Efficient Construction and Explanation of Machine Learning Models through Database Techniques

by

SONA HASANI

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

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2020

To maman and baba who gave me wings to fly.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my PhD advisor Dr. Gautam Das for his immense knowledge, patience, and continued support. His trust in me has encouraged me to be a responsible and independent researcher who is willing to explore new challenges. I am always fascinated by his unique ability in transforming a vague and complex problem into a set of clear and concrete sub-problems. As an amazing teacher of my algorithm courses, he has taught me the best ways to approach and think about problems and algorithms. Last but not least, Dr. Das always supported team collaboration and gave me the opportunity to work with many great researchers all around the world. His remarkable style in research, teaching, and mentoring made my PhD journey a most memorable experience.

I would like to sincerely thank my dissertation committee: Dr. Ramez Elmasri, Dr. Leonidas Fegaras, and Dr. Gergely Zaruba, for their insightful comments and encouragement during my study. My sincere gratitude also goes to Dr. Chengkai Li for his support and for teaching me the elements of research. He mentored me when I was a newcomer and transformed me into a researcher. His high standards, attention to detail, and dedication to his students were truly inspiring.

I am grateful to my amazing collaborators and mentors during my PhD journey. Without their precious support it would not be possible to conduct this research. My sincere thanks go to Dr. Saravanan Thirumuruganathan. He has been a great mentor and a wonderful friend since I started my PhD. I always admire his vast knowledge and skills, and his seemingly unlimited capacity to multitask. I would also like to thank Dr. Nick Koudas for introducing me to the world of machine

learning research and his very insightful comments and collaboration throughout this work. Without his collaboration, this research would not have been possible. During the past few years, I was fortunate to collaborate with Dr. Amal Isaiah on projects from the medical domain. His enthusiasm for exploring new avenues and his thirst for knowledge were really inspiring.

My sincere gratitude goes to Dr. Moshe Vardi and the LAPIS group from Rice university. I feel very fortunate to have had the chance to participate in Dr. Vardi's lectures and meetings. He welcomed me in the Rice community where I found professional support and friendship. Dr. Vardi has been a constant source of inspiration both as a scientist and as a human being.

I thank my dear friends and lab-mates at DBXlab, Jees, Shohed, Suraj, Yeshwanth, Farhad, Sadia, Mary, Lincoln, Faezeh, and Abolfazl for the stimulating discussions, for their support and for all the fun we had. I would also like to thank members of IDIR lab for their friendship. In particular, I would like to thank my friend/office-mate Fatma for all the love, support, and freshly brewed coffee she gave me throughout these years. I would like to extend my appreciation to the amazing faculty and staff at the computer science and engineering department at the University of Texas at Arlington for their support.

Last but not the least, I would like to thank my family: Special thanks to my husband/best friend Pooya. Pooya has been by my side every single moment since the day I started applying for this position until now that I am completing this dissertation. His continuous love and support made every second of my PhD journey memorable. My greatest appreciation goes to my amazing parents, maman and baba and my wonderful sister/best friend Dorna. They have been my true inspiration during my whole life. Their unconditional love and support have been the main source of encouragement with every step I take in life.

August 3, 2020

ABSTRACT

Efficient Construction and Explanation of Machine Learning Models through Database Techniques

SONA HASANI, Ph.D.

The University of Texas at Arlington, 2020

Supervising Professor: Gautam Das

Machine learning (ML) has been widely adopted in the last few years and it has had an undeniable impact on the ways many organizations make decisions. While great advances have been made in developing new ML algorithms and applications, there is a major need for scalable ML solutions in order to meet the demands of the Big data era. In this dissertation, we focus on improving the efficiency of two main machine learning solutions through database techniques: i) efficient construction of machine learning models, and ii) efficient explanation of machine learning models for multiple predictions.

First, we introduce application of machine learning in complex analytic processing. Recently, there has been extensive interest in the database community for supporting quick and interactive ad-hoc analytic queries on ML models trained over large datasets. Data is typically stored in large data warehouses with multiple dimension hierarchies. In this dissertation, 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*. We pro-

pose 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 also propose a cost based optimization framework that identifies appropriate ML models to combine at query time.

The second ML problem we tackle in this dissertation is in the area of explanation. ML algorithms are increasingly used for automated decision making in diverse domains. The widespread use of ML models has necessitated the development of algorithms for explaining their predictions. Generating concise and accurate explanations often increases user trust and understanding of the model prediction. The research community has mobilized to develop sophisticated algorithms for generating explanations. Usually, the implementations of popular explanation algorithms are highly optimized for a single prediction. However, in practice, explanations often have to be generated in a batch for multiple predictions at a time. We propose a principled and lightweight approach for identifying redundant computations and several effective heuristics for speeding up multiple explanation generation. Our approach is inspired by Multi Query Optimization. Our techniques are general and could be applied to a wide variety of explanation algorithms.

For all the problems, we provide extensive experiments over real-world and synthetic datasets, using popular ML algorithms and popular explainers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Machine Learning (ML) has become an invaluable tool that is used by organizations to glean insight from their data. While ML has made incredible progress in the recent years in providing valuable insight, given the sheer size of data available it can immensely benefit from improved efficiency. In this dissertation we investigate how to use established database techniques in order to improve the efficiency of machine learning solutions. In particular, we look at two specific machine learning applications: Ad-hoc ML model construction for analytic queries and generating explanations for predictive ML models for multiple predictions. These two areas in ML are described briefly as follows.

1.1 Ad-hoc ML Models Construction for Analytic Queries

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 an ML model is often a major bottleneck and consumes a lot of time due to the sheer size of the datasets involved. 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*.

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 an ML model and using the model for analytic processing subsumes a large portion of analytic workflows. We argue that these ad-hoc analytic queries on ML models often exhibit a number of appealing properties that enable 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. The data chosen for analysis often belongs to explicit domain hierarchies over country, year, department, vendor, product category, etc.
- *Tolerance for Approximate ML Models.* 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 these models have been *materialized* (instead of being discarded), then one can build an approximate ML model for the superset by *reusing* the models for the various subsets.

In this dissertation, we investigate opportunities for model materialization and reuse to speed up analytic queries [1]. 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 dissertation, 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. 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.

1.2 Generating Explanations for Multiple Predictions

Nowadays Machine Learning has become an inseparable part of many industries. In various domains such as finance, management, and medicine Machine Learning is used to build predictive models. These models are trained on the available data and will be used to make a prediction for future data. Depending on the nature and size of the data, different machine learning algorithms can be used including regression, classification, and clustering. While some machine learning algorithms such as linear regression and decision trees are self-explanatory and are easy to understand, other algorithms such as deep learning or random forest are more difficult to interpret for domain experts. Many computer scientists, statisticians, and mathematicians have a general understanding about machine learning algorithms and how they work, but even for the simple ML algorithms many domain experts are unaware of their details. It is clear that Machine Learning algorithms are strongly dependant on their training data. ML algorithms report evaluation metrics

such as accuracy and confusion matrix to evaluate their prediction quality on the test data. Additionally, you can evaluate the probability assigned to the predicted label for a given data point as a measure of model's confidence in its prediction. However, studies show that even models with high accuracy can be unreliable because of their training data. In some domains these mistakes can lead to a disaster. For example, in medicine or crisis management fields if we trust an unreliable model it can have horrible consequences. Therefore, it is crucial to provide interpretable explanations about the prediction of an ML model to equip the domain expert with some additional knowledge and help him evaluate the trustworthiness of the model. In order to make ML models more transparent, a line of research is focused on generating human understandable explanations on ML algorithms' output. If the users cannot evaluate the trustworthiness of a model, they are unable to decide if they should use it or not. Since machine learning algorithms make their decision only based on their training data, if the training data is not fair, many problems can leak into the decision function of the ML models trained on that data.

Machine learning model explanations can be categorized into two broad categories: **Global Explanations**: There are several research studies that attempt to explain the global behaviour of a model. Although there exist globally interpretable ML models such as linear models or decision trees, many ML models are too complicated to be explained globally and as a whole. In such cases, even the global explanation can be very confusing and hard to understand. Therefore, some studies focus on providing several simpler explanations where each explanation is applicable to a part of the model. **Local Explanations**: Some models may be very hard to be explained as a whole but explanations can be provided to explain their behavior in different local neighborhoods. A few examples of local explanation methods include LIME [2], Anchor [3], and SHAP [4].

Alternatively, the ML explanation methods can be categorized into *model agnostic* and *model specific* methods. Model specific explainers have prior knowledge about what ML model is used and can leverage this additional information to provide more accurate and customized explanations for the outcome of the ML model. On the other hand, model agnostic explanation methods treat the ML model as a black box and generate the explanations without having additional knowledge about the internal detail of the ML model.

In this work, we focus on local explanations. Available local explanation algorithms are optimized for explaining *individual* predictions. In applications such as responsible AI [5, 6] or explanations summarization [7, 8], explaining data cleaning [8, 9, 10] there is a need to generate explanations for multiple predictions in a *batch* setting. Generating explanations often cannot be done in real-time (in milliseconds). For example, generating a single explanation using LIME takes 17, 15, 6, 6 and 5 seconds respectively for the 5 datasets evaluated in the paper.

So, an organization might pre-compute all the explanations in a batch setting and retrieve them as needed.

Sequentially processing one explanation at a time could take too much time. Using a cluster and parallelizing the explanation generation would give results faster but could waste precious computing resources. Given the rapidly increasing carbon footprint of ML algorithms [11], and the widespread deployment of explanation algorithms, there is a pressing need for smarter algorithms for this critical problem.

Our Proposed Approach. In this study, we propose a principled and scalable approach for generating explanations for multiple predictions. The key insight is that there are a number of redundant computations that could be avoided by leveraging techniques such as materialization and reuse. Our techniques were inspired by

Multi-Query Optimization (MQO) [12, 13]. Given a query workload, MQO seeks to identify common sub-expressions across queries, so that the reevaluation cost could be minimized. We describe a general set of heuristics for speeding up explanation algorithms and discuss how these ideas could be instantiated for popular explanation algorithms requiring minimal changes and very low overhead. Our proposed approach achieves significant speedup without compromising the explanation quality. In short, we adapt the techniques pioneered by the database community to solve a practical problem in data science.

1.3 Dissertation Organization

In chapter 2, we advocate treating ML models as first class citizens and investigate opportunities for model materialization and reuse to speed up analytic queries. Given the relative acceptability of approximate ML models in exploratory analysis, we believe that data scientists would willingly sacrifice model accuracy for near real-time model building. 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 chapter 3, we introduce ApproxML, a system implemented based on the solution described in chapter 2.

Finally, in chapter 4, we introduce the problem of Explaining Multiple Predictions and propose a principled and scalable approach for generating explanations for multiple predictions. We describe scalable algorithms for three popular explanation algorithms – LIME, Anchor, and SHAP – and present empirical analysis of the speedups achieved by our approach.

CHAPTER 2

Efficient Construction of Approximate Ad-Hoc ML Models through Materialization and Reuse

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 [1], 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.

2.1 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*.

2.1.1 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 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.

2.1.2 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?

2.1.3 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.

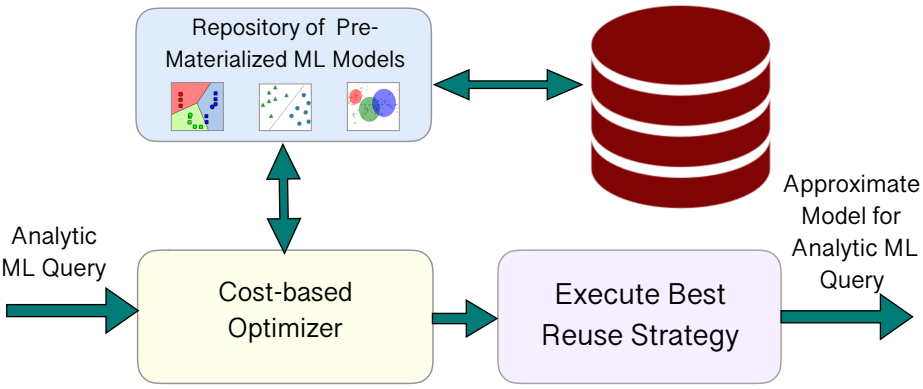


Figure 2.1. Overview of Our Approach.

- **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.

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 2.8 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 2.7.

2.2 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 time-stamp 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 $City \Rightarrow State \Rightarrow Country \Rightarrow Continent \Rightarrow All$.

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 $\mathcal{M}_{q_2} = \{\mathcal{M}([51, 100] \cup [501, 550]), M_2, M_3, M_4, M_5\}$

2.2.1 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 [14]. Given a set of data points $\mathcal{X}$ and centroids C, the SSE is defined as

$$SSE(\mathcal{X}, C) = \sum_{x \in \mathcal{X}} d(x, C)^2 = \sum_{x \in \mathcal{X}} min_{c \in C} ||x - c||_2^2 \tag{2.1}$$

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 [15]. 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.

2.3 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 regres-

sion coefficients while for SVM, it corresponds to the coefficients of the separating hyperplane.

2.3.1 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++ [16] 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 2.2 provides an illustration.

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 $\widetilde{C}_q$

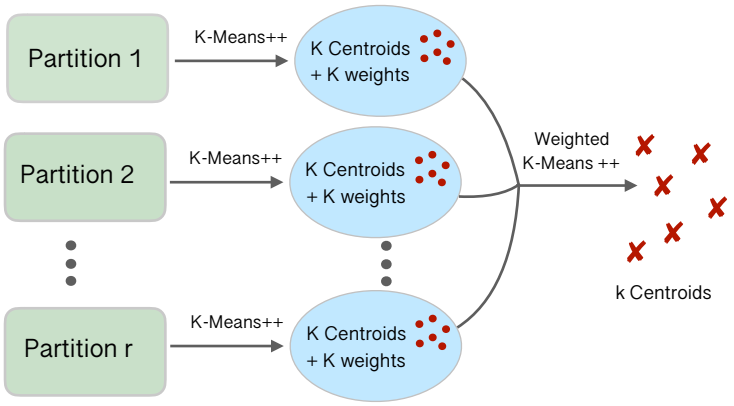


Figure 2.2. Illustration of Two Level K-Means Merging Approach.

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 [17, 18]. Please refer to the appendix of [19] for the proof. The only difference is that we

Table 2.1. 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

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 [19].

2.3.2 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 $\theta(M_j) = [(w_{j_1}, \mu_{j_1}, \Sigma_{j_1}), \quad \ldots, (w_{j_K}, \mu_{j_K}, \Sigma_{j_K})]$. 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.

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 [20]. 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 2.3 an illustration.

Algorithm 2 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

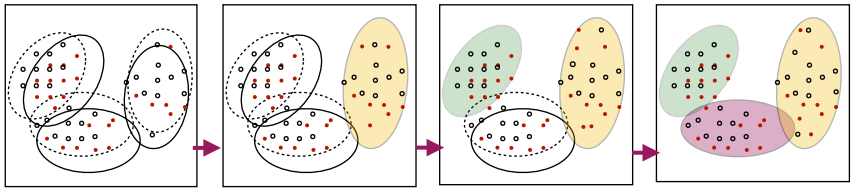


Figure 2.3. Merging for GMM.

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 community about appropriate measures to select components for merging [21, 22, 23]. 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:

$$\begin{aligned} D_B(\mathcal{N}_1, \mathcal{N}_2) &= \frac{1}{8}(\mu_1 - \mu_2)^T \Sigma^{-1} (\mu_1 - \mu_2) \\ &\quad + \frac{1}{2} \ln \left(\frac{|\Sigma|}{\sqrt{|\Sigma_1||\Sigma_2|}} \right) \\ \Sigma &= \frac{\Sigma_1 + \Sigma_2}{2} \end{aligned} \tag{2.2}$$

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 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 [21, 22, 23] is described by $\mathcal{N}(\mu, \Sigma)$ with mixing weights w where,

$$\begin{aligned} w &= w_1 + w_2 \\ \mu &= \frac{1}{w}[w_1\mu_1 + w_2\mu_2] \\ \Sigma &= \frac{w_1}{w} \left[\Sigma_1 + (\mu_1 - \mu)^T (\mu_1 - \mu) \right] \\ &\quad + \frac{w_2}{w} \left[\Sigma_2 + (\mu_2 - \mu)^T (\mu_2 - \mu) \right] \\ &= \frac{w_1}{w}\Sigma_1 + \frac{w_2}{w}\Sigma_2 + \frac{w_1 w_2}{w^2} \left((\mu_1 - \mu_2)(\mu_1 - \mu_2)^T \right) \end{aligned} \tag{2.3}$$

2.3.3 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 2.6, 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

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** $\theta(\widetilde{M}_q) = \frac{1}{r}\sum_{i=1}^{r}\theta(M_i)$

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) [24], Perceptron-type algorithms [25] and for a larger class of stochastic approximation models in [26]. This algorithm was formally analyzed in [26] and [27]. [26] 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 [27] 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.

2.4 Approximation by Coresets

The model merging approach proposed in Section 2.3 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 [15] 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 2.2 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 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

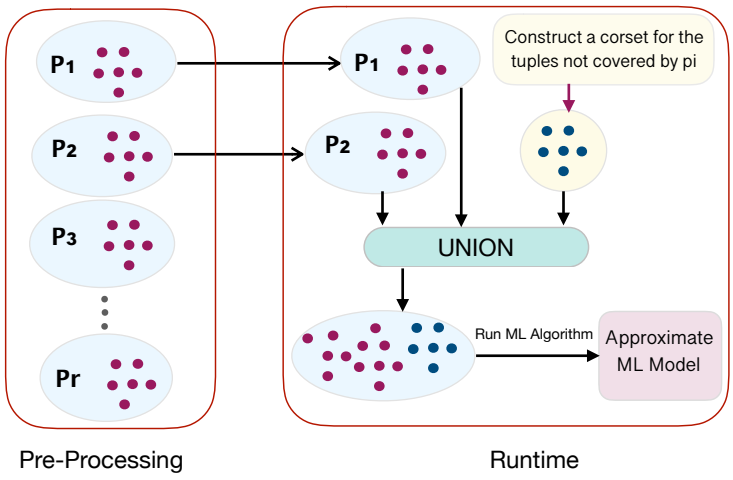


Figure 2.4. Two Phase Approach.

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.

2.4.1 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 [28].

$$p(x) = \frac{1}{2}\frac{1}{|D_i|} + \frac{1}{2}\frac{d(x, \mu(D_i))^2}{\sum_{x' \in D_i} d(x', \mu(D_i))^2} \tag{2.4}$$

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 [28]. Coreset algorithms exist for ML models such as GMM [29], SVM [30], Logistic Regression [31], 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

1: **Input:** Set of data points D_q, Coreset size m
2: **for** each $x \in D_q$ **do**
3: Compute contribution of x based on a surrogate of ϕ
4: $\forall x \in D_q$, Compute sampling probability $p(x)$
5: $C_i = m$ points from D_q chosen through importance sampling
6: $\forall x \in C_i$, compute the weight $w(x)$
7: **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 [28] requires two passes - one to compute the mean of all data points and one to perform importance sampling.

2.4.2 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 $\Omega(\frac{dk+\log\frac{1}{\delta}}{\epsilon^2})$ [28] 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.

2.5 Optimization Considerations

2.5.1 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 tradi-

tional 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.

Choosing ML Models to Reuse for Arbitrary Queries When the queries are range predicates on individual attributes, Algorithm 5 provides the optimal strategy.

Algorithm 5 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

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.

2.5.2 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.

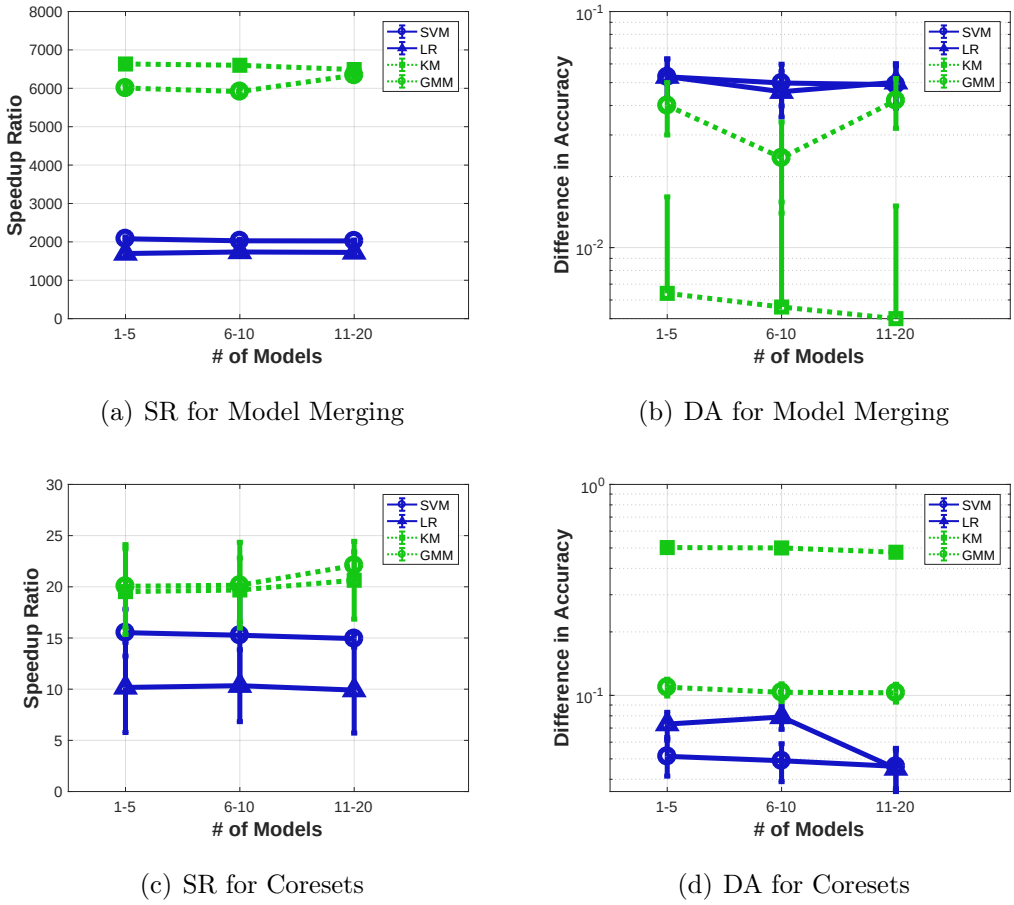
If the workload information is not available, one could use some simple strategies to choose which models to materialize. The equi-width strategy creates L partitions by splitting the range $[1, n]$ into L equal sized parts $\{[1, \lfloor \frac{n}{L} \rfloor], [\lceil \frac{n}{L} \rceil, \lfloor \frac{2n}{L} \rfloor] \ldots\}$. 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.

2.6 Experiments

2.6.1 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.



(a) SR for Model Merging

(b) DA for Model Merging

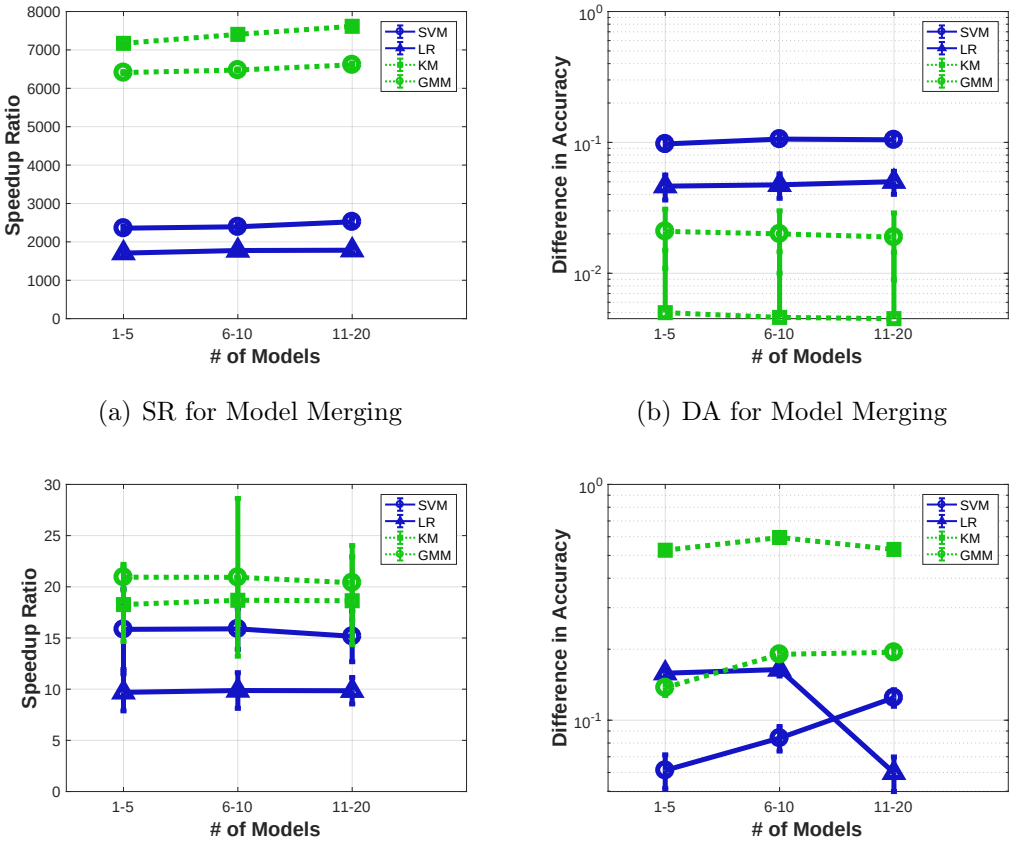
(c) SR for Coresets

(d) DA for Coresets

Figure 2.5. Evaluating approaches for building an approximate ML model for a workload of OLAP queries on the Hamlet datasets..

Scikit-Learn (version 0.19.1) was used to train the ML models [32]. Vowpal wabbit [33] (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 [34] - Movies, Yelp, Walmart, Books and Flights. We also selected two large datasets - SUSY and HIGGS from LibSVM repository [35, 36]. The size of the datasets vary from 200K tuples all the way to 11M tuples. For evaluating clustering, we generated



(a) SR for Model Merging

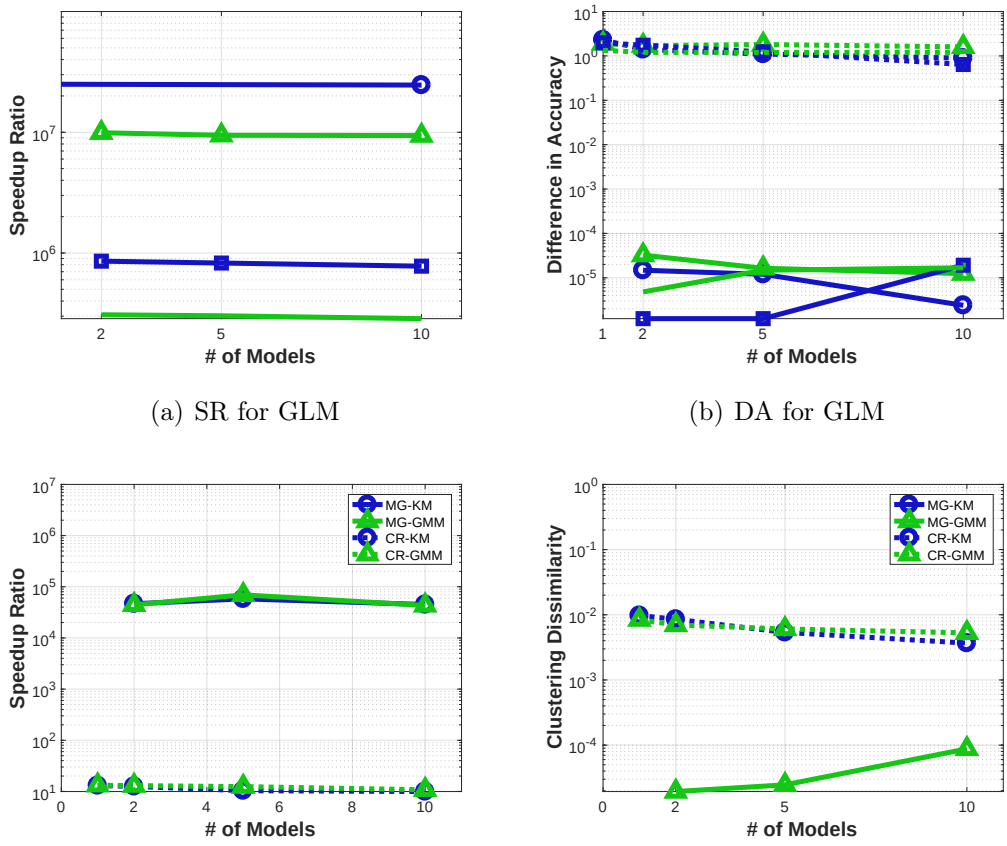
(b) DA for Model Merging

(c) SR for Coresets

(d) DA for Coresets

Figure 2.6. Evaluating approaches for building an approximate ML model for a workload of random queries on the Hamlet datasets..

a synthetic dataset with 5M data points, 20 features and 10 clusters using publicly available generator [32]. 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.



(a) SR for GLM

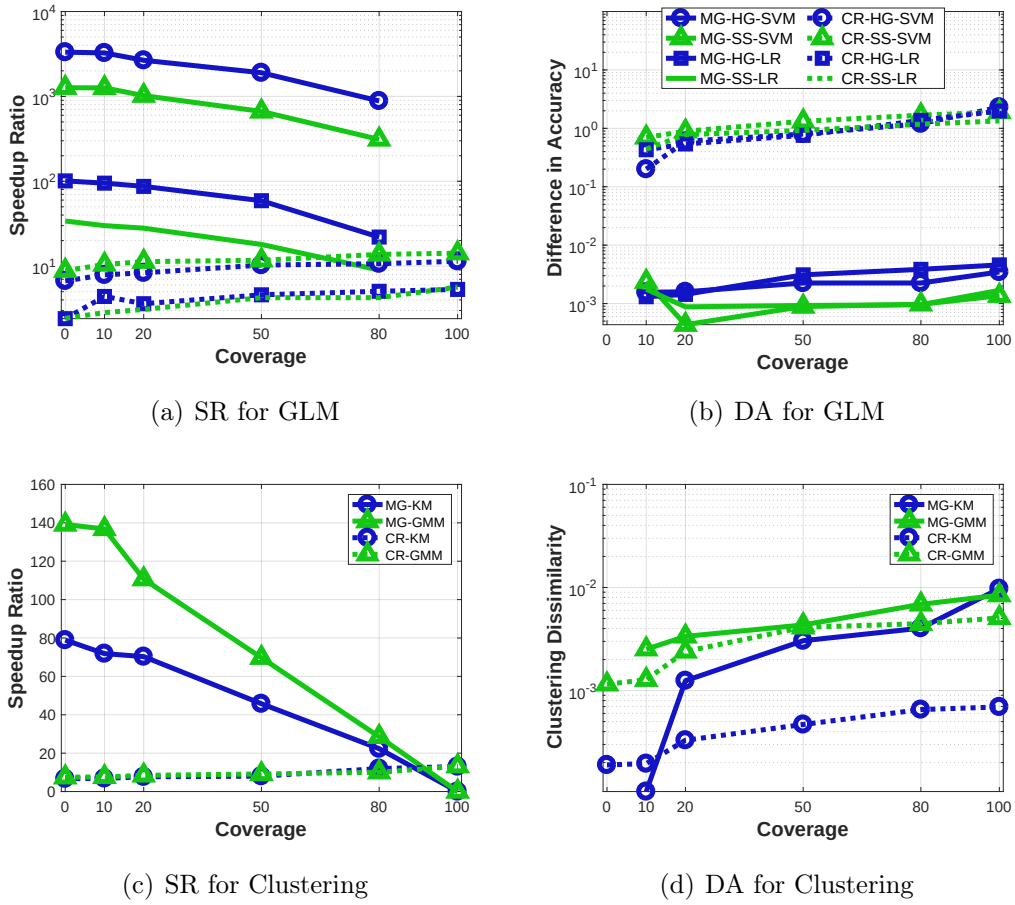
(b) DA for GLM

(c) SR for Clustering

(d) DA for Clustering

Figure 2.7. Evaluating approaches for building an approximate ML model for queries over the entire dataset. Abbreviations: Same as Figure 2.8. Legend for Figures 7a, 7b and 8a same as that of 8b..

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



(a) SR for GLM

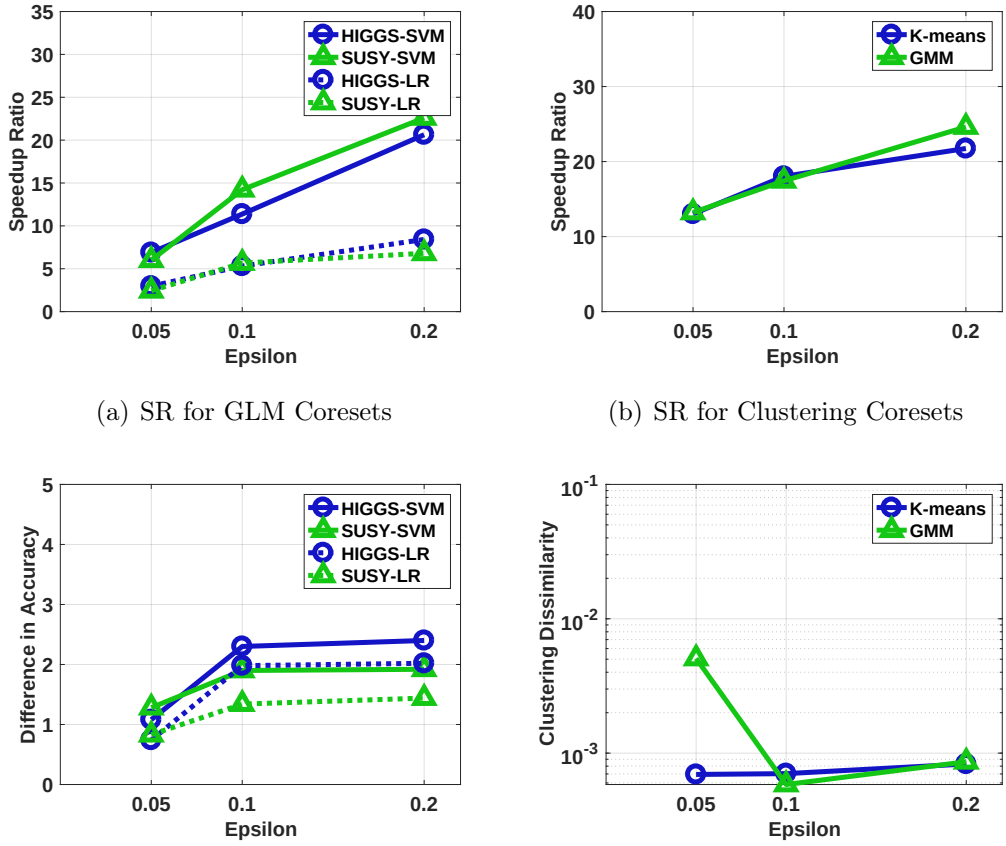
(b) DA for GLM

(c) SR for Clustering

(d) DA for Clustering

Figure 2.8. Evaluating impact of Coverage Ratio on building an approximate ML model. Abbreviations: MG/CR - Merging/Coresets; HG/SS - HIGGS/SUSY, KM - KMeans.

(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



(a) SR for GLM Coresets

(b) SR for Clustering Coresets

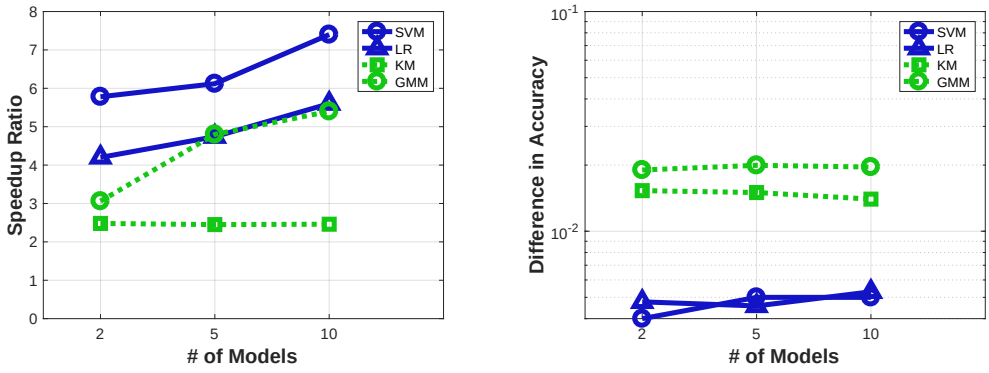
(c) DA for GLM Coresets

(d) DA for Clustering Coresets

Figure 2.9. Evaluating the impact of Approximation Ratio for Coresets..

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 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), K-Means and Gaussian Mixture Models (GMM). We used the implementations provided from scikit-learn. In our exper-



(a) SR (b) DA

Figure 2.10. Evaluating the impact of Coreset compression..

iments, 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 ([28, 29, 30, 31] 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.

2.6.2 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 2.5(a)-2.7(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 2.5(b)-2.7(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 2.5 and 2.6. 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 2.5(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 [27].

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 2.5(c) and 2.6(c) show the performance of the

coreset based approach for OLAP and random query workloads. Figure 2.7(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 2.5(d), 2.6(d), 2.7(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 2.8(a) and 2.8(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 2.9(a) and 2.9(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 2.9(c) and 2.9(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 2.10 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.

2.7 Discussion

The experimental results show that 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.

2.8 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 [37] and MADLib [38] also support similar integrations. There has been work on integrating ML primitives into database engines such as [39], using SQL style declarative languages for ML model training [40, 41]. Recent work also tried to use key concepts from data management for speeding up ML analytic tasks. These include materialization for feature selection [34, 42], using database style cost optimizer for predicting performance of ML tasks [43, 44, 41]. Incremental processing of ML based analytic queries was considered in [45]. Please

refer to [46] 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 [47, 48] and datasets [49]. [48, 50] focus on a unified data, model and lifecycle management for deep learning while [51, 52] 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 [53] 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 [54, 55]. 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 [56]. There has been extensive followup work that computed interesting statistical aggregates on OLAP cubes such as [57, 58, 59, 60]. The work [57] 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 [57] by building approximate ML models for data cubes from its component cubes. Another recent work [61] 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 [14, 62]. 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 [28, 63, 64, 65], GMM [29], kernel density estimation [66], logistic regression [31, 67], SVM [30, 68, 69], Bayesian networks [70] and so on. A recent work [42] 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 [71] 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.

2.9 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.

CHAPTER 3

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

Machine learning (ML) has gained a pivotal role in answering complex predictive analytic queries. Model building for large scale datasets is one of the 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 paper, we propose 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 variety of ML models such as generalized linear models for supervised learning, and K-means and Gaussian Mixture model for unsupervised learning.

3.1 Introduction

Machine learning has become a fundamental tool to gain insight from data. During the exploratory phase data scientists repetitively build numerous ML models in order to achieve higher accuracy. Consider a typical workflow of a data scientist. She issues a query to retrieve data from a data warehouse and builds an ML model (classification, clustering, etc.) on the retrieved data. The model is then used for analytic processing such as predicting revenue of a particular product. These analytic queries on ML models often have properties that allow a faster approach compared to building models from scratch. First, they usually have a specific business interpretation rather than being chosen at random. For example, a data scientist may want to retrieve data for a specific time period (month, semester, year) or for a specific location (city, state, country), etc. Moreover, data scientists are often willing to sacrifice some accuracy in the exploratory phase if they can obtain *good enough* approximate ML models very fast. In addition, data scientists and engineers from the same organization create many ML models for exploratory purposes that are discarded after one-time use. There is a very high chance that in future another member of that organization wants to build an ML model using the same data or a superset of it. Such properties render analytic queries good candidates for approximation as well as enable the potential to reuse their results fully or partially.

ML development revolves around experimentation. Recently systems such as mlflow [72] and modelDB [73] are developed to streamline the process by treating ML models as first class citizens and allow them to be stored with associated metadata. Nevertheless, building an ML model still remains a major bottleneck and consumes huge amount of time and resources due to sheer size of the datasets. If we can speed up the model building process by producing approximate models during an exploratory phase, it will dramatically improve the efficiency of the data scientist. In this paper, we introduce ApproxML, a system based on [1] that rapidly builds approximate ML models for analytic queries by utilizing two fundamental techniques *materialization* and *reuse* from database optimization. Suppose the analyst has access to pre-built ML models for each month and wants to build a model for the entire year. Currently, one builds the model from scratch using the data from the desired year. ApproxML allows one to combine the pre-built models to create model for that year much more efficiently.

ApproxML is a system that enables ML model approximation and reuse for popular supervised and unsupervised learning approaches. A demonstration session of ApproxML consists of three parts. The first part details the core features of the system by showcasing the approximate models supported. Next, the user will experience tradeoffs in accuracy of these approximate models compared with exact approaches. Finally, the user will be exposed on the practical utility of the materialization of ML models for a given workload and the ability to reuse them experiencing the associated speedups.

3.2 ApproxML Overview

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.

Technical Challenges 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 model 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.

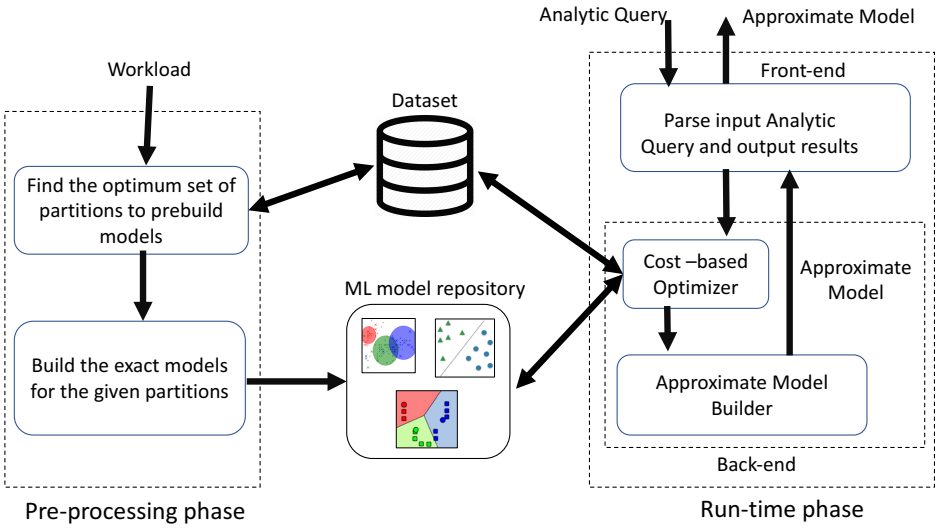


Figure 3.1. ApproxML Overview.

ApproxML offers two orthogonal methods for generating approximate ML models, a)*model merging* approach, and b)*coreset-based* approach. In model merging approach, the relevant models are carefully merged without going back to the data. The merging based approach is often extremely fast. If there is a need for approximate ML models with tunable approximation ratio, ApproxML offers coreset-based approach where it builds a model from a chosen set of 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. There has been extensive work on coresets for various ML models. For more details about the algorithms we used please refer to [1]. We implemented logistic Regression and linear SVMs as examples of Generalized Linear Models (GLMs) for supervised learning. In unsupervised learning, we implemented K-means and Gaussian Mixture Model (GMM). Figure 3.1 demonstrates the system overview of ApproxML.

3.2.1 Run-time phase

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 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 relevant pre-built models 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. Details of each step is explained as follows.

Cost-based optimizer To identify optimum pre-built models to reuse is a major problem. Consider a dataset with 1 million tuples from year 2010 to 2015. Let's assume the relevant ML models for every month and every quarter are materialized. Given a new query for the entire year of 2014, there are many ways to answer it. One can build a model from scratch. Another option is to retrieve the materialized models for all 12 months of the year 2014 and build an approximate model by combining them. Alternatively, one can build a model by combining the materialized models for four quarters of year 2014. There are many more possible options for this simple example. Cost-based optimizer will retrieve all relevant pre-built models from the repository of pre-built models for the given analytic query. The cost-based optimizer finds the optimum set of pre-built models by taking into account different costs involved such as (a) cost of building a model from original data, (b) cost of merging a model and (c) cost of building a coreset. It is possible that some parts of the input query are not covered by the pre-built models. In that case, the exact model for those partitions has to be built using the relevant data from original dataset. At the end, a set of ML models are passed to the *"approximate model builder"* component.

Approximate model builder In the coreset-based approach, the approximate model is built by training the intended ML model using the *union* of the coresets as training data. In model merging approach, for each ML algorithm the parameters of partial models are combined through a principled manner such that the objective function of the approximate model is very close to the objective function of the model built from the scratch. The reuse strategy for each algorithm is briefly explained as follows.

For K-means, given a set of K centroids and their corresponding weights, the weighted variant of K-means++ algorithm is applied to the union of the given centroids and the final K cluster centroids are returned as the output. For GMM, the mean vectors and covariance matrices of previously built GMM models on different partitions of the data are combined by running a hierarchical clustering algorithm and iteratively merging two closest Gaussian components till only K of them are left. We used Bhattacharyya distance to measure the distance between two Gaussian components. For supervised learning, the parameters of the approximate model are calculated by averaging the corresponding parameters of the pre-built models. For Logistic Regression, model parameter corresponds to the coefficients while for linear SVM, it corresponds to the coefficients of the separating hyperplane. Please refer to [1] for further details.

3.2.2 Pre-processing phase

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 highest benefit for the given workload. These selected models are materialized and stored in the pre-built model repository.

Repository of ML models In the pre-processing phase, exact ML models are built for several partitions of data and their corresponding metadata is stored in a repository. These pre-built models may contribute to future approximate models. For model merging approach the model parameters 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 the 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 hyperplane.

3.2.3 User Interface

The user interface of ApproxML consists of three main sections including configuration panel, results section, and building partial models. Each section is described in detail as follows.

3.2.3.1 Configuration section

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

ML task: The user can choose if she wants to build a classifier or a clustering model. If she chooses clustering option, she can specify the number of clusters as well.

ML algorithm: The user can choose between Logistic Regression and Linear SVM for classification task and K-means and GMM for clustering task.

Dataset: The user will select a dataset in this section. An overview of the dataset will be shown to the user. For any selected dataset, appropriate query range or OLAP hierarchy 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 as shown in Figure 3.2(a).

Approximate/Exact: The user has the option to select between the approximate and exact models. If she chooses the exact model, 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 the approximate option 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, an approximation ratio should also be selected.

Figure 3.2(a) illustrates the configuration panel for building a Logistic Regression classifier on Flights dataset using the data from 10 April 2015 to 15 October 2015 through a model merging approach. Figure 3.3(a)shows the configuration for building a K-means clustering model on the FIFA2019 dataset using coreset-based approach with approximation ratio of 10% for the data of Europe.

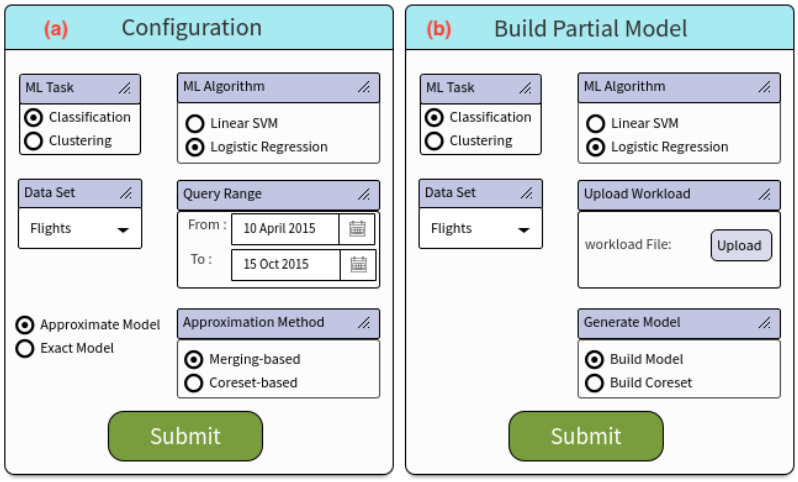


Figure 3.2. (a)Configuration panel for approximate Logistic Regression,(b) Build partial Logistic Regression models.

3.2.3.2 Results Section

In the results section of ApproxML, the statistics and quantitative measures of the generated ML model is reported to the user. For classifiers, training accuracy and testing accuracy are shown. Cost of building the model is also reported to the user. In clustering scenario, Adjusted Rand Index (ARI) and likelihood are shown for K-means and GMM respectively. If an approximate model is built, in addition to the total cost of building the approximate model, the partial costs including the merging cost, cost of building coresets, and costs of building the partial models for the new partitions are also reported. Finally, the user can see which pre-built models were retrieved from the pre-built model repository and reused for this particular approximate ML model. Figure 3.3(b) shows an example of results section for an approximate K-means model with 5 clusters using coreset-based approach.

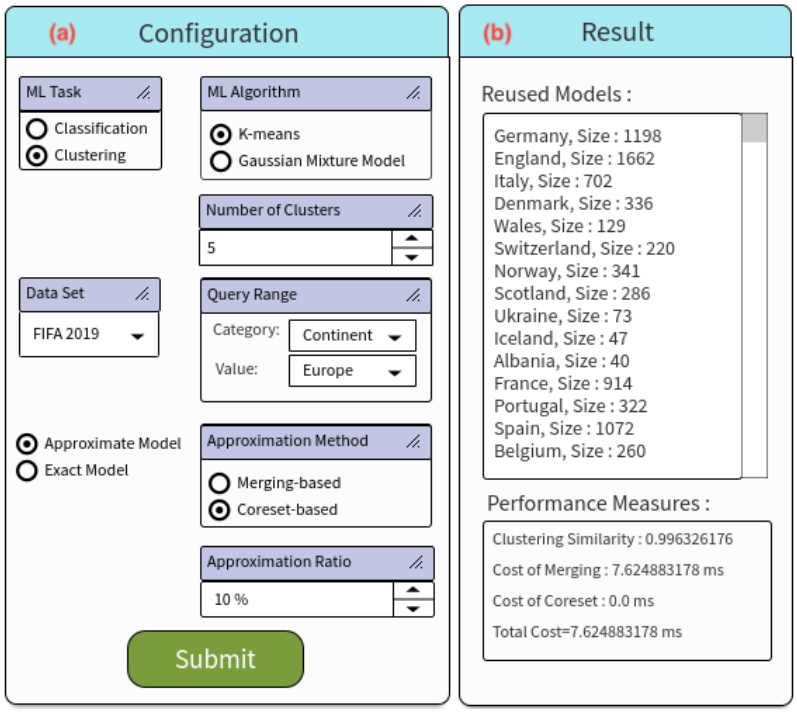


Figure 3.3. (a)Configuration panel for approximate K-means,(b)Results.

3.2.3.3 Build Partial Models

In this section, the user can upload a workload, select a dataset, and customize the parameters of an ML model. The data is then retrieved from the dataset, partitioned into optimum partitions, the exact model is built for each partition, and the corresponding metadata for the models are saved in the ML model repository. Figure 3.2(b) shows an example of this section.

3.2.4 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 dataframes. We used Flask for session management and database connection tools.

Datasets: For classification, we selected 5 datasets from the Hamlet repository [34] including Movies, Yelp, Walmart, Books and Flights. In addition, we used Flights dataset [1]. For evaluating clustering algorithms we used Santander customer transaction data [2] and FIFA 2019 dataset [3]. Additionally, we generated a synthetic dataset with 5M data points, 20 features and 10 clusters using publicly available generator [32].

3.3 Demonstration Plan

3.3.1 System Setup and Audience Interactions

We shall provide 3 laptops with ApproxML pre-installed on them. The datasets and repository of the pre-built models are stored on a server. We will also keep local copies of the datasets and the ML repository on the demo laptops in case of a broken internet. We will store a set of pre-built models in the ML model repository for each dataset. Visitors to the demo can freely select the dataset and ML algorithm of interest, specify the query of interest, and then observe the accuracy and efficiency of the output approximate models and compare them with the exact model built from the scratch. Additionally, they can experience the pre-processing phase by materializing ML models for a given workload.

3.3.2 Demonstration Scenarios

In this section, we describe several scenarios about how the audience can interact with ApproxML.

[1] https://www.kaggle.com/miquar/explore-flights-csv-airports-csv-airlines-csv/data
[2] www.https://www.kaggle.com/c/santander-customer-transaction-prediction/data
[3] https://www.kaggle.com/karangadiya/fifa19

A: Classification using Flights 2015 data In the configuration panel, the user can choose classification for ML task and Logistic Regression as ML algorithm. After selecting the Flights dataset and adjusting query range, the user can once build the approximate model with model merging and once with coreset-based approach. The pre-built models that are reused for the approximate model will be reported to the user along with different costs involved in making the approximate models. She can then choose to build the exact model from scratch for the same configuration and compare its efficiency and accuracy with those of the approximate models. The same process can be repeated for generating approximate linear SVM model.

B: Clustering using FIFA 2019 data In the configuration panel, the user can select clustering as the ML task, K-means as the ML algorithm, and set the number of clusters. After selecting the FIFA 2019 dataset, and setting “category" to “continent" and “value" to “Europe", she can build an approximate model once with model merging and once with coreset-based approach. To compare the efficiency and accuracy of the approximate models, she can choose to build an exact model with the same configuration and evaluate the Adjusted Rand Index for goodness of clustering, cost of building the model for the exact model and various costs involved in the approximate models. She can repeat the same steps for GMM as well. In order to compare the approximate and exact GMM models, the likelihood measure is shown for clustering similarity.

C: Materialize models for Flights 2015 data In the "Build partial model" panel, user can choose the ML model as clustering, the ML algorithm as K-means, select number of clusters, and choose Flights 2015 dataset. We will provide a text file containing a workload that the user can upload to the system. The user can submit the request once for model merging approach and once for coreset-based approach with approximation ratio set to 10%. The user will see the optimum partitions identified by ApproxML for materialization. We will also show the saved centroids and coresets and their corresponding weights in the pre-built model repository. The same process can be repeated for Logistic Regression, linear SVM, and GMM.

3.4 Summary

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 saves 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.

CHAPTER 4

Barracuda: Faster Algorithms for Generating Explanations for Multiple Predictions

Machine learning (ML) models have achieved widespread adoption in the last few years. Generating concise and accurate explanations often increases user trust and understanding of the model prediction. Usually, the implementations of popular explanation algorithms are highly optimized for a single prediction. In practice, explanations often have to be generated in a batch for *multiple* predictions at a time. To the best of our knowledge, there has been no work for efficiently generating explanations for more than one prediction. While one could use multiple machines to generate explanations in parallel, this approach is sub-optimal as it does not leverage higher-level optimizations that are available in a batch setting. We propose a principled and lightweight approach for identifying redundant computations and several effective heuristics for dramatically speeding up explanation generation. Our techniques are general and could be applied to a wide variety of explanation algorithms. We demonstrate this over a diverse set of algorithms including, LIME, Anchor, and SHAP. Our empirical experiments show that our methods impose very little overhead and require minimal modification to the explanation algorithms. They achieve more than 20x speedup over baseline approaches that generate explanations in a sequential manner.

4.1 Introduction

The widespread use of ML models has necessitated the development of algorithms for explaining their predictions. Explanations increase user trust and understanding of the model. It also has diverse applications, including model transparency [74, 75, 76], debugging [8], accountability [6], auditing [74, 77], fairness [5], explanation summarization [7, 8] and others [78, 79]. An increasing number of countries espouse a "right to explanation" [74]. Popular techniques for explaining predictions are widely used and have been incorporated into the ML offerings of Google [75], AzureML [76], and others [80]. However, current algorithms are optimized for explaining *individual* predictions. In applications such as responsible AI [5, 6] or explanations summarization [7, 8], explaining data cleaning [8, 9, 10] there is a need to generate explanations for multiple predictions in a *batch* setting. Generating explanations often cannot be done in real-time (in milliseconds). For example, generating a single explanation using LIME takes 17, 15, 6, 6 and 5 seconds respectively for the 5 datasets evaluated in the paper.

So, an organization might pre-compute all the explanations in a batch setting and retrieve them as needed.

To the best of our knowledge, there has been no prior work on speeding up explanations for multiple predictions. Sequentially processing one explanation at a time could take too much time. Using a cluster and parallelizing the explanation generation would give results faster but could waste precious computing resources. Given the rapidly increasing carbon footprint of ML algorithms [11], and the widespread deployment of explanation algorithms, there is a pressing need for smarter algorithms for this critical problem.

Our Proposed Approach. In this paper, we propose a principled and scalable approach for generating explanations for multiple predictions. The key insight is that there are a number of redundant computations that could be avoided by leveraging techniques such as materialization and reuse. Our techniques were inspired from Multi-Query Optimization (MQO) [12, 13]. Given a query workload, MQO seeks to identify common sub-expressions across queries, so that the reevaluation cost could be minimized. We describe a general set of heuristics for speeding up explanation algorithms and discuss how these ideas could be instantiated for popular explanation algorithms requiring minimal changes and very low overhead. Our proposed approach achieves significant speedup without compromising the explanation quality. In short, we adapt the techniques pioneered by the database community to solve a practical problem in data science.

Opportunities for Optimization. Consider two popular algorithms that build a local surrogate model – LIME [2] and Anchor [3]. Intuitively, both these approaches perturb the data, apply the blackbox classifier on the perturbations, and use the resulting predictions to generate explanations. Naive optimization techniques such as persisting all the perturbations are not viable as it requires a large amount of memory while providing minor improvements. A more effective approach is to do a lightweight preprocessing of the dataset and use the collected statistics to generate perturbations smartly. For example, if two tuples t_i and t_j have some overlap, one could generate perturbations that could be used for the explanations of both of them. We identify a number of such opportunities and propose effective heuristics for speeding up the explanations.

Summary of Contributions.

- We identify an important problem of generating explanations for multiple predictions over *tabular* data.
- We analyze popular explanation algorithms, identify multiple redundant computations and develop scalable algorithms inspired by database techniques.
- We conduct extensive experiments that show that BARRACUDA can achieve significant speedups over diverse datasets with very little overhead

4.2 Preliminaries

In this section, we formally define the problem of generating explanations for multiple predictions. Next, we introduce BARRACUDA, a framework that adapts popular explanation algorithms for this problem and provides a consistent and transparent API to the end-user.

4.2.1 Problem Statement

We are given a *batch* of tuples $B = \{t_1, t_2, \ldots, t_n\}$, a classifier $\mathcal{C}$ and an explainer $\mathcal{E}$. Let $y_i = \mathcal{C}(t_i)$ be the prediction for tuple t_i and $e_i = \mathcal{E}(t_i, \mathcal{C})$ be the corresponding explanation. Our goal is to generate explanations for the predictions of all the tuples in B. The explanations could be in the form of a rule "IF $A_i = u$ then class=Positive". It could be weights associated with each attributes such that attributes vital for the prediction getting a higher value. We shall describe the different type of explanations in Section 4.3.

EXPLANATIONS FOR MULTIPLE PREDICTIONS (EMP) PROBLEM: Given a batch of tuples B, classifier $\mathcal{C}$ and explainer $\mathcal{E}$, efficiently generate explanations for each of tuples t_i such that the cumulative cost of computing explanations is minimized.

A straightforward approach would assign more resources by running the explanation algorithms in parallel on very many machines. Instead, we propose a more promising and simpler approach inspired by multi-query optimization [12, 13] that achieves speed up by avoiding redundant computations by materializing them. As we shall show in our experiments, our approach outperforms the parallelization strategy for batches as small as 1000.

Offline and Online Variants. There are two concrete variants of EMP problem that are of interest to a practitioner. In the *offline* variant, we are provided with a batch of individual predictions that must be explained. In a number of data science applications, the set of tuples on which the predictions need to be made and explained is available beforehand. Emerging scenarios in responsible AI and explanation summarization work by generating explanation for each tuple in the test set and post-processing the generated explanations. This allows a number of optimization opportunities by performing a lightweight pre-processing to identify the redundant computations. These could then be pre-computed and materialized for later use. The other is the *online* scenario where the predictions arrive one at a time and we need to compute explanations for them *immediately*. We do not have the luxury of pre-hoc identifying the redundant computations and might also have additional constraints on resource consumption. We need to identify promising candidates for redundant computations in a principled manner. Our proposed approaches are generic enough to handle both the scenarios. Figure 4.1 illustrates the key components of our proposed approach.

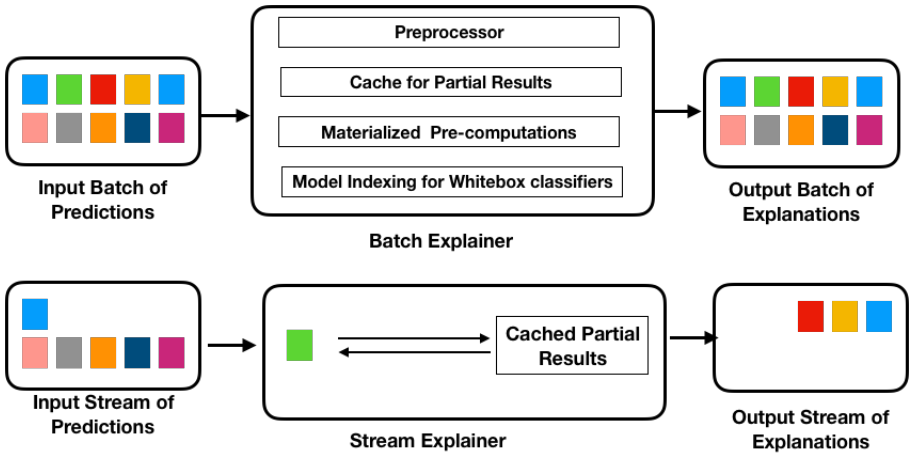


Figure 4.1. Illustration of our proposed approach.

4.2.2 BARRACUDA API

We have built BARRACUDA, a Python library that provides efficient explanations for a group of tuples. We support major algorithms for individual explanations, including LIME, Anchor, KernelSHAP and many others. It provides a unified interface for invoking explanations for individual and multiple predictions. BARRACUDA is also modular and can easily accommodate new optimizations and other popular explanation algorithms. Figure 4.2 shows how BARRACUDA could be invoked for LIME explainer. Note that the optimizations are transparent to the user.

```
from Barracuda import
    BatchLIMEExplainer
explainer = BatchLIMEExplainer
    ()
#Explain a single prediction
explanation = explainer.explain
    (x)
#Explain multiple predictions
explanations = explainer.
    explain(X_test)
```

```
from Barracuda import
    StreamingLIMEExplainer
explainer =
    StreamingLIMEExplainer()
for x in X_test:
    #Explain tuples one at a time
    explanation = explainer.
    explain(x)
#Clear the partial computations
explainer.clear()
```

Figure 4.2. Code snippet showcasing the Offline and Online variant of BARRACUDA..

4.3 Explaining Multiple Predictions for LIME, Anchor and SHAP

In this section, we provide the necessary background for three explanation algorithms and describe how BARRACUDA modified them for explaining multiple predictions. For the sake of exposition, we focus on the optimizations that provide the most bang for the buck, easy to implement, and are generalizable to other explanation algorithms.

Desiderata for Explanation Algorithm Selection. We focus on three algorithms – LIME [2], Anchor [3], and KernelSHAP [4]. These algorithms use diverse techniques and are exemplars in showing the generality of our approach. The algorithms are widely used in academia, industry, and incorporated into the explainable AI services of major cloud ML providers [75, 76, 81].

Algorithm	**Model Agnostic?**	**Technique**	**Explanation**
LIME	Y	Perturbation	Feature importance
Anchor	Y	Perturbation	Rules
KernelSHAP	Y	Shapley values	Feature contribution

Table 4.1. Categorization of major Explanation Algorithms discussed in the paper

We next provide necessary details of these three algorithms and describe how BARRACUDA modified them for EMP problem. BARRACUDA takes a uniform random sample of the batch and applies a traditional frequent itemset algorithm. Each frequent itemset f is of the form $\{A_i = u, A_j = v, \ldots\}$ where $A_i, A_j, \ldots$ are arbitrary features and $u, v, \ldots$, are the corresponding values of those features that are frequent in the batch. The sample size is heuristically chosen as *max(1000, 1% of batch)*.

4.3.1 LIME for EMP Problem

LIME Primer. Local interpretable model-agnostic explanations (LIME) [2] is a seminal work that trains an interpretable surrogate model to approximate the black-box classifier $\mathcal{C}$ and generate explanations. The explanation of LIME corresponds to a small set of attributes with *relative weights*. For a two class classification problem, attributes that contribute to positive class will have positive weights while those contribute to the other class will have negative weights. We can obtain an ordering of the importance of the attributes to the prediction by sorting based on the weights. LIME consists of four key operations: (1) perturbing t_i to obtain samples S; (2) running blackbox classifier $\mathcal{C}$ on S; (3) training an interpretable model C_{int} on S that acts as a surrogate for $\mathcal{C}$; (4) generating explanations for t_i by analyzing C_{int} Profiling on LIME shows that the steps (1) and (2) account for more than 95% of total execution time.

Adapting LIME for EMP**.** One can simultaneously minimize the cost of both steps (1) and (2). First, let us dig deeper into how the perturbations are done for tabular data. For categorical attributes, LIME perturbs them by sampling values according to the data distribution from the training dataset. By default, it discretizes numerical data and treats it as categorical. Alternatively, it can perturb a numerical feature by sampling from a unit Normal distribution and performing inverse operation of mean-centering and scaling [2]. BARRACUDA relies on two key insights. First, the perturbations are performed for each feature independently and based on a distribution that is *fixed* for each tuple. For example, consider two arbitrary tuples t_i and t_j. During perturbation, the probability that a feature A_i will be set the value u, for both t_i and t_j, is exactly same as the proportion of u in the training dataset. Second, given two arbitrary perturbations made by LIME, we must prefer those that could be reused for multiple tuples.

We compute the frequent itemsets F and for each $f \in F$, we compute τ perturbations. For example, if $f = (A_i = u, A_j = v)$, then we create τ perturbations such that all of them have $(A_i = u, A_j = v)$ while the values for other features are obtained using LIME's perturbation techniques. The parameter τ is set automatically by BARRACUDA based on the resource constraints. The classifier is invoked on each of the perturbations and its output is stored. Given a new tuple t_i for explanation, we check if t_i contains any of the frequent itemsets. If so, we pool the τ perturbations corresponding to those itemsets. For the remaining perturbations, we follow the same procedure as LIME. The reused perturbations and their labels result in savings in terms of both classifier invocations and needless creation of perturbations. The pseudocode can be found in Algorithm 6.

Algorithm 6 LIME for EMP Problem

Input: A batch B, blackbox classifier $\mathcal{C}$, number of samples N
Compute frequent itemsets F over B
Generate τ perturbations $\forall f \in F$
Invoke $\mathcal{C}$ on all perturbations and store the output in P
for each tuple $t_i \in B$ **do**
    S = retrieve reusable samples and labels from P
    S' = Obtain $N - |S|$ perturbations, invoke $\mathcal{C}$ on them
    $S = S \cup S'$
    Compute proximity between t_i and each $s \in S$
    Train interpretable model M using S
    Generate explanations for t_i using M
return explanations for B

4.3.2 Anchor for EMP Problem

Anchor Primer. Anchor [3] is another popular perturbation based explantion algorithm that outputs easy to understand rules of the form IF $A_i = u$ AND $A_j = v$ THEN class=1. For each rule, Anchor also provides two metrics – precision and coverage. Precision is the proportion of tuples in which the rule holds. Coverage is the fraction of tuples on which the predicates of the rule holds. Given a tuple t_i and a desired threshold on precision, Anchor provides a rule with high coverage whose precision exceeds the bound. Anchor consists of three key steps: (1) Generating candidate rules; (2) Estimating their precision; (3) Identify K best candidates with high precision and coverage and repeating from step 1 till the precision constraints are satisfied.

Adapting Anchor for EMP**.** Similar to LIME, Anchor provides multiple opportunities for optimization. The key bottleneck in Anchor is the estimation of precision of a candidate rule. For example, let IF $A_i = u$ THEN class=1 be such a rule. A naive approach would be to generate various samples where $A_i = u$ and the other attributes are obtained by using training data distribution. For each of the sample data points, we invoke the classifier and report the proportion in which class=1 occurs. Since classifier invocation is very expensive, Anchor uses a sophisticated multi-armed-bandits to minimize the number of such calls.

We begin by identifying the frequent itemsets F. For each $f \in F$, we estimate the precision of rules. If the precision of a rule containing f as its predicate is higher than the user provided threshold, then the rule could be used as an Anchor for all tuples containing f. Since f was a frequent itemset, it is likely to have a high coverage. The second optimization is to bootstrap the computation of precision for candidate rules containing a superset of frequent itemsets. Let $A_i = u$ be a frequent itemset. Then the process of estimating its precision required the generation of various sample data points and the invocation of classifier on it. Consider a rule IF $A_i = u$ AND $A_j = v$ THEN class=1. Instead of estimating the precision from scratch, we can scan the samples generated for $A_i = u$, find the subset that also contains $A_j = v$. By computing the proportion of those that have class=1, we can obtain a preliminary estimate that could be refined as needed. Finally, the coverage of the rules are fixed for each candidate rule. Hence, we materialize the coverage of all the candidate rules so that they are not recomputed again and again. Given multiple candidate rules satisfying the requirements, we pick the rule with least predicates. Algorithm 7 describes coarse-grained pseudocode.

Algorithm 7 Anchor for EMP Problem

Input: A batch B, blackbox classifier $\mathcal{C}$
Compute frequent itemsets F over B
Generate candidate rules R using F and estimate their precision
for each tuple $t_i \in B$ **do**
  $R' = \{\}$
  loop
    Generate candidate rules by appending new predicates from t_i to R'
    if precision of any candidate rule $r \in R'$ satisfies precision constraints **then**
      Use r as Anchor
    else
      Bootstrap precision for the candidate rules R' from materialized samples
      Add current precision estimates of rules to R
      Find best candidate rules $R'' \subseteq R'$
      Update precision of rules R''
return explanations for B

4.3.3 KernelSHAP for EMP Problem

Shapley Values. SHAP [4] is a family of explanation algorithms that use Shapley values which is a principled approach to allocate credit for a feature. Given a tuple t_i, one could use Shapley values to compute the contribution/importance of each feature value $A_j = v$ in t_i. Intuitively, Shapley values computes the marginal contribution for each feature over all possible subsets of features. Computing the exact Shapley value requires exponential time. Hence, the values are computed approximately through sampling [82]. Feature importances computed via Shapley values have a number of appealing theoretical properties.

KernelSHAP Primer. KernelSHAP can estimate the feature importances of any blackbox functions. It consists of four major steps: (1) Generate multiple random feature subsets, estimate its weight through SHAP kernel and convert each feature subset to a random perturbations through sampling from the training data distribution; (2) Apply blackbox classifier on the random data perturbations; (3) Build a weighted linear and interpretable model; (4) Compute the Shapley values for each feature

Adapting KernelSHAP for EMP**.** First, we obtain the frequent itemsets F and generate τ random perturbations. Suppose the feature subset is $A_i = u, A_j = v$. Then for each of the τ perturbations, we set their A_i to u and A_j to v. The other attribute values are filled by sampling according to their data distribution. We invoke the classifier on each of these random data perturbations and store the predictions. KernelSHAP computes M random data perturbations before feeding them to an interpretable model. Given a tuple t_i for explanation, we identify if it contains any feature subset that is frequent. If so, then we can reuse all random perturbations and their labels. For the remaining budget, we randomly chose a feature subset and check if it is a superset of any other frequent itemset. If so, we can again reuse those data perturbations. For example, if $A_i = u$ is frequent but $A_i = u, A_j = v$ is not, we can still scan the random data perturbations of $A_i = u$ for those that also have $A_j = v$. Another key optimization is to choose random feature subsets in proportion to the weight provided by SHAP kernel defined as [4].

$$\pi(m, s) = \frac{m - 1}{\binom{m}{s} \times s \times (m - s)} \tag{4.1}$$

Here m is the maximum size of the feature subset while s is the size of current feature subset. We can see that when s is small or $s \sim m$, the value $\pi(m, s)$ is large. In other words, generating feature subsets that are either very small (1 or 2) or very large (as large as m or $m-1$) is preferable to feature subsets of intermediate size such as $m/2$. This optimization has been previously observed in [83]. Algorithm 8 puts together all these ideas.

Algorithm 8 KernelSHAP for EMP Problem

Input: A batch B, blackbox classifier $\mathcal{C}$, number of samples N
Compute frequent itemsets F over B
Generate τ random data perturbations $\forall f \in F$
Invoke $\mathcal{C}$ on all perturbations and store the output in P
for each tuple $t_i \in B$ **do**
    $S = S' = \{\}$
    if t_i contains any frequent itemset **then**
        $S = S \cup$ retrieve the perturbations and their labels from P
    while $|S| + |S'| < N$ **do**
        Pick feature subset size according to Equation 4.1
        Pick a random subset s
        if s is a superset of any frequent itemset **then**
            $S' = S' \cup$ any relevant perturbations from P
    Invoke $\mathcal{C}$ on perturbations from S'
    $S = S \cup S'$
    Compute weight of each $s \in S$ using SHAP kernel
    Train an interpretable model M
    Generate explanations for t_i using M
return explanations for B

4.3.4 Optimization Principles used by BARRACUDA

In this subsection, we abstract the generic ideas behind BARRACUDA that could be used to speedup other perturbation based explanation algorithms over tabular data.

Materialization and Reuse of Perturbations. The key insight is to identify the expensive computations that are repeated and materialize the intermediate results. For example, invoking blackbox classifiers is often the biggest bottleneck accounting for 88% of the execution time for LIME and 92% for Anchor for Census-Income dataset. However, it is often unlikely that random perturbations of two tuples would have produced a common sample whose classifier invocation could be minimized. In such a case, we need to *engineer* opportunities for reuse by leveraging additional techniques. For example, our approach for LIME exploited the fact that for tabular data, perturbations are based on training data distribution. By preferentially selecting perturbations that could be reused, we achieved tremendous speedups.

Caching Other Invariant Results. The precision and coverage of a rule are invariant and do not change for different tuples. Even if they are inexpensive, it is sub-optimal to repeatedly calculate them. One can design a cache to store these invariant results for reuse. In some cases such as coverage, it is often clear that it is an invariant. In other cases, such as precision in Anchor, the parameter is often derived/estimated using a complex approach like multi-armed bandit which makes the invariance non-obvious. Similarly, other notions of invariance might exist and one could achieve good speedup by pre-computing them.

4.3.5 Streaming Variant of Barracuda

Barracuda could also be invoked in a streaming setting where individual predictions arrive one at a time and explanations have to be generated for them. Barracuda uses a simple adaptation to retrofit the ideas developed for the batch setting for application in the streaming setting. Barracuda is given a memory budget that constrains the amount of auxiliary information that could be saved such as frequent itemsets and perturbations.

Let us consider the LIME explanation algorithm. At the beginning, the tuples arrive one at a time and we do not have sufficient information to identify which of them are frequent itemsets. For the first tuple t_1, we generate and store all the perturbations along with their labels to a repository P. Clearly, there is no saving yet. For the second tuple to be explained t_2, we check if we can reuse any of the perturbations of t_1. If so, we reuse the perturbations as appropriate. If not, we generate perturbations of t_2, invoke the classifier and store these perturbations with labels to P. We also store the set of tuples that are being explained $\{t_1, t_2, \ldots, \}$.

This process repeats until one of the conditions get satisfied: (a) P exceeds the memory budget or (b) number of tuples exceed a certain threshold (automatically chosen by BARRACUDA such as 100). When the former happens, we kick out perturbations based on the LRU (least recently used) policy. At the limit, this approach implicitly ensures that perturbations containing frequent itemsets will not be kicked out. When the latter (b) happens, we run a frequent itemset mining algorithm and also store the negative border of the frequent itemsets. An itemset $\{A_i = u, A_j = v\}$ is in the negative border if it is not frequent but all of its *immediate subsets* i.e. $\{A_i = u\}$ and $\{A_j = v\}$ are frequent.

Let F be the set of frequent itemsets and their negative border. For each item $f \in F$, we compute the frequency of f in the set of tuples. Once this is done, we purge the tuples as they no longer are needed. Due to the way in which F is constructed, the perturbation repository P already consists of perturbations containing itemsets from F. If not, we purge that perturbation and use the obtained savings to generate perturbations of $f \in F$.

When new set of tuples arrive, we update the frequency of itemsets in F and reuse perturbations as appropriate from P. We also persist the next set of tuples so that the frequent itemsets can be recomputed. Any frequent itemset $f \in F$ that becomes infrequent is kicked out along its perturbations from P. We can see that this intuitive approach computes useful perturbations while keeping fresh by periodically recomputing the frequent itemsets. This property makes the algorithm dynamic and responsive to changes in the input stream.

4.3.6 Discussion

Handling Numeric Data. BARRACUDA relies on frequent itemsets which are well defined for categorical attributes. Both LIME and Anchor *discretize* numeric data (such as by quartile discretization) before generating perturbations. BARRACUDA computes the frequent itemset over the discretized data. Of course, if discretization is not done the opportunity for identifying redundant computation decreases.

BARRACUDA **Optimizations and Anchor Explanation Quality.**

The two optimizations used by BARRACUDA to speed up Anchor are exact. First, BARRACUDA caches invariant results such as precision and coverage of a candidate rule and avoids recomputing it. Second, BARRACUDA speeds up precision computation of candidate rules by reusing perturbations. For e.g. the precision of a rule R_1 such as "IF $A_i = u$ and $A_j = v$ THEN class=1" is obtained by generating various perturbations where $A_i = u$ and $A_j = v$, invoking classifier and finding the fraction for which class=1. We can see that one could reuse the perturbation of R_1 for candidate rules R_2 ("IF $A_i = u$ THEN class=1") and R_3 ("IF $A_j = v$ THEN class=1"). Furthermore, the reverse is also possible. We can reuse any of the perturbations of R_2 and R_3 that matches the predicate of R_1 to compute the precision of R_1. Once again, this optimization is exact.

BARRACUDA **Optimizations and LIME/KernelSHAP Explanation Quality.**

Both LIME and KernelSHAP generate a perturbation of tuple t_i as follows. First, they fix the value of a random subset of attributes (say that of A_1, A_2). For each of the other attribute ($A_j \in \{A_3, A_4, \ldots\}$), they choose a value from *Domain(A_j)* according to frequency distribution. This perturbation is passed to a classifier for getting the label. Then the whole process is repeated to get another perturbation. BARRACUDA pre-generates perturbations for frequent itemsets. If tuple t_i has a frequent itemset f, one could reuse the pre-computed perturbations of f as and when LIME or KernelSHAP picks the set of attributes f. Once again, this on-demand reuse of cached perturbations does not introduce any approximation.

Dataset	**#Tuples**	**#CatA**	**#NumA**	**#MaxDC**	**SeqB**	**BC-B**	**BC-S**
Census (KDD)	299285	27	15	18	261.2	30.8	93.3
Recidivism	9549	14	5	20	93.7	10.4	31.2
lendingclub	42536	26	24	837	292.5	29.3	91.4
KDD Cup 1999	4000000	13	27	490	101.7	8.5	26.8
Covertype	581012	44	10	7	100.8	9.2	28

Table 4.2. Performance of BARRACUDA over diverse datasets. #CatA, #NumA are the number of categorical and numerical attributes. #MaxDC is largest domain cardinality of the categorical attribute. SeqB gives the time taken *in minutes* by the sequential baseline for explaining *1000 tuples*. BC-B and BC-S show the time taken (in minutes) by the batch and streaming variant of BARRACUDA for the same set of 1000 tuples in the same order.

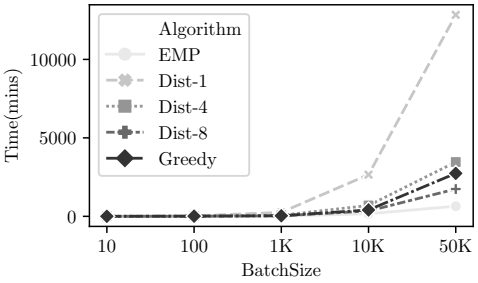


Figure 4.3. Comparison with Baseline Algorithms.

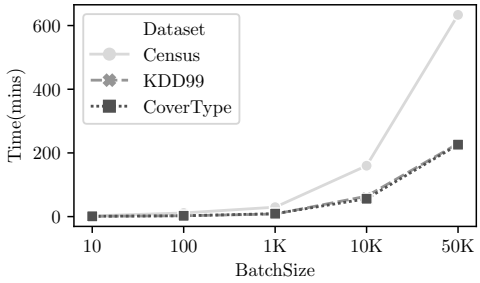


Figure 4.4. Speedup for LIME Explainer.

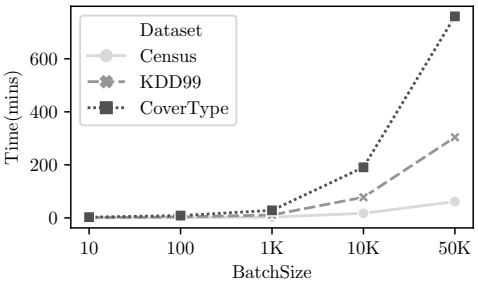


Figure 4.5. Speedup for Anchor Explainer.

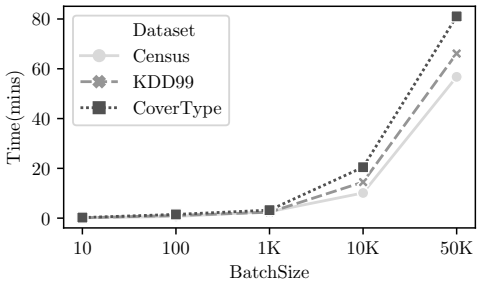


Figure 4.6. Speedup for SHAP Explainer.

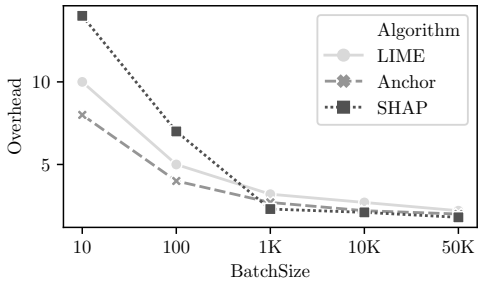


Figure 4.7. Overhead of BARRACUDA.

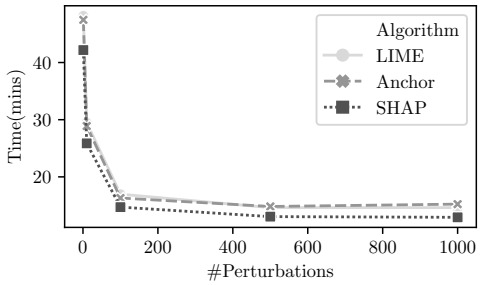


Figure 4.8. Impact of #Perturbations, τ.

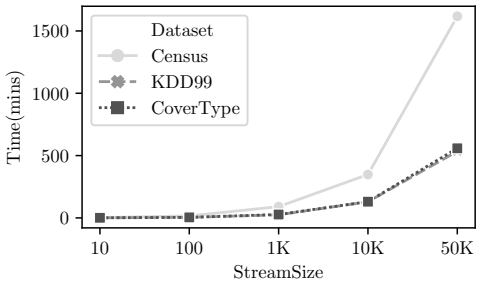


Figure 4.9. Speedup for LIME Explainer (Streaming).

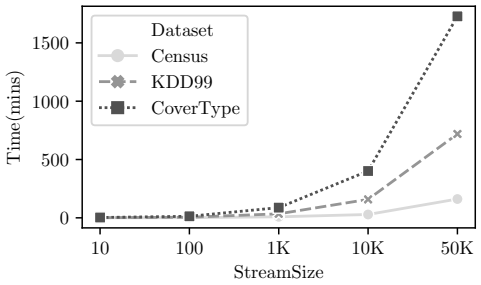


Figure 4.10. Speedup for Anchor Explainer (Streaming).

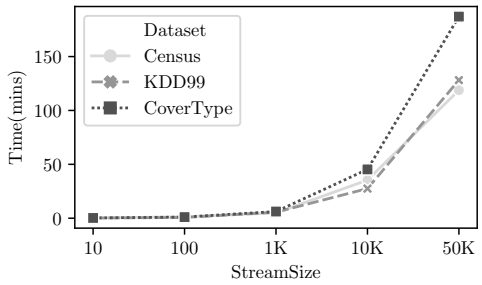


Figure 4.11. Speedup for SHAP Explainer (Streaming).

4.4 Experiments

We conducted extensive empirical experiments that demonstrate that BARRACUDA achieves significant speedup over baseline approaches by minimizing redundant computations for three widely used explanation algorithms.

4.4.1 Experimental Setup

Hardware and Platform. All our experiments were performed on a quad-core 2.2 GHz machine with 16 GB of RAM. BARRACUDA and the various explanation algorithms are all implemented in Python.

Datasets. We conduct experiments over five diverse benchmark datasets. The Census-Income dataset [84] consists of 42 demographic and employment related variables and predicts whether the person makes $\$50K$ annually. The recidivism dataset is used to predict recidivism for individuals released from prison [85]. The lending club dataset predicts whether a loan will result in default or late payment. These three datasets have been used in prior explanation work such as [3]. KDDCup 1999 dataset seeks to build a network intrusion detector by predicting whether a connection is normal or abnormal [86]. The CoverType dataset from UCI repository tries to predict forest cover type from cartographic variables [86]. Each of these datasets are diverse in number of total, categorical and numerical attributes. The categorical attributes also have a wide spectrum in terms of domain cardinality. The details about these datasets are provided in Table 4.2. We partition the dataset into 1/3 and 2/3. We used the former for training the ML model and used the latter for prediction and explanation.

Explanation Algorithms. We focus on three representative algorithms – LIME, Anchor and SHAP as they use different algorithmic techniques and produce very different explanations. Our implementation of BARRACUDA was based on Microsoft's InterpretML library [79]. Due to space limits, we conduct our experiments on random forest classifier. Since BARRACUDA achieves speedup by minimizing the *number of classifier invocations*, this does not materially affect the conclusions. We used the default hyperparameters for the explanation algorithms (such as $\epsilon = 0.1, \delta = 0.05$ for Anchor). The default value of τ was set to 100.

Baseline Algorithms. We consider two baseline approaches. The first approach generates explanations sequentially one prediction at a time. Specifically, we consider a distributed variant where the explanation generation is spread across 1, 4 or 8 machines. Each machine process the same amount of data. We dub this approach as DIST-1, DIST-4 and DIST-8 respectively. Given a batch of 10000 predictions to explain, DIST-8 will split them into 1250 predictions and spread them to 8 machines. The second baseline is GREEDY. Given a memory budget, this approach stores all the perturbations until the budget is exhausted. When generating explanation, it reuses existing perturbations and their labels if possible. Otherwise, it generates new explanations and uses the LRU (least recently used) policy to replace unused perturbation. By default, we assumed that the space budget is 10x the size of the batch.

Batch Sizes. We evaluate both the offline and online variants of BARRACUDA using the same set of tuples processed. We vary the batch size from 10 to 50K tuples. The order in which the tuples are processed is the same for all the algorithms. In the offline scenario, this denotes the set of tuples for which explanations has to be generated. In the online scenario, BARRACUDA receives an explanation request one at a time. We randomly generated 10 different permutations and report the average results.

Performance Measures. We measure our proposed algorithms based on two dimensions – scalability and explanation quality. The former measures how much faster are the algorithms of BARRACUDA over the baselines described above. Specifically, we use two key metrics – runtime and overhead. We report the wall-clock time. None of the existing implementations of LIME, SHAP and Anchor use multiprocessing. While BARRACUDA does use multiprocessing, we disable it to show that our superior performance comes from algorithmic improvements that minimize the number of classifier invocations. For the distributed version of the baseline (say DIST-8), we report the average time taken by the 8 machines as the runtime. Note that BARRACUDA runs only on a single core of a single machine. The overhead measures the percentage of time taken by BARRACUDA for housekeeping purposes such as computing frequent itemsets and retrieving relevant perturbations. The second dimension is that of explanation quality. Even though the optimizations of BARRACUDA are exact, we nevertheless empirically show that the explanations are equivalent to the sequential approach. We empirically measure the fidelity of explanations generated by the sequential approach along two metrics: feature importance values and rank. For LIME and SHAP, we can represent the contribution of each feature as a real-valued number. We measure the Euclidean distance between the explanation generated by original LIME/SHAP and the BARRACUDA variants. Using the importance values, we can compute a ranked list of features. We use Kendall-τ to measure the rank correlation between the ranking produced by original LIME/SHAP and BARRACUDA variant. Given a batch of tuples, we compute the Kendall-τ for each of the tuple in the batch and compute its average.

4.4.2 Experimental Results

Comparison against Baseline Approaches. We begin by evaluating BARRACUDA against the distributed and greedy baselines. We use LIME explainer to explain the predictions of random forest classifier trained over Census-Income dataset. Figure 4.3 shows the performance of BARRACUDA as the batch size is varied from 10 to 50K. Overall, BARRACUDA provides the best results and achieves a speedup of 10.8 for a batch size of 1000 which increases to 20.9 for a batch size of 50K. Even though BARRACUDA ran on a single core of a single machine, it outperforms DIST-8 for batch sizes of 1000. The disparity in performance increases as the batch size is increased with BARRACUDA being almost 3x faster than DIST-8 for a batch size of 50K. In the offline scenario, one might have to generate explanations for a large batch – such as for the entire test set consisting of thousands of tuples. This demonstrates the efficacy of algorithmic ideas of BARRACUDA.

The speedup increases with larger batch sizes as one could make informed decisions about the redundant computations through frequent itemsets. Given that the biggest bottleneck is invocation of the blackbox classifier, this shows that BARRACUDA reduces the number of invocations by a factor of 10! The other baseline GREEDY achieves substantial speedup for small batches that decreases for larger batches. This decrease is due to the algorithm's sub-optimal decisions regarding which perturbations to persist and which to remove. Blindly persisting all perturbations is not an effective strategy. In contrast, BARRACUDA takes a holistic approach and produces consistent and significant speedups.

Further evidence of this can be found in Table 4.2. Here, we compare the performance of the sequential approach (DIST-1) against the offline and online variant of BARRACUDA for explaining 1000 tuples using LIME explainer for random forest classifier. Even for a small batch size of 1000, BARRACUDA achieves a speedup of 7x or higher. We can see that the mixture of categorical and numerical attributes do not materially affect the performance of BARRACUDA. In fact, the performance improves for datasets with large number of numerical attributes such as KDDCup. This is primarily due to the discretization applied by the explainers that converts numerical attributes to categorical attributes. Often, these derived categorical attributes have a smaller domain cardinality than the original one and thereby provides an additional speedup. We can also see that the number of attributes does not materially affect the performance.

Offline Setting. Figures 4.4- 4.6 show the speedup achieved by BARRACUDA for three explainers – LIME, Anchor and SHAP – for three classifiers for a variety of batch settings. One can make the following observations: BARRACUDA achieves as much as 20x speedup over the baseline sequential approach. There is only minor difference between the speedups for different classifiers. Since invoking blackbox classifier accounts for more than 90% of the time taken, we can infer that speedup is primarily achieved by minimizing the number of calls to the classifier by intelligently materializing and reusing the perturbations. Among the algorithms, LIME achieves the least speedup of $\sim 11x$ for a batch size of 1000 when explaining LR classifier. The corresponding speedups for Anchor and SHAP are 9x and 8x respectively. Due to the simplicity of LIME, the major speedup is achieved by materializing the perturbations.

Online Setting. Figures 4.9- 4.11 shows the performance of BARRACUDA in an online scenario. Overall, the trends largely mirror those from the batch scenario. One can see that the speedups achieved in the batch setting dwarves that from the streaming setting. This is not surprising due to the ability of BARRACUDA to pre-process the data and carefully chose the perturbations to materialize. While the streaming based approach has a slower start obtaining only 25% of the speedup of that of batch setting, it improves to more than 60% for larger batches. Of course, one could achieve higher speedups through smarter frequent itemset computation.

BARRACUDA **Overhead.** We evaluate the overhead of BARRACUDA in the batch setting – for LIME explainer for random forest over Census-Income dataset. BARRACUDA mines the frequent itemsets, computes the perturbations and invokes the classifiers on the perturbations. For each tuple in the batch, it retrieves the relevant perturbations. The computation of frequent itemsets is done on a uniform random sample of size 1% of the batch size or 1000 whichever is larger). The small size ensures that this step is not very expensive. The computation of perturbations and classifier invocation gets amortized overall. Retrieving relevant perturbations is also a lightweight operation. Figure 4.7 shows that the percentage overhead imposed by BARRACUDA is as little as 3% and 2% for batch size of 10K and 50K respectively. Given the significant speedup achieved by BARRACUDA, this is a small price to pay.

We next study the impact of varying the resource budget for BARRACUDA for random forest classifier for different explanation algorithms. We vary the number of perturbations stored for each frequent itemset between 1 to 1000. Figure 4.8 shows the result. Even storing 10 perturbations provides a 5x speedup for LIME. Interestingly, storing beyond 100 perturbations per frequent itemset does not provide any additional benefit. This is due to the fact that each tuple usually contains a handful of frequent itemsets. Hence, the number of perturbations generated by the traditional LIME explainer that contains them is also limited.

Explanation Quality. We also conducted additional experiments on how the optimizations of BARRACUDA affects the explanation quality. For LIME and SHAP, the explanation could be construed as a real-valued vector over the features. For LIME, these values could be positive (that feature contributes towards the predicted label) or negative (that feature contributes against the predicted label). For SHAP, the values are positive with features having higher values contributing more to the prediction. Using these importance values, one could also rank the features.

Given two sets of explanations – one from the unmodified LIME/SHAP and another from BARRACUDA's – we measure the explanation quality along two dimensions: (a) the amount of deviation between the feature importances (b) the deviation in ranking of features. BARRACUDA *always* achieves the *same ranking* of features. We found that the maximum deviation in explanation values to be as small as 0.1. For KernelSHAP, this is not surprising as our algorithm from Section 4.3.3 avoids needless recomputations through materialization. Our optimizations do not have any major impact due to LIME giving different weights to perturbations based on the distance from the tuple being explained. Finally, our modifications to Anchor also does not affect the explanation quality in anyway. It lets the algorithm work as is while minimizing needless recomputations by materializing perturbations and caching the precision and coverage of candidate rules which are invariant.

4.5 Related Work

Explanations of ML Models. ML explanation has applications in model debugging, detecting bias, ensuring operational safety, among others [78, 79]. In the last few years, there has been a groundswell of work for explaining and/or interpreting ML models sometimes with different motivations [87, 77]. Explaining complex models is often very challenging [88, 89, 90]. Hence, a number of recent research focus instead on generating explanations for individual predictions. Some local explanation algorithms explain the model introducing interpretable surrogate models in the local neighborhood of an individual prediction such as LIME [2] or Anchor [3]. Some popular explanation techniques provide explanations in the form of computing the contribution of individual features (such as SHAP [4]). These algorithms are incorporated into ML platforms of the industry sector including Google [75] and AzureML [76] among others. Robustness of the existing explanation methods are studied in [91]. Recently, there has been a number of work on generating explanations for database and data curation settings [10, 9, 8, 92].

Speeding up ML through Database Techniques. There has been extensive work focusing on making ML algorithms more efficient through well established database techniques. Some have used the concept of Materialization and reuse to speedup the ML model construction [1, 93]. Similarly, other database techniques like pipelining and operator pushdown are used to speedup ML tasks [94]. Query optimization techniques have been used to speedup random forest inference [95] and video queries [96, 97]. A recent work [98] used Multi query optimization [12] techniques to speedup the *individual* explanations of a CNN based model. There has been extensive work on implementing efficient and extensible algorithms for MQO [13, 99].

4.6 Conclusion

In this paper, we introduced a practical data science problem – efficiently generating explanations for a batch of predictions. Given the increasing popularity of explanation algorithms and their adoption in academia and industry, there is a pressing need to develop scalable algorithms. Our proposed BARRACUDA introduces a number of non-trivial adaptations for the ML context in order to identify redundant computations. It achieves significant speedups over baseline approaches. There are a number of interesting next steps such as extending this techniques for non-tabular data and other major class of explanation algorithms. While the proposed optimizations are exact, it is possible that one could achieve substantial speedup by allowing certain approximation in the explanations generated.

REFERENCES

[1] S. Hasani, S. Thirumuruganathan, A. Asudeh, N. Koudas, and G. Das, "Efficient construction of approximate ad-hoc ml models through materialization and reuse," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1468–1481, 2018.

[2] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you? explaining the predictions of any classifier," in *KDD*, 2016, pp. 1135–1144.

[3] ——, "Anchors: High-precision model-agnostic explanations," in *AAAI*, 2018.

[4] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *NeurIPS*, 2017, pp. 4765–4774.

[5] J. Dodge, Q. V. Liao, Y. Zhang, R. K. Bellamy, and C. Dugan, "Explaining models: an empirical study of how explanations impact fairness judgment," in *Proceedings of the 24th International Conference on Intelligent User Interfaces*, 2019, pp. 275–285.

[6] A. Smith-Renner, R. Fan, M. Birchfield, T. Wu, J. Boyd-Graber, D. Weld, and L. Findlater, "No explainability without accountability: An empirical study of explanations and feedback in interactive ml."

[7] K. Natesan Ramamurthy, B. Vinzamuri, Y. Zhang, and A. Dhurandhar, "Model agnostic multilevel explanations," *arXiv*, pp. arXiv–2003, 2020.

[8] A. Ebaid, S. Thirumuruganathan, W. G. Aref, A. Elmagarmid, and M. Ouzzani, "Explainer: entity resolution explanations," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 2000–2003.

[9] V. Di Cicco, D. Firmani, N. Koudas, P. Merialdo, and D. Srivastava, "Interpreting deep learning models for entity resolution: an experience report using lime," in *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2019, pp. 1–4.

[10] S. Thirumuruganathan, M. Ouzzani, and N. Tang, "Explaining entity resolution predictions: Where are we and what needs to be done?" in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2019, pp. 1–6.

[11] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai," *arXiv preprint arXiv:1907.10597*, 2019.

[12] T. K. Sellis, "Multiple-query optimization," *TODS*, vol. 13, no. 1, pp. 23–52, 1988.

[13] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *SIGMOD*, 2000, pp. 249–260.

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

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

[16] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *SODA*. SIAM, 2007, pp. 1027–1035.

[17] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams: Theory and practice," *TKDE*, vol. 15, no. 3, pp. 515–528, 2003.

[18] N. Ailon, R. Jaiswal, and C. Monteleoni, "Streaming k-means approximation," in *NIPS*, 2009, pp. 10–18. [Online]. Available: http://www.cs.columbia.edu/~rjaiswal/ajmNIPS09.pdf

[19] ——, "Streaming k-means approximation," http://www.cs.columbia.edu/~rjaiswal/ajmNIPS09.pdf, accessed: Jul 20, 2018.

[20] 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*, vol. 15, no. 3, pp. 712–734, 2006.

[21] C. Hennig, "Methods for merging gaussian mixture components," *Advances in data analysis and classification*, vol. 4, no. 1, pp. 3–34, 2010.

[22] A. R. Runnalls, "Kullback-leibler approach to gaussian mixture reduction," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 43, no. 3, 2007.

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

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

[25] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *NAACL*. ACL, 2010, pp. 456–464.

[26] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *NIPS*, 2010, pp. 2595–2603.

[27] Y. Zhang, M. J. Wainwright, and J. C. Duchi, "Communication-efficient algorithms for statistical optimization," in *NIPS*, 2012, pp. 1502–1510.

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

[29] D. Feldman, M. Faulkner, and A. Krause, "Scalable training of mixture models via coresets," in *NIPS*, 2011, pp. 2142–2150.

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

[31] J. Huggins, T. Campbell, and T. Broderick, "Coresets for scalable bayesian logistic regression," in *NIPS*, 2016, pp. 4080–4088.

[32] 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*, vol. 12, no. Oct, pp. 2825–2830, 2011.

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

[34] 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*. ACM, 2016, pp. 19–34.

[35] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

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

[37] 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*, vol. 17, no. 1, pp. 1235–1241, 2016.

[38] 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*, vol. 5, no. 12, pp. 1700–1711, 2012.

[39] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *SIGMOD*. ACM, 2015, pp. 1969–1984.

[40] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang, "Mlog: Towards declarative in-database machine learning," *PVLDB*, vol. 10, no. 12, pp. 1933–1936, 2017.

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

[42] C. Zhang, A. Kumar, and C. Ré, "Materialization optimizations for feature selection workloads," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 1, p. 2, 2016.

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

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

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

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

[47] A. Kumar, R. McCann, J. Naughton, and J. M. Patel, "Model selection management systems: The next frontier of advanced analytics," *ACM SIGMOD Record*, vol. 44, no. 4, pp. 17–22, 2016.

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

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

[50] H. Miao, A. Li, L. S. Davis, and A. Deshpande, "Modelhub: Deep learning lifecycle management," in *ICDE*. IEEE, 2017, pp. 1393–1394.

[51] 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*, 2018, pp. 1285–1300.

[52] 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*. ACM, 2016, p. 14.

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

[54] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy, "Join synopses for approximate query answering," in *ACM SIGMOD Record*, vol. 28. ACM, 1999, pp. 275–286.

[55] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic sample selection for approximate query processing," in *SIGMOD*. ACM, 2003, pp. 539–550.

[56] 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*, vol. 1, no. 1, pp. 29–53, 1997.

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

[58] D. Barbará and X. Wu, "Loglinear-based quasi cubes," *Journal of Intelligent Information Systems*, vol. 16, no. 3, pp. 255–276, 2001.

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

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

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

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

[63] S. Har-Peled and S. Mazumdar, "On coresets for k-means and k-median clustering," in *STOC*. ACM, 2004, pp. 291–300.

[64] 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*. SIAM, 2013, pp. 1434–1453.

[65] M. Shindler, A. Wong, and A. W. Meyerson, "Fast and accurate k-means for large datasets," in *NIPS*, 2011, pp. 2375–2383.

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

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

[68] I. W. Tsang, J. T. Kwok, and P.-M. Cheung, "Core vector machines: Fast svm training on very large data sets," *JMLR*, vol. 6, no. Apr, pp. 363–392, 2005.

[69] I. W. Tsang, A. Kocsor, and J. T. Kwok, "Simpler core vector machines with enclosing balls," in *ICML*. ACM, 2007, pp. 911–918.

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

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

[72] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the machine learning lifecycle with mlflow," in *IEEE Data Engineering Bulletin, 41(4)*, 2018. [Online]. Available: https://cs.stanford.edu/~matei/papers/2018/ieee_mlflow.pdf

[73] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia, "Modeldb: A system for machine learning model management," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '16. New York, NY, USA: ACM, 2016, pp. 14:1–14:3. [Online]. Available: http://doi.acm.org.ezproxy.uta.edu/10.1145/2939502.2939516

[74] B. Goodman and S. Flaxman, "European union regulations on algorithmic decision-making and a âright to explanationâ," *AI magazine*, vol. 38, no. 3, pp. 50–57, 2017.

[75] Google, "Increasing transparency with google cloud explainable ai," 2020, https://cloud.google.com/explainable-ai, Last accessed on 2020-02-12.

[76] Microsoft, "Model interpretability in azure machine learning," 2020, https://docs.microsoft.com/en-us/azure/machine-learning/how-to-machine-learning-interpretability, Last accessed on 2020-02-12.

[77] C. Rudin, "Please stop explaining black box models for high stakes decisions," *CoRR*, vol. abs/1811.10154, 2018. [Online]. Available: http://arxiv.org/abs/1811.10154

[78] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[79] H. Nori, S. Jenkins, P. Koch, and R. Caruana, "Interpretml: A unified framework for machine learning interpretability," *arXiv preprint arXiv:1909.09223*, 2019.

[80] H20.AI Interpretability, "H20.ai driverless ai," 2020, http://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/interpret-non-ts.html, Last accessed on 2020-02-12.

[81] H20.AI, "H20.ai driverless ai," 2020, http://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/index.html, Last accessed on 2020-02-12.

[82] E. Štrumbelj and I. Kononenko, "Explaining prediction models and individual predictions with feature contributions," *Knowledge and information systems*, vol. 41, no. 3, pp. 647–665, 2014.

[83] C. Molnar, *Interpretable Machine Learning*, 2019, https://christophm.github.io/interpretable-ml-book/.

[84] UCI Repository, "Census income dataset," 2020, https://archive.ics.uci.edu/ml/datasets/Census-Income+%28KDD%29, Last accessed on 2020-04-01.

[85] P. Schmidt and A. D. Witte, *Predicting recidivism in north carolina, 1978 and 1980*. Inter-university Consortium for Political and Social Research, 1988.

[86] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[87] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, no. 3, p. 31â57, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3236386.3241340

[88] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM computing surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[89] A. Adadi and M. Berrada, "Peeking inside the black-box: A survey on explainable artificial intelligence (xai)," *IEEE Access*, vol. 6, pp. 52 138–52 160, 2018.

[90] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, "Explaining explanations: An overview of interpretability of machine learning," in *DSAA*. IEEE, 2018, pp. 80–89.

[91] D. Alvarez-Melis and T. S. Jaakkola, "On the robustness of interpretability methods," *arXiv preprint arXiv:1806.08049*, 2018.

[92] S. Thirumuruganathan, N. Tang, M. Ouzzani, and A. Doan, "Data curation with deep learning," *EDBT*, 2020.

[93] A. Asudeh, A. Nazi, J. Augustine, S. Thirumuruganathan, N. Zhang, G. Das, and D. Srivastava, "Leveraging similarity joins for signal reconstruction," *Proc. VLDB Endow.*, vol. 11, no. 10, p. 1276â1288, Jun. 2018. [Online]. Available: https://doi.org/10.14778/3231751.3231752

[94] Z. Kaoudi, J.-A. Quiane-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal, "A cost-based optimizer for gradient descent optimization," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD â17. New York, NY, USA: Association for Computing Machinery, 2017, p. 977â992. [Online]. Available: https://doi.org/10.1145/3035918.3064042

[95] A. Ardalan, A. Doan, A. Akella *et al.*, "Smurf: self-service string matching using random forests," *Proceedings of the VLDB Endowment*, vol. 12, no. 3, pp. 278–291, 2018.

[96] N. Koudas, R. Li, and I. Xarchakos, "Video monitoring queries," *arXiv preprint arXiv:2002.10537*, 2020.

[97] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: Optimizing neural network queries over video at scale," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1586â1597, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137664

[98] S. Nakandala, A. Kumar, and Y. Papakonstantinou, "Incremental and approximate inference for faster occlusion-based deep cnn explanations," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1589–1606.

[99] W. Le, A. Kementsietsidis, S. Duan, and F. Li, "Scalable multi-query optimization for sparql," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 666–677.