [7] Flink a. apache flink.
URL <http://flink.apache.org>
- [8] B. Friedman, *Introduction to Apache Flink : stream processing for real time and beyond*, O’Reilly Media, Sebastopol, CA, 2016.
- [9] D. García-Gil, S. Ramírez-Gallego, S. García, F. Herrera, A comparison on scalability for batch big data processing on apache spark and apache flink, *Big Data Analytics* 2 (1) (2017) 1.
URL <https://doi.org/10.1186/s41044-016-0020-2>
- [10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink : Stream and batch processing in a single engine, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36 (4), qC 20161222.
URL <http://sites.computer.org/debull/A15dec/issue1.htm>
- [11] I. Katakis, G. Tsoumakas, I. Vlahavas, On the utility of incremental feature selection for the

- classification of textual data streams, in: P. Bozanis, E. N. Houstis (Eds.), *Advances in Informatics*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 338–348.
- [12] L. Yu, H. Liu, Feature selection for high-dimensional data: A fast correlation-based filter solution, 2003, pp. 856–863.
- [13] J. Wang, P. Zhao, S. C. H. Hoi, R. Jin, Online feature selection and its applications, *IEEE Transactions on Knowledge and Data Engineering* 26 (3) (2014) 698–710.
- [14] G. I. Webb, Contrary to popular belief incremental discretization can be sound, computationally efficient and extremely useful in: *Proceedings of the 2014 IEEE International Conference on Data Mining, ICDM '14*, IEEE Computer Society, Washington, DC, USA, 2014, pp. 1031–1036.
URL <https://doi.org/10.1109/ICDM.2014.123>
- [15] S. Ramírez-Gallego, S. García, F. Herrera, Online entropy-based discretization for data streaming classification, *Future Generation Computer Systems* 86 (2018) 59 – 70.
URL <http://www.sciencedirect.com/science/article/pii/S0167739X17325815>
- [16] C. Pinto, Discretization from data streams: applications to histograms and data mining, in: *In Proceedings of the 2006 ACM symposium on Applied computing (SAC'06, 2006*, pp. 662–667.
- [17] A. Alcalde, *elbaulp/dpasf: 0.1.1 release (Oct. 2018)*.
URL <https://github.com/elbaulp/DPASF>
- [18] S. Landset, T. M. Khoshgoftaar, A. N. Richter, T. Hasanin, A survey of open source tools for machine learning with big data in the hadoop ecosystem, *Journal of Big Data* 2 (1) (2015) 24.
URL <https://doi.org/10.1186/s40537-015-0032-1>
- [19] T. R. Rao, P. Mitra, R. Bhatt, A. Goswami, The big data system, components, tools, and technologies: a survey, *Knowledge and Information Systems*.
- [20] S. Ramírez-Gallego, A. Fernández, S. García, M. Chen, F. Herrera, Big data: Tutorial and guidelines on information and process fusion for analytics algorithms with MapReduce, *Information Fusion* 42 (2018) 51–61. doi:10.1016/j.inffus.2017.10.001.
URL <https://doi.org/10.1016/j.inffus.2017.10.001>
- [21] Apache hadoop.
URL <https://hadoop.apache.org/>
- [22] Laney d. 3d data management: controlling data volume, velocity and variety. meta group; 2001.
- [23] J. Gama, *Learning from data streams : processing techniques in sensor networks*, Springer, Berlin New York, 2007.
- [24] J. a. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, A. Bouchachia, A survey on concept drift adaptation, *ACM Comput. Surv.* 46 (4) (2014) 44:1–44:37.
URL <http://doi.acm.org/10.1145/2523813>
- [25] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, H. Liu, Feature selection: A data perspective, *ACM Computing Surveys (CSUR)* 50 (6) (2017) 94.
- [26] S. Ramírez-Gallego, H. Mouriño-Talín, D. Martínez-Rego, V. Bolón-Canedo, J. M. Benítez, A. Alonso-Betanzos, F. Herrera, An information theory-based feature selection framework for big data under apache spark, *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48 (9) (2018) 1441–1453.
- [27] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical recipes in c*, Cambridge University Press 1 (1988) 3.
- [28] S. Ramírez-Gallego, S. García, H. Mouriño-Talín, D. Martínez-Rego, V. Bolón-Canedo, A. Alonso-Betanzos, J. M. Benítez, F. Herrera, Data discretization: taxonomy and big data challenge, *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 6 (1) (2016) 5–21.
- [29] J. S. Vitter, Random sampling with a reservoir, *ACM Trans. Math. Softw.* 11 (1) (1985) 37–57.
URL <http://doi.acm.org/10.1145/3147.3165>
- [30] D. Wood, V. Informatica, B. C. H, J. V. Leeuwen, J. V. Leeuwen, Interval heaps, *The Computer Journal* 36 (1987) 209–216.
- [31] U. M. Fayyad, K. B. Irani, On the handling of continuous-valued attributes in decision tree generation, *Machine Learning* 8 (1) (1992) 87–102.
URL <https://doi.org/10.1023/A:1022638503176>
- [32] U. Fayyad, K. Irani, Multi-interval discretization of continuous-valued attributes for classification learning.
- [33] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, *Data mining : practical machine learning tools and techniques*, Morgan Kaufmann Publisher, Cambridge, MA, 2017.
- [34] D. A. Zighed, S. Rabaséda, R. Rakotomalala, FUSINTER: A method for discretization of continuous attributes, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06 (03) (1998)

307-326.

URL <https://doi.org/10.1142/s0218488598000264>