ApproxML: Efficient Approximate Ad-Hoc ML Models Through Materialization and Reuse

By Faezeh Ghaderi

Supervising Professor: Dr. Gautam Das

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2019

Acknowledgements

I would like to express my deepest appreciation and gratitude to my Advisor, Dr. Gautam Das, for giving me the opportunity to work with him and for his constant support and guidance throughout my Master thesis. I feel extremely honored to be a part of DBXLAB, where I have had the opportunity to explore new horizons, which I never knew existed. I am also extremely grateful to Dr. Ramez Elmasri and Dr. Sharma Chakravarthy for accepting to be on the thesis committee.

I sincerely appreciate the efforts of Ms. Sona Hasani of DBXLAB for her remarkable guidance and support throughout this project. It was wonderful working with all the members of the lab throughout my time as a DBXLAB member and I am extremely grateful for that.

Finally, a huge thank you to my parents and my husband Mahdi for their moral support and encouragement in the graduation days and also GOD, for his grace in me.

December 3, 2019

Abstract

Machine Learning (ML) has become an essential tool in answering complex predictive analytic queries. Model building for large scale datasets is one of the most time-consuming parts of the data science pipeline. Often data scientists are willing to sacrifice some accuracy in order to speed up this process during the exploratory phase. In this report, we aim to demonstrate ApproxML, a system that efficiently constructs approximate ML models for new queries from previously constructed ML models using the concepts of model materialization and reuse. ApproxML supports a wide variety of ML models such as generalized linear models for supervised learning and K-Means and Gaussian Mixture model for unsupervised learning. The Implementation is compatible with different datasets and ML algorithms, as it is a cost-based optimization framework that identifies best reuse strategy at query time.

Table of Contents

List of Illustrations

Chapter 1: Introduction

Machine Learning has undergone a phase transition from a pure academic endeavor to being one of the main drivers of modern commerce and science. However, when it comes to very big dataset, the capabilities of statistical machine learning methods are limited by the computing time rather than the sample size. Even though there has been extensive work from the ML community on developing faster algorithms, building an ML model is often a major bottleneck and consumes a lot of time due to the sheer size of the datasets involved. In this report, the feasibility of building faster ML models for a popular class of analytic queries has been demonstrated by leveraging two fundamental concepts from database optimization materialization and reuse [1].

Generally, data science is about analysis of different types of data and one of the most interesting approaches is ad-hoc analytics on ML models. The process of ad hoc analysis is to answer one specific business question. This type of analysis may be done in response to an event, like a decrease in sales. It can provide a report that isn't part of any other teams, or it can dig deeper into an existing report to get more detail. This process consists of retrieving data, building an ML model and using the model for analytic processing which consumes a large amount of workflow.

The queries in most businesses are usually aligned with the OLAP hierarchies of the company based on some explicit domain required in the scope of the business for example regional or monthly queries. On the other hand, ML models in most of exploratory analysis are one time used, means that a model won't be reused in other parts of the queries. With these in mind and the fact that for explanatory models we can sacrifice a small amount of accuracy, the new method of materialization and reuse of ML models have been used and implemented.

There are two phases in the main workflow. In the first phase which is pre-processing phase, we store ML models along with a small amount of additional meta-data and statistics in the repository; During the second phase of runtime phase, we identify the relevant ML models to reuse and quickly construct an approximate ML model from them [2]. We selected some ML models based on their popularity among 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. In unsupervised learning, we consider two canonical clustering approaches: K-Means and Gaussian Mixture Models (GMMs).

In the process of storing ML models in the repository, two orthogonal approaches have been used. In this part, let us briefly explain two different approaches:

- Model Merging:

  In this method, after construction of some ML models and storing their metadata in repository in the pre-processing phase, we select and combine the relevant prebuilt ML models for specific queries and construct a complete ML model without consuming time on going back to original data and building the model from scratch in run-time phase. This 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.

- Coresets:

  Coresets are succinct, small summaries of large data sets, so that the solutions found on the summary are provably competitive with solution found on the full data set [3]. 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.

Chapter 2: Architecture

ApproxML enables the user to build approximate model for popular supervised and unsupervised ML models for a given analytic query and a set of materialized models. There are several challenges to tackle in order to build efficient approximate ML models such as a) If we have access to a set of pre-built models, would it be possible to combine them in a few milliseconds to construct an approximate model instead of spending minutes/hours to build a model from scratch? b)How can we efficiently identify the relevant models among many possible choices? c)What information should be materialized for each model to make it reusable in future? ApproxML generates approximate ML models in a two- phase approach. During ”pre-processing phase”, the model passively stores the ML models built by the data analyst to a model DB along with small amount of additional meta-data such as the data used and parameters; During the “run- time phase", for a new query, it identifies the relevant and reusable pre-built ML models and efficiently constructs an approximate ML model from them. ApproxML offers two orthogonal methods for generating approximate ML models, a) model merging approach, and b) coreset-based approach. Figure 1 demonstrates the system overview of ApproxML.

During the run-time phase, we assume we have access to a repository of the pre-built models. The user submits an analytic query through the front-end. Front-end will parse the information about the dataset, the intended ML algorithm, the approximation method (model merging, coreset-based), etc. and will pass them to the cost-based optimizer in the back-end. The optimizer will retrieve all pre-built models relevant to the given analytic query from the pre-built model repository. It will identify which of the retrieved pre-built models should be reused and what additional partial models have to be built from scratch.

Then these partial models are passed to the "Approximate model builder" component to be combined efficiently to get the final approximate model.

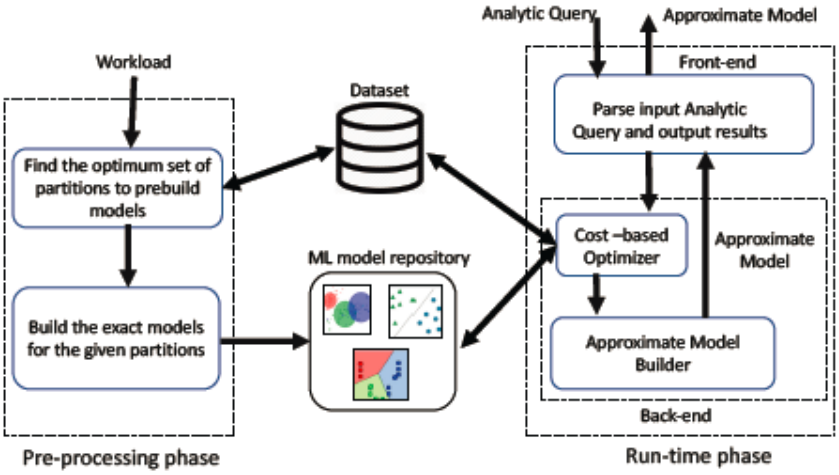

Figure 1 System overview

In the pre-processing phase, a set of models are built and stored in the pre-built model repository. These models are selected to be reused for the future queries in the best way using a workload or analytic query logs from the past. To identify the best models to materialize, first, the list of possible ML models to build for a given workload history is enumerated. In the next step a greedy algorithm is applied to identify the models with the highest benefit for the given workload. These selected models are materialized and stored in the pre-built model repository.

In the pre-processing phase, exact ML models are built for several partitions of data and their corresponding meta data is stored in a repository. These pre-built models may contribute to future approximate models. For model merging approach the parameters of the models are materialized while for coreset-based approach the coresets and their corresponding weights are recorded. In the model merging scenario for K-means, ApproxML stores K centroids and the weight associated with each cluster. In GMM, it stores the mean vector and covariance matrix of each component along with their relative weights. For Logistic regression it stores the coefficients and for SVM it stores the coefficients of the separating hyper plane.

Chapter 3: Technical Background

In this part, we elaborate the technical background for enabling interactive ad-hoc analytics on ML models. For a given query among all the dataset, we should know some definitions about the dataset attribute and different type of queries and basic concepts of materialization of the models. Some required explanation about ML models that we implement in this project will be explained below. The used algorithms include K-Means, Gaussian Mixture Models and Generalized Linear Models.

2.1. Dataset and queries

Let's say a dataset D consists of n tuples which have d attributes A = {A_1, A_2, …, A_d}. We partition the schema A into X and 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, Z_2, …, Z_g} that are associated with pre-defined dimensional hierarchies such as city.

Each tuple ti is also associated with a unique identifier tid that imposes a total ordering in D. As an example, ti_d could be an automatically incrementing sequence or timestamp indicating when the tuple was created.

Query Model: Let q be the analytic query specified on D that returns a result set Dq 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 Xi and range [a, b] such that they filter all tuples with value of Xi falling between a and b.
- Dimension based predicate: These queries filter tuples that have specific values for one or more dimensional attributes.

- Arbitrary Predicates: These queries use complex query predicates (including a combination of range and dimension based) to select relevant data.

In this project, based on selected dataset we just focused on the first type, ranged based predicate.

3.2. Pre-Materialized Models

We denote the exact model built on Dq as M(Dq) while its approximation as M'(Dq). 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 Mq be the set of pre-built models that could be used to answer it approximately where |Mq| = r.

Example: Consider a database D = {1, … ,1000} where we have a set of built ML models {M1, … ,M10} over ranges $\{P_1 = [1; 100]; P_2 = [101; 200], \ldots, P_{10} = [901; 1000]\}$. Given a query $q_1 = [101; 500]$, then Mq_1 = {M2, M3, M4, M5}. If necessary, one can build appropriate models for tuples from Dq for which no pre-built models exist. Given a query $q_2 = [51; 550]$, the set of models to answer them will be Mq2 = {M([51; 100] [501; 550]), M2, M3, M4, M5}

3.3. Machine Learning algorithms

3.3.1. K-means

K-means clustering is one of the simplest and most popular unsupervised machine learning algorithms. Typically, unsupervised algorithms make inferences from datasets using only input vectors without referring to known, or labelled, outcomes[4].

A cluster refers to a collection of data points aggregated together because of certain similarities. We will define a target number k, which refers to the number of centroids needed in the dataset. A centroid is the imaginary or real location representing the center of the cluster. Every data point is allocated to each of the clusters through reducing the

in-cluster sum of squares. Given a set of points X∈ R^d, the K-Means clustering seeks to find cluster centers in R^d (centroids) such that the sum of squared errors (SSE) is minimized. Given a set of data points X and centroids C, the SSE is defined as:

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

In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible. 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. This algorithm starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids

It halts creating and optimizing clusters when either:

- The centroids have stabilized, there is no change in their values because the clustering has been successful.
- The defined number of iterations has been achieved.

3.3.2. Gaussian Mixture Models (GMM)

Gaussian mixture (GM) is defined as a convex combination of Gaussian densities. A Gaussian density in a d-dimensional space, characterized by its mean m ∈ Rd and d×d covariance matrix [5].

Gaussian mixture model is parameterized by two types of values, the mixture component weights, and the component means and variances/covariances. For a Gaussian mixture model with K components, the kth component has a mean of μ_k and variance of ∑ for the univariate case and a vector of mean of μ_k and covariance matrix of Σ_k for the multivariate case. Noted that the total probability distribution normalizes to 1.

GMMs have been used for feature extraction from speech data and have also been used extensively in object tracking of multiple objects, where the number of mixture components and their means predict object locations at each frame in a video sequence.

Suppose we are given a set of d-dimensional data points $X = \{x_1, x_2, \dots, x_n\}$. We fit X as Gaussian mixture model parameterized by $= [(w_1, \mu_1, \sum_1), (w_2, \mu_2, \sum_2), \dots, (w_k, \mu_k, \sum_k),]$ where the i-th mixture component is a d-dimensional multivariate Gaussian $(w_i, \mu_i, \sum_i)$ with w_i being its prior probability. Note that the prior probabilities of the components sum up to 1. Given the data X, GMM estimates the parameters that maximizes the likelihood through the Expectation-Maximization (EM) algorithm.

3.3.3. Generalized Linear Models (GLM)

Among many classes of GLM we focused on the most popular ones, including logistic regression (LR) and support vector machines (SVM). Logistic regression method calculates the probabilities for classification problems with two possible outcomes. It's an extension of the linear regression model for classification problems. Instead of fitting a straight line or hyperplane (in linear regression model), the logistic regression model uses the logistic function to squeeze the output of a linear equation between 0 and 1. The logistic function is defined as:

$$\text{logistic}(\eta) = \frac{1}{1 + exp(-\eta)}$$

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is binary. Like all regression analyses, logistic regression is a predictive analysis.

SVM is a supervised machine learning algorithm which can be used for classification or regression problems. In this project, we'll focus on using SVM for classification. A hyperplane is a line that splits the input variable space. In SVM, a hyperplane is selected to best separate the points in the input variable space by their class, either class 0 or

class 1. In two-dimensions you can visualize this as a line and let's assume that all our input points can be completely separated by this line.

The distance between the line and the closest data points is referred to as the margin. The best or optimal line that can separate the two classes is the line that as the largest margin. This is called the Maximal-Margin hyperplane.

The margin is calculated as the perpendicular distance from the line to only the closest points. Only these points are relevant in defining the line and in the construction of the classifier. These points are called the support vectors. They support or define the hyperplane (Figure 2).

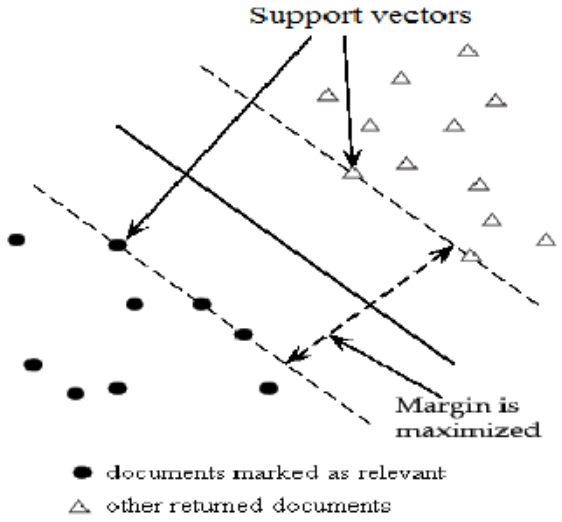

Figure 2 Maximum-margin hyperplane and margins for an SVM trained with samples from two classes

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.

Chapter 4: Methodologies

In this part, we will elaborate the main idea of implementation Approximation ML models. It consists of two main types: approximation by model merging and approximation by coreset points.

In model merging, we merge prebuilt (exact) ML models for a query and the result will approximate the exact model which is built for that query. The advantage is that we can generate a model without retrieving the data. Suppose we have data D and the exact ML model M(D). we have also n number of pre-built models for a specific algorithm in range of D, which are needed to be merged and generate an approximation of M(D). For each algorithm, the relevant parameters of pre-built models must be materialized. The models are used and merged with help of reusing the parameters which have been stored in the repository. Each algorithm has its own method for merging pre-built models which will be elaborated later in this chapter.

When we want to train the model on a smaller number of data points, we can candidate a portion of the original points that can provide accurate enough model compared to the model built from scratch. The candidate points and their associated weights should be selected in a way that the model which is built on top of them have enough accuracy.

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 [6]. Specifically, a weighted set C is said to be a coreset for dataset D, if $(1-\epsilon)\emptyset_D(.) \leq \emptyset_D(.) \leq (1+\epsilon)\emptyset_D(.)$ where $\emptyset(.)$ 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 + ε) of SSE obtained by running K-Means on the entire data.

For selecting the coreset points, the natural approach of uniform sampling often does not work well in practice or requires very large sample size for sufficient approximation. So,

for each ML algorithm, we explain the method than can be leveraged to get the best points as the data points coresets.

Coresets are a natural solution to the problem of obtaining ML models with tunable approximation – by varying the value of epsilon (), we can achieve coresets with higher or lower approximation. Naturally, lower requires a larger size coreset.

These two approaches enable a data analyst to tradeoff 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 (1+ ε) of SSE obtained by running K-Means on the entire data.

4.1. K-Means

4.1.1. Model Merging

Given an arbitrary query q, our objective is to efficiently output K centroids Cq such that SSE for Cq is close to SSE of Cq where Cq is the set of centroids obtained by running K-Means algorithm from scratch on the entire Dq. We seek to do this by only using the information (Mi) - the cluster centroids and the number of data points assigned to it. K-Means++ [4] 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 O(logK) 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 Cw represent the union of all cluster centroids from all the models Mi Mq. As before, if there were some tuples in Dq that were not covered by models Mq, one can readily run K-Means on those tuples and add those cluster centroids to Cw. For each centroid cj Cw, 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 Cw 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 3 provides an illustration. The time complexity of the algorithm is to store the cluster centroids for each of the partition which is O(Kd) in which k is the number of clusters and d is the number of features of dataset..

Algorithm 1 Merging K-Means Centroids

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

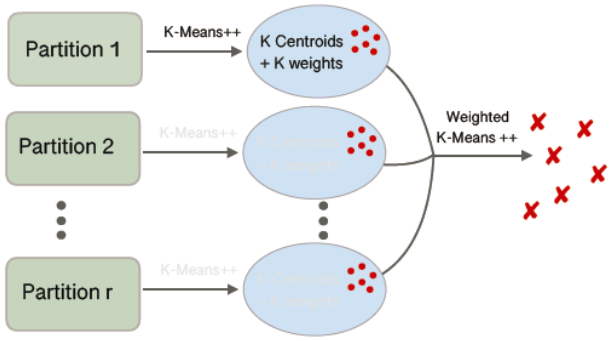

Figure 3 Merging of K-Means centroids

4.1.2. coreset approximation

Coresets are compact representations of data sets and models trained on a coreset points are provably competitive with models trained on the full data set. The coreset construction is based on important sampling. Let q(x) be any probability distribution on X and Q any set of k centers in R^d. Then the quantization error may be rewritten as:

$$\phi_X(Q) = \sum_{x \in X} q(x) \frac{d(x, Q)^2}{q(x)}$$

The quantization error can hence be approximated by sampling m points from X using q(x) and assigning them weights inversely proportional to q(x). Olivier Bachem et.al. suggested the following proposal distribution [7]:

$$q(x) = \frac{1}{2} \underbrace{\frac{1}{|X|}}_{(A)} + \frac{1}{2} \underbrace{\frac{d(x, \mu(X))^2}{\sum_{x' \in X} d(x', \mu(X))^2}}_{(B)}$$

The resulting coreset construction is provided as pseudo code in Algorithm 2 and is extremely simple and practical: One calculates the mean of the data and then uses it to compute the importance sampling distribution q(x). Finally, m points are sampled with probability q(x) from X and assigned the weight 1/(m·q(x)). The algorithm only requires two full passes through the data set resulting in a total computational complexity of O(nd). There is no additional linear dependence on the number of clusters k as in previous constructions which is crucial in the setting where k is even moderately large.

Algorithm 2 Lightweight coreset construction

Require: Set of data points X, coreset size m

1: $\mu \leftarrow$ mean of X
2: **for** $x \in X$ **do**
3: $\quad q(x) \leftarrow \frac{1}{2} \frac{1}{|X|} + \frac{1}{2} \frac{d(x, \mu)^2}{\sum_{x' \in X} d(x', \mu)^2}$
4: **end for**
5: $C \leftarrow$ sample m weighted points from X where each point x has weight $\frac{1}{m \cdot q(x)}$ and is sampled with probability $q(x)$
6: **Return** lightweight coreset C

4.2. GMM

4.2.1. Model Merging

In this part we will discuss about how to merge to models which are GMM type. Given a query q, we assume the availability of pre-built ML models $M_q = M_1, \ldots, M_r$ that are parameterized by $(M_i) = [(w_{j1}, \mu_{j1}, \sum_{j1}), (w_{j2}, \mu_{j2}, \sum_{j2}), \ldots, (w_{jk}, \mu_{jk}, \sum_{jk})]$. We seek to post-processes the Gaussian mixtures obtained from each partition to approximate the GMM on D_q. There are totally K_r Gaussian components that we must process to just K components. 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, so we cannot use the method of model merging for K-Means models which were parametrized by the vector of centroids. Given a set of data points, GMM works by estimating the parameters of a Gaussian mixture that maximizes the likelihood. Instead, we used another approximate method for merging in the easy, yet with a good enough accuracy as follows: We begin by normalizing the prior probabilities of all the Gaussian mixtures by $w_{ji} = w_{ji}/Z$ where $Z = \sum_{j=1}^{r} \sum_{i=1}^{k} w_{ji}$. 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 3 provides the pseudocode and Figure 4 an illustration.

Algorithm 3 Iterative Merging of Gaussian Components

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

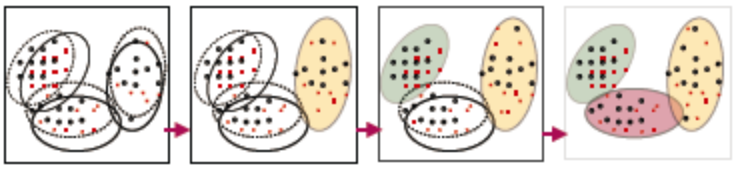

Figure 4 Merging of GMM

Selecting Components to Merge: One of the key steps in Algorithm 3 is the selection of two Gaussian components to merge. There has been extensive work in statistical community about appropriate measures to select components for merging [8]. 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 multivariate Gaussian distributions N1(μ_1, $\sum_1$) and N2(μ_2, $\sum_2$), their Bhattacharyya distance is computed as:

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

$$\Sigma = \frac{\Sigma_1 + \Sigma_2}{2}$$

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 multivariate Gaussian distributions N1(μ_1, $\sum_1$) and N2(μ_2, $\sum_2$) with mixing weights w_1 and w_2, the merged component is described by N(μ,$\sum$) with mixing weights w where,

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

4.2.2. coreset construction for GMM

The method for selecting the candidate points in GMM model is a simple procedure which iteratively samples a small number of points and removes half of the data set closest to the sampled points, provides a sufficiently accurate first approximation B for this purpose. This initial clustering is then used to sample the data points comprising coreset C according to probabilities which are roughly proportional to the squared distance to the set B. This non-uniform random sampling can be understood as an importance-weighted estimate of the log-likelihood $L(D_j)$, where the weights are optimized in order to reduce the variance[9]. The pseudocode for obtaining the approximation B, and for using it to obtain coreset C is given in Algorithm 4. This algorithm can be achieved in O(ndk).

Algorithm 4 Coreset construction

Input: Data set D, ε, δ, k
Output: Coreset $C = \{(\gamma(\mathbf{x}_1), \mathbf{x}_1), \ldots, (\gamma(\mathbf{x}_{|C|}), \mathbf{x}_{|C|})\}$
$D' \leftarrow D$; $B \leftarrow \emptyset$;
while $|D'| > 10dk\ln(1/\delta)$ **do**
  Sample set S of $\beta = 10dk\ln(1/\delta)$ points uniformly at random from D';
  Remove $\lceil |D'|/2 \rceil$ points $\mathbf{x} \in D'$ closest to S (i.e., minimizing $\mathrm{dist}(\mathbf{x}, S)$) from D';
  Set $B \leftarrow B \cup S$;
Set $B \leftarrow B \cup D'$;
for *each* $b \in B$ **do** $D_b \leftarrow$ the points in D whose closest point in B is b. Ties broken arbitrarily;
for *each* $b \in B$ *and* $\mathbf{x} \in D_b$ **do**
  $m(\mathbf{x}) \leftarrow \left\lceil \frac{5}{|D_b|} + \frac{\mathrm{dist}(\mathbf{x}, B)^2}{\sum_{\mathbf{x}' \in D} \mathrm{dist}(\mathbf{x}', B)^2} \right\rceil$;
Pick a non-uniform random sample C of $10\lceil dk|B|^2\ln(1/\delta)/\varepsilon^2 \rceil$ points from D, where for every $\mathbf{x}' \in C$ and $\mathbf{x} \in D$, we have $\mathbf{x}' = \mathbf{x}$ with probability $m(\mathbf{x})/\sum_{\mathbf{x}' \in D} m(\mathbf{x}')$;
for *each* $\mathbf{x}' \in C$ **do** $\gamma(\mathbf{x}') \leftarrow \frac{\sum_{\mathbf{x} \in D} m(\mathbf{x})}{|C| \cdot m(\mathbf{x}')}$;

4.3. Generalized Linear Models (GLM)

4.3.1 Model merging

Algorithm 4 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 Mq. This surprisingly simple algorithm works extremely well for most ML models and especially so for GLMs[2]. 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 4 AVGM: Average Mixture Algorithm

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

4.3.2. Coreset construction

For Logistic regression :

In logistic regression, the covariates are real feature vectors $X_n \epsilon R_D$, the observations are labels $Y_n \epsilon \{-1,1\}, \theta \in R^D$, and the likelihood is defined as:

$$p(Y_n \mid X_n, \theta) = p_{logistic}(Y_n \mid X_n, \theta) := \frac{1}{1+\exp\left(-Y_n X_n \cdot \theta\right)}.$$

Huggins has designed coreset construction algorithm and proved its correctness using a quantity $\sigma_n(\theta)$ called the sensitivity [3], which quantifies the redundancy of a particular data point n. the larger the sensitivity, the less redundant. In the setting of logistic regression, we have that the sensitivity is

$$\sigma_n(\Theta) := \sup_{\theta \in \Theta} \frac{N\,\phi(Z_n \cdot \theta)}{\sum_{\ell=1}^{N}\phi(Z_\ell \cdot \theta)}.$$

Intuitively, $\sigma_n(\theta)$ captures how much influence data point n has on the log-likelihood $L_n(\theta)$ when varying the parameter $\theta \epsilon \vartheta$, and thus data points with high sensitivity should be included in the coreset. Evaluating $\sigma_n(\vartheta)$ exactly is not tractable, however, so an upper bound $m_n \geq \sigma n(\vartheta)$ must be used in its place. Thus, the key challenge is to efficiently compute a tight upper bound on the sensitivity. The upper bound of sensitivity for any k clustering is proven to be:

$$\sigma_n(\mathbb{B}_R) \leq m_n := \frac{N}{1 + \sum_{i=1}^{k} |G_i^{(-n)}| e^{-R\|Z_{G,i}^{(-n)} - Z_n\|_2}}.$$

Which can be calculated in O(k) time. The size of the coreset depends on the mean sensitivity bound, the desired error. Combining these pieces we obtain Algorithm 5, which constructs an ε-coreset.

Algorithm 5 Construction of logistic regression coreset

Require: Data $\mathcal{D}$, k-clustering $\mathcal{Q}$, radius $R > 0$, tolerance $\varepsilon > 0$, failure rate $\delta \in (0, 1)$

for $n = 1, \ldots, N$ **do** ▷ calculate sensitivity upper bounds using the k-clustering
  $m_n \leftarrow \frac{N}{1+\sum_{i=1}^{k} |G_i^{(-n)}| e^{-R\|Z_{G,i}^{(-n)} - Z_n\|_2}}$
end for
$\bar{m}_N \leftarrow \frac{1}{N} \sum_{n=1}^{N} m_n$
$M \leftarrow \left\lceil \frac{c\bar{m}_N}{\varepsilon^2}[(D+1)\log \bar{m}_N + \log(1/\delta)] \right\rceil$ ▷ coreset size; c is from proof of Theorem B.1
for $n = 1, \ldots, N$ **do**
  $p_n \leftarrow \frac{m_n}{N\bar{m}_N}$ ▷ importance weights of data
end for
$(K_1, \ldots, K_N) \sim \text{Multi}(M, (p_n)_{n=1}^{N})$ ▷ sample data for coreset
for $n = 1, \ldots, N$ **do** ▷ calculate coreset weights
  $\gamma_n \leftarrow \frac{K_n}{p_n M}$
end for
$\tilde{\mathcal{D}} \leftarrow \{(\gamma_n, X_n, Y_n) \mid \gamma_n > 0\}$ ▷ only keep data points with non-zero weights
return $\tilde{\mathcal{D}}$

For svm:

The algorithm is based on the idea that for any given dataset P, we assign an importance $\gamma(p_i)$ to each data point pi and then sample from the dataset according to the multinomial distribution emerging from this procedure. The crucial insight to this method is how we assign the importance. In particular, we use an over approximation of the sensitivity s(pi) of each point, i.e., $\gamma(p_i)$, to assign importance, which are obtained from the analysis from the previous section. Following the sampling of points, we

further assign weights u(pi) to each data points, which are proportional to the number of times the point has been sampled.

The overall method to compute the desired coreset is outlined in Algorithm 6. Given a set of input data P, an error parameter ε, and the desired failure probability δ, the algorithm returns an ε -coreset (S, u) from the query space F with probability at least 1- δ . In Line 2 we compute the importance of a point, i.e., the upper bound on the sensitivity s(pi) of a point pi. In Line 4, we compute the necessary number of samples to include in (S, u), and we then sample from the resulting multinomial distribution, see Line 5. More details are in [10].

Algorithm 6 CORESET$(\mathcal{P}, \varepsilon, \delta)$

Input: A set of training points $\mathcal{P} \subseteq \mathbb{R}^d$ containing n points, an error parameter $\varepsilon \in (0,1)$, and failure probability $\delta \in (0,1)$.

Output: An ε-coreset $(\mathcal{S}, u)$ for the query space $\mathcal{F}$ with probability at least $1-\delta$.

1 **for** $i \in [n]$ **do**

2 $\quad \gamma(p_i) \leftarrow \frac{1}{n} + \frac{\log n + ||x_i||_2 \log^2 n}{n}$

3 $t \leftarrow \sum_{i \in [n]} \gamma(p_i)$

4 Let

$$m \leftarrow \Omega\left(\frac{t}{\varepsilon^2}\left(d \log t + \log\left(\frac{1}{\delta}\right)\right)\right),$$

5 $(K_1, \ldots, K_n) \sim \text{Multinomial}\,(m, \pi_i = \gamma(p_i)/t \;\; \forall i \in [n])$

6 $\mathcal{S} \leftarrow \{p_i \in \mathcal{P} : K_i > 0\}$

7 *//Compute the weights* $u : \mathcal{P} \to \mathbb{R}_{\geq 0}$ *for every point* $p_i \in \mathcal{S}$.

8 **for** $i \in [n]$ **do**

9 $\quad u(p_i) \leftarrow \frac{tK_i}{\gamma(p_i)|\mathcal{S}|}$

10 **return** $(\mathcal{S}, u)$

4.4. Combining the coreset points

Our approach consists of two phases. In the pre-processing phase, we compute C coreset points for each of the pre-built models. In the runtime phase, we identify the set of partitions Pq that could be used to answer q. We construct a coreset for all the tuples that were not covered by pre-existing partitions. For combining the points, coreset compression method is implemented. We select the topmost weighted points from each partition is selected and the number of points is proportional to the number

of data points which has been used for the model construction to the total number of data points in all the 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. Noted that in this method the final number of coreset points after combination will be C.

4.5. Optimization Consideration

4.5.1. Choosing ml model to reuse

After building the pre-process models and store them in the repository, we need to identify an optimal execution strategy. In other words, we need to build an approximate ML model efficiently from set of materialized models and an analytic query.

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 $[lb_0, ub_0]$ is considered relevant if it is a subset of q = [lb, ub]. For example, for q = [250K, 1M], the model M1= [1, 100] is not relevant. Second, we collect the set of distinct lb, ub values from the relevant models including q. As an 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 eij exists between nodes vi and vj if vi < vj . If there exists a model with lb and ub corresponding to vi and vj , then weight(eij) = C_{Merge}(vi , vj). This corresponds to the cost of directly using this model. If not, weight(eij) = C_{Build}([250K, 300K]) for the merging approach and weight(eij) = $C_{coreset}$([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 eij = (vi, vj) in the shortest path either corresponds to a pre-existing ML model built on (vi, vj) or requires one to build one between (vi, vj). Algorithm 7 provides the pseudocode for this approach.

Algorithm 7 Optimal Execution Strategy

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

4.5.2. Selecting Models for Prebuilding

Suppose we are given a 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 consider the problem of selecting L models to materialize so as to maximize the number of queries in Q that can be speed 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 = {q1 = [lb1; ub1], q2 = [lb2; ub2] , … qM = [lbM, ubM]} , 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 7 to estimate the optimal cost of building a given query. The difference between Cost({}) and Cost({ri1, ri2 , …}) provides the utility of choosing models ri1 , ri2 , … 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 ri 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. For example, if one of the dimensions is Country, then one could choose to pre-build models for the L largest countries.

Chapter 5: User interface

As it is shown in Fig. 5, the user interface of ApproxML consists of two main sections: building partial models and building approximate models. Each section is described in detail as follows.

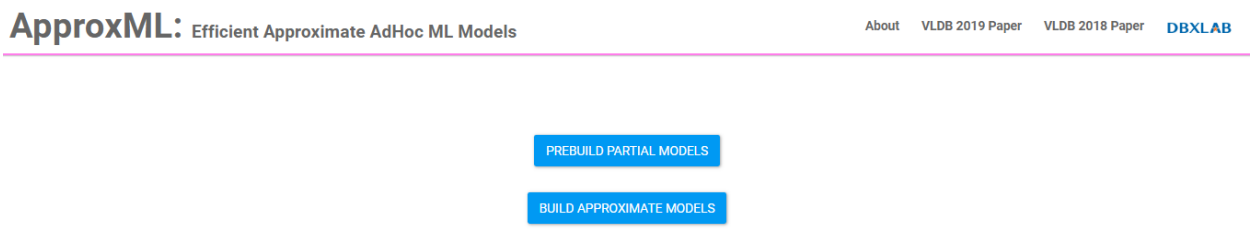

Figure 5 Main user interface of ApproxML

5.1. building approximate models

In this section the user can submit an analytic query and customize the following options for approximate ML model.

Dataset: The user will select a dataset in this section. Each dataset based on the datatypes and labels is assigned to a suitable ML model category i.e. classification and clustering. For any selected dataset, appropriate query range options for customizing the query becomes available. For example for Flights dataset, the user can customize the analytic query range by specifying the FROM and TO parameters with min and max of 1 and 365, respectively, as day of the year (Figure 6).

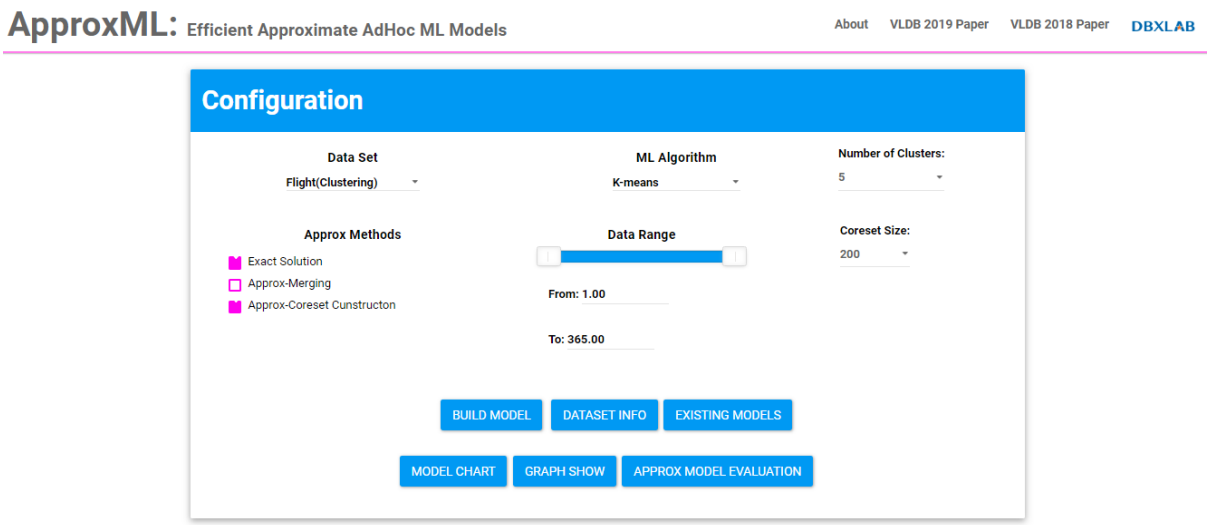

Figure 6 ApproxML: building approximate models configuration for flight dataset

ML algorithm: The user can choose between Logistic Regression and Linear SVM for classification task and K-means and Gaussian Mixture Model for clustering task. If clustering option is chosen, the number of clusters should be specified as well.

Approximation method: The user has the option to select between the approximate and exact models. If she chooses the exact solution, the entire data for the given analytic query will be retrieved from the selected dataset, and the exact model will be built on the entire data from scratch. If Approx-merging or Approx-coreset-construction is selected, the user can then choose between model merging and coreset-based methods. Model merging/Coreset-based: Based on the user's input in this section, the approximate model will be built using either model merging or the coreset-based methods. If coreset-based method is chosen, a coreset size should also be selected. Figure 4 shows the configuration for building a K-means clustering model on the Flight (Clustering) dataset using exact and coreset-based approach with coreset size of 200 for the data between 1 and 365. As another example, Figure 7 illustrates the configuration panel for building an SVM classifier on Santander (Classification) dataset using the data from 5.43 to 17.00 through a model merging approach and exact model.

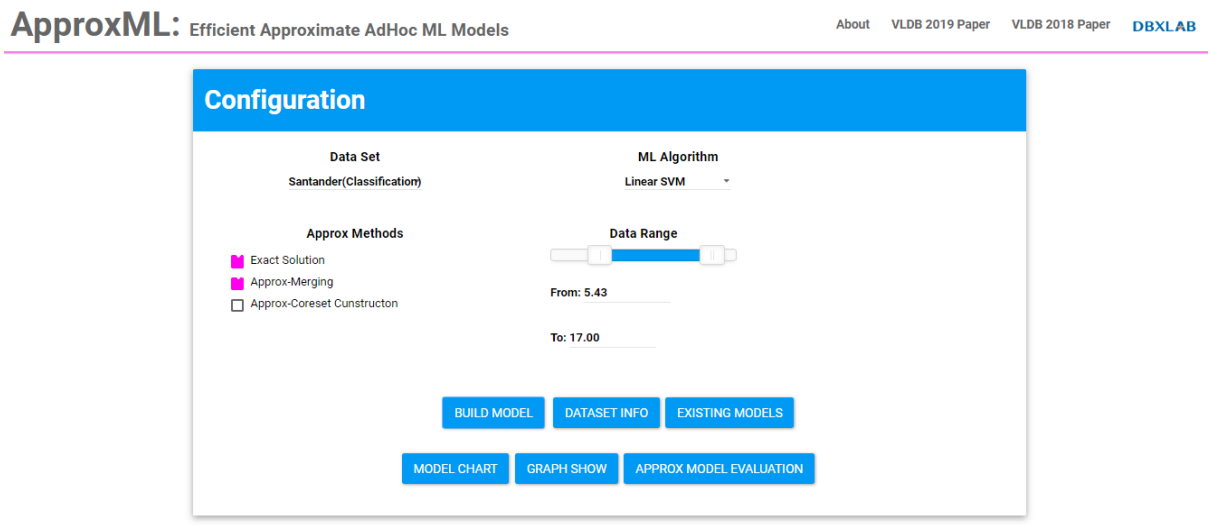

Figure 7 ApproxML: building approximate models configuration for Santander dataset

Functionalities:

There is a total of 6 option that can be selected after setting the criteria. The first one is for 'building models' based on the range and criteria selected in previous paragraphs. In the results section of ApproxML, quantitative measures of the generated ML model are reported to the user. For classifier, accuracy as well as time of building the model are reported to the user. In clustering scenario, accuracy in terms of Adjusted Rand Index (ARI) and likelihood are shown for K-means and GMM respectively. Figure 8 illustrates an example for the result, when all the options for approximation method have been selected.

Performance Comparison

Method	Accuracy	Time	Cost
Exact	1	12.06	217.89
Merging	0.8	0.71	12.05
Coreset	0.67	0.12	0.57

Figure 8 ApproxML: results for approximate models

The second option is ‘dataset info’ which shows general information about the dataset selected.

The ‘existing model’ option demonstrate a table containing all the prebuilt models that have been built between the range selected in the configuration and their meta data exists in the repository. The table shows important information about the data such as ML model information as well as date and time of their generation (Fig. 9).

Search Results

Row	Dataset	Algorithm	From	To	Num_k	Model	DateTime
1	santander	svm	12.64	12.8	NA	Merge	11/24,23:43
2	santander	svm	3.05	3.26	NA	Merge	11/24,23:43
3	santander	svm	3.78	13.96	NA	Merge	11/24,23:41
4	santander	svm	12.41	12.64	NA	Merge	11/24,23:34
5	santander	svm	4.78	5.134	NA	Merge	11/24,23:34
6	santander	svm	4.62	10.59	NA	Merge	11/24,23:33
7	santander	svm	12.156	12.41	NA	Merge	11/24,23:22
8	santander	svm	5.91	13.24	NA	Merge	11/24,23:21
9	santander	svm	5.23	15.25	NA	Merge	11/24,23:16
10	santander	svm	13.828	16.503	NA	Merge	11/24,23:11
11	santander	svm	11.153	13.828	NA	Merge	11/24,23:11
12	santander	svm	8.478	11.153	NA	Merge	11/24,23:11

Figure 9 list of prebuilt models in repository

The prebuilt models in the repository can be visually demonstrated in another option of Model Chart. As it is shown in Figure 10, the horizontal axis represents the selected feature of the dataset and the models have been built in the different ranges are colored blocks in the specified ranges. Each ML algorithm has its own color. By moving the curser on each block, it will show the information including ML algorithm and the range of that block.

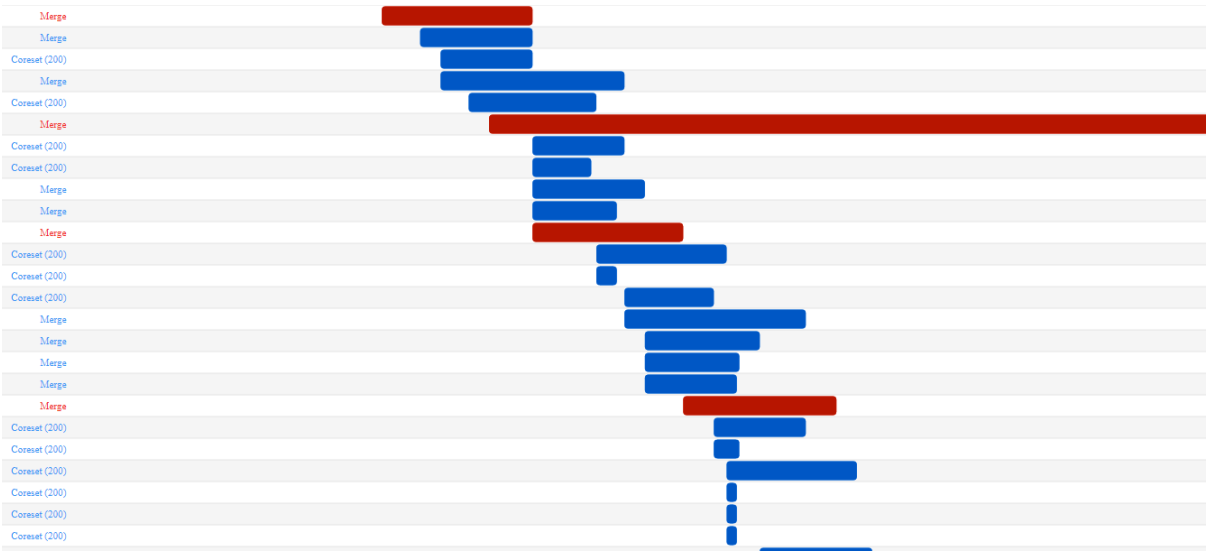

Figure 10 demonstration of prebuilt models in the chart

In the graph show option, the user can see which pre-built models are retrieved from the pre-built model repository and reused for this particular approximate ML model. Figure 8 shows an example of graph for an ApproxML system. As it is explained in the legend, each color of the edges and nodes have a specific meaning. The colored edges are selected among all the edges with the shortest path methods (Dijkstra in our project) and building the model on the query from 2.87 to 11.55 will have minimum cost with the colored path illustrated in figure 11.

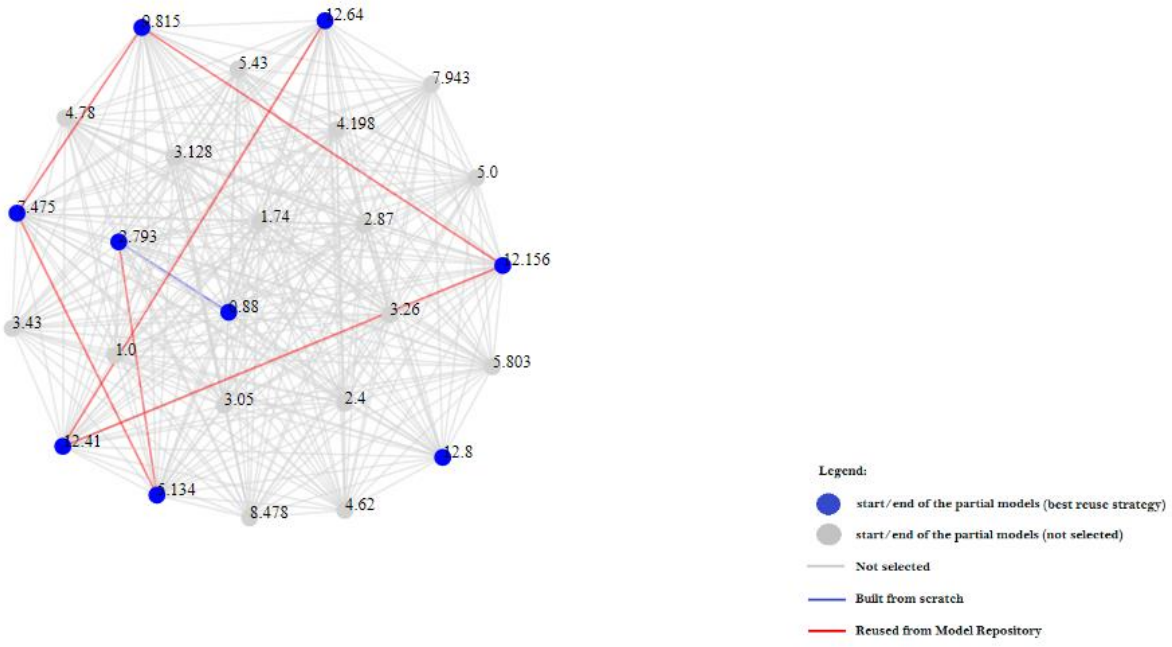

Figure 11 models have been used in building the specified model

5.2. building partial models

In this section, the user can select a dataset, method of approximation and customize the parameters of an ML model. There are three different way for setting the partitions that models will be built on top of them. These three ways are : workload, partition and range. User can select workload and upload the workload file of desired ranges for model construction. The data is then retrieved from the dataset, partitioned into optimum partitions, the exact model is built for each partition, and the corresponding meta data for the models are saved in the ML model repository. For partition section the user selects the number of partitions k. Data will be divided into k equal partitions and exact model is built for each partition, and the corresponding meta data for the models are saved in the ML model repository. For range section, range of one partition in the slide bar can be selected. The exact model is built for the partition, and the corresponding meta data for the model are saved in the ML model repository. Figure 12 shows an example of this section.

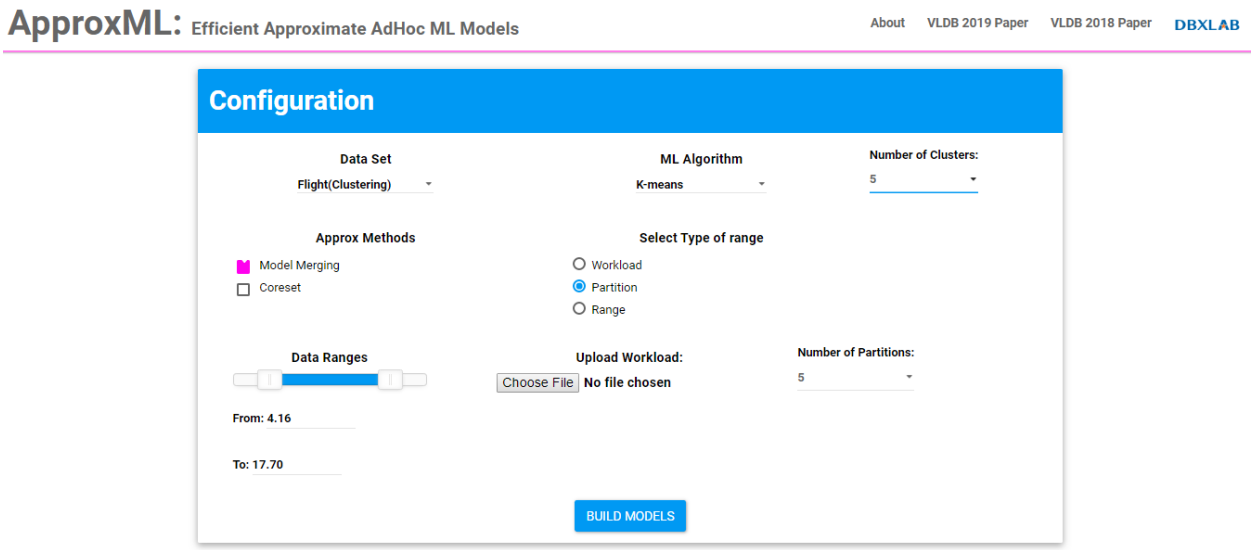

Figure 12 building partial models section

5.3. System Implementation

ApproxML's backend is implemented in Python 3.6. Scikit- Learn (version 0.19.1) was used to train the ML models. Pandas library was used to save the query results in data frames. We used Flask for session management and database connection tools. Datasets: For classification, we used Santander datasets from the UCL repository which includes 10500 rows and 200 features and Flight dataset with one attribute as a label. For evaluating clustering algorithms, we used Flight dataset with 580000 rows and 32 features.

Chapter 6: Conclusion

We demonstrate ApproxML, a system that efficiently constructs approximate ML models for new queries from previously constructed ML models by leveraging the concepts of model materialization and reuse. In order to generate approximate ML models, ApproxML takes a two-phase approach. In the pre-processing phase it partitions the data and builds exact ML models on each partition and stores their meta data in a pre-built model repository. During the run-time phase, it reuses the pre-built models and combines them efficiently to create an approximate model for a new analytic query.

Reference :

1. Hasani S, Ghaderi F, Hasan S, Thirumuruganathan S, Asudeh A, Koudas N, et al. ApproxML: Efficient Approximate Ad-Hoc ML Models Through Materialization and Reuse. Proceedings VLDB Endowment. 12. Available: http://www.vldb.org/pvldb/vol12/p1906-hasani.pdf

2. Hasani S, Thirumuruganathan S, Asudeh A, Koudas N, Das G. Efficient Construction of Approximate Ad-hoc ML Models Through Materialization and Reuse. Proceedings VLDB Endowment. 2018;11: 1468–1481.

3. Huggins J, Campbell T, Broderick T. Coresets for scalable Bayesian logistic regression. Advances in Neural Information Processing Systems. 2016. pp. 4080–4088.

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

5. Verbeek JJ, Vlassis N, Kröse B. Efficient greedy learning of gaussian mixture models. Neural Comput. 2003;15: 469–485.

6. Agarwal PK, Har-Peled S, Varadarajan KR. Geometric approximation via coresets. Combinatorial and computational geometry. 2005;52: 1–30.

7. Bachem O, Lucic M, Krause A. Scalable K -Means Clustering via Lightweight Coresets. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. New York, NY, USA: ACM; 2018. pp. 1119–1127.

8. Hennig C. Methods for merging Gaussian mixture components. Adv Data Anal Classif. 2010;4: 3–34.

9. Feldman D, Faulkner M, Krause A. Scalable Training of Mixture Models via Coresets. In: Shawe-Taylor J, Zemel RS, Bartlett PL, Pereira F, Weinberger KQ, editors. Advances in Neural Information Processing Systems 24. Curran Associates, Inc.; 2011. pp. 2142–2150.

10. Baykal C, Liebenwein L, Schwarting W. Training Support Vector Machines using Coresets. arXiv [cs.DS]. 2017. Available: http://arxiv.org/abs/1708.03835