Coreset-based Data Compression for Logistic Regression

Nery Riquelme-Granada[1], Khuong An Nguyen[2], and Zhiyuan Luo[1]

[1] Royal Holloway University of London, Surrey TW20 0EX, United Kingdom
[2] University of Brighton, East Sussex BN2 4AT, United Kingdom
{Nery.RiquelmeGranada,Zhiyuan.Luo}@rhul.ac.uk
K.A.Nguyen@brighton.ac.uk

Abstract. The coreset paradigm is a fundamental tool for analysing complex and large datasets. Although coresets are used as an acceleration technique for many learning problems, the algorithms used for constructing them may become computationally exhaustive in some settings. We show that this can easily happen when computing coresets for learning a logistic regression classifier. We overcome this issue with two methods: **A**ccelerating **C**lustering **v**ia **S**ampling (ACvS) and **R**egressed **D**ata **S**ummarisation **F**ramework (RDSF); the former is an acceleration procedure based on a simple theoretical observation on using Uniform Random Sampling for clustering problems, the latter is a coreset-based data-summarising framework that builds on ACvS and extend it by using a regression algorithm as part of the construction. We tested both procedures on five public datasets, and observed that computing the coreset and learning from it, is 11 times faster than learning directly from the full input data in the worst case, and 34 times faster in the best case. We further observed that the best regression algorithm for creating summaries of data using the RDSF framework is the Ordinary Least Squares (OLS).

Keywords: Coresets · Data Compression · Logistic Regression.

1 Introduction

Data-compression techniques are a valuable set of tools for allowing learning algorithms to scale with large datasets. These techniques deviate from the classic algorithmic approach where one needs to write new algorithms in order to improve the running time of the old ones. One then expects these newer versions to converge faster. On the other hand, compression techniques allow the use of the existing, potentially inefficient, algorithms on reduced versions of their input data.

In a nutshell, given some input data $\mathcal{D}$ and a learning algorithm $\mathcal{A}$, a typical compression algorithm would attempt to reduce $\mathcal{D}$ to a much smaller, more manageable, dataset $\mathcal{S}$; once this compressed dataset has been obtained, the original one can be discarded and $\mathcal{A}$ can be run on $\mathcal{S}$ as many times as necessary.

Since $\mathcal{S}$ is much smaller than $\mathcal{D}$, the learning process is accelerated and hence the computational burden associated with $\mathcal{A}$ can be better controlled.

Coresets (or core-sets) [10] are a representative framework of the data-compression family of algorithms, a powerful data-compression paradigm that *provably correctly* approximates the input data $\mathcal{D}$ by constructing a small coreset $\mathcal{C}$, with $|\mathcal{C}| \ll |\mathcal{D}|$. Although there are other interesting compression techniques such as Sketches [15] and Uniform Random Sampling (URS) [4], coresets are very attractive since (i) as opposed to URS, they keep the learning loss bounded, and (ii) as opposed to Sketches, coresets reside in input space *i.e.* the same space where the input data lives. Furthermore, there are well-established approaches for converting, with little effort, any batch algorithm to construct coresets into an online one (See [6], Section 7).

This paper expands the line of work started in [17], where the algorithm for constructing coresets for the problem of Logistic Regression (LR) was shown to produce a bottleneck, which in turn caused undesirable extra computational effort in the process of computing the coreset. Hence, some procedures were necessary in order to guarantee the fast compression of the input data. Specifically, in this work we look in-depth at the *Accelerating Clustering via Sampling* (ACvS) and *Regressed Data Summarisation Framework* (RDSF) ideas proposed in [17]. The former is a straight-forward procedure that accelerates the computation of coresets for the problem of Logistic Regression without sacrificing any performance; the latter, is a coreset-based framework that uses machine learning to predict what the most *important* portion of the training set is, and constructs a data compression using its predictions. Our contributions are summarised as follows:

- We present and explain the ACvS procedure, proposed in [17], as a simple and effective acceleration technique for reducing the computational overheads that constructing coresets may cause for the problem of Logistic Regression.
- Following the work of [17], we re-visit the RDSF method as a coreset-based compression technique that benefits from a regression algorithm to learn how important the input points are with respect to the LR problem. We also show how RDSF enjoys, following the same acceleration principles involved in ACvS, a fast running time.
- We expand the empirical study presented in [17] in two major ways: (i) we consider two new datasets, ijcnn1 and w8a, and show that ACvS and RDSF efficiently produce good summaries of data; (ii) we present a new set of experiments where different regressors are used as part of the RDSF method. Specifically, we consider the *Ordinary Least Squares* (OLS), *Ridge*, *Lasso* and *Elastic Net* regressors.

The rest of the paper is organised as follows. Section 2 provides the fundamental definitions and discussions on coresets. Section 3 presents an exposition of the ideas originally proposed in [17]: ACvS and RDSF. Section 4 shows our empirical evaluations and the results obtained by using both ideas. Finally, Section 5 offers our conclusions and future work.

2 Coresets and Learning

In this section, we consider the problem of efficiently learning a binary LR classifier over a small data summary obtained from the input data. To approach this, we first introduce coresets and their construction. Then, we discuss the well-known Logistic Regression classifier and the state-of-the-art coreset approach for this particular learning problem called "Coreset Algorithm for Bayesian Logistic Regression" (CABLR) [13]. Finally, we review the computational bottleneck that seems inevitable when using the coreset construction algorithm CABLR for LR.

2.1 Coresets and Their Construction

The framework of coresets is a well-established approach used to reduce both the volume and dimensionality of large and complex datasets. A coreset is a small set of points that approximates a much bigger set of points with respect to a specific function. Formally, let function f be the objective function of some learning problem and let $\mathcal{D}$ be the input data. The set $\mathcal{C}$ is said to be an ϵ-coreset for $\mathcal{D}$ with respect to f if the following condition holds [10]:

$$|f(\mathcal{D}) - f(\mathcal{C})| \leq \epsilon f(\mathcal{D}) \tag{1}$$

where $\epsilon \in (0, 1)$ accounts for the error incurred for evaluating f over $\mathcal{C}$ [3]. This expression establishes the main error bound offered by coresets. Hence, we can potentially suffer some loss when using the coreset; but this loss is bounded and controlled via the ϵ error parameter. Notice that it is expected that $|\mathcal{C}| \ll |\mathcal{D}|$ holds.

The natural question to ask at this point is how to construct such a set $\mathcal{C}$ for our input data so that we can enjoy the kind of guarantee in formula (1). There are well-known techniques for constructing coresets and they can be summarised as follows:

- **geometric decomposition [2]:** this is a fundamental approach for constructing coresets, because the first coreset constructions followed this line of thinking; it relies on discretising the input space of points into cells or grids, and then *snapping* representative points from each grid. This approach has been extensively used in the analysis of *shape fitting* problems such as the *Minimum Enclosing Ball* (MEB) [5].
- **random sampling [9]:** this is a more recent result and currently is one of the most successful approaches for constructing coresets. The idea is to compute a probability distribution that, in a well-defined sense, reflects the importance of each input point with respect to function f. Then, one samples the points following the importance distribution and assigns *weights* to each sampled point. Thus, the coreset in this case is a *weighted subset* of the original input data.

[3] instead of using the full input data $\mathcal{D}$

– **gradient descent [16]:** this line of work consists in using results from convex optimisation to reduce the coreset construction process to an special case of the popular gradient descent optimiser. Hence, the coreset is constructed iteratively and often implicitly as part of the optimisation of the function f.
– **projection-based methods [15]:** the most notable projection-based method is that of *sketches*: the idea is to perform projections to sub-spaces of the input space in order to have lower-dimensional points. Hence, coresets constructed following this approach may have the same number of points as the original input data, but because the coreset resides in a much lower dimension *i.e.* the number of dimensions in the coreset is strictly less than the number of dimensions in the original input data, learning-related computations are faster. This set of techniques is largely inspired by the Johnson-Lindenstrauss lemma [7], a result that states that the input data can be embedded in a much lower-dimensional space such that the distances among the points are approximately preserved.

No matter what approach we choose to construct the coreset $\mathcal{C}$, we need a precise definition of f. In machine learning, f is defined as the objective function for the learning problem we are trying to solve. In our case, we want to compute a coreset that allows us to learn a logistic regression classifier with small loss. In the next section we formally define this fundamental learning problem; then, we present a powerful method that falls into the random-sampling family of algorithms to compute coresets.

2.2 Logistic Regression and Sensitivity Framework

Logistic Regression is a well-known statistical method for binary classification problems. Given an input data $\mathcal{D} := \{(x_n, y_n)\}_{n=1}^N$, where x_n is d-dimensional feature vector and $y_n \in \{-1, 1\}$ is its corresponding label, the likelihood of observing $y_n = 1$ for x_n and some parameters $\theta \in \mathbb{R}^{d+1}$ can be defined as $[p_{logistic}(y_n = 1|x_n; \theta) := 1/(1 + \exp(-x'_n \cdot \theta))]$, where $\theta \in \mathbb{R}^{d+1}$ and $x'_n \in \mathbb{R}^{d+1}$ which is x_n with an additional column of 1.

Similarly, we have

$$\begin{aligned} p_{logistic}(y_n = -1|x_n; \theta) &:= 1 - \frac{1}{(1 + \exp(-x'_n \cdot \theta))} \\ &= \frac{\exp(-x'_n \cdot \theta)}{(1 + \exp(-x'_n \cdot \theta))} = \frac{1}{(1 + \exp(x'_n \cdot \theta))}. \end{aligned} \quad (2)$$

Therefore, for any y_n $p_{logistic}(y_n|x_n; \theta) := 1/(1 + \exp(-y_n x'_n \cdot \theta))$.

Because the input data $\mathcal{D}$ is assumed to be independent and identically distributed, the log-likelihood function $LL_N(\theta|\mathcal{D})$ can be defined as in [20]:

$$LL_N(\theta|\mathcal{D}) := \sum_{n=1}^{N} \ln p_{logistic}(y_n|x_n; \theta) = -\sum_{n=1}^{N} \ln(1 + \exp(-y_n x_n \cdot \theta)) \quad (3)$$

which is the objective function for the LR problem. The optimal parameter $\hat{\theta}$ can be obtained using maximum likelihood estimation. Maximising $LL_N(\theta|\mathcal{D})$ is equalivent to minimising $\mathcal{L}_N(\theta|\mathcal{D}) := \sum_{n=1}^{N} \ln(1 + \exp(-y_n x_n \cdot \theta))$ over all $\theta \in \mathbb{R}^{d+1}$. Finally, the optimisation problem can be defined as:

$$\hat{\theta} := \operatorname*{arg\,min}_{\theta \in \mathbb{R}^{d+1}} \mathcal{L}_N(\theta|\mathcal{D}), \tag{4}$$

where $\hat{\theta}$ is the best solution found. Once we have solved Equation (4), we use the estimated $\hat{\theta}$ to make prediction for any unseen data point.

A Bayesian approach to Logistic Regression allows us to specify a prior distribution for the unknown parameter θ, $p(\theta)$ based on our real-life domain knowledge and derive the posterior distribution $p(\theta|\mathcal{D})$ for a given data $\mathcal{D}$ by applying Bayes' theorem:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})} = \frac{p(\mathcal{D}|\theta)p(\theta)}{\int p(\mathcal{D}|\theta)p(\theta)d\theta}.$$

Exact Bayesian inference for Logistic Regression is intractable. Therefore, no closed-form maximum likelihood solution can be found for determining θ and approximation methods are typically used to find the solution.

The state-of-the-art coreset construction for LR problem, namely *Coreset Algorithm for Bayesian Logistic Regression* (CABLR) was proposed by Huggins *et al.* [13]. Designed for the Bayesian setting, this algorithm uses random sampling for constructing coresets that approximate the log-likelihood function on the input data $\mathcal{D}$.

Huggins *et al.* followed a well-established framework know as the *sensitivity framework* [9] for constructing coresets for different instances of clustering problems such as K-means and K-median. The main idea is to formulate coreset construction as the problem of finding an ϵ-approximation [14], which can be computed using non-uniform sampling based on the importance of each data point, in some well-defined sense [4]. Hence, each point in the input data is assigned an importance score, *a.k.a.* the sensitivity of the point. An approximation to the optimal clustering of the input data is required in order to calculate such importance scores. For each point, the sensitivity score is computed by taking into account the distance between the point and its nearest (sub-optimal) cluster centre obtained from the approximation. The next step is to sample M points from the distribution defined by the sensitivity scores, where M is the size of the coreset. Finally, each of the M points in the coreset is assigned a positive real-valued *weight* which is the inverse of the point's sensitivity score. The sensitivity framework returns a coreset consisting of M weighted points. The theoretical proofs and details can be found in [9].

However, careless use of this algorithm in the optimisation setting, as we shall see in the next section, may be devastating as computing the clustering of

[4] for our discussion, it is enough to state that the sensitivity score is a real value in the half-open interval $[0, \infty)$

the input data, even for the minimum number of iteration, can be too expensive. The description of CABLR is shown in Algorithm 1 where k number of cluster centres $\mathcal{Q}$, from the input data are used to compute the sensitivities (lines 2-4); then, sensitivities are normalised and points get sampled (line 5); finally, the weights which are inverse proportional to the sensitivities, are computed for each of the sampled points (lines 6-12). Thus, even though the obtained coreset is for LR, CABLR still needs a clustering of the input data as it is common for any coreset algorithms designed using the sensitivity framework.

```
   Input: $\mathcal{D}$: input data, $Q_{\mathcal{D}}$: $k$-clustering of $\mathcal{D}$ with $|Q_{\mathcal{D}}| := k$, $M$: coreset size
   Output: $\epsilon$-coreset $\mathcal{C}$ with $|\mathcal{C}| = M$
 1 initialise;
 2 for $n = 1, 2, ..., N$ do
 3     $m_n \leftarrow$ Sensitivity($N, Q_{\mathcal{D}}$) ;     // Compute the sensitivity of each point
 4 end
 5 $\bar{m}_N \leftarrow \frac{1}{N}\sum_{n=1}^{N} m_n$ ;
 6 for $n = 1, 2, ..., N$ do
 7     $p_n = \frac{m_n}{N\bar{m}_N}$ ;     // compute importance weight for each point
 8 end
 9 $(K_1, K_2, ..., K_N) \sim$ Multi($M$,$(p_n)_{n=1}^{N}$) ;     // sample coreset points
10 for $n = 1, 2, ..., N$ do
11     $w_n \leftarrow \frac{K_n}{p_n M}$ ;     // calculate the weight for each coreset point
12 end
13 $\mathcal{C} \leftarrow \{(w_n, x_n, y_n) | w_n > 0\}$;
14 return $\mathcal{C}$
```

Algorithm 1: CABLR ([13]): an algorithm to construct coresets for Logistic Regression.

Remark 1. In the description of Algorithm 1, we hide the coreset dependence on the error parameter ϵ, defined in Section 2.1. There is a good reason for doing this. When theoretically designing a coreset algorithm for some fixed problem, there are two error parameters involved: $\epsilon \in [0, 1]$, the "loss" incurred by coresets, and $\delta \in (0, 1)$, the probability that the algorithm will fail to construct a coreset. Then, it is necessary to define the minimum coreset size M in terms of these error parameters. The norm is to prove there exists a function $t : [0, 1] \times (0, 1) \rightarrow \mathcal{Z}^+$, with $\mathcal{Z}^+$ being the set of all positive integers, that gives the corresponding coreset size for all possible error values *i.e.* $t(\epsilon_1, \delta_1) := M_1$ implies that M_1 is the minimum number of points needed in the coreset for achieving, with probability $1 - \delta$, the guarantee, defined in inequality (1), for ϵ_1. However, in practice, one does not worry about explicitly giving the error parameters as inputs; since each coreset algorithm comes with its own definition of t, one only needs to give the desired coreset size M and the error parameters can be computed using t. Finally,

t defines a fundamental trade-off for coresets: the smaller the error parameters, the bigger the resulting coreset size *i.e.* smaller coresets may potentially lose more information than bigger coresets.[5]

2.3 Challenges and Problems

Clustering on large input data is computationally hard [3]. This is why approximation and data reduction techniques are popular choices for accelerating existing clustering algorithms. In fact, the paradigm of coresets has seen great success in the task of approximating solutions for clustering problems (see [10], [12], [4], [22] and [1]). The sensitivity framework, originally proposed for constructing coresets for clustering problems, requires a sub-optimal clustering of the input data $\mathcal{D}$ in order to compute the sensitivity for each point in $\mathcal{D}$. This requirement transfers to CABLR, described in the previous section. If we assume a Bayesian setting as in[13], the time necessary for clustering $\mathcal{D}$ is dominated by the cost of the posterior inference algorithms (see [13]). However, if we remove the burden of posterior inference and consider the optimisation setting, then the situation is very different.

Figure 1 sheds some light on the time spent on finding a clustering compared to all the other steps taken by CABLR to construct a coreset, namely: *sensitivity computation* and *sampling*. The time spent on *learning* from the coreset is included as well.

Evidently, applying a clustering algorithm, even to get a sub-optimal solution as done here, can be dangerously impractical for LR in the optimisation setting, as it severely increases the overall coreset-construction time. Even worse, constructing the coreset is slower than learning directly from $\mathcal{D}$, defeating the purpose of using the coreset as an acceleration technique.

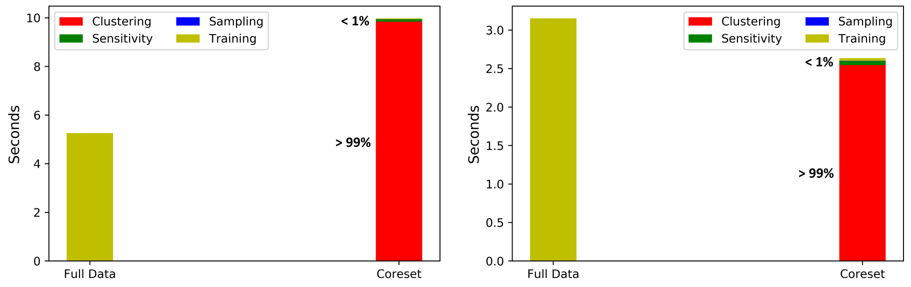
Another interesting issue can be seen at display here: the algorithm used for computing coresets must give us summaries of data that are both quickly-computable and quality-preserving. Disregarding one of these objectives makes our task much easier; that is, we can simply take an uniform random sampling from our input data; no compression algorithm will finish faster, but we will suffer a quality loss that is out of our control. Similarly, we can just avoid performing any data compression/reduction at all, and we will for sure be able to obtain a very high-quality solution; but we should be ready to deal with lots of computational stress, storage overflows and even loss of random access[6].

With coresets for LR, we propose the following research question: *'Can we still benefit from good coreset acceleration in the optimisation setting?'*

[5] For CABLR, Huggins *et al.* proved that $t(\epsilon, \delta) := \lceil \frac{c\bar{m}_N}{\epsilon^2}[(D+1)log\,\bar{m}_N + log(\frac{1}{\delta})]\rceil$, where D is the number of features in the input data, $\bar{m}_N$ is the average sensitivity of the input data and c is a constant. The mentioned trade-off can be appreciated in the definition of t.

[6] we loose random access when the input data is so large that accessing some parts of the data is more expensive, computationally speaking, than accessing other parts.

We give an affirmative answer to this question through two approaches described in the next section. The key ingredient for both methods is the use of *Uniform Random Sampling* alongside coresets.



(a) Webspam (b) Covertype

Fig. 1: The bottleneck induced by the clustering process when constructing coresets, as presented in [17].

3 Two Accelerating Procedures

In this section, we propose two different procedures for efficiently computing coresets for Logistic Regression in the optimisation setting. Both approaches are similar in the following sense: they both make use of Uniform Random Sampling (URS) for speeding up the coresets computation. URS is the most straightforward and simple way to reduce the size of input dataset by picking as many points as permissible uniformly at random. This is not the first time that the concept of URS comes up alongside coresets; in fact, URS can be seen as a naive approach for computing coresets and it is the main motivation for deriving a more sophisticated sampling approach [4]. In our procedures, however, we see URS as a complement to coresets, not as an alternative to them, which is usually the case in coresets works.

3.1 Accelerating Clustering via Sampling

Our first procedure, **A**ccelerated **C**lustering **v**ia **S**ampling (ACvS), uses a straightforward application of URS. The procedure is described in Algorithm 2. First, we extract b input points from $\mathcal{D}$ and put them into a new set S. We require that $b \ll N$, where $N := |\mathcal{D}|$. Then, we cluster S to obtain k cluster centres, namely Q_S with $|Q_S| := k$. We finally run the CABLR algorithm using Q_S as input and compute a coreset. Since we have that $|S| \ll |\mathcal{D}|$, obtaining Q_S is substantially faster than obtaining $Q_{\mathcal{D}}$. Notice that the coreset algorithm CABLR is parameterised by the coreset size M. As a simple example, suppose we have a large

```
    Input: CABLR: coreset algorithm, $\mathcal{D}$: input data, $A$: a clustering algortithm,
           $k$: number of cluster centres, $b \ll |\mathcal{D}|$: uniform random sample size
    Output: $\mathcal{C}$: coreset
1   $S \leftarrow \emptyset$;
2   $B \leftarrow |S|$;
3   while $B < b$ do
4       $s \leftarrow$ SamplePoint($\mathcal{D}$) // Sample without replacement
5       $S \leftarrow S \cup \{s\}$ // Put $s$ in $S$
6   end
7   $Q_S \leftarrow A(S,k)$ // Run Clustering algorithm on $S$
8   $\mathcal{C} \leftarrow \mathrm{CABLR}_M(\mathcal{D}, Q_S)$ // Run Coreset Algorithm
9   return $\mathcal{C}$
```

Algorithm 2: ACvS procedure, as defined in [17].

dataset that we would like to classify using logistic regression, and to do it quite efficiently, we decide to compress the input data into a coreset. The standard coreset algorithm for LR dictates that we have to find a clustering of our dataset as the very first step. Then we use the clustering to compute the sensitivity of each point. The next step is to sample points according to their sensitivity and to put them in the coreset. Finally, we compute the weight for each point in the coreset. What ACvS proposes is: instead of computing the clustering of our dataset, compute the clustering of a very small set of uniform random samples of it, then proceed as the standard coreset algorithm dictates.

We design this procedure based on the following observation: uniform random sampling can provide unbiased estimation for many cost functions that are additively decomposable into non-negative functions. We prove this fact below.

Let $\mathcal{D}$ be a set of points and let $n := |\mathcal{D}|$; also, let Q be a *query* whose cost value we are interested in computing. We can define a cost function which is decomposable into non-negative functions as $cost(\mathcal{D}, Q) := \frac{1}{n}\sum_{x\in\mathcal{D}} f_Q(x)$.

Most machine learning algorithms can be cast to this form. Here, we are mainly concerned about k-means clustering, where Q is a set of k points and $f_Q(x) := min_{q\in Q}||x-q||_2^2$. Let us now take a random uniform sample $S \subset \mathcal{D}$ with $m := |S|$, and define the cost of query Q with respect to S as $cost(S, Q) := \frac{1}{m}\sum_{x\in S} f_Q(x)$. To show that $cost(S, Q)$ is an unbiased estimator of $cost(\mathcal{D}, Q)$ we need to prove that $\mathbb{E}[cost(S, Q)] = cost(\mathcal{D}, Q)$

Claim. $\mathbb{E}[cost(S, Q)] = cost(\mathcal{D}, Q)$

Proof. By definition, we have that $cost(S, Q) := \frac{1}{m}\sum_{x\in S} f_Q(x)$. Expanding this, we get

$$\mathbb{E}[cost(S, Q)] = \mathbb{E}[\frac{1}{m}\sum_{x\in S} f_Q(x)] \tag{5}$$

The crucial step now is to compute the above expectation. To do this, it is useful to construct the set $\mathbb{S}$ that contains all the possible subsets S in $\mathcal{D}$.

The number of such subsets has to be $\binom{n}{m}$, which implies $|\mathbb{S}| := \binom{n}{m}$. Then, computing the expectation over $\mathcal{S}$ and re-arranging some terms we get

$$\frac{1}{\binom{n}{m}} \frac{1}{m} \sum_{S \in \mathbb{S}} \sum_{x \in S} f_Q(x) \tag{6}$$

Next, we get rid of the double summation as follows: we count the number of times that $f_Q(x)$ is computed. By disregarding overlapping computation of $f_Q(x)$ due to the fact that a point x may belong to multiple subsets $S \in \mathbb{S}$, we quickly obtain that the count has to be $\binom{(n-1)}{(m-1)}$. Then, we write (6) as

$$\frac{1}{\binom{n}{m}} \frac{1}{m} \binom{(n-1)}{(m-1)} \sum_{x \in \mathcal{D}} f_Q(x) \tag{7}$$

Notice that now we have a summation that goes over our original set $\mathcal{D}$. Finally, by simplifying the factors on the left of the summation we have

$$\frac{1}{n} \sum_{x \in \mathcal{D}} f_Q(x) = cost(\mathcal{D}, Q) \tag{8}$$

which concludes the proof.

The imminent research question here is: how does using the ACvS procedure affect the performance of the resulting coreset? We shall show in Section 4 that the answer is more benign than we originally thought.

3.2 Regressed Data Summarisation Framework

Our second method builds on the previous one and it gives us at least two important benefits on top of the acceleration benefits given by ACvS:

- *sensitivity interpretability:* it unveils an existing (not-obvious) linear relationship between input points and their sensitivity scores.
- *instant sensitivity-assignment capability:* apart from giving us a coreset, it gives as a trained regressor capable of assigning sensitivity scores *instantly* to new unseen points.

Before presenting the method, however, it is useful to remember the following: the CABLR algorithm (Algorithm 1) implements the sensitivity framework, explained in Section 2.2, and hence it relies on computing the sensitivity (importance) of each of the input points.

We call our second procedure *Regressed Data Summarisation Framework* (RDSF). We can use this framework to (i) accelerate a sensitivity-based coreset algorithm; (ii) unveil information on how data points relate to their sensitivity scores; (iii) obtain a regression model that can potentially assign sensitivity scores to new data points.

The full procedure is shown in Algorithm 3: RDSF starts by using ACvS to accelerate the clustering phase. The next step is to separate the the input data

```
Input: $\mathcal{D}$: input data, $A$: clustering algortithm, $k$: number of cluster centres,
        $b \ll |\mathcal{D}|$: uniform random sample size, $M$: coreset size
Output: $\tilde{\mathcal{C}}$: Summarised Version of $\mathcal{D}$, $\phi$: Trained Regressor
1  initialise;
2  $S \leftarrow \emptyset$;
3  $B \leftarrow |S|$;
4  $N \leftarrow |\mathcal{D}|$;
5  while $B < b$ do
6      $s \leftarrow$ SamplePoint($\mathcal{D}$) // Sample without replacement
7      $S \leftarrow S \cup \{s\}$ // Put $s$ in $S$
8  end
9  $Q_S \leftarrow A(S,k)$ // Run Clustering algorithm on $S$
10 $R \leftarrow \mathcal{D} \setminus S$;
11 $Y \leftarrow \emptyset$;
12 for $n = 1, 2, ..., b$ do
13     $m_n \leftarrow$ Sensitivity($b, Q_S$) // Compute the sensitivity of each point
           $s \in S$
14     $Y \leftarrow Y \cup \{m_n\}$;
15 end
16 $\hat{Y}, \phi \leftarrow$ PredictSen($S, Y, R$) // Train regressor on $S$ and $Y$, predict
       sensitivity for each $r \in R$
17 $Y \leftarrow Y \cup \hat{Y}$;
18 $\bar{m}_N \leftarrow \frac{1}{N} \sum_{y \in Y} y$;
19 for $n = 1, 2, ..., N$ do
20     $p_n = \frac{m_n}{N \bar{m}_N}$ ;            // compute importance weight for each point
21 end
22 $(K_1, K_2, ..., K_N) \sim$ Multi($M$,$(p_n)_{n=1}^N$) ;          // sample coreset points
23 for $n = 1, 2, ..., N$ do
24     $w_n \leftarrow \frac{K_n}{p_n M}$ ;         // calculate the weight for each coreset point
25 end
26 $\tilde{\mathcal{C}} \leftarrow \{(w_n, x_n, y_n) | w_n > 0\}$;
27 return $\tilde{\mathcal{C}}$,$\phi$
```

Algorithm 3: The Regressed Data Summarisation Framework ([17]) uses a coreset construction to produce coreset-based summaries of data.

$\mathcal{D}$ in two sets: S, the small URS picked during ACvS, and R, all the points in $\mathcal{D}$ that are not in S. The main step in RDSF starts at line 12: using the clustering obtained in the ACvS phase, Q_S, we compute the sensitivity scores *only* for the points in S and place them in a predefined set Y. A linear regression problem is then solved using the points in S as feature vectors and their corresponding sensitivity scores in Y as targets. This is how RDSF sees the problem of summarising data as the problem of 'learning' the sensitivity of the input points. The result of such learning process is a trained regressor ϕ and RDSF uses it to predict the sensitivities of all the points in R. Hence, RDSF uses S as training set and R as test set.

We finally see that after merging the computed and predicted sensitivities of S and R (line 16 in Algorithm 3), respectively, RDSF executes the same steps as CABLR *i.e.* compute the mean sensitivity (line 19), sample the points that will be in the summary (line 22) and compute the weights (line 23).

4 Evaluations

In this section, we show our evaluation results. Similar to [17], [18] and [19], we rigorously test coresets and coresets-based methods by applying a set of metrics which are standard in machine learning. This work hence puts considerable emphasis on investigating coresets from an empirical standpoint, a perspective that still remains largely unexplored in the coreset community.

4.1 Strategy

We tested our procedures on 5 datasets, shown in Table 1, which are publicly available [7] and are well-known in the coreset community.

Table 1: Overview of the datasets considered for evaluation of our procedures.

Dataset	Examples	Features
ijcnn1	141,691	22
Webspam	350,000	254
Covertype	581,012	54
Higgs	11,000,000	28
w8a	64,700	300

All of our experiments are based on the following five procedures:

– **Full:** no coreset or summarisation technique is used. That is, we simply train a LR model on the entire training set, then predict the labels for the instances in the test set.
– **CABLR:** we obtain a clustering of the input data and run CABLR (Algorithm 1) to obtain a coreset. We then train a LR model on the coreset to predict the labels for the test instances.
– **ACvS:** we use the procedure 'Accelerated Clustering via Sampling', described in Algorithm 2, to accelerate the coreset computation. Once we have obtained the coreset in accelerated fashion, we proceed to learn a LR classifier over it.

[7] https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/ - last accessed in 4/2021.

– **RDSF:** summaries of data are generated via the 'Regressed Data Summarisation Framework', described in Algorithm 3. Hence, we compute the sensitivity scores only for a handful of instances in the training set. Then, we train a regressor to predict the sensitivity scores for the remaining of the training instances. We sample points according to the sensitivities, compute their weights, and return the data summary. We then proceed as in the previous coreset-based procedures.
– **URS:** for the sake of completeness, we include 'Uniform Random Sampling' as a baseline for reducing the volume of input data; we simply pick the required input points uniformly at random and then train a LR classifier over them.

Our evaluation pipeline can be described as follows: for each of the above approaches, and for each dataset in Table 1, we take the below steps:

a) **data shuffling:** we randomly mix up all the available data.
b) **data splitting:** we select 50% of the data as training set and leave the rest for testing purposes. The training set is referred as the input data $\mathcal{D}$.
c) **data compressing:** we proceed to compress the input data. As previously mentioned, the *CABLR* approach computes a coreset without any acceleration, *ACvS* computes a coreset by using CABLR with an accelerated clustering phase, *RDSF* compresses the input data into a small data summary in an accelerated fashion, and *URS* performs a naive compression by taking uniform random samples of the input data. *Full Data* is the only approach that does not perform any compression on the input data.
d) **data training:** we train a Logistic Regression classifier on the data obtained in the previous step. *CABLR*, *ACvS*, *RDSF* and *URS* produce a reduced version of the input data while *Full Data* trains the classifier on the full input data $\mathcal{D}$.
e) **data assessing:** we finally use the trained Logistic Regression classifiers to predict the labels in the test set and apply our performance metrics, detailed in Section 4.2.

We applied the above steps 10 times for each of the five different approaches. Hence, the results we present in the next sections are averaged ones. Regarding the hardware, our experiments were performed on a single desktop PC running the Ubuntu-Linux operating system, equipped with an Intel(R) Xeon(R) CPU E3-1225 v5 @ 3.30GHz processor and 32 Gigabytes of RAM.

For the coreset implementation, we adapted to our needs the CABLR algorithm as shared by its authors [8]. All of our programs were written in Python. The method used for clustering the input data is the well-known K-means algorithm; and for RDSF, we used linear regression to learn the sensitivities of input points.

[8] https://bitbucket.org/jhhuggins/lrcoresets/src/master - last accessed in 2/2020

4.2 Metrics

As previously mentioned, the empirical performance of coresets has mainly remained a grey area in the past years. We apply the following performance metrics in order to shed some light on, first, the performance of coresets in general, second, the performance of our proposed methods. We consider the following five performance metrics:

1. **Computing time (in seconds):** we measure acceleration in seconds. To make a more meaningful analysis, we further break down time into 5 different stages :
   a) **Clustering:** the time needed to obtain the k-centres for coreset-based approaches.
   b) **Sensitivity:** the time required to compute the sensitivity score for each input point.
   c) **Regression:** the time needed for training a regressor in order to predict the sensitivity scores for input points. The prediction time is also taken into account.
   d) **Sampling:** the time required to sample input points.
   e) **Training:** the time required for leaning an LR classifier.

   Notice that the *Training phase* is the only one present in all of our approaches. Hence, for example, the *coreset* approach does not learn any regressor and thus it is assigned 0 second for that phase. The URS approach does not perform any clustering or sensitivity computation, hence those phases get 0 second for this method, etc.
2. **Classification Accuracy:** this measure is given by the percentage of correctly classified test examples. It is commonly used as the baseline metric for measuring performance in a supervised-learning setting.
3. **Area Under the Precision & Recall Curve (PREC/REC):** Precision is defined as $\frac{TP}{TP+FP}$ and Recall can be computed as $\frac{TP}{TP+FN}$ [8], where TP,FP and FN stand for the *True Positives*, *False Positives* and *False Negatives* achieved by a binary classifier, respectively. The curve is obtained by putting the Recall on the x-axis and the Precision on the y-axis. Once the curve has been generated, the area under the curve can be calculated in the interval between 0 and 1. The greater the area, the better the performance.
4. **F1 Score:** is the harmonic average of precision and recall [11] and hence can be computed as $F_1 := 2\frac{PR}{R+P}$, where P is Precision and R is recall [11]. The greater the value, the better the classifier's performance.
5. **Area Under the ROC Curve (AUROC):** provides an aggregate measure of performance across all possible classification thresholds. The curve can be computed by placing the *False Positives Rate* (FPR) on the x-axis and the *True Positives Rate* (TPR) on the y-axis, with FPR $:= \frac{FP}{FP+TN}$ and TPR $:= \frac{TP}{TP+FN}$; here, once more, TP,FP and FN stand for the *True Positives*, *False Positives* and *False Negatives* achieved by any binary classifier, respectively. Simillar to the area under the precision and recall curve, AUROC is obatined from the curve.

4.3 Acceleration via ACvS and RDSF

We categorise our results according to the five different metrics we just described. Notice that their values are shown as functions of the size of the summaries used for training the LR classifier. Thus, if we look at Figures 3, 5, 4 and 6, we can see that summary sizes on the x-axes correspond to very small percentages of the training set. Specifically, for the Higgs dataset, which is the largest one, the summary sizes are 0.005 %, 0.03 %, 0.06 % and 0.1 % of the input data. For Webspam and Covertype, which are smaller then Higgs, we show results with summary sizes of 0.05 %, 0.1 % , 0.3 %, 0.6 % and 1 % of the total input data. Finally, for w8a and ijcnn1, which are the smaller datasets, the sizes shown are 1 %, 3 % , 6 %, and 10 %. The reason why there are different summary sizes for some of our datasets has to do with the difference in size across datasets. For example, computing a summary of 0.005 % of the w8a or the ijcnn1 datasets is unfeasible since these datasets are not very large and hence it is very likely to end up with a extremely small summary of data that only contains points of one class. In other words: the larger the dataset, the more we can compress it.

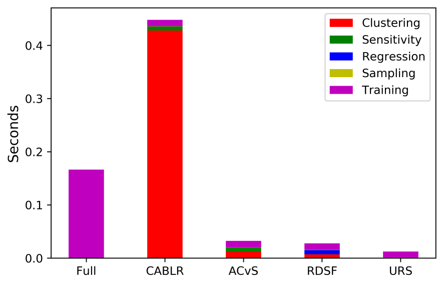
Computing Time Figure 2 summarises our results in terms of computing time. We show in the stacked-bars plots the time spent for each *phase* of the different approaches.

We can clearly see that the *CABLR* approach, which clusters the full input data in order to constructs coresets , is not suitable for the optimisation setting we are considering. Specifically, with respect to the Full method, we notice that for the Covertype dataset, the coreset approach gives a modest acceleration of approximately 1.2 times. The situation becomes more severe for the Webspam, ijcnn1, w8a and Higgs datasets, where using the traditional coreset approach incurs in a learning process which is about 1.9, 2.6, 1.8 and 1.3 times slower than not using coreset at all, respectively.

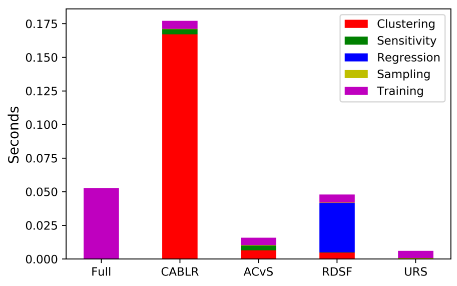
Hence, by removing the bottleneck produced by the clustering phase, our two proposed methods show that we can still benefit from coreset acceleration to solve our particular problem; that is, our methods take substantially shorter computing time when compared to the *CABLR* approach. In particular, and with respect to Full Data approach, our approach *ACvS* achieves a minimum acceleration of 3.5 times (w8a) and a maximum acceleration of 34 times (Higgs) across our datasets. Regarding *RDSF*, the minimum acceleration obtained was 1.15 times (w8a) and the maximum was 27 times (Covertype).

Notice that our accelerated methods are only beaten by the naive URS method, which should most certainly be the fastest approach.

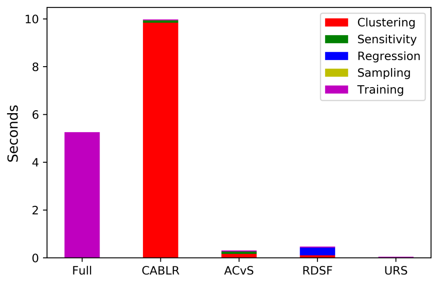
Finally, notice that the *RDSF* approach is slightly more expensive than *ACvS*. This is the computing price we pay for obtaining more information *i.e.* *RDSF* outputs a trained regressor that can immediately assign sensitivities to new unseen data points; this can prove extremely useful in settings when learning should be done *on the fly*. According to our evaluations, RDSF's time can be improved by reducing the number of points used for training the underlying regression algorithm. In general, using 1 % of the training set for training the



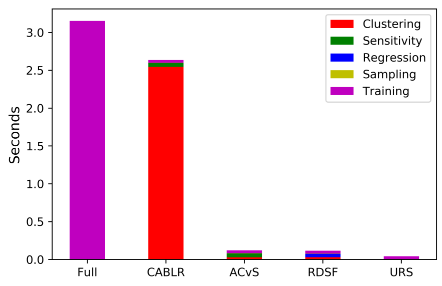
(a) ijcnn1



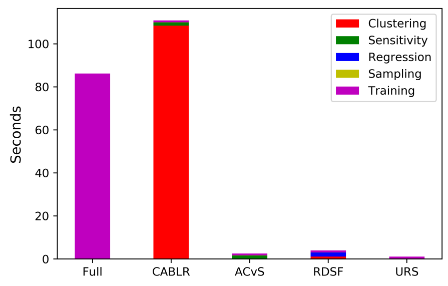
(b) w8a



(c) Webspam



(d) Covertype



(e) Higgs

Fig. 2: Comparison of the computing time required by the different procedures.

regressor worked well; however, depending on the data, this could be reduced (or increased) in order to achieve better computing time (learning quality).

The natural follow-up question is whether our methods' resulting classifiers perform well. We address this question in great detail in the following sections.

Accuracy We first look into the baseline metric for measuring the success of a classifier: the accuracy. Figure 3 shows how the accuracy of the methods changes as the sample sizes increase on different datasets. To recall, each of the different methods considered relies on reducing the input data via a coreset-based compression or a random uniform sample, as described in Section 4.1. Hence, we here report the different accuracy scores obtained by training our LR classifier on different samples sizes. As reference, we also include the accuracy of the *Full Data* method as a straight line.

The first observation is that all the methods perform better as the sample sizes increase. Quite surprisingly, we see that in all cases, without exceptions, the ACvS approach achieves the exact same accuracy that the CABLR approach achieves, for all sample sizes. Hence, for coresets, clustering over a sub-sample of the input data does not seem to deteriorate the rate of correct predictions of the resulting classifiers, and greatly accelerates the overall coreset computation, as we could appreciate in the previous section.

We also see that the RDSF approach performs as good as the rest of the coreset-based approaches, with accuracy never lower than the baseline approach (URS). It is generally expected that coresets outperform the URS approach in most situations; and RDSF summaries, even without being strictly a coreset [9], show this behaviour.

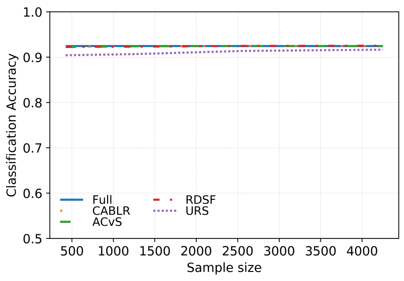
Finally, we see that, as sample sizes increase, the gap between coreset performance and URS performance reduces *i.e.* they both get closer and closer to the Full Data approach. A particularly interesting case is that of w8a, which shows a very similar performance for all the methods. This could be an indicator that points in the dataset contribute *almost* equally to the learning problem considered: LR, in this case.

F1 Score We now present the results of applying the F1 score metric to our LR classifiers for different sample sizes, see Figure 4.

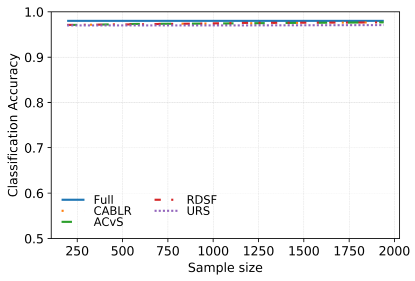
The first observation we make is that, similar to accuracy results, ACvS and CABLR obtain exactly the same scores, and RDSF remains competitive against them. Furthermore, it is fair to say that for the Covertype and Higgs datasets, RDSF has preferable performance compared to the other compression approaches. Hence, we once more see that the advantages of coresets are available in our setting as long as we carefully accelerate the underlying algorithm.

Our experiments reveal that the performance gap between URS and the rest of the approaches becomes even greater for the F1 Score; showing that classifiers

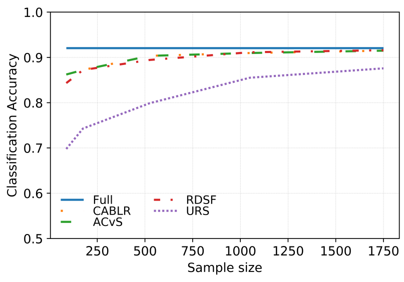
[9] we carefully distinguish between a coreset and a coreset-based summary. The former requires a theoretical proof on the quality loss.



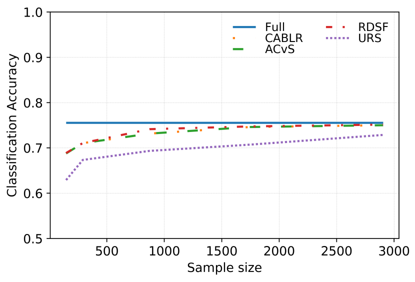
(a) ijcnn1



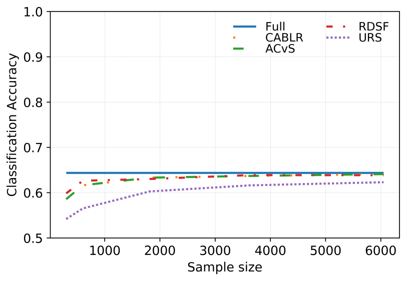
(b) w8a



(c) Webspam



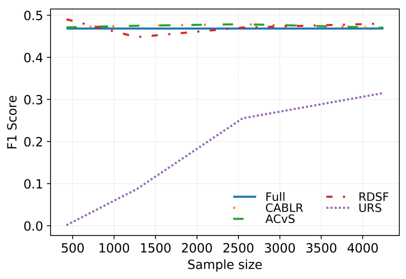
(d) Covertype



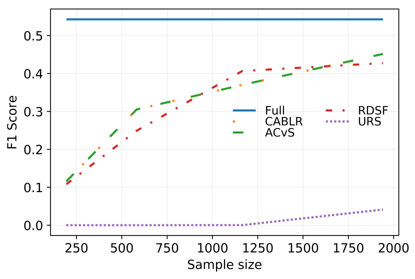
(e) Higgs

Fig. 3: Comparison of the prediction accuracy achieved by the considered methods.

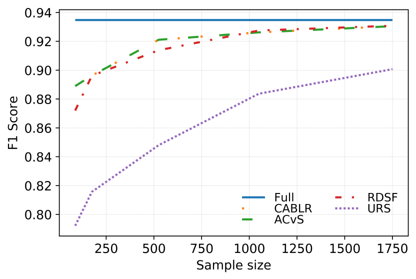
trained over coresets are more useful and informative than the ones trained over uniformly randomly selected samples. Furthermore, if we look at F1 score for ijcnn1 (see Figure 6a) we can have a glimpse of an interesting phenomenon: we actually obtain better results using coresets, and hence *less* data, than using the full dataset. We leave this as an open problem for future exploration.



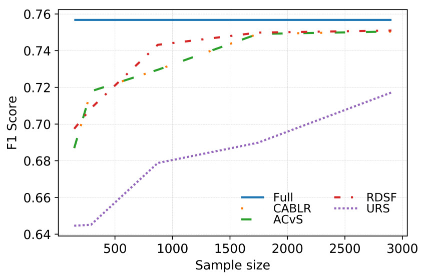
(a) ijcnn1



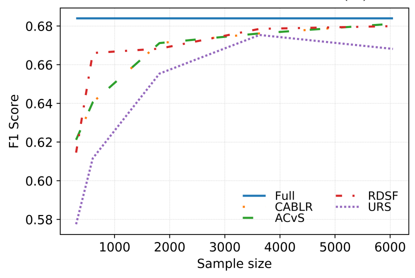
(b) w8a



(c) Webspam



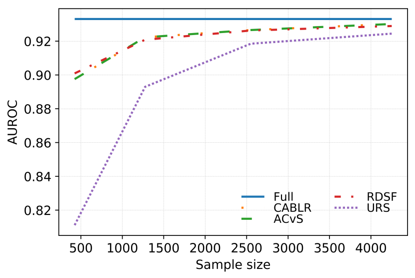
(d) Covertype



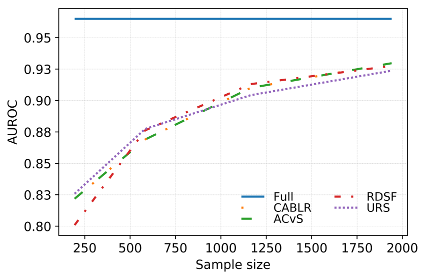
(e) Higgs

Fig. 4: F1 scores obtained by each of the methods.

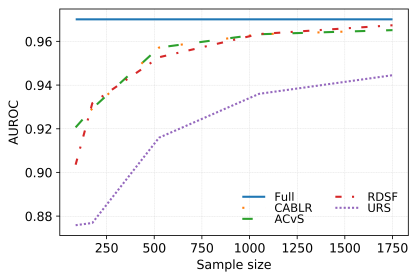
AUROC We now present the AUROC score for each of our 5 approaches. Figure 5 shows comparison of the AUROC scores for all methods. The picture is similar to that of F1-score and Accuracy: coreset-based approaches consistently outperform URS. We see indeed that ACvS and RDSF perform competitively traditional coreset approach, achieving their performance in substantially less computing time than coresets (see Section 4.3). An intriguing case is that of w8a, which shows that URS is actually slightly better than coreset approaches for small sample sizes. As we previously mentioned, it is highly probably that this is an indication that the input points in the dataset are not very different in terms of their contribution to the LR objective function; or, it could also be the case that the sensitivity distribution, as computed by the coreset-based methods, does not fully account for the structure of the data.



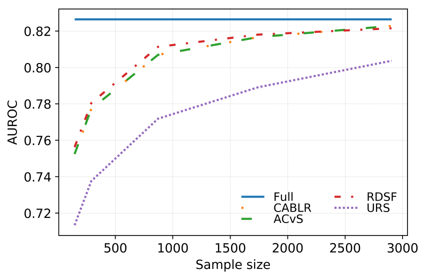
(a) ijcnn1



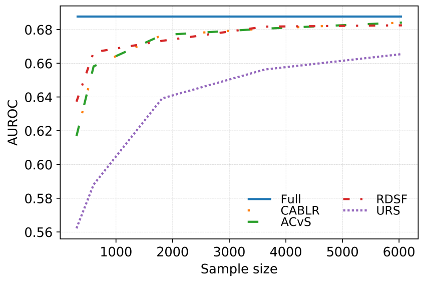
(b) w8a



(c) Webspam

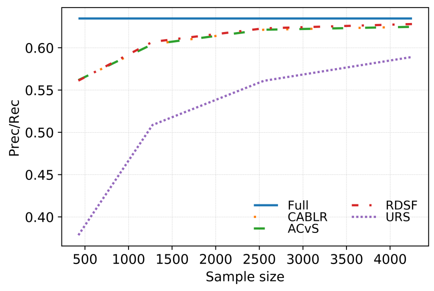


(d) Covertype

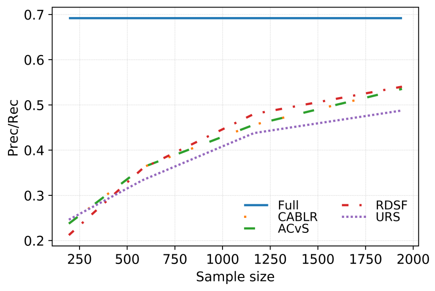


(e) Higgs

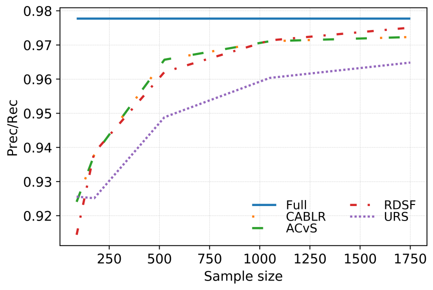
Fig. 5: Comparison of the area under the ROC curve of different methods.



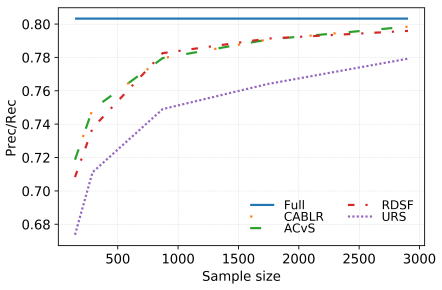
(a) ijcnn1



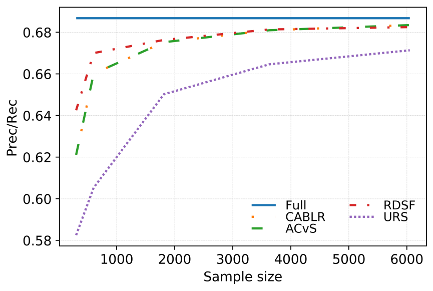
(b) w8a



(c) Webspam



(d) Covertype



(e) Higgs

Fig. 6: Comparison of the area under the precision/recall curve of different methods.

Precision & Recall Finally, we present the results concerning the precision and recall score. The behaviour here is similar to that of AUROC. The performance of w8a seems to be different from the rest of the data once more: we see that, for very small sample sizes, URS is even slightly better that the coreset and coreset-based approaches. As the sample sizes increase, the latter outperform the former, although not by much. As we previously mentioned, this could mean that input points in w8a are more or similar for LR and hence we cannot strictly distinguish between redundant and important points. We also see that ACvS gets exactly the same scores when compared to its non-accelerated version; which shows that clustering over the whole input data is not necessary *i.e.* clustering over a small uniform random sample is sufficient.

Summary of Results This section provides detailed performance information of our procedures, alongside the rest of the methods, on the five datasets shown in Table 1. Specifically, Tables 2, 3, 4, 5 and 6 show the different metric scores obtained over the these 5 datasets, respectively.

Overall, and as shown in the plots in previous sub-sections, the empirical results demonstrated that ACvS and RDSF do provide meaningful acceleration to the traditional Coreset approach, while maintaining competitive performance, in all datasets considered. One important observation is that, even though our procedures give good speed-ups, the performance of coresets are dataset dependent. If the structure of the data can be captured correctly by doing an uniform random sample, then coreset performance should not be expected to be very different than URS. We can see an example of this in w8a, where coresets in general do not give very meaningful improvement. On the other hand, for the rest of the datasets, we can indeed see how compressing the input data in more involved fashion gives a significant improvement over the naive URS.

4.4 The RDSF Technique with Different Regressors

To finish our experiments exposition, we report the results obtained when using RDSF with different regressors in a plug-in/plug-out fashion. More concretely, we consider four different regression approaches: Ordinary Least Squares (OLS), Ridge Regression (RR), Lasso Regression (LSR) and Elastic Net (EN). To remind the reader, RR consists in fitting an OLS regressor with a L_2 regulariser; on the other hand, LSR trains an OLS regressor using a L_1 regulariser. Finally, EN finds an OLS regressor by using a convex combination of both L_1 and L_2 as its the regularisation term.

Depending of what regression algorithm was used to predict the sensitivities of input points, we can have one the following four RDSF instances:

– **RDFS-OLS:** an RDSF instance in which the input points' sensitivity scores were predicted using the ordinary-least-squares regression method. It is worth mentioning that this is the RDSF instance used in [17] and in Section 4.3.
– **RDFS-RR:** an instance of RDSF where the sensitivity for input points was predicted using the ridge regression method.
– **RDFS-LSR:** an RDSF instance that uses the lasso method for regression for predicting sensitivity scores.
– **RDFS-EN:** an instance of RDSF in which the sensitivity scores for input points are predicted via the elastic net regression method.

It is important to mention that for this study we only consider three metrics: Precision & Recall, AUROC and F1 Score. The reason for this decision already sheds the first lights on using different regressors for RDSF: different regressors give summaries of data that, when used to solve the LR problem, give classifiers

Table 2: Performance comparison, as presented in [17], on the Covertype dataset. Here, "size" is the percentage of training data used for coresets and coreset-based summaries.

Size (%)	**Method**	**F1 score**	**ROC**	**Accuracy**	**Time (seconds)**
0.05	Full	0.76	0.83	0.76	3.35
0.05	CABLR	0.69	0.75	0.69	2.78
0.05	ACvS	0.69	0.75	0.69	0.12
0.05	RDSF	0.70	0.76	0.69	0.12
0.05	URS	0.65	0.71	0.63	0.04
0.3	Full	0.76	0.83	0.76	3.31
0.3	CABLR	0.73	0.81	0.73	2.78
0.3	ACvS	0.73	0.81	0.73	0.13
0.3	RDSF	0.74	0.81	0.74	0.13
0.3	URS	0.68	0.77	0.69	0.05
1	Full	0.76	0.83	0.76	3.71
1	CABLR	0.75	0.82	0.75	3.38
1	ACvS	0.75	0.82	0.75	0.18
1	RDSF	0.75	0.82	0.75	0.18
1	URS	0.72	0.80	0.73	0.07

that do not meaningfully differ in their accuracy score. However, we will see that when more involved performance metrics are considered, the difference in performance becomes more meaningful.

We also highlight that the point of this set of experiments is not to find out which of the RDSF instances considered is faster as the OLS method is by definition the fastest regression algorithm of all, and ridge regression is known to be faster than lasso. Instead, the aim of these experiments is to see whether more involved regression algorithms can help us obtain RDSF summaries that obtain better learning performance.

Figures 7, 8 and 9 show the performance obtained when compressing the input data using different instances of the RDSF framework, specifically in terms of the precision & recall, AUROC and F1 score, respectively. Notice that these plots, as the ones found in the previous subsection, have the different sample sizes of the (RDSF) summaries on the x axis and the corresponding performance value on the y axis.

Table 3: Performance comparison, as presented in [17], on the Webspam dataset. Here, “size” is the percentage of training data used for coresets and coreset-based summaries.

Size (%)	**Method**	**F1 score**	**ROC**	**Accuracy**	**Time (seconds)**
0.05	Full	0.92	0.94	0.97	5.39
0.05	CABLR	0.86	0.89	0.92	10.15
0.05	ACvS	0.86	0.89	0.92	0.31
0.05	RDSF	0.84	0.87	0.90	0.49
0.05	URS	0.70	0.79	0.88	0.05
0.3	Full	0.92	0.94	0.97	6.54
0.3	CABLR	0.90	0.92	0.96	13.40
0.3	ACvS	0.90	0.92	0.96	0.41
0.3	RDSF	0.89	0.91	0.95	0.64
0.3	URS	0.80	0.85	0.92	0.06
1	Full	0.92	0.94	0.97	6.35
1	CABLR	0.92	0.93	0.97	13.44
1	ACvS	0.92	0.93	0.97	0.45
1	RDSF	0.92	0.93	0.97	0.68
1	URS	0.88	0.90	0.95	0.08

Quite surprisingly, RDSF trained with the simple OLS method seems to be the preferable approach for computing RDSF summaries in general: it is not always the best. but more often than not it gets really close to the best performing method. This is an enlightening result as OLS is the simplest method of the ones considered, and is seems to provide a very good explanation of the relationship between the input points and their sensitivities. Furthermore, we can make the conclusion that regularisation does not help us much with the task of predicting sensitivities. If we also add up the fact that OLS has a closed-form solution and hence it is computationally very efficient, it becomes really hard to justify the use of any of the other regression algorithms to compress input data via the RDSF framework.

Finally, it is useful to mention that we also considered different sample sizes for the regression problem of sensitivity prediction *i.e.* different values for the parameter b in Algorithm 3; specifically, we tested setting b to 0.01%, 0.05%, 1%, 5%, 10% and 15% of the input data size; however, the behaviour observed is very similar to the results presented here.

Table 4: Performance comparison on the Higgs dataset. Here, "size" is the percentage of training data used for coresets and coreset-based summaries.

Size (%)	Method	F1 score	ROC	Accuracy	Time (seconds)
0.005	Full	0.68	0.69	0.64	89.22
0.005	CABLR	0.62	0.62	0.59	112.44
0.005	ACvS	0.62	0.62	0.59	2.61
0.005	RDSF	0.62	0.64	0.60	4.16
0.005	URS	0.58	0.56	0.54	1.12
0.03	Full	0.68	0.69	0.64	92.22
0.03	CABLR	0.67	0.68	0.633	121.25
0.03	ACvS	0.67	0.68	0.63	2.70
0.03	RDSF	0.67	0.67	0.63	4.46
0.03	URS	0.66	0.64	0.60	1.20
0.1	Full	0.68	0.69	0.64	90.18
0.1	CABLR	0.68	0.68	0.64	123.97
0.1	ACvS	0.68	0.68	0.64	2.61
0.1	RDSF	0.68	0.68	0.64	4.50
0.1	URS	0.69	0.67	0.62	1.29

5 Conclusion

As modern ever-growing sets of data overshadow our computing resources, scaling up machine learning algorithm is not a trivial task. The most direct algorithmic approach is to write new learning algorithms that overcome the inefficiencies of their old counterparts. A less direct approach consists of using the well-known algorithms we currently possess over a reduced version of their input data. We have presented the paradigm of coresets: a framework that correctly compresses the input data with respect to an specific learning problem. We have shown that for the optimisation setting, the algorithm for constructing coresets for the problem of Logistic Regression relies on a clustering phase that, more often than not, creates a bottleneck in the compression process.

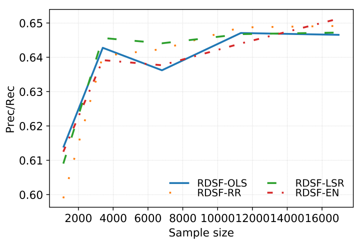
To circumvent this, we proposed two methods that ease this computational bottleneck: Accelerating Clustering via Sampling (ACvS) and Regressed Data Summarisation Framework (RDSF). Both methods achieved substantial overall learning acceleration while maintaining the performance accuracy of coresets.

Table 5: Performance comparison on the w8a dataset. Here, "size" is the percentage of training data used for coresets and coreset-based summaries.

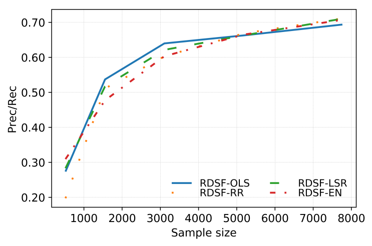
Size (%)	**Method**	**F1 score**	**ROC**	**Accuracy**	**Time (seconds)**
1	Full	0.543	0.965	0.98	0.06
1	Coreset	0.116	0.822	0.971	0.18
1	ACvS	0.116	0.822	0.971	0.02
1	RDSF	0.108	0.801	0.971	0.05
1	URS	0	0.826	0.97	0.007
3	Full	0.543	0.965	0.98	0.06
3	Coreset	0.305	0.869	0.973	0.19
3	ACvS	0.305	0.869	0.97321	0.02
3	RDSF	0.249	0.876	0.97237	0.06
3	URS	0	0.877	0.97	0.01
6	Full	0.543	0.965	0.98	0.05
6	Coreset	0.37	0.91	0.975	0.19
6	ACvS	0.37	0.91	0.975	0.02
6	RDSF	0.407	0.913	0.975	0.05
6	URS	0.0003	0.904	0.97	0.01

Table 6: Performance comparison, as presented in [17], on the ijcnn1 dataset. Here, “size” is the percentage of training data used for coresets and coreset-based summaries.

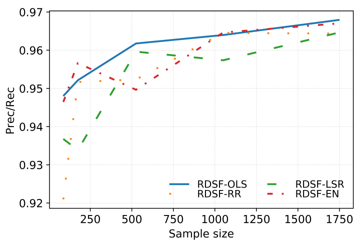
Size (%)	**Method**	**F1 score**	**ROC**	**Accuracy**	**Time (seconds)**
1	Full	0.46836	0.93313	0.92462	0.18
1	Coreset	0.47105	0.89755	0.92229	0.49
1	ACvS	0.47105	0.89755	0.92229	0.03
1	RCP	0.49022	0.90092	0.92261	0.03
1	URS	0.00123	0.81114	0.90412	0.01
3	Full	0.46836	0.93313	0.92462	0.19
3	Coreset	0.47487	0.92229	0.92391	0.45
3	ACvS	0.47487	0.92229	0.92391	0.04
3	RCP	0.44788	0.92085	0.92305	0.03
3	URS	0.08715	0.89298	0.90677	0.02
6	Full	0.46836	0.93313	0.92462	0.19
6	Coreset	0.47861	0.92658	0.92456	0.46
6	ACvS	0.47861	0.92658	0.92456	0.04
6	RCP	0.47063	0.92630	0.92477	0.04
6	URS	0.25508	0.91849	0.91363	0.02



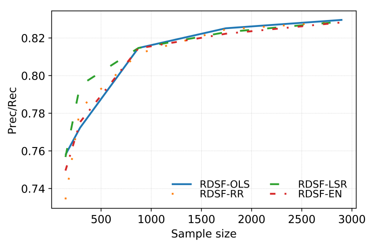
(a) ijcnn1



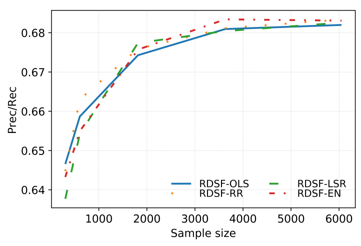
(b) w8a



(c) Webspam



(d) Covertype



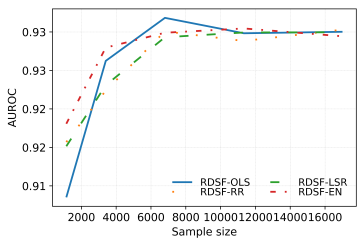
(e) Higgs

Fig. 7: Comparison of the areas under the precision & recall curve obtained from LR classifiers learned over RDSF summaries. The summaries were computed using different underlying regressors.

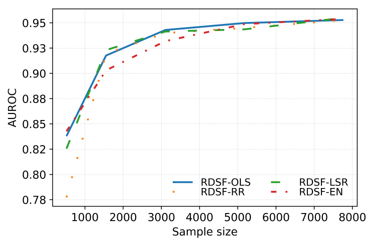
This implies that coresets can still be efficiently used to learn a logistic regression classifier in the optimisation setting.

Interestingly, we observed that, even though CABLR involves input data clustering, this can be relaxed in the practical sense. Furthermore, our calculations indicate that CABLR must be used with the clustering done over a small subset of the input data in the optimisation setting (*i.e.* the ACvS approach). Our empirical evaluations confirm that, by doing so, one will not be sacrificing learning performance.

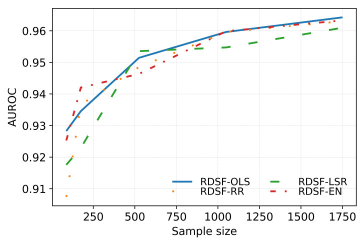
With respect to RDSF, we believe this will open a new research branch for coresets: we could pose the computation of data compression via coresets as solving a small-scale learning problem in order to solve a large-scale one. It is



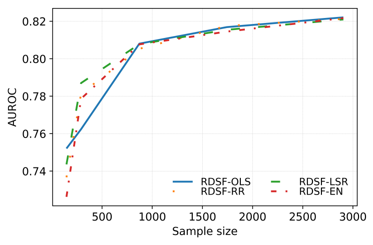
(a) ijcnn1



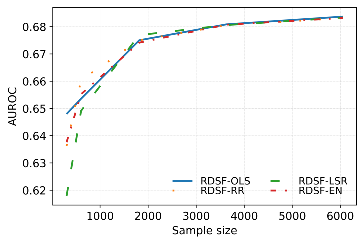
(b) w8a



(c) Webspam

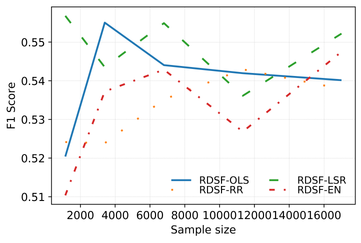


(d) Covertype

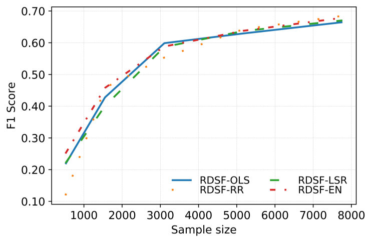


(e) Higgs

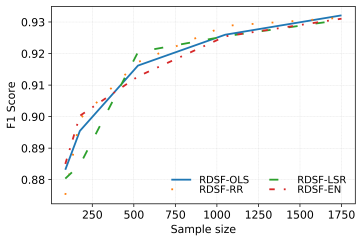
Fig. 8: Comparison of the AUROC obtained from LR classifiers learned over RDSF summaries. The summaries were computed using different underlying regressors.



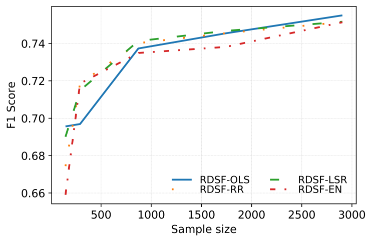
(a) ijcnn1



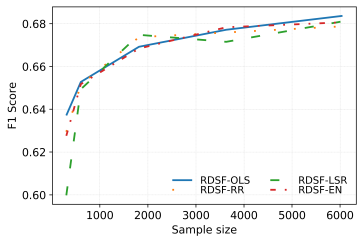
(b) w8a



(c) Webspam



(d) Covertype



(e) Higgs

Fig. 9: Comparison of the F1 scores obtained from LR classifiers learned over RDSF summaries. The summaries were computed using different underlying regressors.

interesting to see that the sensitivities of input points can be explained by a simple linear regressor. Most importantly, we believe that this method could be of powerful use in the *online learning setting* [21]. This is because RDSF allows us to obtain a fully trained regressor capable of assigning sensitivity scores to new incoming data points. We leave the use of these methods in different machine learning tasks as future work.

Acknowledgements

This research is supported by AstraZeneca and the Paraguayan Government.

References

1. Ackermann, M.R., Märtens, M., Raupach, C., Swierkot, K., Lammersen, C., Sohler, C.: Streamkm++: A cluste ing algorithm for data streams. Journal of Experimental Algorithmics (JEA) **17**, 2–4 (2012)
2. Agarwal, P.K., Har-Peled, S., Varadarajan, K.R.: Geometric approximation via coresets. Combinatorial and computational geometry **52**, 1–30 (2005)
3. Arthur, D., Vassilvitskii, S.: k-means++: The advantages of careful seeding. In: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 1027–1035. Society for Industrial and Applied Mathematics (2007)
4. Bachem, O., Lucic, M., Krause, A.: Practical coreset constructions for machine learning. arXiv preprint arXiv:1703.06476 (2017)
5. Bădoiu, M., Clarkson, K.L.: Optimal core-sets for balls. Computational Geometry **40**(1), 14–22 (2008)
6. Braverman, V., Feldman, D., Lang, H.: New frameworks for offline and streaming coreset constructions. CoRR **abs/1612.00889** (2016), http://arxiv.org/abs/1612.00889
7. Dasgupta, S., Gupta, A.: An elementary proof of the johnson-lindenstrauss lemma. International Computer Science Institute, Technical Report **22**(1), 1–5 (1999)
8. Davis, J., Goadrich, M.: The relationship between precision-recall and roc curves. In: Proceedings of the 23rd international conference on Machine learning. pp. 233–240 (2006)
9. Feldman, D., Langberg, M.: A unified framework for approximating and clustering data. In: Proceedings of the forty-third annual ACM symposium on Theory of computing. pp. 569–578. ACM (2011)
10. Feldman, D., Schmidt, M., Sohler, C.: Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In: Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms. pp. 1434–1453. SIAM (2013)
11. Goutte, C., Gaussier, E.: A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In: European Conference on Information Retrieval. pp. 345–359. Springer (2005)
12. Har-Peled, S., Mazumdar, S.: On coresets for k-means and k-median clustering. In: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing. pp. 291–300. ACM (2004)

13. Huggins, J., Campbell, T., Broderick, T.: Coresets for scalable bayesian logistic regression. In: Advances in Neural Information Processing Systems. pp. 4080–4088 (2016)
14. Mustafa, N.H., Varadarajan, K.R.: Epsilon-approximations and epsilon-nets. arXiv preprint arXiv:1702.03676 (2017)
15. Phillips, J.M.: Coresets and sketches. arXiv preprint arXiv:1601.00617 (2016)
16. Reddi, S.J., Póczos, B., Smola, A.J.: Communication efficient coresets for empirical loss minimization. In: UAI. pp. 752–761 (2015)
17. Riquelme-Granada, N., Nguyen., K.A., Luo., Z.: On generating efficient data summaries for logistic regression: A coreset-based approach. In: Proceedings of the 9th International Conference on Data Science, Technology and Applications - Volume 1: DATA,. pp. 78–89. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009823200780089
18. Riquelme-Granada, N., Nguyen, K., Luo, Z.: Coreset-based conformal prediction for large-scale learning. In: Conformal and Probabilistic Prediction and Applications. pp. 142–162 (2019)
19. Riquelme-Granada, N., Nguyen, K.A., Luo, Z.: Fast probabilistic prediction for kernel svm via enclosing balls. In: Conformal and Probabilistic Prediction and Applications. pp. 189–208. PMLR (2020)
20. Shalev-Shwartz, S., Ben-David, S.: Understanding machine learning: From theory to algorithms. Cambridge university press (2014)
21. Shalev-Shwartz, S., et al.: Online learning and online convex optimization. Foundations and Trends in Machine Learning **4**(2), 107–194 (2012)
22. Zhang, Y., Tangwongsan, K., Tirthapura, S.: Streaming k-means clustering with fast queries. In: Data Engineering (ICDE), 2017 IEEE 33rd International Conference on. pp. 449–460. IEEE (2017)