Efficient Construction of Approximate Ad-Hoc ML models Through Materialization and Reuse

Sona Hasani‡, Saravanan Thirumuruganathan‡†, Abolfazl Asudeh†,
Nick Koudas‡‡, Gautam Das‡
‡University of Texas at Arlington; ‡†QCRI, HBKU; †University of Michigan; ‡‡University of Toronto
‡\{sona.hasani@mavs, gdas@cse\}.uta.edu, ‡†sthirumuruganathan@hbku.edu.qa,
†asudeh@umich.edu, ‡‡koudas@cs.toronto.edu

ABSTRACT

Machine learning has become an essential toolkit for complex analytic processing. Data is typically stored in large data warehouses with multiple dimension hierarchies. Often, data used for building an ML model are aligned on OLAP hierarchies such as location or time. In this paper, we investigate the feasibility of efficiently constructing approximate ML models for new queries from previously constructed ML models by leveraging the concepts of *model materialization* and *reuse*. For example, is it possible to construct an approximate ML model for data from the year 2017 if one already has ML models for each of its quarters? We propose algorithms that can support a wide variety of ML models such as generalized linear models for classification along with K-Means and Gaussian Mixture models for clustering. We propose a cost based optimization framework that identifies appropriate ML models to combine at query time and conduct extensive experiments on real-world and synthetic datasets. Our results indicate that our framework can support analytic queries on ML models, with superior performance, achieving dramatic speedups of several orders in magnitude on very large datasets.

PVLDB Reference Format:
Sona Hasani, Saravanan Thirumuruganathan, Abolfazl Asudeh, Nick Koudas and Gautam Das. Efficient Construction of Approximate Ad-Hoc ML models Through Materialization and Reuse. *PVLDB*, 11 (11): 1468-1481, 2018.
DOI: https://doi.org/10.14778/3236187.3236199

1. INTRODUCTION

Machine Learning (ML) has become an invaluable tool that is used by organizations to glean insight from their data. Almost all the major database vendors have added analytical capabilities on top of their database engines. Even though there has been extensive work from the ML community on developing faster algorithms, building a ML model is often a major bottleneck and consumes a lot of time due to the sheer size of the datasets involved. In this paper, we investigate the feasibility of building faster ML models for a popular class of analytic queries by leveraging two fundamental concepts from database optimization - *materialization* and *reuse*.

1.1 Analytic Queries on ML Models

Recently, there has been extensive interest in the database community for enabling interactive ad-hoc analytics on ML models. Consider a typical workflow of a data scientist. She issues a query (SQL or otherwise) to retrieve relevant data that is stored in a data warehouse. This data is used to build an ML model for classification, clustering, etc. The model is then used for performing complex analytic processing such as predicting customer churn in a geographic region. This process of retrieving data, building a ML model and using the model for analytic processing subsumes a large class of analytic workflows. We argue that these ad-hoc analytic queries on ML models often exhibit a number of appealing properties that enables a better and faster approach than building models from scratch every time. Some of these properties include:

- *Meaningful SQL Predicates.* The queries that are used to retrieve relevant data are not chosen at random and often have a specific business interpretation. Data warehouses often impose OLAP hierarchies and most of the analytic queries are aligned along the hierarchy. The data chosen for analysis often belongs to explicit domain hierarchies over country, year, department, vendor, product category, etc. For example, the domain scientist might want to retrieve data for years 2018/2017 or for continents Asia/Europe/North America, etc. Building models on an arbitrary subset of the data is typically rare.
- *Tolerance for Approximate ML Models.* Building a ML model takes a lot of time for large datasets which is inconvenient when it is primarily used for exploratory analysis. Data scientists are often willing to sacrifice some accuracy of exploratory analysis if they can obtain "close enough" estimates from approximate ML models quickly.
- *Opportunities for ML Model Reuse.* In a typical enterprise, data scientists and engineers often create hundreds to thousands of ML models for exploratory purposes that are then discarded after one-time use. If a data scientist needs to build an ML model for all the data from year 2017, it is likely that some other data scientist(s) has created ML models for the various

Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07.
DOI: https://doi.org/10.14778/3236187.3236199

quarters of 2017. If these models have been *materialized* (instead of being discarded), then one can build an approximate ML model for 2017 by *reusing* the models for the various quarters of 2017.

1.2 Technical Challenges

There are a number of technical challenges that one must overcome before ML models are reused for building approximate ML models for exploratory purposes. While there has been extensive work on building a ML model efficiently, there is a relative paucity of work in combining multiple pre-built ML models. Consider a straightforward scenario whereby the data is already partitioned and both supervised (*e.g.,* SVMs) and unsupervised (*e.g.,* K-Means) models have been built for each partition. Given a set of partitions and their corresponding SVMs, how can one construct a single SVM that performs comparably to one that is built from scratch on the combined data from the partitions? Similarly, given a set of K-Means centroids for each of the partitions, is it possible to approximately compute K-Means centroids for the union of the partitions? Further, is it possible to give any theoretical guarantees for the approximate ML model? How can we trade-off time and space to get an ML model with a better approximation? Is there a cost model that enables to decide when building an ML model from scratch is preferable to combining pre-existing ML models? Is it possible to come up with an optimization framework that decides which models to reuse, how to combine those models with minimum cost? Given an analytic workload and a space budget, is it possible to identify ML models to materialize to achieve significant speedup to later queries?

1.3 Outline of Technical Results

In this paper, we advocate treating ML models as first class citizens and investigate opportunities for model materialization and reuse to speed up analytic queries. We propose a two-phase approach. We store ML models along with small amount of additional meta-data and statistics during a "pre-processing phase"; During the "runtime phase", we identify the relevant ML models to reuse and quickly construct an approximate ML model from them.

In this paper, we investigate reuse of popular supervised and unsupervised ML models. In supervised learning, we consider Generalized Linear Models (GLMs) that subsumes many popular classifiers such as logistic regression and linear SVMs. Note that our approach extends to any ML algorithm that uses Stochastic Gradient Descent (SGD) for training. In unsupervised learning, we consider two canonical clustering approaches: K-Means and Gaussian Mixture Models (GMMs). For each of them, we propose two orthogonal approaches for generating approximate ML models.

- **Model Merging.** In this approach, we store some additional metadata during the pre-processing phase such that during the run-time phase, one can combine the ML models in a principled manner without going back to data.
- **Coresets.** Coresets are a small weighted set of tuples such that ML models built from the coresets are *provably closer* to ML models built on the entire data. During the pre-processing phase, one can construct coresets for each pre-built model. During the run-time phase, we build the ML model from the union of coresets in a fraction of time.

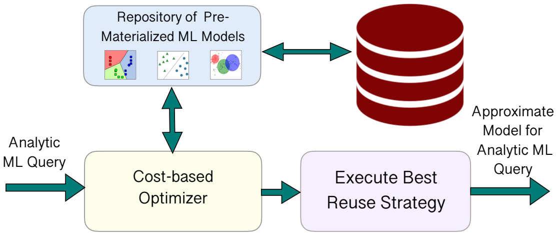

Repository of Pre-Materialized ML Models
Analytic ML Query
Cost-based Optimizer
Execute Best Reuse Strategy
Approximate Model for Analytic ML Query

Figure 1: Overview of Our Approach

These two approaches enable a data analyst to trade-off performance and model approximation. The merging based approach is often extremely fast but does not provide tunable approximation of the objective function. On the other hand, the coreset based approach might take more time (though much less than re-training from scratch) but is more flexible and allows one to approximate the objective function within a factor of ϵ.

There has been increasing interest from the database community on building systems for ML model management (see Section 8 for further details). Our approaches can easily be retrofitted over these systems to facilitate rapid construction of approximate ML models. We further discuss our potential limitations in Section 7.

2. BACKGROUND

Dataset. Let D denote a relation with n tuples and d attributes $A = \{A_1, \ldots, A_d\}$. We partition the schema A into X, Y where X is the set of predictor/independent attributes and Y the predicted/dependent attribute(s). The schema also has a set of dimension attributes $Z = \{Z_1, \ldots, Z_l\}$ that are associated with pre-defined dimensional hierarchies. Each tuple t_i is also associated with a unique identifier *tid* that imposes a total ordering in D. As an example, *tid* could be an automatically incrementing sequence or timestamp indicating when the tuple was created. For example, an ML model for credit card approval might have $X = \{$*Age, Gender, Salary,Education, City*$\}$ with Y=$\{$*Approval*$\}$. The dimensional attribute $Z = \{$*City*$\}$ is associated with the hierarchy *City* ⇒ *State* ⇒ *Country* ⇒ *Continent* ⇒ *All*.

Query Model. Let q be the analytic query specified on D that returns a result set D_q over which the ML model is built. We consider the following types of queries that subsumes most queries used for model building.

- *Range based Predicate:* These queries are specified by an attribute X_i and range $[a, b]$ such that they filter all tuples with value of X_i falling between a and b.
- *Dimension based Predicate:* These queries filter tuples that have specific values for one or more dimensional attributes. Using the example above, the predicate `State = 'Texas'` filters all credit card applications from Texas.
- *Arbitrary Predicates:* These queries use complex query predicates (including a combination of range and dimension based) to select relevant data.

Pre-Materialized Models. We denote the *exact* model built on D_q as $M(D_q)$ while its approximation as $\widetilde{M}(D_q)$.

We assume the availability of pre-materialized *exact* models $\{M_1, M_2, \ldots, M_R\}$ built from previous analytic queries. Each of these models is annotated with relevant information (such as `State = 'Texas'`). Given an arbitrary query q, let $\mathcal{M}_q \subseteq \mathcal{M}_D$ be the set of pre-built models that could be used to answer it approximately where $|\mathcal{M}_q| = r$.

Example. Consider a database $D = \{1, \ldots, 1000\}$ where we have a set of built ML models $\{M_1, \ldots, M_{10}\}$ over ranges $\{P_1 = [1, 100], P_2 = [101, 200], \ldots, P_{10} = [901, 1000]\}$. Given a query $q_1 = [101, 500]$, then $\mathcal{M}_{q_1} = \{M_2, M_3, M_4, M_5\}$. If necessary, one can build appropriate models for tuples from D_q for which no pre-built models exist. Given a query $q_2 = [51, 550]$, the set of models to answer them will be $\mathcal{M}_{q_2} = \{\mathcal{M}([51, 100] \cup [501, 550]), M_2, M_3, M_4, M_5\}$

2.1 ML Primer

K-Means. K-Means is a widely used clustering algorithm that partitions data into K clusters. Formally, given a set of points $\mathcal{X} \in \mathbb{R}^d$, the K-Means clustering seeks to find K cluster centers in $\mathbb{R}^d$ (also called as centroids) such that the sum of squared errors (SSE) is minimized [11]. Given a set of data points $\mathcal{X}$ and centroids C, the SSE is defined as

$$SSE(\mathcal{X}, C) = \sum_{x \in \mathcal{X}} d(x, C)^2 = \sum_{x \in \mathcal{X}} min_{c \in C} ||x - c||_2^2 \quad (1)$$

Even though clustering with K-Means objective is known to be a NP-Complete problem, there are a number of efficient heuristics and approximation algorithms. The most popular heuristic algorithm is Lloyd's algorithm. It works by randomly choosing K initial centroids from $\mathcal{X}$. Each point $x \in \mathcal{X}$ is assigned to the nearest cluster centroid. Then, the cluster centroid is updated as the mean of the points assigned to the cluster. This process of cluster assignment and centroid update is repeated till the change in cluster centroids between iterations is below some threshold.

Gaussian Mixture Models (GMM). GMM is one of the most popular mixture models used for unsupervised clustering. GMM models the data in terms of mixtures of multiple components where each component is a multi-variate Gaussian distribution. A multi-variate Gaussian distribution generalizes the one-dimensional Gaussian distribution (specified by a mean and variance) to higher dimensions and is specified by a mean vector μ of dimension d and a covariance matrix Σ of dimension $d \times d$. GMM is a probabilistic/soft version of K-Means where each data point could be assigned to multiple clusters with different probabilities.

Suppose we are given a set of d-dimensional data points $\mathcal{X} = \{x_1, x_2, \ldots, x_n\} \subseteq \mathbb{R}^d$. We fit $\mathcal{X}$ as Gaussian mixture model parameterized by $\theta = [(w_1, \mu_1, \Sigma_1), (w_2, \mu_2, \Sigma_2), \ldots, (w_k, \mu_K, \Sigma_K)]$ where the i-th mixture component is a d-dimensional multi-variate Gaussian $\mathcal{N}(\mu_i, \Sigma_i)$ with w_i being its prior probability. Note that the prior probabilities of the components sum up to 1 - i.e. $\sum_{i=1}^{K} w_i = 1$. Given the data $\mathcal{X}$, GMM estimates the parameters θ that maximizes the likelihood through the Expectation-Maximization (EM) algorithm.

Generalized Linear Models (GLM). GLM covers a large class of popular ML models including logistic regression (LR), support vector machines (SVM). Due to their widespread applicability and popularity, GLMs have been extensively studied and shown to have a number of appealing theoretical properties. They have natural convex optimization formulations wherein every local minima is also a global minima. While we restrict our attention to popular supervised ML models, we would like to note that our methods described in this section can be easily adapted for other GLMs such as linear regression and other log-linear models.

Coresets. A coreset is a weighted subset of the data such that an ML model built on the coreset very closely approximates one built on the entire data [2]. Specifically, a weighted set C is said to be a ϵ-coreset for dataset D if $(1 - \epsilon)\phi_D(\cdot) \leq \phi_C(\cdot) \leq (1 + \epsilon)\phi_D(\cdot)$ where $\phi(\cdot)$ corresponds to the objective function of a model - such as Sum of Squared Errors (SSE) for K-Means. The SSE for the cluster centroids obtained by running K-Means algorithm on the coreset is within a factor of $(1 + \epsilon)$ of SSE obtained by running K-Means on the entire data.

3. APPROXIMATION BY MODEL MERGING

In this section, we investigate how to construct approximate ML models for a query q by merging pre-built (exact) ML models. Specifically, we focus on scenarios where we can construct the approximate model purely from the pre-built models *without retrieving* data D_q. Our proposed approach has a number of appealing properties such as: (a) orders of magnitude faster than building the model from scratch; (b) provable guarantees on approximation; (c) minimal sacrifice of model accuracy.

Pre-built ML Models. Let $\mathcal{M}_q = \{M_1, M_2, \ldots, M_r\}$ be the set of pre-built ML models that must be merged to obtain the approximate ML model $\widetilde{M}(D_q)$. Let $\theta(M_i)$ be the relevant parameters of model M_i that must be materialized. This information is dependent on the ML algorithm. For K-Means, $\theta(M_i)$ is the set of K centroids and the number of data points assigned to each of the clusters. For GMM, $\theta(M_i) = [(w_1, \mu_1, \Sigma_1), (w_2, \mu_2, \Sigma_2), \ldots, (w_K, \mu_K, \Sigma_K)]$ where the i-th mixture component is a d-dimensional multi-variate Gaussian $\mathcal{N}(\mu_i, \Sigma_i)$ with w_i being its prior probability. For GLM such as Logistic Regression, $\theta(M_i)$ corresponds to the regression coefficients while for SVM, it corresponds to the coefficients of the separating hyperplane.

3.1 Model Merging for K-Means

Given an arbitrary query q, our objective is to efficiently output K centroids $\widetilde{C}_q$ such that SSE for $\widetilde{C}_q$ is close to SSE of C_q where C_q is the set of centroids obtained by running K-Means algorithm from scratch on the entire D_q. We seek to do this by only using the information $\theta(M_i)$ - the cluster centroids and the number of data points assigned to it.

K-Means++ [6] is one of the most popular algorithms for solving K-Means clustering. It augments the classical Lloyd's algorithm with a careful randomized seeding procedure and results in a $O(\log K)$ approximation guarantee. Due to its simplicity and speed, K-Means++ has become the default algorithm of choice for K-Means clustering. Hence, we assume that all the cluster centroids were obtained through the K-Means++ algorithm.

Let C_w represent the union of all cluster centroids from all the models $M_i \in \mathcal{M}_q$. As before, if there were some tuples in D_q that were not covered by models $\mathcal{M}_q$, one can readily run K-Means on those tuples and add those cluster centroids to C_w. For each centroid $c_j \in C_w$, we assign the number of data points associated with it in the original partition as its weight $w(j)$. We then run the weighted variant of K-Means++ algorithm on C_w and return the K cluster centroids as the output. If the centroids were obtained using some other algorithm, our algorithm proposed below still works as an effective heuristic but does not provide any provable approximation guarantees. Algorithm 1 provides the pseudocode of the approach while Figure 2 provides an illustration.

Algorithm 1 Merging K-Means Centroids

```
Input: Set of ML models $M_q$, K
$C_w = \cup_{i=1}^{r}$ K-Means centroids for $M_i$
$\forall$ clusters $c_j \in C_w$, $w(c_j)$ = number of data points assigned to $c_j$
Run weighted K-Means++ on $C_w$
return the cluster centroids $\widetilde{C}_q$
```

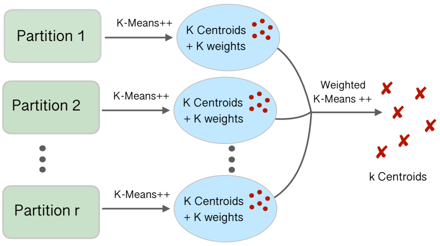


Figure 2: Illustration of Two Level K-Means Merging Approach

Complexity Analysis. The time complexity is $O(n' \times K \times d \times L)$ where n' is the number of cluster centroids and L is the number of iterations required K-Means++ before convergence. Since the number of clusters are much smaller than the number of data points, the clustering results can be obtained extremely fast. In order to run Algorithm 1, we only need to store the cluster centroids for each of the partitions that requires $O(K \times d)$ space.

THEOREM 1. *The SSE of cluster centroids produced by Algorithm 1 has an approximation ratio of $O(\log K)$ to the SSE of cluster centroids C_q obtained by running K-Means++ on D_q. Furthermore, they also have an approximation ratio of $O(\log^2 K)$ to the SSE of the optimal cluster centroids C_q^*.*

Our theorem can be proved by directly adapting the proofs from [21, 4]. Please refer to the appendix of [3] for the proof. The only difference is that we use K-Means++ for both the stages. Since K-Means++ provides an bi-criteria approximation of $(O(\log K), O(1))$, the proof directly follows from [3].

3.2 Model Merging for GMM

We next investigate the problem of reusing pre-built Gaussian mixture models to efficiently answer other GMM based ML queries. Given a query q, we assume the availability of pre-built ML models $\mathcal{M}_q = M_1, \ldots, M_r$ that are parameterized by $\theta(M_j) = [(w_{j1}, \mu_{j1}, \Sigma_{j1}), \ \ldots, (w_{jK}, \mu_{jK}, \Sigma_{jK})]$. We seek to post-processes the Gaussian mixtures obtained from each partition to approximate the GMM on D_q. There are totally $K \times r$ Gaussian components that we must process to just K components.

Table 1: Summary of Notations

Symbol	Description
D_q	Data selected by query q
$M(D_q)$	Model trained on entire data from scratch
$\widetilde{M}(D_q)$	Approximate model by merging
$M_1, \ldots, M_r$	Pre-built models for constructing $\tilde{M}(D_q)$
$D_1, \ldots, D_r$	Data used to train model M_i
C_q	Clusters centers through K-Means++ on D_q
$\widetilde{C}_q$	Cluster centers through merging
C_q^*	Optimal cluster centers for data D_q
C_i	Cluster centers through K-Means++ for D_i. i.e. $C_i = \{c_{i1}, \ldots, c_{iK}\}$
C_w	Union of cluster centers C_i with number of tuples in cluster c_{ij} as its weight $w(c_{ij})$
$NC(C_i, x)$	Nearest cluster center in C_i to x
$d(x, y)$	Euclidean distance between x and y
w_i, μ_i, Σ_i	Prior probability, mean vector and covariance matrix of a GMM component

Ineffective Approaches. The approach that we used for merging K-Means models does not work here. The output of the K-Means algorithm can be parameterized by the centroids that are simply vectors and can be re-clustered. In contrast, the output of GMM is a Gaussian mixture where each Gaussian distribution in it is parameterized by mean vector, covariance matrix and a prior probability. Given a set of data points, GMM works by estimating the parameters of a Gaussian mixture that maximizes the likelihood. While the likelihood that a point is generated by a Gaussian distribution is straightforward to compute, the likelihood that a Gaussian distribution generated another is not.

Another approach is to try some clustering algorithm other than GMM such as K-Means. We begin by randomly choosing K distributions as initial centroids. Using the Bhattacharya distance, we can easily identify the closest centroid for each Gaussian distribution. We could also re-compute the centroids by averaging the Gaussian distributions. However there are two issues with this approach: (a) the process of merging multiple Gaussian distributions to one is very expensive and (b) the resulting Gaussian distributions could be arbitrarily far away from the ones that we could have obtained by running GMM from scratch.

Iterative Merging of GMM Components. The key idea is to use another popular clustering algorithm - hierarchical clustering. We begin by normalizing the prior probabilities of all the Gaussian mixtures by $w_{j_i} = \frac{w_{j_i}}{Z}$ where $Z = \sum_{j=1}^{r} \sum_{i=1}^{K} w_{j_i}$. One can also use a sophisticated normalization technique such as those described in [49]. We can consider the problem of obtaining GMM for D_q as analogous to constructing a *mixture* of Gaussian mixture models. This can be achieved by iteratively merging two Gaussian components till only K of them are left. Algorithm 2 provides the pseudocode and Figure 3 an illustration.

Selecting Components to Merge. One of the key steps in Algorithm 2 is the selection of two Gaussian components to merge. There has been extensive work in statistical com-

Algorithm 2 Iterative Merging of Gaussian Components

```
Input: Set of ML models $M_q$, K
$\mathbf{T} = \cup_{i=1}^{r}$ Gaussian mixture components of $M_i$
Normalize the weights of all GMM in $\mathbf{T}$
while number of components $> K$ do
    Merge the two most similar Gaussian components
    Recompute the parameters of the merged components
return the parameters of the Gaussian mixture
```

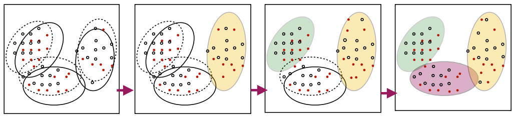


Figure 3: Merging for GMM

munity about appropriate measures to select components for merging [27, 47, 14]. Intuitively, one seeks to select two distributions that are very similar to each other. In our work, we use the Bhattacharyya dissimilarity measure for this purpose and choose the pair of components with least distance between them. Given two multi-variate Gaussian distributions $\mathcal{N}_1(\mu_1, \Sigma_1)$ and $\mathcal{N}_2(\mu_2, \Sigma_2)$, their Bhattacharyya distance is computed as:

$$\begin{aligned} D_B(\mathcal{N}_1, \mathcal{N}_2) &= \frac{1}{8}(\mu_1 - \mu_2)^T \Sigma^{-1} (\mu_1 - \mu_2) \\ &+ \frac{1}{2} \ln \left(\frac{|\Sigma|}{\sqrt{|\Sigma_1||\Sigma_2|}} \right) \\ \Sigma &= \frac{\Sigma_1 + \Sigma_2}{2} \end{aligned} \tag{2}$$

Merging Gaussian Components. Once the two components with the least Bhattacharyya distance has been identified, we merge them into a single Gaussian component while taking into account their respective mixing weights, mean vectors and covariance matrices. Given two multi-variate Gaussian distributions $\mathcal{N}_1(\mu_1, \Sigma_1)$ and $\mathcal{N}_2(\mu_2, \Sigma_2)$ with mixing weights w_1 and w_2, the merged component [27, 47, 14] is described by $\mathcal{N}(\mu, \Sigma)$ with mixing weights w where,

$$\begin{aligned} w &= w_1 + w_2 \\ \mu &= \frac{1}{w}[w_1\mu_1 + w_2\mu_2] \\ \Sigma &= \frac{w_1}{w}\left[\Sigma_1 + (\mu_1 - \mu)^T(\mu_1 - \mu)\right] \\ &+ \frac{w_2}{w}\left[\Sigma_2 + (\mu_2 - \mu)^T(\mu_2 - \mu)\right] \\ &= \frac{w_1}{w}\Sigma_1 + \frac{w_2}{w}\Sigma_2 + \frac{w_1 w_2}{w^2}\left((\mu_1 - \mu_2)(\mu_1 - \mu_2)^T\right) \end{aligned} \tag{3}$$

3.3 Classifier Combination by Parameter Mixtures

In this subsection, we describe an effective approach for merging supervised ML models. As before we are given a query q representing the subset D_q and the corresponding pre-built ML models $\mathcal{M}_q$. Our objective is to post-process the ML models $M_i \in \mathcal{M}_q$ to produce an approximate model $\widetilde{M}(D_q)$ such that it approximates the classifier $M(D_q)$ trained on D_q.

Algorithm 3 shows the pseudocode for the approach. Given a set of pre-built ML models, we average their corresponding model parameters and return that as the model $\widetilde{M}_q$. As we shall show in Section 6, this surprisingly simple algorithm works extremely well for most ML models and especially so for GLMs. This approach can be considered as analogous to distributed statistical inference where we partition the data into a number of chunks, build optimal models for each individually and then in a single round of communication average the parameters.

Algorithm 3 AVGM: Average Mixture Algorithm

```
Input: ML Models $\mathcal{M}_q$ for partitions covering $D_q$
Collect model parameters $\theta(M_i) \quad \forall M_i \in \mathcal{M}_q$
return $\theta(\widetilde{M}_q) = \frac{1}{r}\sum_{i=1}^{r}\theta(M_i)$
```

Complexity Analysis and Approximation Guarantees. Algorithm 3 is a linear time algorithm whose complexity is proportional to the number of models being merged. The parameter averaging method, dubbed Average Mixture (AVGM), has been previously described for a number of ML models such as MaxEnt models including Conditional Random Fields (CRFs) [38], Perceptron-type algorithms [37] and for a larger class of stochastic approximation models in [58]. This algorithm was formally analyzed in [58] and [57]. [58] showed that one of the key advantages of AVGM is that averaging r parameter vectors reduces the variance by $O(r^{-\frac{1}{2}})$. A sharper analysis was provided by [57] that showed the surprising result that this simple approach matches the error rate of the traditional (centralized) approach that builds the model from scratch over D_q. This is achieved under mild conditions such as the number of partitions is less than the data points in each partition - specifically $|\mathcal{P}_q| < \sqrt{|D_q|}$ which holds almost all the time.

4. APPROXIMATION BY CORESETS

The model merging approach proposed in Section 3 is very efficient with the approximate ML suffering from minimal loss in accuracy compared to the ML model built from scratch. However, in many cases, one might desire for tunable guarantees on the degree of approximation of the ML model. An alternate approach to speedup model building is to train the model on a smaller number of data points. However, these data points must be carefully chosen so that they provide a close approximation of the objective function of the ML model trained from scratch. The natural approach of uniform sampling often does not work well in practice or requires very large sample size for sufficient approximation. In this section, we describe how one can leverage the concept of Coresets [2] from computational geometry for arbitrarily approximating the objective function with a smaller number of data points.

Coresets. Coresets provide a systematic approach for sampling tuples proportional to their contribution to the objective function. Recall from Section 2 that a weighted set C is said to be a ϵ-coreset for dataset D if $(1-\epsilon)\phi_D(\cdot) \leq \phi_C(\cdot) \leq (1+\epsilon)\phi_D(\cdot)$. Coresets are a natural solution to the problem of obtaining ML models with tunable approximation - by varying the value of ϵ, we can achieve coresets with higher or lower approximation. Naturally, lower ϵ requires a larger

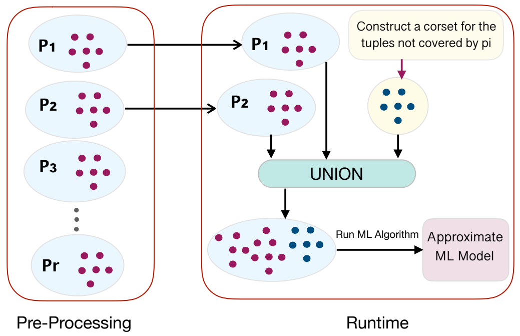


Figure 4: Two Phase Approach

sized coreset. Coresets can be stored as a pair (w_i, t_i) where w_i is the weight of tuple $t_i \in C$.

Two Phase Approach. Our proposed approach consists of two phases. In the *pre-processing* phase, we compute an ϵ-coreset C_i for the selected by each of the pre-built models. In the *runtime* phase, we identify the set of partitions $\mathcal{P}_q$ that could be used to answer q. We construct a coreset for all the tuples that were not covered by pre-existing partitions. Finally, we do a union of all the relevant coresets and run an appropriate ML model on it and provide the resulting model as an approximation.

4.1 Coreset Construction

We now provide a brief description of how to construct coresets for each partition for various ML models. Note that the definition of coreset is intrinsically tied to the objective function of the ML model. For example, coresets are defined based on SSE for K-Means, log likelihood for GMM and so on. Hence, one cannot reuse the coreset constructed for one ML model (such as K-Means) for the other (such as GMM).

A common strategy for computing coreset is to sample the data points proportional to their contribution for ϕ. Consider the K-Means clustering problem that seeks to minimize the SSE. Suppose we are given a set of *optimal* cluster centroids and an arbitrary data point x. If x is close to its nearest cluster centroid, then one need not include x in the coreset with high probability. Instead, one can "approximate" x's contribution to SSE through its cluster centroid (by increasing its weight by 1). If a point is distant from its nearest centroid, then it has a large contribution to SSE and hence must be included in the coreset with high probability. Intuitively, coreset construction can be considered as an *importance sampling* problem where data points are sampled based on their contribution to ϕ (such as SSE for K-Means). In practice, one does not have the optimal cluster centroids. The key research problem in coreset construction is to approximate the importance of a data point to SSE (ϕ in general), without knowing the optimal cluster centroids. This is often achieved by choosing a careful surrogate function ϕ' that is a good approximation of ϕ and can be computed efficiently.

Surrogate Functions for Coresets. A surrogate function for a coreset must satisfy two desirable properties: (a) it must provide an ϵ-coreset with small number of data points and (b) it must be lightweight so as to compute the importance of a data point in one or two passes over the entire data. Consider the surrogate function for K-Means defined in [8].

$$p(x) = \frac{1}{2}\frac{1}{|D_i|} + \frac{1}{2}\frac{d(x, \mu(D_i))^2}{\sum_{x' \in D_i} d(x', \mu(D_i))^2} \tag{4}$$

Given a set of points D_i and $x \in D_i$, it computes the importance of x by measuring the distance of x to the mean vector $\mu(D_i)$. Data points that are far away from the mean vector are provided with higher importance. The first term of the equation ensures that every data point has a non-zero probability of being picked. Note that this function is lightweight, efficient, computable in just two passes over the data and embarrassingly parallel to implement. It also provides an ϵ-coreset as proved in [8]. Coreset algorithms exist for ML models such as GMM [17], SVM [10], Logistic Regression [28], etc. Each of these algorithms vary in how the importance of the data point is computed.

Algorithm 4 provides the pseudocode for coreset construction.

Algorithm 4 Coreset Construction

```
Input: Set of data points $D_q$, Coreset size $m$
for each $x \in D_q$ do
    Compute contribution of $x$ based on a surrogate of $\phi$
$\forall x \in D_q$, Compute sampling probability $p(x)$
$C_i = m$ points from $D_q$ chosen through importance sampling
$\forall x \in C_i$, compute the weight $w(x)$
return coreset $C_i$
```

Complexity Analysis. For most of the popular ML models, there exist efficient coreset construction algorithms that run in time linear on the size of the dataset. For example, the algorithm proposed in [8] requires two passes - one to compute the mean of all data points and one to perform importance sampling.

4.2 Coreset Compression

The size of the coreset depends on the value of ϵ which is often end-user defined. A smaller value of ϵ requires better approximation and thereby larger coresets. Most of the state-of-the-art coreset algorithms often have the intriguing property that the coreset size depends primarily on ϵ and is independent of the size of the dataset. For example, one needs a K-Means coreset of size $\Omega(\frac{dk + \log \frac{1}{\delta}}{\epsilon^2})$ [8] to ensure that with probability of at least $1 - \delta$, coreset C_i is an ϵ-lightweight coreset. Given $K = 10$ and $d = 5$, one can obtain an $\epsilon = 0.1$-coreset with probability 0.95 by getting a sample of at least 5000 *regardless of the size of dataset.* $\epsilon = 0.2$-coreset requires approximately 1250 samples.

Consider a scenario where one needs to build a ML model for the entire USA where pre-built coresets exist for each state. Based on the observation above, we store approximately 5000 tuples as coreset for each state. When we pool them together, there are 250K tuples for the entire USA. Since coresets have the *compositional* property where if C_1 is an ϵ-coreset for D_1 and C_2 is an ϵ-coreset for D_2, then $C_1 \cup C_2$ is an ϵ-coreset for $D_1 \cup D_2$. Hence, the set of 250K tuples is an 0.1-coreset for entire USA. However, if we had constructed coreset directly over the entire data from USA, we would have only gotten 5000 tuples. We solve this conundrum by coreset compression. Simply put, we invoke a coreset construction algorithm with the same ϵ on the pooled set of tuples and choose a smaller number of tuples with highest importance - say of size 10,000 instead of 250,000. As we shall show in experiments, this approach works well in practice with minimal loss of accuracy.

5. OPTIMIZATION CONSIDERATIONS

5.1 Choosing ML Models to Reuse

The first major problem is to identify an optimal execution strategy - given a set of materialized models and an analytic query, how can one build an approximate ML model efficiently? For ease of exposition, we describe our approach for analytic queries specified as ranges such as building an ML model for tuples $[lb, ub]$. This approach can easily be adapted for OLAP queries over a single dimension.

Example. Consider a dataset with 1 million tuples with 4 materialized models for tuples $M_1 = [1, 500K]$, $M_2 = [500K, 1M]$, $M_3 = [300K, 900K]$, $M_4 = [900K, 1M]$. Given a new query $q = [250K, 1M]$, there are many ways to answer it. The traditional strategy S_1 builds an ML model from scratch for all of q. Or one could build an ML model from scratch for $[250K, 300K]$ and then merge it with M_3 and M_4 (Strategy S_2). Alternatively, one could build an ML model for $[250K, 500K]$ and then merge it with M_2 (Strategy S_3). Furthermore, each of these options can be either done using coresets or by model merging.

Cost Model. In order to perform cost based optimization, we need an objective cost model that quantifies various execution strategies. Broadly speaking, the cost involves three components: (a) cost of building a model C_{Build} from a set of (possibly weighted) tuples (b) cost of merging a model C_{merge} and (c) cost of building a coreset $C_{coreset}$. For example, the cost of strategy S_1 is $C_{Build}([250K, 1M])$ and S_2 is $C_{Build}([250K, 300K]) + C_{Merge}(M_2) + C_{Merge}(M_3)$. Optionally, one could also use a cost component for penalizing the loss of accuracy. In practice, efficiently estimating the accuracy of a model before building it is a non-trivial task. All the algorithms described in the paper provide rigorous worst case guarantees about the approximate model that we use as a proxy for their eventual performance. As an example, if the models have a coreset with $\epsilon = 0.1$, it provides an approximation of 10%. Henceforth, we focus on the scenario where the models obtained by either merging or through coresets already exceed the quality requirements of the analyst. If this is not acceptable, the analyst can build the model from scratch.

In our paper, we treat the cost model as an orthogonal issue that is often domain specific. The only constraint that must be satisfied by the cost model is that it is *monotonic.* In other words, all things being equal, building a model with N_i tuples should cost more than one with N_j tuples if $N_i > N_j$. Our algorithm produces an optimal execution strategy as long as the cost function is monotonic.

Finding the Optimal Execution Strategy. We formulate the problem of finding the optimal execution strategy as finding the shortest path in a graph with minimum weight. Our approach involves three steps. First, we retrieve a set of materialized models that can be used to answer q. A model built on $[lb', ub']$ is considered relevant if it is a subset of $q = [lb, ub]$. For example, for $q = [250K, 1M]$, the model M_1 is not relevant. Second, we collect the set of distinct lb, ub values from the relevant models including q. In the running example, it will be $V = \{250K, 300K, 500K, 900K, 1M\}$. Third, we construct an *execution strategy graph* - a weighted, directed and complete graph - that succinctly encodes all possible execution strategies to solve q. We build two graphs - one to identify the best execution strategy using the coreset approach and another for the merging approach. Informally, each of the distinct lb, ub values collected in Step 2 form the nodes. A directed edge e_{ij} exists between nodes v_i and v_j if $v_i < v_j$. If there exists a model with lb and ub corresponding to v_i and v_j, then $weight(e_{ij}) = C_{Merge}(v_i, v_j)$. This corresponds to the cost of directly using this model. If not, $weight(e_{ij}) = C_{Build}([250K, 300K])$ for the merging approach and $weight(e_{ij}) = C_{coreset}([250K, 300K]) + C_{Build}(\mathcal{C}([250K, 300K]))$ for coreset based approach. This corresponds to the cost of directly building an ML model for this range or building a coreset for this range and building an ML model over the coreset. Once the graph is constructed, the minimum cost execution strategy can be obtained by identifying the shortest path between the nodes corresponding to *lb* and *ub* - say by using Dijkstra's algorithm. Each edge $e_{ij} = (v_i, v_j)$ in the shortest path either corresponds to a pre-existing ML model built on (v_i, v_j) or requires one to build one between (v_i, v_j). Algorithm 5 provides the pseudocode for this approach.

Algorithm 5 Optimal Execution Strategy

```
Input: $q = [lb, ub]$, all materialized models $\mathcal{M}_D$
$\mathcal{M}_q$ = Filter the relevant models from $\mathcal{M}_D$
$V$ = Distinct end points for $\{q \cup \mathcal{M}_q\}$
for each pair $(v_i, v_j) \in V$ with $v_i < v_j$ do
    Add edge with appropriate weight ($C_{build}(v_i, v_j)$ or $C_{merge}(v_i, v_j)$)
$S_{opt}$ = Shortest path between $v_{lb}$ and $v_{ub}$
Execute strategy $S_{opt}$ to build an approximate ML model for $q$
```

Choosing ML Models to Reuse for Arbitrary Queries When the queries are range predicates on individual attributes, Algorithm 5 provides the optimal strategy. However, when the queries has predicates over multiple attributes or over OLAP hierarchies, then optimally choosing the models for reuse becomes an instantiation of exact set cover - a known NP-complete problem. To see why, each pre-built model can be considered as a set of tuples from which they were built. The query q can be considered as the set of tuples retrieved by it - i.e. D_q. Our objective is to select a small number of sets such that each tuple in D_q is covered by exactly one set.

We propose a natural greedy approach that works well in practice even when the number of queries is large. Of course, when the problem instances are small, one can essentially use a brute force approach to identify the optimal solution. We begin by pruning all pre-built models that are not proper subsets of D_q. This eliminates all models that contain tuples that are not retrieved by D_q. Using the cost model, we choose the model from the set of candidates that provide the most benefit (*e.g.,* it covers most tuples with least cost). Once the model M_i is chosen, we do two types of pruning. First, we remove all the tuples covered by M_i from D_q so that in the next rounds, the cost model gives higher weight to tuples that are not yet covered. Second, we remove all pre-built models that are not proper subsets of $D_q \setminus M_i$. This ensures that the same tuple is not covered by multiple chosen models and thereby having higher impact.

5.2 Selecting Models for Prebuilding

Suppose we are given set of queries Q that is representative of the ad-hoc analytic queries that could be issued in the future. These could be obtained from a workload or analytic query logs from the past. In this subsection, we first consider the problem of selecting L models to materialize so as to maximize the number of queries in Q that can be sped up through model reuse. We then briefly discuss the case where workload Q is not available. We address this problem in two stages. In the *candidate generation* step, we enumerate the list of possible ML models to build. In the *candidate selection* step, we propose a metric to evaluate the utility of selecting a model and use it to pick the best L models.

Candidate Generation. Given a workload $Q = \{q_1 = [lb_1, ub_1], q_2 = [lb_2, ub_2], \ldots, q_M = [lb_M, ub_M]\}$, our objective is to come up with L ranges such that they could be used to answer Q. Note that we are not limited to selecting ranges from Q. As an example, one could identify a sub-range that is contained in multiple queries to materialize. We generate the set of candidate models as follows. First, we select the list of all distinct lb, ub values. We then consider all possible ranges (l, u) such that $l < u$ and there exists at least one query in Q that contains the range (l, u). This ensures that we consider all possible ranges that could be reused to answer at least one query in Q.

Candidate Selection. In this step, we design a simple cost metric to compare two sets of candidate models. We can see that the cost of not materializing any model is equivalent to the traditional approach of building everything from scratch. So we have $Cost(\{\}) = \sum_{i=1}^{M} C_{build}(q_i)$. This gives us a natural method to evaluate a candidate set. We assume the availability of the corresponding models and compute the cost of answering Q. We use Algorithm 5 to estimate the optimal cost of building a given query. The difference between $Cost(\{\})$ and $Cost(\{r_{i_1}, r_{i_2}, \ldots\})$ provides the utility of choosing models $r_{i_1}, r_{i_2}, \ldots$ to materialize. Given this setup, one can use a greedy strategy to select the L models with highest utility. At each iteration, we pick a range r_i such that it provides the largest reduction in cost of answering all queries in Q.

If the workload information is not available, one could use some simple strategies to choose which models to materialize. The equi-width strategy creates L partitions by splitting the range $[1, n]$ into L equal sized parts $\{[1, \lfloor \frac{n}{L} \rfloor], [\lceil \frac{n}{L} \rceil, \lfloor \frac{2n}{L} \rfloor] \ldots\}$. Alternatively, one could also choose the L largest values of a given OLAP dimension. For example, if one of the dimensions is Country, then one could choose to pre-build models for the L largest countries.

Selecting Models for Arbitrary Queries. The above proposed approach can be naturally adapted for arbitrary queries. Given a set of workload Q, we generate the set of candidates as follows. Let $M = \{\}$ be the set of candidate models to pre-build. For each pair of queries $(q_i, q_j) \in Q$, we add $\{q_i, q_j, q_i \cup q_j, q_i \cap q_j\}$ to the set of candidate models M. Once the set of candidate models are constructed, we compute its weight based on how much it can contribute for speeding up queries in Q. We greedily choose the model from M with most benefit and re-compute the benefits of remaining candidate models. We repeat this iterative process till L models are chosen.

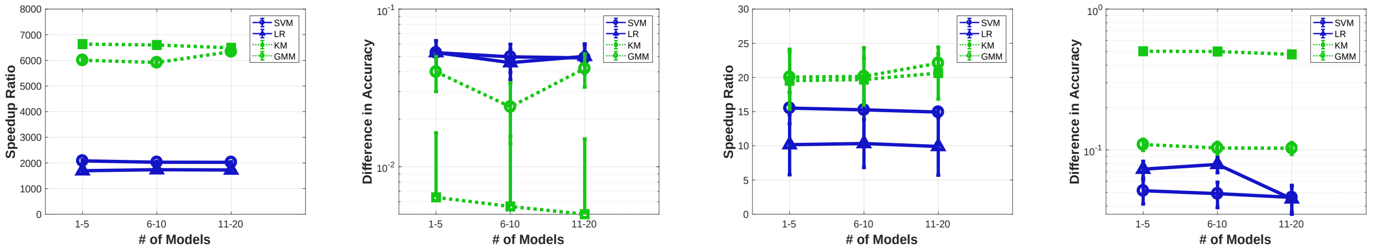
6. EXPERIMENTS

6.1 Experimental Setup

Hardware and Platform. All our experiments were performed on a quad-core 2.2 GHz machine with 16 GB of RAM. The algorithms were implemented in Python. Scikit-Learn (version 0.19.1) was used to train the ML models [45]. Vowpal wabbit [34] (version 8.5.0) was used for online learning.

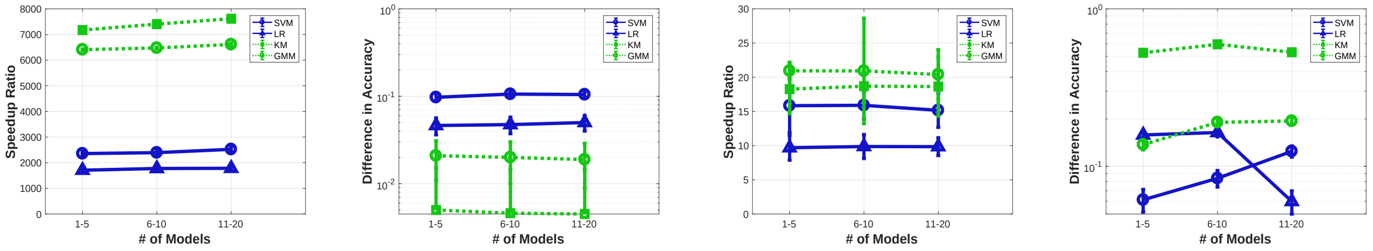
Datasets and Algorithms. For evaluating our classification algorithms (SVM and LR), we used 7 diverse datasets for binary classification. For datasets with OLAP style hierarchies, we selected 5 datasets from the Hamlet repository [33] - Movies, Yelp, Walmart, Books and Flights. We also selected two large datasets - SUSY and HIGGS from LibSVM repository [12, 15]. The size of the datasets vary from 200K tuples all the way to 11M tuples. For evaluating clustering, we generated a synthetic dataset with 5M data points, 20 features and 10 clusters using publicly available generator [45]. Each of the experiments was run with 5 different random seeds and the results are averaged. We evaluated a total of 8 algorithms - coreset and merging based algorithms for K-Means, GMM, SVM and Logistic Regression respectively. We compared each of these algorithms against two baseline algorithms where the analytic query is answered by running the ML model from scratch and by an incremental algorithm.

Performance Measures. We evaluate the efficacy of our algorithms against the baseline approach along two dimensions: time and ML model accuracy. *Speedup Ratio* (SR) is defined as the ratio of time taken for building an ML model over the data to the time taken to build a model by reusing existing ML models. It measures the time savings that one can obtain by building an approximate ML model from other ML models as against building it from scratch. We also evaluate the difference in model performance in order to ensure that the benefit in time does not come at the cost of model accuracy. For classification, we measure the difference in accuracy (DA) between the exact and approximate models. For example, a difference of 0.1 is obtained when the exact and approximate models have an accuracy of 99.9 and 100.0 respectively. For K-Means, we measure the clustering similarity through Adjusted Rand Index (ARI). ARI can be informally described as the ratio of agreements between two clusterings with respect to all possible pairs of data points. Specifically, if n_{ss} and n_{dd} are the number of pair of tuples that were assigned to same cluster and different clusters respectively, then $RI = \frac{n_{ss}+n_{dd}}{\binom{|D_q|}{2}}$. Adjusted Rand Index performs chance normalization on Rand Index such that it has an expected value of 0 for independent clusterings and 1 for identical clusterings. For GMM, we use relative error between the likelihood for the entire produced by the two models. If the value is closer to 1, then we obtained a model that is very close to the exact one. Note that both these methods are in the same range of $[0, 1]$ with a value closer to 1 being preferred. In the charts, we use the generic



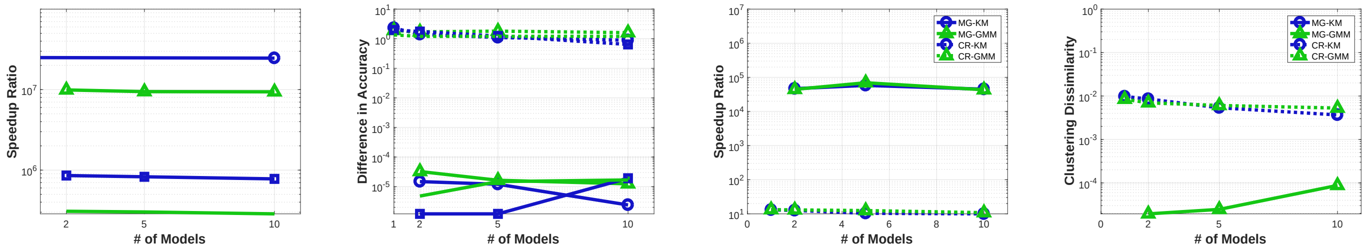
(a) SR for Model Merging (b) DA for Model Merging (c) SR for Coresets (d) DA for Coresets

Figure 5: Evaluating approaches for building an approximate ML model for a workload of OLAP queries on the Hamlet datasets.



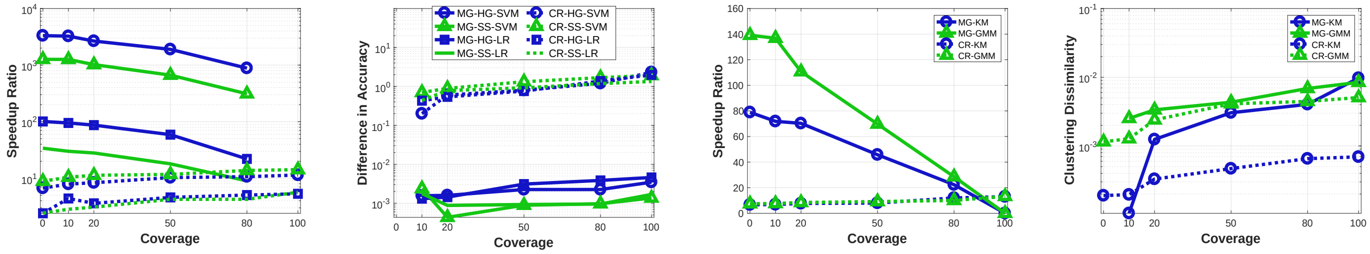
(a) SR for Model Merging (b) DA for Model Merging (c) SR for Coresets (d) DA for Coresets

Figure 6: Evaluating approaches for building an approximate ML model for a workload of random queries on the Hamlet datasets.



(a) SR for GLM (b) DA for GLM (c) SR for Clustering (d) DA for Clustering

Figure 7: Evaluating approaches for building an approximate ML model for queries over the entire dataset. Abbreviations: Same as Figure 8. Legend for Figures 7a, 7b and 8a same as that of 8b.

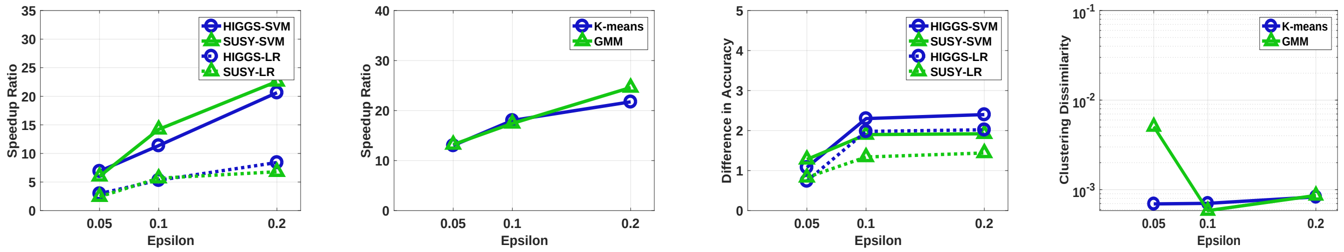


(a) SR for GLM (b) DA for GLM (c) SR for Clustering (d) DA for Clustering

Figure 8: Evaluating impact of Coverage Ratio on building an approximate ML model. Abbreviations: MG/CR - Merging/Coresets; HG/SS - HIGGS/SUSY, KM - KMeans

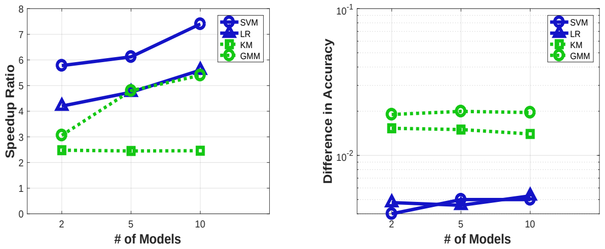
term “clustering dissimilarity” to denote the corresponding distance measure (i.e. 1 - ARI or 1 - relative error).

Evaluation Methodology. We consider four ML models: Logistic Regression (LR), Support Vector Machines (SVM),



(a) SR for GLM Coresets (b) SR for Clustering Coresets (c) DA for GLM Coresets (d) DA for Clustering Coresets

Figure 9: Evaluating the impact of Approximation Ratio for Coresets.



(a) SR (b) DA

Figure 10: Evaluating the impact of Coreset compression.

K-Means and Gaussian Mixture Models (GMM). We used the implementations provided from scikit-learn. In our experiments, we did not use any hyper parameter tuning and set those to the default values of scikit-learn. Our experiments showed that the impact of hyper parameters (such as learning rate or regularizer) was minimal when each of the merged model used the same or similar value. For K-Means and GMM, the number of clusters was set to 10. We used the classical non-nested 5-fold cross validation and report the average of accuracy over the testing data for 5 runs. Each of the chosen coreset algorithms ([8, 17, 10, 28] also provide a closed form solution to compute the size of the coreset to achieve ϵ-approximation. Conservatively, we multiply the estimate by 4.

Query Workload. We consider two types of query workloads. Random workload involves queries where the query predicate is chosen at random (such as build a model for tuples with id in the range [1M, 2M]). We shuffled the data using 5 different random seeds to ensure that the results are not due to chance. OLAP workload involves queries that have predicate over the attributes that have OLAP hierarchies imposed on them. Specifically, we considered all OLAP cuboids that contained at least 1% of the tuples. Then, we generated queries by considering all possible subsets of the cuboids of the same predicate. As an example, in the movies dataset, the ratings vary between 1-5. We considered all 26 possible combinations of the ratings of size at least 2 (i.e. $\sum_{j=2}^{5}\binom{5}{j} = 26$). This process was repeated for each attribute and each dataset.

In our final set of experiments, we study the impact of various parameters on our algorithm. These experiments are performed on the synthetic dataset for a specific query: build an ML model over the entire data. The data has been randomly partitioned into 2, 5, and 10 partitions. As an example, the data is partitioned into 10 equal sized partitions and an ML model has been pre-materialized for each partition. For each query in the workload, we assume that there is always a set of pre-built models that can be used to approximate the query. We also investigate the impact of queries that are not fully covered by pre-built models.

6.2 Experimental Results

Evaluating the Model Merging based Approach. In our first set of experiments, we evaluate the performance of the model merging based approach for both classification and clustering based ML models. Figures 5(a)-7(a) show the results for all three types of query workloads. For each type of queries, our approach achieves substantial speedup over both baseline approaches with significant speedup whenever the analytic query has high selectivity. For example, our approach achieves a significant speedup as much as 10^7 for HIGGS. In concrete numbers, training a Linear SVM on the entire HIGGS dataset with 11M tuples takes around 1.5 hours while simply merging the models takes just milliseconds. Another key thing to notice is that the benefit improves dramatically with larger datasets such as HIGGS and SUSY getting orders of magnitude speedup over the smaller datasets. The benefit is especially significant for compute intensive training algorithms such as SVM. Furthermore, the number of models to be merged that correspond to the number of partitions has at most negligible impact. These observations support our original hypothesis that one can significantly speed up analytic queries by reusing ML models. Our approach has the potential to make ML more interactive and near real-time. These benefits extend for clustering also. We achieve a speedup of at least 10^4 for both K-Means and GMM based clustering over large datasets.

Figures 5(b)-7(b) show that this substantial speedup does not have any major impact on the model accuracy. The performance of the exact and approximate ML models are almost identical with their accuracy values varying only in the third or fourth decimal places. Even for a large dataset such as HIGGS that had a testing set with 500K tuples, this only corresponds to a handful of mis-classifications. For a number of exploratory analytic ML tasks this is an acceptable trade-off when one can get results in many orders of magnitude faster. Since our experiments were conducted on multiple datasets in the Hamlet repository, we also provide the error bars for Figures 5 and 6. Note that while the error bars seem quite large, the actual differences were very small (*e.g.,* in the order of 0.1 for Figure 5(b). The differences for results on speedup ratio is also miniscule. Another interesting observation is that the difference in model accuracy actually decreases for larger datasets. This is consistent with the theoretical results from [57].

Evaluating Coreset based Approach. In the next set of experiments, we evaluate the performance of the coreset based approach for both classification and clustering based ML models for various query workloads. We assume the availability of coresets for each partition that are then pooled together and a weighted variant of the ML model algorithm was invoked on it. By default, we set the coreset approximation ratio as $\epsilon = 0.1$. Figures 5(c) and 6(c) show the performance of the coreset based approach for OLAP and random query workloads. Figure 7(a) and (c) show the same for queries over the entire dataset.

Our approach typically provides a speedup between 5-15 with larger speedups for bigger datasets and expensive algorithms such as SVM. This is due to the fact that most coreset algorithms can approximate a dataset with at most logarithmic number of points thereby providing substantial speedups over the traditional approach that runs on the entire dataset. While the coreset based approach provides an order of magnitude speedup over the current approach, it pales in comparison against the speedups provided by model merging based approaches. This is due to the fact that one has to run the expensive model training algorithm (for *e.g.,* $O(n^3)$ for SVM) on the coreset. Figures 5(d), 6(d), 7(b) and (d) show the impact on accuracy is minimal. Even though $\epsilon = 0.1$, the difference in accuracy was much lower around 1-2% for GLM and almost negligible for clustering. Furthermore, as the number of partitions increases, the difference in model accuracy decreases. This is due to the fact that the union of coresets often have *slightly* more redundant information that helps in improving the performance. Overall, this set of experiments show that coresets can provide approximate models that are as accurate as the exact ones and often sufficient enough for exploratory ML purposes.

Impact of Coverage Ratio. In our next set of experiments, we studied the impact of coverage ratio on the performance of our algorithms. Informally, the coverage ratio corresponds to the ratio of the query for which we could reuse pre-built ML models. So a coverage ratio of 100% means that we can completely answer an analytic query using pre-built models while a coverage ratio of 0% means that we have to build the model from scratch. In our experiments, we focused on a scenario where the data is partitioned in 10 partitions. So for a coverage ratio of 20%, we assumed that pre-built models exist for two randomly chosen partitions. We then build a single exact model for the remaining 8 partitions and then combine it with the 2 models. For coresets, this corresponds to running a coreset algorithm on the 80% of the data, combining it with pre-computed coresets for the other two partitions and running the ML model.

Figures 8(a) and 8(c) show the time taken in seconds for model building. As expected, the running time of our approach depends significantly on the availability of pre-built models. For example, if pre-built models are completely available, our model merging approach just requires a few milliseconds. However, if pre-built models only exist for 80% of the data, it provides a speedup of 5x as one needs to train the model only for 20% of the data. As shown previously, the impact on accuracy of the model - for both classification and clustering - is minimal to non-existent. A similar behavior can be observed for coresets. As the coverage ratio increases, the speedup ratio provided by the coreset also increases with minimal impact on model accuracy.

Impact of Coreset Approximation Ratio. In our final set of experiments, we vary the approximation ratio of the coreset from $\epsilon = [0.05, 0.1, 0.2]$. A smaller value of ϵ provides a tighter approximation at the cost of a larger coreset. Figures 9(a) and 9(b) show that as value of ϵ increases, the speedup ratio also improves. This is due to the fact that a smaller coreset suffices to guarantee a larger approximation of ϵ. Figures 9(c) and 9(d) show an interesting result wherein increasing the value of ϵ - say by doubling it - does not result in a significant reduction in model accuracy. Instead, the impact is quite minimal! This seems to confirm the central observation in coreset theory that real-world data often have substantial redundant information that can be effectively approximated by coresets.

Impact of Coreset Compression. In the final set of experiments, we study the efficacy of coreset compression. Figure 10 shows the results. As expected, coreset compression has minimal impact on accuracy yet has a significant improvement in reducing the time take for model building. This behavior is pronounced when there are multiple models to be merged which is precisely the scenario where coreset compression has the most impact.

7. DISCUSSION

As shown by the experimental results, our proposed approaches can be used for efficiently generating approximate ML models. In this section, we briefly discuss the scenarios in which our approach is relevant and when it might not be.

In general, our approach is often geared to be used in exploratory ML analysis. In this stage of the analysis pipeline, the data scientist is often exploring various hypotheses and is often willing to trade accuracy for real-time response. In the production environment where the data scientist would want to maximize accuracy, our approach might not be applicable.

We would like to note that the suitability of queries for building ML models is an orthogonal issue that is determined by the domain expert. For example, it is possible that building a model over the union of data from 2017 and 2018 is inadvisable due to issues such as staleness or concept drift. In such a case, building of an exact ML model (via traditional methods) or an approximate ML model (via our approach) are both inappropriate. Our focus is on building an approximate ML model efficiently when the data scientist deems such a model to be relevant.

Thoroughly understanding the limitations of our approach is a key focus of our future work. A non exhaustive list of scenarios where our approach might provide less robust results include:

- *Concept Drift* is said to occur when the statistical properties of the target variable changes over time. As an example, the data for 2017 and 2018 might be so different that building a model over the union of the data is not meaningful.

- *Skewness of Model Data Sizes.* If the constituent models have very skewed distribution of selectivity, model merging does not provide robust results. As an example, consider individual models that are built for ratings=1 to ratings=5. Often, the ratings express a U-shaped distribution where there are more tuples with ratings 1 or 5 with substantially less tuples for ratings 2, 3, and 4. If the ratings 1 and 5 account for 90% of the tuples, the results could also be skewed.
- *Skewness of Labels.* If in a binary classification problem, 90% of the tuples belong to one class, naive merging could result in a biased classifier.

8. RELATED WORK

Data Management Challenges in Machine Learning. Recently there has been extensive interest in integrating ML capabilities into databases from both industry and academia. Most major commercial database products such as IBM System ML, Oracle ORE, SAP HANA already support analytic queries over database engines. Academic product such as MLLib [39] and MADLib [26] also support similar integrations. There has been work on integrating ML primitives into database engines such as [32], using SQL style declarative languages for ML model training [35, 29]. Recent work also tried to use key concepts from data management for speeding up ML analytic tasks. These include materialization for feature selection [33, 56], using database style cost optimizer for predicting performance of ML tasks [44, 55, 29]. Incremental processing of ML based analytic queries was considered in [22]. Please refer to [30] for additional details about various data management related issues in ML.

Management of ML Models. Recently, there is increasing interest in managing key artifacts of ML process such as ML models [31, 41] and datasets [23]. [41, 40] focus on a unified data, model and lifecycle management for deep learning while [53, 54] seek to manage models for other applications such as model diagnosis, visualization and provenance. As ML model management systems become mainstream, they could be used to identify relevant models for a new ML model query. Our approaches can be easily retrofitted on top of these systems.

Speeding Up Analytic Queries. There has been extensive work on speeding up analytic queries in databases. Two techniques are especially relevant: approximate query processing (AQP) and cube materialization. AQP [19] relies on the fact that exact answers are not always required and provides approximate answers - often for aggregate queries - at interactive speeds. The common techniques include sampling and construction of synopses [1, 7]. Our coreset based approach can be considered analogous to synopses for AQP. There also has been extensive work on efficiently materializing OLAP cubes by leveraging partial computations [20]. There has been extensive followup work that computed interesting statistical aggregates on OLAP cubes such as [13, 9, 36, 16]. The work [13] is especially relevant to our problem. Prediction cubes summarizes a predictive model trained on the data corresponding to an OLAP data cube. Our approach can be used to speedup [13] by building approximate ML models for data cubes from its component cubes. Another recent work [5] is complementary to our effort as it focuses on speeding mean value and multi-variate regression queries. In contrast, we focus on ML analytic queries for classification and clustering.

Approximating ML Models. A number of ML algorithms often use iterative algorithms such as gradient descent. A common approach for approximating the ML model is to stop after a fixed number of iterations [11, 43]. However, this typically does not provide any rigorous guarantees. Coresets were originally proposed in computational geometry that provide strong approximation guarantees. There has been extensive work on coresets for various ML models such as K-Means [8, 25, 18, 48], GMM [17], kernel density estimation [46], logistic regression [28, 24], SVM [10, 52, 51], Bayesian networks [42] and so on. A recent work [56] also used the concept for coresets. They focused on speeding up analytic queries for feature selection process by using coresets as a principled sampling appproach. In contrast, our work assumes that feature selection/engineering is already completed and use coresets to build models with strong approximation guarantees. They also have an elegant idea of warm starting where they train some models more efficiently by reusing prior models with related *features*. For example, a model with feature set F can be used to speed up another one that has feature set $F \setminus f$ or $F \cup f$ where f is a single feature. In contrast, our approach is for a fixed feature set F where the data that is used to train the model varies.

Another area related to our work is transfer learning [50] where the objective is to train a model for one domain/dataset and reuse it for another. Our work primarily considers a single dataset. How to adapt ideas from transfer learning so that we can transfer the model trained on a query Q to a related query Q' is an interesting research problem.

9. CONCLUSION

In this paper, we presented an approach to answer ad-hoc analytic queries on ML models in an approximate manner at interactive speeds. Our key observation was that most of these queries are often aligned on OLAP hierarchies and it must be possible to materialize and reuse ML models. We presented two orthogonal approaches based on coresets and model merging to answer popular ML algorithms in classification and clustering. We also proposed an algorithm to identify an optimal execution strategy for an analytic query and to determine which models to materialize. Our experimental results on a wide variety of real-world datasets show that our approach can result in orders of magnitude in speedup with negligible approximation cost.

10. REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD Record*, volume 28, pages 275–286. ACM, 1999.

[2] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Geometric approximation via coresets. *Combinatorial and computational geometry*, 52:1–30, 2005.

[3] N. Ailon, R. Jaiswal, and C. Monteleoni. Streaming k-means approximation. http://www.cs.columbia.edu/~rjaiswal/ajmNIPS09.pdf. Accessed: Jul 20, 2018.

[4] N. Ailon, R. Jaiswal, and C. Monteleoni. Streaming k-means approximation. In *NIPS*, pages 10–18, 2009.

[5] C. Anagnostopoulos and P. Triantafillou. Efficient scalable accurate regression queries in in-dbms analytics. In *ICDE*, pages 559–570. IEEE, 2017.

[6] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *SODA*, pages 1027–1035. SIAM, 2007.

[7] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD*, pages 539–550. ACM, 2003.

[8] O. Bachem, M. Lucic, and A. Krause. Scalable and distributed clustering via lightweight coresets. *arXiv preprint arXiv:1702.08248*, 2017.

[9] D. Barbará and X. Wu. Loglinear-based quasi cubes. *Journal of Intelligent Information Systems*, 16(3):255–276, 2001.

[10] C. Baykal, L. Liebenwein, and W. Schwarting. Training support vector machines using coresets. *arXiv preprint arXiv:1708.03835*, 2017.

[11] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[12] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[13] B.-C. Chen, L. Chen, Y. Lin, and R. Ramakrishnan. Prediction cubes. *PVLDB*, pages 982–993, 2005.

[14] A. Declercq and J. H. Piater. Online learning of gaussian mixture models-a two-level approach. In *VISAPP (1)*, pages 605–611, 2008.

[15] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository. `https://archive.ics.uci.edu/ml/datasets.html`, 2017. Accessed: Jul 20, 2018.

[16] G. Dong, J. Han, J. Lam, J. Pei, and K. Wang. Mining multi-dimensional constrained gradients in data cubes. *PVLDB*, pages 321–330, 2001.

[17] D. Feldman, M. Faulkner, and A. Krause. Scalable training of mixture models via coresets. In *NIPS*, pages 2142–2150, 2011.

[18] D. Feldman, M. Schmidt, and C. Sohler. Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In *SODA*, pages 1434–1453. SIAM, 2013.

[19] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. *PVLDB*, pages 343–352, 2001.

[20] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[21] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *TKDE*, 15(3):515–528, 2003.

[22] P. Gupta, N. Koudas, E. Shang, R. Johnson, and C. Zuzarte. Processing analytical workloads incrementally. *arXiv preprint arXiv:1509.05066*, 2015.

[23] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *SIGMOD*, pages 795–806. ACM, 2016.

[24] L. Han, T. Yang, and T. Zhang. Local uncertainty sampling for large-scale multi-class logistic regression. *arXiv preprint arXiv:1604.08098*, 2016.

[25] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *STOC*, pages 291–300. ACM, 2004.

[26] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *PVLDB*, 5(12):1700–1711, 2012.

[27] C. Hennig. Methods for merging gaussian mixture components. *Advances in data analysis and classification*, 4(1):3–34, 2010.

[28] J. Huggins, T. Campbell, and T. Broderick. Coresets for scalable bayesian logistic regression. In *NIPS*, pages 4080–4088, 2016.

[29] Z. Kaoudi, J.-A. Quiané-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal. A cost-based optimizer for gradient descent optimization. In *SIGMOD*, pages 977–992. ACM, 2017.

[30] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD*, pages 1717–1722. ACM, 2017.

[31] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[32] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984. ACM, 2015.

[33] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD*, pages 19–34. ACM, 2016.

[34] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007.

[35] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang. Mlog: Towards declarative in-database machine learning. *PVLDB*, 10(12):1933–1936, 2017.

[36] D. Margaritis, C. Faloutsos, and S. Thrun. Netcube: A scalable tool for fast data mining and compression. *PVLDB*, pages 311–320, 2001.

[37] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *NAACL*, pages 456–464. ACL, 2010.

[38] R. Mcdonald, M. Mohri, N. Silberman, D. Walker, and G. S. Mann. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, pages 1231–1239, 2009.

[39] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(1):1235–1241, 2016.

[40] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Modelhub: Deep learning lifecycle management. In *ICDE*, pages 1393–1394. IEEE, 2017.

[41] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards unified data and lifecycle management for deep learning. In *ICDE*, pages 571–582. IEEE, 2017.

[42] A. Molina, A. Munteanu, and K. Kersting. Coreset based dependency networks. *arXiv preprint arXiv:1710.03285*, 2017.

[43] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[44] X. Pan, S. Venkataraman, Z. Tai, and J. Gonzalez. Hemingway: Modeling distributed optimization algorithms. *arXiv preprint arXiv:1702.05865*, 2017.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *JMLR*, 12(Oct):2825–2830, 2011.

[46] J. M. Phillips and W. M. Tai. Improved coresets for kernel density estimates. *arXiv preprint arXiv:1710.04325*, 2017.

[47] A. R. Runnalls. Kullback-leibler approach to gaussian mixture reduction. *IEEE Transactions on Aerospace and Electronic Systems*, 43(3), 2007.

[48] M. Shindler, A. Wong, and A. W. Meyerson. Fast and accurate k-means for large datasets. In *NIPS*, pages 2375–2383, 2011.

[49] R. J. Steele, A. E. Raftery, and M. J. Emond. Computing normalizing constants for finite mixture models via incremental mixture importance sampling (imis). *Journal of Computational and Graphical Statistics*, 15(3):712–734, 2006.

[50] L. Torrey and J. Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 1:242, 2009.

[51] I. W. Tsang, A. Kocsor, and J. T. Kwok. Simpler core vector machines with enclosing balls. In *ICML*, pages 911–918. ACM, 2007.

[52] I. W. Tsang, J. T. Kwok, and P.-M. Cheung. Core vector machines: Fast svm training on very large data sets. *JMLR*, 6(Apr):363–392, 2005.

[53] M. Vartak, J. M. da Trindade, S. Madden, and M. Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In *SIGMOD*, pages 1285–1300, 2018.

[54] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Model db: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

[55] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[56] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

[57] Y. Zhang, M. J. Wainwright, and J. C. Duchi. Communication-efficient algorithms for statistical optimization. In *NIPS*, pages 1502–1510, 2012.

[58] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.