# On Generating Efficient Data Summaries for Logistic Regression: A Coreset-based Approach

Nery Riquelme-Granada, Khuong An Nguyen and Zhiyuan Luo
*Department of Computer Science, Royal Holloway University of London, Egham, Surrey, TW20 0EX, U.K.*

Keywords: Coresets, Data Summaries, Logistic Regression, Large-data, Computing Time.

Abstract: In the era of datasets of unprecedented sizes, data compression techniques are an attractive approach for speeding up machine learning algorithms. One of the most successful paradigms for achieving good-quality compression is that of coresets: small summaries of data that act as proxies to the original input data. Even though coresets proved to be extremely useful to accelerate unsupervised learning problems, applying them to supervised learning problems may bring unexpected computational bottlenecks. We show that this is the case for Logistic Regression classification, and hence propose two methods for accelerating the computation of coresets for this problem. When coresets are computed using our methods on three public datasets, computing the coreset and learning from it is, in the worst case, 11 times faster than learning directly from the full input data, and 34 times faster in the best case. Furthermore, our results indicate that our accelerating approaches do not degrade the empirical performance of coresets.

## 1 INTRODUCTION

Classification is one of the most fundamental machine learning problems: given labeled data $\mathcal{D}$, we aim to *efficiently* learn a model that predicts properties of unseen data with minimal error. This leads to the following dilemma: on one hand, our learning process benefits greatly as the size of the input data grows; on the other hand, the computational cost of our leaning algorithms grows at least linearly with the input data size. Hence, large-scale datasets tend to provide rich knowledge to the learning process at the cost of very high computing time.

The classic approach for tackling the above challenge in the algorithmic field of Computer Science is through the design of new algorithms with better running times (*i.e.* an efficient new algorithm which scales with the input data). A more recent, novel approach involves using the same (potentially inefficient) algorithm over a *reduced* version of the input data (*i.e.* compressing the input data so that the same algorithm may complete faster).

One of the most popular frameworks to achieve a compression of (large-scale) input data is that of 'coresets' (or core-sets) (Har-Peled and Mazumdar, 2004), which allows us to identify and remove *less important* information from our input data while retaining certain properties. Roughly speaking, a coreset is a small *weighted* subset of data [1] that provably correctly approximates the original input data (Phillips, 2016). After obtaining a coreset, we may safely discard the original input data and run our learning algorithm over the weighted data summary, dramatically reducing storage and computational overheads.

In this paper we investigate the performance of coresets for Logistic Regression (LR) and show that careless use of coresets may result in devastating impact for the learning process. More concretely, we empirically show that clustering, which is used as a subroutine in the coreset construction, imposes a *severe bottleneck* on the coreset computation for LR.

To circumvent the above problem, we present a method called 'Accelerated Clustering via Sampling' (ACvS), and shall demonstrate that it greatly accelerates the coreset construction. Additionally, we introduce the *Regressed Data Summarisation Framework* (RDSF), a meta procedure that not only accelerates the coreset construction process, but also opens a new perspective for coresets: we apply machine learning, in the form of regression, to decide what input information is important to include in the data summary. Thus, RDSF "learns" the *importance* of each input point and allows us to retain this knowledge in the

[1] Although this can be relaxed.

form of a trained linear regressor. The latter can be extremely useful for deciding the importance of previously unseen points in almost constant time.

We summarise our contributions as follows:

- We show the existence of a heavy bottleneck in the clustering step of the coreset construction algorithm for LR. In response, we empirically demonstrate that this issue can be mitigated by taking the clustering over small uniform random samples of the input data. We call this method *Accelerated Clustering via Sampling* (ACvS), and we show that using it alongside coresets greatly accelerates the coreset computation while retaining the coreset performance.
- Piggybacking on ACvS, we propose the method of *Regressed Data Summarisation Framework* (RDSF) to output coreset-based summaries of data. In essence, given a dataset $\mathcal{D}$ and a coreset construction algorithm $\mathcal{A}$, we can *plug-in* $\mathcal{A}$ into RDSF to obtain (i) a small summary of $\mathcal{D}$ (ii) a trained regression model capable of identifying *important points* in unseen data. Thus, RDSF adds an extra layer of information to the whole data compression process.
- We shed light on the empirical performance of coresets for Logistic Regression with and without our methods; for this, we measure their performance not only in terms of computing time, but also in terms of classification accuracy, F1 score and ROC curves.

The rest of the paper is organised as follows: Section 2 gives a gentle introduction to coresets and discusses the coreset inefficiencies for the defined learning problem. Section 3 provides a full description of our two methods: ACvS and RDSF. Our empirical evaluations are presented in Section 4. Finally, we conclude our work in Section 5.

## 2 BACKGROUND AND PROBLEM

In this section we give a gentle introduction to coresets. We define the classification problem of Logistic Regression in the optimisation setting, for which we present the state-of-the-art algorithm for constructing coresets. Finally, we show the computational bottleneck imposed by computing a clustering of the whole input data as part of the LR coreset construction algorithm.

### 2.1 Coresets

Coresets are a data approximation framework that has been used to reduce both the volume and dimensionality of big and complex datasets. We can formally define a coreset as follows: let function $f$ be the objective function of some learning problem and let $\mathcal{D}$ be the input data. We call $\mathcal{C}$ an $\varepsilon$-coreset for $\mathcal{D}$ if the following inequality holds:

$$|f(\mathcal{D}) - f(\mathcal{C})| \leq \varepsilon f(\mathcal{D}) \qquad (1)$$

where $\varepsilon$ is the error parameter. This expression establishes the main error bound offered by coresets. Hence, coresets are lossy compressed versions of the input data $\mathcal{D}$ and the amount of information loss is quantified by $\varepsilon$. Note that we need $|\mathcal{C}| << |\mathcal{D}|$.

The design of an algorithm to construct a set $\mathcal{C}$ that fulfils (1) depends naturally on the function $f$. In machine learning, $f$ is defined as the objective function of the learning problem of interest. Thus, we consider $f$ as the function that minimises the *log-likelihood*, which corresponds to learning a Logistic Regression classifier.

Even though coresets have been widely studied in the context of clustering (Feldman et al., 2013) (Har-Peled and Mazumdar, 2004) (Bachem et al., 2017) (Zhang et al., 2017) (Ackermann et al., 2012), their uses remain largely unexplored for supervised learning problems. In this light, the most studied problem in the coreset community is Logistic Regression (LR) (Huggins et al., 2016) (Munteanu et al., 2018). Current coreset construction algorithms for this problem also rely on solving a clustering problem under the hood. Specifically, the current state-of-the-art coreset algorithm for LR (Huggins et al., 2016) showed good speed-ups in the Bayesian setting (*i.e.* Bayesian Logistic Regression), where posterior inference algorithms are computationally demanding. However, these computing-time gains can easily vanish as we move to a non-Bayesian setting due to the computational cost incurred in clustering.

### 2.2 Logistic Regression

We define Logistic Regression in *the optimisation setting* and hereafter we will refer to the below definition simply as LR. Let $\mathcal{D} := \{(x_n, y_n)\}_{n=1}^{N}$ be our input data, where $x_n \in \mathbb{R}^d$ is a feature vector and $y_n \in \{-1, 1\}$ is its corresponding label. We define the likelihood of observation $y_n$, given some parameter $\theta \in \mathbb{R}^{d+1}$, as $p_{logistic}(y_n = 1|x_n;\theta) := 1/(1+\exp(-x_n \cdot \theta))$ and

$$p_{logistic}(y_n = -1|x_n;\theta) := 1 - \frac{1}{(1+\exp(-x_n\cdot\theta))}$$
$$= \frac{\exp(-x_n\cdot\theta)}{(1+\exp(-x_n\cdot\theta))} = \frac{1}{(1+\exp(x_n\cdot\theta))}. \quad (2)$$

Therefore, we have $p_{logistic}(y_n|x_n;\theta) := 1/(1+\exp(-y_n x_n\cdot\theta))$ for any $y_n$ and define the log-likelihood function $LL_N(\theta|\mathcal{D})$ as in (Shalev-Shwartz and Ben-David, 2014):

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

which is the objective function for the LR problem. The optimal parameter $\hat{\theta}$ can be estimated by maximising $LL_N(\theta|\mathcal{D})$. In other words, we want to minimise $\mathcal{L}_N(\theta|\mathcal{D}) := \sum_{n=1}^{N} \ln(1+\exp(-y_n x_n\cdot\theta))$ over all $\theta\in\mathbb{R}^{d+1}$. Finally, our optimisation problem can be written as:

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

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

A Bayesian approach to Logistic Regression requires us to specify a prior distribution for the unknown parameter $\theta$, $p(\theta)$ and derive the posterior distribution $p(\theta|\mathcal{D})$ by applying Bayes' theorem:

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

Notice that both the Bayesian and optimisation settings require the computation of the likelihood function. However, the Bayesian framework needs to use posterior inference algorithms to estimate $p(\theta|\mathcal{D})$, hence high computational cost incurred as no closed form solution can be found.

Having defined LR and contrasted it to the Bayesian framework, we can now proceed to present the state-of-the-art coreset construction for this particular classification problem.

## 2.3 The Sensitivity Framework

We focus on the state-of-the-art coreset algorithm for LR classification propossed by Huggins *et al.* (Huggins et al., 2016), to which we shall refer as *Coreset Algorithm for Bayesian Logistic Regression* (CABLR). Designed for the Bayesian setting, this algorithm uses random sampling for constructing small coresets that approximate the log-likelihood function on the input data.

Before describing CABLR in detail, however, it is important to shed some light on the roots of this algorithm in order to understand why the method can easily become computationally unfeasible for LR in the optimisation setting as defined in Section 2.2.

To design CABLR, Huggins *et al.* followed a well-established framework for defining coresets commonly know as the *sensitivity framework* (Feldman and Langberg, 2011). This framework provides a systematic general approach for constructing coresets for different instances of clustering problems *e.g.* K-means, K-median, K-line and projective clustering. In essence, this abstract framework consists in reducing coresets to the problem of finding an $\varepsilon$-approximation (Mustafa and Varadarajan, 2017), which can be computed using non-uniform sampling. Hence, the sensitivity framework falls under the random-sampling category for defining coresets. The non-uniform distribution is based on the importance of each data point, in some well-defined sense [2]. Hence, each point in the input data is assigned an importance score, *a.k.a.* the sensitivity of the point. To compute such importance scores, the framework requires an approximation to the optimal clustering of the input data. Then, for each input 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, to each of the $M$ points in the coreset, a positive real-valued *weight* is assigned by taking the inverse of the point's sensitivity score. Hence, the sensitivity framework allows us to design coreset construction algorithms that return a coreset consisting of $M$ weighted points. The theoretical proofs and details can be found in (Feldman and Langberg, 2011).

The sensitivity framework proved to be an extremely powerful tool for constructing coresets for many clustering instances as the time required for computing the necessary clustering approximation is small compared to the time needed for clustering the full input data.

Huggins *et al.* made the first incursion in using the sensitivity framework to design a coreset algorithm for the supervised learning problem of Bayesian Logistic Regression, proving that CABLR indeed produce $\varepsilon$-coresets of size $M$ with probability $1-\delta$. However, their algorithm still depends on approxi-

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

mating the clustering of the input data. This is not a major problem in the Bayesian setting, as posterior inference algorithms are computationally expensive.

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 the input data, even for the minimum number of iteration, can be too expensive. The description of CABLR is shown in Algorithm 1.

```
Input: $\mathcal{D}$: input data, $Q_{\mathcal{D}}$: $k$-clustering of $\mathcal{D}$
       with $|Q_{\mathcal{D}}| := k$, $M$: coreset size
Output: $\varepsilon$-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: an algorithm to construct coresets for Logistic Regression.

**Remark.** *In the description of Algorithm 1, we hide the coreset dependence on the error parameter $\varepsilon$, 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: $\varepsilon \in [0,1]$, the "loss" incurred by coresets, and $\delta \in (0,1)$, the probability that the algorithm will fail to compute 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 $t : [0,1] \times (0,1) \to \mathcal{Z}^+$, with $\mathcal{Z}^+$ being the set of all positive integers, that gives the corresponding coreset size for all possible error values i.e. $t(\varepsilon_1, \delta_1) := M_1$ 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 $\varepsilon_1$. 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.*[3]

Notice that Algorithm 1 implements the sensitivity framework almost as described in Section 2.3: $k$ cluster centres, $\mathcal{Q}$, from the input data are used to compute the sensitivities; then, sensitivities are normalised and points get sampled; finally, the weights, which are inverse proportional to the sensitivities, are computed for each of the sampled points. Thus, even though the obtained coreset is for LR, CABLR still needs a clustering of the input data as is common for any coreset algorithms designed using the sensitivity framework.

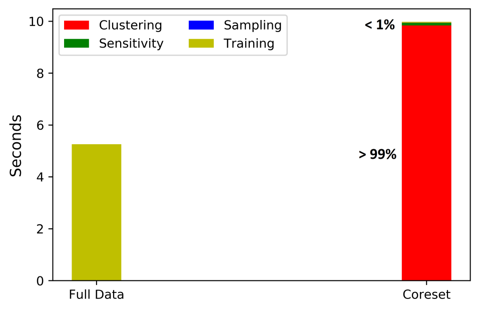
## 2.4 The Clustering Bottleneck

Clustering is known to be a computationally hard problem (Arthur and Vassilvitskii, 2007). This is why approximation and data reduction techniques are useful for speeding up existing algorithms. The sensitivity framework, originally proposed for designing coresets for clustering problems, requires a sub-optimal clustering of the input data $\mathcal{D}$ in order to compute the sensitivity for each input point. This requirement transfers to CABLR, described in the previous section. In the Bayesian setting, the time necessary for clustering $\mathcal{D}$ is strongly dominated by the cost of posterior inference algorithms (see (Huggins et al., 2016)). However, if we remove the burden of posterior inference and consider the optimisation setting, then the situation is dramatically different.

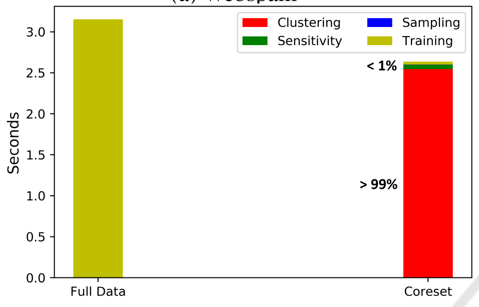
Figure 1 sheds some light on the clustering time with respect to all the other steps taken by CABLR to construct a coreset, namely: *sensitivity computation* and *sampling*. The time spent on *learning* from the coreset is also included.

We can clearly see that obtaining the clustering can be dangerously impractical for constructing coresets 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.

[3]For CABLR, Huggins *et al.* proved that $t(\varepsilon, \delta) := \lceil \frac{c\bar{m}_N}{\varepsilon^2}[(D+1)log\,\bar{m}_N + log(\frac{1}{\delta})]\rceil$, 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 $t$.



(a) Webspam



(b) Covertype

Figure 1: Comparison of the computing time needed for constructing LR coresets for the Webspam and Covertype datasets. The clustering process consumed the majority of the processing time.

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

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*.

## 3 PROPOSED 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. 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 (Bachem et al., 2017). 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

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

Algorithm 2: ACvS procedure.

To recap, we are interested in efficiently learning a good linear classifier using Logistic Regression in the optimisation setting, as defined in Section 2.2. Furthermore, we want the learning process to be as computationally efficient as possible. Coresets allow us to achieve this, at the cost of compromising on the quality of the resulting classifier. The good news is that coresets maintain this quality loss bounded.

For LR classification, however, the state-of-the-art algorithm to compute a coreset, CABLR, is not adequate for our optimisation setting. Specifically, the computing time required to compress the input data overshadows the time needed for learning from the compressed data. We identified that the source of such inefficiency comes from the clustering step and hence we reduced the problem of speeding-up CABLR to the problem of accelerating the clustering phase.

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, say we have a large 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, and 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 designed 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 $cost(\mathcal{D},Q) := \frac{1}{n}\sum_{x\in\mathcal{D}} f_Q(x)$.

Many 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 $x$ can belong to multiple subsets $S \in \mathbb{S}$, we can quickly obtain that the count has to be $\binom{(n-1)}{(m-1)}$. Then, we can 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

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

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 2.3, 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 $\mathcal{D}$ 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

```
Input: $\mathcal{D}$: input data, $A$: clustering
       algortithm, $k$: number of cluster
       centres, $b \ll |\mathcal{D}|$: number of samples
       in the summary, $M$: 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 (RDSF) uses a coreset construction to produce coreset-based summaries of data.

summarising data as the problem of ‘learning’ the sensitivity of the input points. The result of such learning process is a trained regressor $\phi$ 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).

It is worth noting that we avoid using the term *coreset* for RDSF’s output; this is because, strictly speaking, the term coreset implies that a theoretical guarantee of the kind presented in formula (1) is proven. We shall see in the next section that the summaries produced by RDSF indeed perform as good as coresets; however, the prove of theoretical guarantees for our summaries are left as a future work.

## 4 EXPERIMENTS & RESULTS

In this section, we present our empirical evaluations. It is worth mentioning that we investigate coresets (and coresets-based approaches) using a set of metrics that are standard in machine learning, and to the best of our knowledge, no previous research work shows how coresets respond to this kind of evaluation.

### 4.1 Description

For illustration purposes, we tested our procedures on 3 datasets shown in Table 1. These chosen datasets are publicly available [4] and well-known in the coreset community.

Table 1: Overview of the datasets used for the evaluation of our methods.

| Dataset | Examples | Features |
|---|---|---|
| Webspam | 350,000 | 254 |
| Covertype | 581,012 | 54 |
| Higgs | 11,000,000 | 28 |

We are interested in analysing the following five different approaches:

- **Full:** no coreset or summarisation technique is used. We simply train a LR model on the entire training set and predict the labels for the new instances in the test set.
- **CABLR:** we make use of coresets via the CABLR algorithm (Algorithm 1). Thus, we obtain a clustering of the training data and run CABLR to obtain a coreset. We then train a LR model on the coreset to predict the test labels.

[4]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/ - last accessed in 2/2020.

- **ACvS:** we use our procedure 'Accelerated Clustering via Sampling', described in Algorithm 2, to accelerate the coreset computation. Once we have obtained the coreset in the accelerated fashion, we proceed to learn a LR classifier over it.
- **RDSF:** summaries of data are generated via the 'Regressed Data Summarisation Framework', described in Algorithm 3. That is, we compute sensitivity scores only for a handful of instances in the training set. Then, we train a regressor to predict the sensitivity scores of the rest 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 settings.
- **URS:** for the sake of completeness, we include 'Uniform Random Sampling' as a baseline for reducing the volume of input data; we simply pick $M$ input points uniformly at random and then train a LR classifier over them.

For each of the above approaches, and for each dataset in Table 1, we take the following steps:

(a) **data shuffling:** we randomly mix up all the available data.

(b) **data splitting:** we randomly select 50% of the data to use as training set and leave the rest for testing. It is the training set to which we refer as input data $\mathcal{D}$.

(c) **data compressing:** we proceed to compress the input data: *CABLR* approach computes a coreset without any acceleration, *ACvS* computes a coreset by using CABLR with accelerated clustering phase, *RDSF* compresses the input data into a small summary data in an accelerated fashion, and *URS* performs a naive compression by taking an uniform random sample of the input data. *Full Data* is the only approach not relying on compressing the input data.

(d) **data training:** we train a Logistic Regression classifier on the input-data compression obtained in the previous step. Again, the *Full Data* approach trains the classifier on the full input data $\mathcal{D}$.

(e) **data assessing:** we finally use the trained Logistic Regression classifier to predict the labels in the test set and apply our performance metrics, detailed in Section 4.2.

We applied the above strategy 10 times for each of the five different approaches. 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.

Lastly, for coresets, we adapted to our needs the CABLR algorithm implementation and shared by its authors [5]. All our programs were written in Python. The algorithm used for clustering the input data is the well-known K-means algorithm and, finally, RDSF uses linear regression to learn sensitivities.

## 4.2 Evaluation

As we previously mentioned, the empirical performance of coresets has mainly remained as 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. The following 5 performance metrics are considered:

- **Computing time (seconds):** our main goal is to accelerate the coreset computation to consequently speed-up the learning process when coreset-based approaches are applied. We distinguish 5 phases for which we measure time:
  (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 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.
- **Classification Accuracy:** is given by the percentage of correctly classified test examples and it is usually the baseline metric for measuring performance.
- **F1 Score:** is the harmonic average of precision and recall (Goutte and Gaussier, 2005).

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

- **Area Under the ROC Curve (AUROC):** provides an aggregate measure of performance across all possible classification thresholds.

## 4.3 Results

We categorise our results according to the four different metrics considered. Notice that our performance metrics are shown as functions of the size of the summaries used for training the LR classifier. For example, if we look at Figure 5, we see that summaries correspond to very small percentages of the Higgs dataset *i.e.* 0.005 %, 0.03 %, 0.06 %. and 0.1 %. For Webspam and Covertype, we show results for summary sizes of 0.05 %, 0.1 % , 0.3 %, 0.6 % and 10 % of the total input data.

Due to limited space, and given that the performance of all the methods follows similar patterns with respect to accuracy, F1 socre and AUROC, we only show figures for one (random) dataset for these particular metrics; however, the tables in Section 4.4 show results for all datasets in a more detailed fashion.

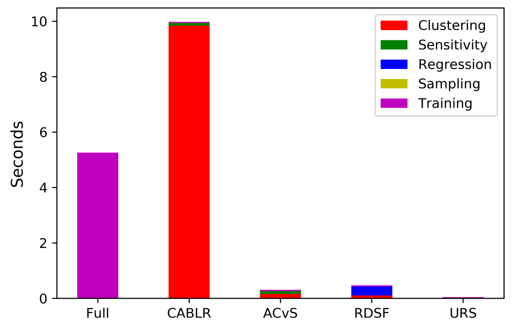
### 4.3.1 Computing Time

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.

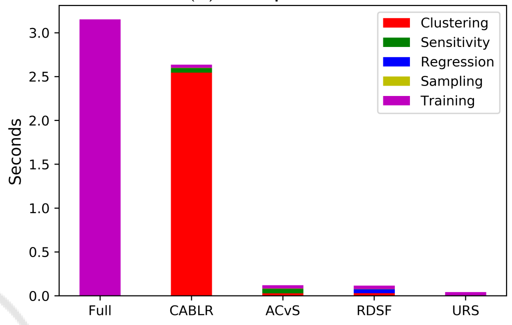
We can clearly see that the *CABLR* approach, which applies coresets in their original form, is not suitable for the optimisation setting we are considering. Specifically, with respect to the Full method, we notice that for the Covertype dataset, the coreset approach gives a modest acceleration of approximately 1.2 times. The situation becomes more severe for the Webspam and Higgs datatsets, where using the traditional coreset approach incurs in a learning process which is about 1.9 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 both the *CABLR* and *Full Data* approaches. In particular, and with respect to Full Data approach, our approach *ACvS* achieves a minimum acceleration of 17 times (Webspam) and a maximum acceleration of 34 times (Higgs) across our datasets. Regarding *RDSF*, the minimum acceleration obtained was 11 times (Webspam) and the maximum was 27 times (Covertype).

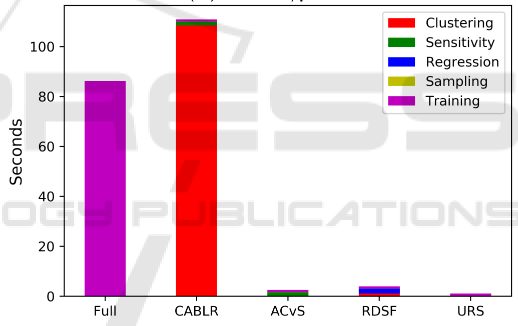
Notice that our accelerated methods are only beaten by the naive URS method, which should most certainly be the fastest approach.



(a) Webspam



(b) Covertype



(c) Higgs

Figure 2: Comparison of the computing time of different methods.

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*.

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

### 4.3.2 Accuracy

We first look into the baseline metric for measuring the success of a classifier: the accuracy. Tables 2, 3

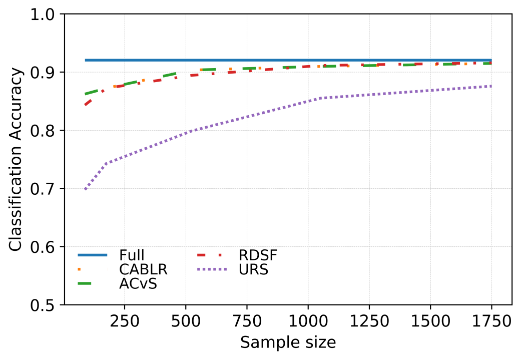


Figure 3: Comparison of the prediction accuracy for all methods on the Webspam dataset.

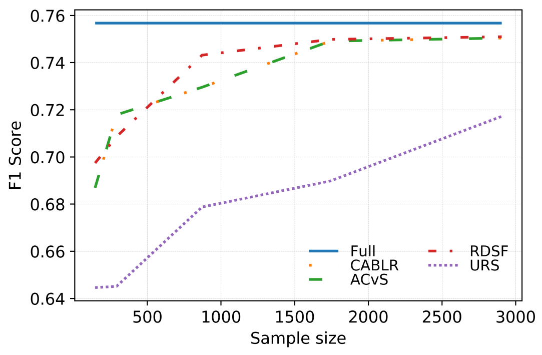


Figure 4: Comparison of the F1 score for all methods on the Covertype dataset.

and 4 show 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, see Figure 3 for an example.

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 subsample 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. On this regard, it is common for coreset to be *lower bounded* in performance by the URS approach: that is, we expect coresets to outperform URS in most situations. 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).

### 4.3.3 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 very first observation is that, similar to the previous metric 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 dataset, RDSF has preferable performance compared to the rest of the coreset-based 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.

Finally, 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 trained over coresets are more useful and informative than the ones trained over uniformly randomly selected samples.

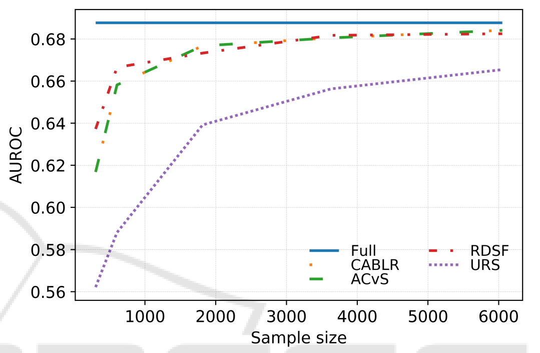


Figure 5: Comparison of the area under the ROC curve for all methods on the Higgs dataset.

### 4.3.4 AUROC

Lastly, we 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 as well as the traditional coreset approach, achieving their performance in substantially less computing time than coresets (see Section 4.3.1).

### 4.4 Summary of Results

This section provides detailed performance metrics of our proposed methods on the Covertype, Webspam, and Higgs datasets (see Tables 2, 3 and 4). Overall, the empirical results demonstrated that our proposed ACvS and RDSF run significantly faster than the traditional Coreset approach, while maintaining competitive performance, in all datasets considered.

Table 2: Performance comparison on the Covertype dataset where size is the percentage of training data. "M" stands for "Methods", "A" stands for "Accuracy" and "T" stands for "Time".

| Size (%) | M | F1 | AUROC | A | T(secs) |
|---|---|---|---|---|---|
| 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 |

Table 3: Performance comparison on the Webspam dataset where size is the percentage of training data. "M" stands for "Methods", "A" stands for "Accuracy" and "T" stands for "Time".

| Size (%) | M | F1 | AUROC | A | T(secs) |
|---|---|---|---|---|---|
| 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 |

## 5 CONCLUSION

Under the presence of ever growing input data, it is absolutely necessary to accelerate the learning time of machine learning algorithms. Instead of improving the algorithms, there are less direct approaches that involve reducing the input data and hence learn from smaller data. Coreset approach is one of such methods. As coresets were originally proposed to be used to solve computationally hard problems, like clustering, they should be carefully applied in computationally-less-demanding scenarios, like classification. However, it is not obvious how to do this. We proposed two methods that can bring all the benefits of coresets to Logistic Regression classification in the optimisation setting: 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.

Our results clearly show that coresets can be used to learn a logistic regression classifier in the optimisation setting. It is enlightening to observe that, even though CABLR needs a clustering of the input data, this can be largely 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.* our ACvS approach). Regarding RDSF, this opens a new direction for perceiving coresets: we could pose the computation of data summary as solving a small-scale learning problem in order to solve a large-scale one. It is illuminating to see that the sensitivities can be largely explained by a simple linear regressor. Most importantly, we believe that this method may be very powerful for *the online learning setting* (Shalev-Shwartz et al., 2012), since RDSF returns a trained regressor capable of assigning sensitivity scores to new incoming data points. We leave as future work the use of these methods in different machine learning contexts.

Table 4: Performance comparison on the Higgs dataset where size is the percentage of training data. “M” stands for “Methods”, “A” stands for “Accuracy” and “T” stands for “Time”.

| Size (%) | M | F1 | AUROC | A | T(secs) |
|---|---|---|---|---|---|
| 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 |

## ACKNOWLEDGMENTS

We thank the reviewers for their helpful and thorough comments. Our research is partially supported by Astra Zeneca and the Paraguayan Government.

## REFERENCES

Ackermann, M. R., Märtens, M., Raupach, C., Swierkot, K., Lammersen, C., and Sohler, C. (2012). Streamkm++: A cluste ing algorithm for data streams. *Journal of Experimental Algorithmics (JEA)*, 17:2–4.

Arthur, D. and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

Bachem, O., Lucic, M., and Krause, A. (2017). Practical coreset constructions for machine learning. *arXiv preprint arXiv:1703.06476*.

Feldman, D. and Langberg, M. (2011). A unified framework for approximating and clustering data. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 569–578. ACM.

Feldman, D., Schmidt, M., and Sohler, C. (2013). 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*, pages 1434–1453. SIAM.

Goutte, C. and Gaussier, E. (2005). A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *European Conference on Information Retrieval*, pages 345–359. Springer.

Har-Peled, S. and Mazumdar, S. (2004). On coresets for k-means and k-median clustering. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 291–300. ACM.

Huggins, J., Campbell, T., and Broderick, T. (2016). Coresets for scalable bayesian logistic regression. In *Advances in Neural Information Processing Systems*, pages 4080–4088.

Munteanu, A., Schwiegelshohn, C., Sohler, C., and Woodruff, D. (2018). On coresets for logistic regression. In *Advances in Neural Information Processing Systems*, pages 6561–6570.

Mustafa, N. H. and Varadarajan, K. R. (2017). Epsilon-approximations and epsilon-nets. *arXiv preprint arXiv:1702.03676*.

Phillips, J. M. (2016). Coresets and sketches. *arXiv preprint arXiv:1601.00617*.

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

Shalev-Shwartz, S. et al. (2012). Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194.

Zhang, Y., Tangwongsan, K., and Tirthapura, S. (2017). Streaming k-means clustering with fast queries. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 449–460. IEEE.