I. INTRODUCTION

0. Overview

0.0. Statistical lerning

- (Making some hypotheses before building models)
- Prefering numerical data
- e.g. SVM and instance-based learning (memmory/lazy learning)

0.1. Symbolic learning

- (Rule-based learning)
- Requiring inherent finite values
- Prefering categorical data (unordered)
- e.g. decision trees and rule induction learning -> They require discretized data or have their own discretization process

1. Discretization

1.0. Overview

- One of the most effective data pre-processing technique in Data Mining
- Translating quantitative data into qualitative data
- Procuring a non-overlapping division of a continuous domain
- Ensuring an association between each numerical value and a certain interval

> -> Diminishing data from a large domain of numeric values to a subset of categorical values -> data reduction mechanism

> - **No free lunch**

- Classical data reduction methods are not expected to scale well when managing huge data - both in number of features and instances -> Need distributed version
- Although many state-of-the-art DM algorithms have been implemented in **MLlib (a part of Spark)**, it is **not the case for discretization** algorithms yet.

1.1 Main Objective:

- Presenting a **distributed version** of the **entropy minimization discretizer** using **Apache Spark**, which is based on **Minimum Description Length Principle (MDLP)**
- proving that well-known discretization algorithms as MDLP can be parallelized in these frameworks, providing good discretization solutions for Big Data analytics
- Transforming the iterativity yielded by the original proposal in a single-step computation.

II. BACKGROUND AND PROPERTIES

1. Discretization process

- Assuming a data set S consisting of N examples, M attributes and c class labels, a discretization scheme D_A would exist on the continuous attribute $A \in M$, which partitions this attribute into k discrete and disjoint intervals:

  $$\{[d_0, d_1], (d_1, d_2], \ldots, (d_{k_{A-1}}, d_{k_A}]\},$$

where d_0 and d_{k_A} are, respectively, the minimum and maximal value, and $P_A = \{d_1, d_2, \ldots, d_{k_{A-1}}\}$ represents the set of **cut points** of A in **ascending order**.

> - A typical **discretization process** generally consists of **four steps**:

- **(1) Sorting** the continuous values of the feature to be discretized
- **(2) Evaluating** a cut point for splitting or adjacent intervals for merging
- **(3) Splitting or merging** intervals of continuous values according to some defined criterion
- **(4) Stopping** at some point

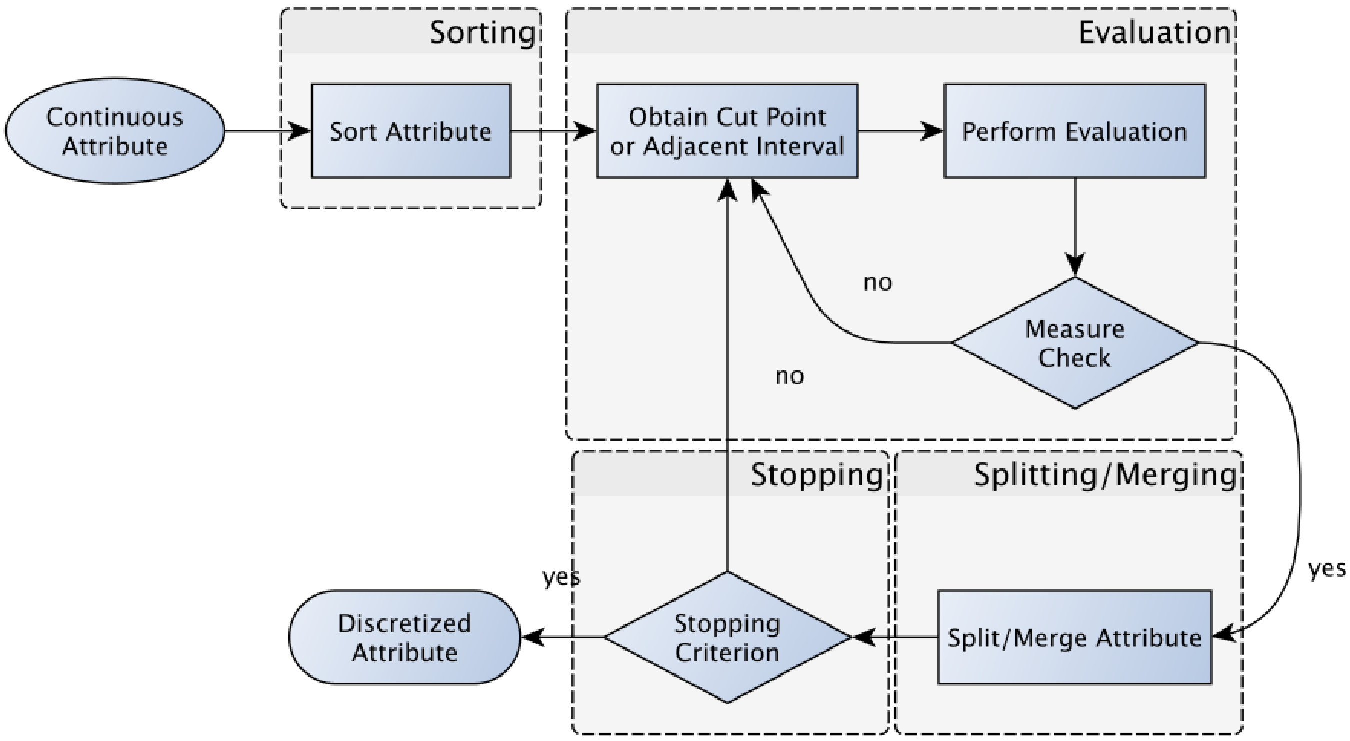

Figure 1: Discretization Process

- **Sorting:** The continuous values for a feature are sorted in either **descending or ascending** order, with a time complexity of **O(NlogN)**
- **Selection of a Cut Point:** After sorting, the **best cut point** or the **best pair of adjacent intervals** should be found in the attribute range in order to **split or merge** in a following required step. An **evaluation measure or function** is used to determine the **correlation, gain, improvement in performance** or **any other benefit** according to **the class label**.
- **Splitting/Merging:** For **splitting**, the **possible cut points** are the **different real values** present in an attribute. For **merging**, the discretizer aims to **find the best adjacent intervals to merge** in **each iteration.**

- **Stopping Criteria:** When to stop the discretization process, **trade-off** between a **final lower number of intervals, good comprehension and consistency.**

2. Discretization Properties

2.1. Static vs. Dynamic:

- **Level of independence** between the **discretizer** and the **learning method**.
- A **static discretizer** is **run prior** to the **learning task** and is **autonomous** from the **learning algorithm**, as a **data pre-processing** algorithm.
- A **dynamic discretizer** responds when the learner requires so, **during the building of the model** -> **embedded in the learner** itself, producing **accurate and compact outcome** together with the **associated learning algorithm.**

2.2. Univariate vs. Multivariate

- **Univariate discretizers** only operate with a **single attribute simultaneously** -> **sort** the attributes **independently**, and then, the **derived discretization** disposal for **each attribute** keeps **unchanged** in the **next phases.**
- **Multivariate**: concurrently consider **all or various attributes**, may accomplish discretization **handling the complex interactions among several attributes** to decide also the attribute in which the **next cut point** will be **splitted or merged.**

2.3. Supervised vs. Unsupervised

- **Supervised** discretizers consider the **class label** whereas **unsupervised** ones do not.
- **Unsupervised discretization** can be applied on both supervised and unsupervised learning
- There is a **growing interest** in **unsupervised discretization** for **descriptive tasks**
- **Unsupervised** also open the door to **transfer the learning between tasks** since the discretization is not tailored to a specific problem.

2.4. Splitting vs. Merging

- **Splitting (Top down):** search for **a cut point** to **divide** the domain into **two intervals** among **all** the possible **boundary points**.
- **Merging (Bottom up):** begin with a **pre-defined partition** and search for a candidate **cut point** to **mix both adjacent intervals after removing it.**
- **Hybrid category:** as the way of **alternating splits with merges** during running time

2.5. Global vs. Local

- consider either **all available information** in the **attribute** or **only partial information.**
- **all the dynamic discretizers** and **some top-down** based methods are **local** (e.g. **MDLP** and **ID3**)

2.6. Direct vs. Incremental

- **Direct discretizer:** the **range associated to an interval** must be **divided into k intervals simultaneously**, requiring an **additional criterion** to **determine** the value of **k.** (e.g. **One-step discretization** methods)

- **Incremental methods: begin** with a **simple discretization** and pass through an **improvement process**, requiring an **additional criterion** to determine when it is the **best moment to stop.**

2.7. Evaluation Measure

- **Information:** Entropy, Gini index, Mutual Information
- **Statistical:** measurement of **dependency/correlation among attributes** ((Zeta, ChiMerge, Chi2), interdependency [40], probability and bayesian properties (MODL), contingency coefficient, etc.
- **Rough Sets:** evaluate the discretization schemes by **using rough set properties and measures**, such as **class separability, lower and upper approximations,** etc.
- **Wrapper: rely on** the **error** provided by a **classifier or a set of classifiers** that are used in each evaluation.
- **Binning:** there is **not an evaluation measure**. This refers to **discretize an attribute** with a **predefined number of bins** in a **simple way**. **A bin** assigns a **certain number of values per attribute** by using a **non sophisticated procedure.** (e.g. **EqualWidth** and **EqualFrequency** are **unsupervised** binning methods)

3. Minimum Description Length-based Discretizer (MDLP)

- Dynamic, univariate, supervised, splitting, local and incremental method
- Using the **Minimum Description Length Principle** to **control the partitioning process.**
- Introducing an **optimization** based on a **reduction of whole set of candidate points**, only formed by the **boundary points** in this set.

Denotation:

- ***A(e)*** : value for attribute ***A*** in the example ***e***.
- ***b*** : A **boundary point *∈ Dom(A)***, which is the **midpoint** between ***A(u)*** and ***A(v)***, assuming that in the **sorted collection** of points in ***A***, there exist two examples ***u, v ∈ S*** with **different class labels**, such that ***A(u) < b < A(v)***; and there does **not exist** other example ***w ∈ S*** such that ***A(u) < A(w) < A(v)***.
- B_A : set of **boundary points** for attribute ***A***
- b_a : a boundary point to evaluate
- $S_1 \subset S$: a subset where $\forall a' \in S1$, $A(a') \leq b_a$, and S_2 be equal to $S - S_1$.
- ***E(S)*** : class entropy in set ***S***
- c , c_1 and c_2: the number of class labels in S , S_1 and S_2 , respectively
- ***N = |S|*** : the number of training examples in set ***S***

MDLP algorithm:

- **Recursively** evaluates **all boundary points**, computing the **class entropy of the partitions** derived as quality measure.
- The **objective is to minimize this measure** to obtain the **best cut decision.** The **class information entropy** yielded by a given **binary partitioning** can be expressed as:

> $EP(A, b_a, S) = \frac{|S1|}{|S|} E(S_1) + \frac{|S2|}{|S|} E(S_2)$

- **Finally**, a **decision criterion** is defined in order to **control** when to **stop the partitioning process**. The use of **MDLP** as a **decision criterion** allows us to **decide whether or not to partition.** Thus a cut point b_a will be applied iff:

> ***$G(A, b_a, S) >$ $\frac{log2(N − 1)}{N}$ + $\frac{Δ(A, ba , S)}{N}$***

where ***$Δ(A, ba, S) = log_2(3^c) − [cE(S) − c_1E(S_1) − c_2E(S_2)]$*** and ***$G(A, b_a, S) = E(S) − EP(A, b_a, S)$***

Important improvements:

- The number of cut points to derive in each iteration.
- A multi-interval extraction of points demonstrating that better classification models - both in error rate and simplicity.

III. TAXONOMY

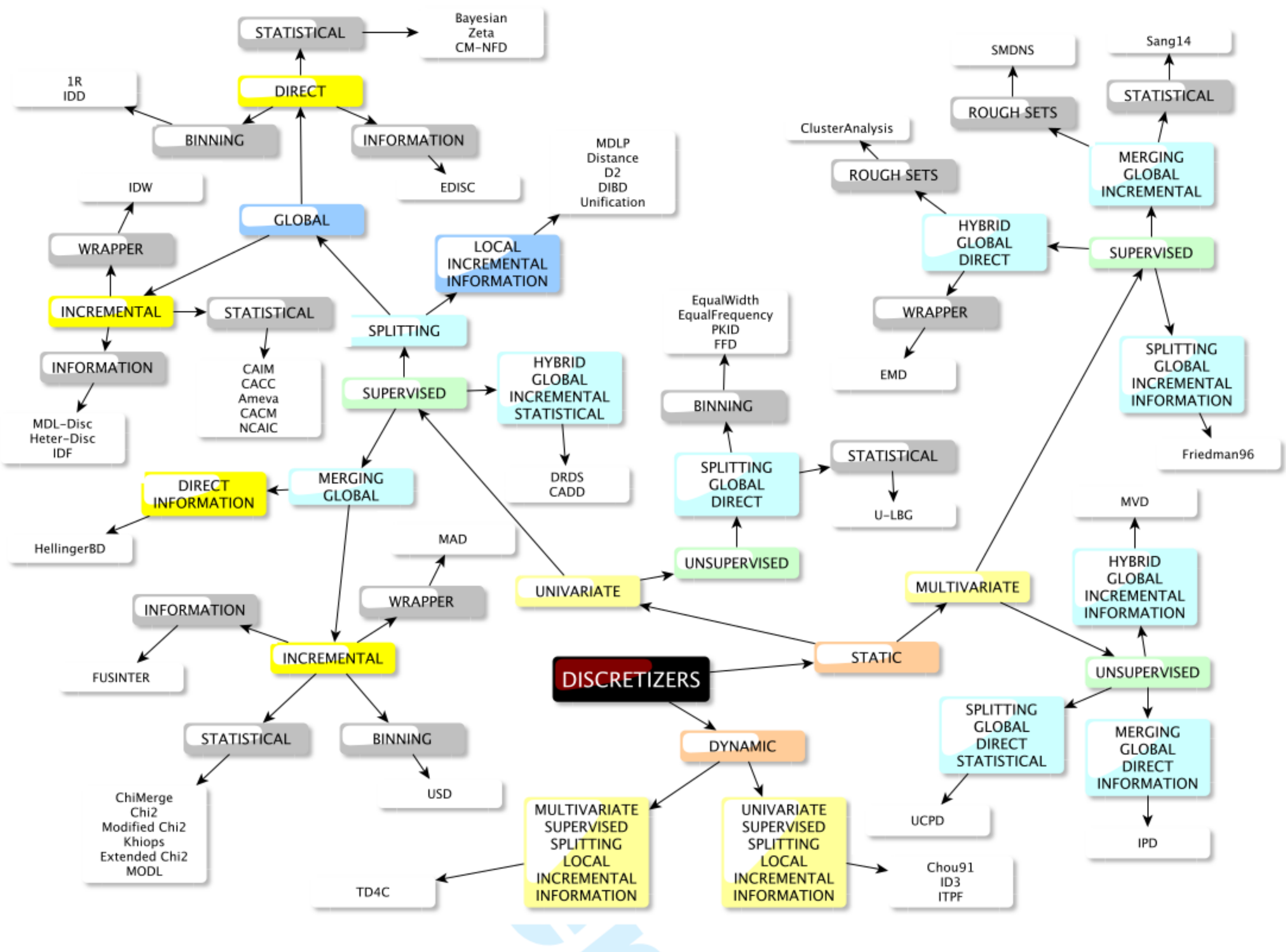

Figure 2: Discretization Taxonomy

- The **purpose of this taxonomy** is **three-fold.**
- **Firstly**, it **identifies the subset** of most representative **state-of-the-art discretizers**
- **Secondly**, it **characterizes the relationships** among techniques, the **extension of the families** and **possible gaps** to be filled in future developments.
- **No relevant methods** in the field of **Big Data**

IV. BIG DATA BACKGROUND

- The **3Vs terms** (extended with **2 additional terms**) that define it as **high volume, velocity and variety information** that require a **new large- scale processing.**
- **Volume:** the massive amount of data that is produced every day is still exponentially growing (from terabytes to exabytes)
- **Velocity:** data needs to be loaded, analyzed and stored as quickly as possible
- **Variety:** data come in many formats and representations
- **Veracity:** the quality of data to process
- **Value:** extracting value from data

MapReduce Model and Other Distributed Frameworks

- Aiming at **processing and generating large-scale datasets**, **automatically processed** in an **extremely distributed fashion** through **several machines**.
- Defining **two primitives** to work with **distributed data**: **Map** and **Reduce**.
- **Map: Splitting** data into t**uples (key/value pairs)** and **distributing** them across **clusters**
- **Reduce: Combining** those conincident **pairs** to form final output
- **Apache Hadoop** -> **Apache Spark** (based on **distributed data structures** called **Resilient Distributed Datasets (RDDs)**: immutable, partitioned collection of elements that can be operated on in parallel)

V. DISTRIBUTED MDLP DISCRETIZATION

- Distribute the complexity of this algorithm across the cluster.
- Two time-consuming operations: 1) Sorting of candidate points $O(|A|\log(|A|))$ complexity (assuming that all points in A are distinct) and 2) Evaluation of these points $O(|B_A|^2)$ -> both are bounded for a single attribute -> Algorithm to **sort and evaluate all points in a single step** -> Using some primitives from Spark's API, such as: **mapPartitions, sortByKey, flatMap and reduceByKey**.

1. Main discretization procedure

Algorithm 1: Main discretization procedure

- Calculates the **minimum-entropy cut points by feature** according to the **MDLP criterion**. It uses a parameter to **limit the maximum number of points** to yield.
- **Step 1**: **Generates tuples** with the value and the index for each feature as key and a class counter as value ***(< (A, A(s)), v >)***.
- **Step 2**: **Reduces tuples** using a function that **aggregates** all **subsequent vectors** with the **same key**, obtaining the **class frequency** for **each distinct value** in the dataset.
- **Step 3: Sorts tuples by key** -> obtains the **list of distinct values** ordered by **feature index and feature value**.
- **Step 4:** Calculates the **first point by partition**
- **Step 5:** Generates the **boundary points**
- **Step 6:** Transforms **boundary points** into **tuples** with **feature index as key** ***(<(att, (point, q)) >)***
- **Step 7: Divides the tuples** in two groups (**big** and **small**) depending on **the number of candidate points by feature** exceeding a **threshold** ***(mc)*** or not and **re-formatted tuples** as: ***(< point, q >)***.

- **Step 8: Evaluates and selects** the **most promising cut points grouped by feature (finally one best cut point per feature)** according to the **MDLP criterion (single-step version)**.
- **Step 9: Joins both sets of cut points** into final result

Algorithm 1 Main discretization procedure

Input: S Data set
Input: M Feature indexes to discretize
Input: mb Maximum number of cut points to select
Input: mc Maximum number of candidates per partition
Output: Cut points by feature

```
comb ←
    map s ∈ S
        v ← zeros(|c|)
        ci ← class_index(v)
        v(ci) ← 1
        for all A ∈ M do
            EMIT < (A, A(s)), v >
        end for
    end map
distinct ← reduce(comb, sum_vectors)
sorted ← sort_by_key(distinct)
first ← first_by_part(sorted)
bds ← get_boundary(sorted, first)
bds ←
    map b ∈ bds
        < (att, point), q >← b
        EMIT < (att, (point, q)) >
    end map
(small, big) ← divide_attributes(bds, mc)
sth ← select_thresholds(small, mb, mc)
for all a ∈ keys(big) do
    bth ← bth+select_thresholds(big(a), mb, mc)
end for
return(union(bth, sth))
```


2. Boundary points selection

Algorithm 2: Function to generate the boundary points (get_boundary)

- Selects those points falling in the class borders, executes an independent function on each partition in order to parallelize the selection process as much as possible so that a subset of tuples is fetched in each thread.
- **Step 1:** For **each instance in a partition**, if the **feature index != index of the previous point** -> emit a **tuple** with the **last point as key** and the **accumulated class counter as value.** If not, checks whether the **current point** is a **boundary w.r.t the previous point** or not -> emits a **tuple** with the **midpoint as** key and the **accumulated counter as value**.
- **Step 2: Repeat the scheme in step 1** for the **last point** in the **current partition** and the **first point** in the **next partition**.
- **Step 3: Joins all tuples** into a new mixed **RDD of boundary points** (***bds***)

Algorithm 2 Function to generate the boundary points *(get_boundary)*

Input: $points$ An RDD of tuples $(<(att, point), q>)$, where att represents the feature index, $point$ the point to consider and q the class counter.
Input: $first$ A vector with all first elements by partition
Output: An RDD of points.

```
boundaries ←
  map partitions part ∈ points
    < (la, lp), lq > ← next(part)
    accq ← lq
    for all < (a, p), q > ∈ part do
      if a <> la then
        EMIT < (la, lp), accq >
        accq ← ()
      else if is_boundary(q, lq) then
        EMIT < (la, (p + lp)/2), accq >
        accq ← ()
      end if
      < (la, lp), lq > ← < (a, p), q >
      accq ← accq + q
    end for
    index ← get_index(part)
    if index < npartitions(points) then
      < (a, p), q > ← first(index + 1)
      if a <> la then
        EMIT < (la, lp), accq >
      else
        EMIT < (la, (p + lp)/2), accq >
      end if
    else
      EMIT < (la, lp), accq >
    end if
  end map
return(boundaries)
```


3. MDLP evaluation

- Features in **each group** are **evaluated differently**
- **Small features** are evaluated in a **single step** where **each feature** corresponds with a **single partition**.
- **Big features** (*less frequent*) are evaluated **iteratively** since **each feature** corresponds with **a complete RDD with several partitions**.
- In **both cases**, the ***select_thresholds*** function is applied to **evaluate and select** the **most relevant cut points by feature**.
- **Small features** -> ***arr_select_ths*** ; **big features** -> ***rdd_select_ths***

3.1. Algorithm 3: Function to select the best cut points for a given feature (select_thresholds)

- **Evaluates and selects** the **most promising cut points grouped by feature** according to the **MDLP criterion (single-step version)**.
- **Step 1: Select** the **best cut point** (minimum entropy <-> maximum entropy gain) satisfying **MDLP** (done by ***arr_select_ths*** and ***rdd_select_ths***)
- **Step 2: Add** this point to ***result*** list and the **current subset** is **divided into two new partitions** using this cut point.
- **Repeats** until **no partition** to evaluate or **the number of selected points** (***mb***) is fulfilled

Algorithm 3 Function to select the best cut points for a given feature *(select_thresholds)*

Input: *cands* A RDD/array of tuples ($< point, q >$), where *point* represents a candidate point to evaluate and q the class counter.
Input: *mb* Maximum number of intervals or bins to select
Input: *mc* Maximum number of candidates to eval in a partition
Output: An array of thresholds for a given feature

$st \leftarrow enqueue(st, (candidates, ()))$
$result \leftarrow ()$
while $|st| > 0$ & $|result| < mb$ **do**
  $(set, lth) \leftarrow dequeue(st)$
  if $|set| > 0$ **then**
    if $type(set) = 'array'$ **then**
      $bd \leftarrow arr\_select\_ths(set, lth)$
    else
      $bd \leftarrow rdd\_select\_ths(set, lth, mc)$
    end if
    if $bd <> ()$ **then**
      $result \leftarrow result + bd$
      $(left, right) \leftarrow divide(set, bd)$
      $st \leftarrow enqueue(st, (left, bd))$
      $st \leftarrow enqueue(st, (right, bd))$
    end if
  end if
end while
$return(sort(result))$

3.2. Algorithm 4: Function to select the best cut point according to MDLP criterion (single-step version) (arr_select_ths)

- **Accumulates frequencies** and then **selects the minimum-entropy candidate** for **small attributes**
- **Step 1**: obtains the **total class counter vector** by **aggregating all candidate vectors**.
- **Step 2**: obtains the **accumulated counters** for the **two partitions** generated by **each point** (done by **aggregating the vectors** from the **most-left point** to the **current one**, and from the **current point** to the **right-most point**) -> ***(< point, q, lq, rq >)***
- **Step 3: Evaluates** the candidates using the ***select_best*** function

Algorithm 4 Function to select the best cut point according to MDLP criterion (single-step version) *(arr_select_ths)*

Input: *cands* An array of tuples ($< point, q >$), where *point* represents a candidate point to evaluate and q the class counter.
Output: The minimum-entropy cut point

$total \leftarrow sum\_freqs(cands)$
$lacc \leftarrow ()$
for $< point, q > \in cands$ **do**
  $lacc \leftarrow lacc + q$
  $freqs \leftarrow freqs + (point, q, lacc, total - lacc)$
end for
$return(select\_best(cands, freqs))$

3.3. Algorithm 5: Function that selects the best cut points according to MDLP criterion (RDD version) (rdd_select_ths)

- Explains the **selection process**; in this case for **"big" features** (more than one partition).
- **Step 1**: **Re-distributes** this RDD into ***npart***
- **Step 2**: Computes the **accumulated counter by partition**.
- **Step 3**: **Aggregates** the **results (by partition)** to obtain the **total accumulated frequency** for the whole subset

- **Step 4**: Computes the **accumulated frequencies by point from both sides Firstly**, the process **accumulates the counter from all previous partitions to the current one**. Then it computes the **accumulated values for each inner point** -> ***(< point, q, lq, rq >)***
- **Step 6: Evaluates** the candidates using the ***select_best*** function

Algorithm 5 Function that selects the best cut points according to MDLP criterion (RDD version) *(rdd_select_ths)*

Input: *cands* An RDD of tuples ($< point, q >$), where *point* represents a candidate point to evaluate and q the class counter.
Input: *mc* Maximum number of candidates to eval in a partition
Output: The minimum-entropy cut point

```
npart ← round(|cands|/mc)
cands ← coalesce(cands, npart)
totalpart ←
  map partitions partition ∈ cands
    return(sum(partition))
  end map
total ← sum(totalpart)
freqs ←
  map partitions partition ∈ cands
    index ← get_index(partition)
    ltotal ← ()
    freqs ← ()
    for i = 0 until index do
      ltotal ← ltotal + totalpart(i)
    end for
    for all < point, q >∈ partition do
      freqs ← freqs + (point, q, ltotal + q, total − ltotal)
    end for
    return(freqs)
  end map
return(select_best(cands, freqs))
```

3.4. Algorithm 6: Function that calculates class entropy values and selects the minimum entropy cut point (select_best)

- **Evaluates** the **discretization schemes** yielded by **each point** by **computing the entropy for each partition generated**, also taking into account the **MDLP criterion**.
- From the **set of accepted points**, the algorithm **selects** the **one with the minimum class information entropy.**

Algorithm 6 Function that calculates class entropy values and selects the minimum-entropy cut point *(select_best)*

Input: $freqs$ An array/RDD of tuples ($< point, q, lq, rq >$), where point represents a candidate point to evaluate, leftq the left accumulated frequency, rightq the right accumulated frequency and q the class frequency counter.
Input: $total$ Class frequency counter for all the elements
Output: The minimum-entropy cut point

$n \leftarrow sum(total)$
$totalent \leftarrow ent(total, n)$
$k \leftarrow |total|$
$accp \leftarrow$
map $< point, q, lq, rq > \in freqs$
  $k1 \leftarrow |lq|; k2 \leftarrow |rq|$
  $s1 \leftarrow sum(lq); s2 \leftarrow sum(rq);$
  $ent1 \leftarrow ent(s1, k1); ent2 \leftarrow ent(s2, k2)$
  $partent \leftarrow (s1 * ent1 + s2 * ent2)/s$
  $gain \leftarrow totalent - partent$
  $delta \leftarrow log_2(3^k - 2) - (k * hs - k1 * ent1 - k2 * ent2)$
  $accepted \leftarrow gain > ((log_2(s-1)) + delta)/n$
  if $accepted = true$ **then**
    $EMIT < partent, point >$
  end if
end map
$return(min(accp))$

VI. EXPERIMENTAL FRAMEWORK AND ANALYSIS

Hardware

- A cluster composed of twenty computing nodes and one master node, each node has: 2 processorsx Intel Xeon CPU E5-2620, 6 cores per processor, 2.00 GHz, 15 MB cache, QDR InfiniBand Network (40 Gbps), 2 TB HDD, 64 GB RAM.

Software

- Hadoop 2.5.0-cdh5.3.1 from Cloudera's open-source Apache Hadoop distribution5, Apache Spark and MLlib 1.2.0, 480 cores (24 cores/node), 1040 RAM GB (52 GB/node).

Data set

Table 2 Summary description for classification datasets

Data Set	#Train Ex.	#Test Ex.	#Atts.	#Cl.
epsilon	400 000	100 000	2000	2
ECBDL14 (ROS)	65 003 913	2 897 917	631	2

- **ECBDL14** is a **binary classification** problem where the **class distribution** is **highly imbalanced**: **2% of positive instances**

Algorithms

- **Handling imbalanced data distribution**: **MapReduce version** of the **Random OverSampling (ROS)** with **sampling_stratery = 1**

- **Classifier**: **Naive Bayes** (**multinomial** version in **MLlib** with ***λ = 1***)

Evaluation Metrics

- **Classification accuracy** -> it is **not a proper metric** because of **imbalanced distribution** -> it should be **F1-measure** if 2 classes are **equally important** or **F2-measure** when **positive class is more costly**

Table 3 Classification accuracy values

Dataset	**NB** *Train*	**NB** *Test*	**NB-disc** *Train*	**NB-disc** *Test*
ECBDL14	0.5260	0.6276	**0.6659**	**0.7260**
epsilon	0.6542	0.6550	**0.7094**	**0.7065**

- **Classification time** (NB vs. NB disc) and **discretization time** (sequential vs. distributed) (in seconds)

Table 4 Classification and discretization time values: with vs. w/o discretization -first two columns-, and sequential vs. distributed -last two columns- (in seconds)

Dataset	**NB**	**NB-disc**	**Sequential**	**Distributed**
ECBDL14	31.06	**26.39**	295 508	**1 087**
epsilon	5.72	**4.99**	5 764	**476**