Processing Analytical Workloads Incrementally

Priyank Gupta
University of Toronto
priyank@cs.toronto.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Europa Shang
University of Toronto
europa@cs.toronto.edu

Ryan Johnson
University of Toronto
ryan.johnson@cs.utoronto.ca

Calisto Zuzarte
IBM Toronto
calisto@ca.ibm.com

ABSTRACT

Analysis of large data collections using popular machine learning and statistical algorithms has been a topic of increasing research interest. A typical analysis workload consists of applying an algorithm to build a model on a data collection and subsequently refining it based on the results.

In this paper we introduce model materialization and incremental model reuse as first class citizens in the execution of analysis workloads. We materialize built models instead of discarding them in a way that can be reused in subsequent computations. At the same time we consider manipulating an existing model (adding or deleting data from it) in order to build a new one. We discuss our approach in the context of popular machine learning models. We specify the details of how to incrementally maintain models as well as outline the suitable optimizations required to optimally use models and their incremental adjustments to build new ones. We detail our techniques for linear regression, naive bayes and logistic regression and present the suitable algorithms and optimizations to handle these models in our framework.

We present the results of a detailed performance evaluation, using real and synthetic data sets. Our experiments analyze the various trade offs inherent in our approach and demonstrate vast performance benefits.

1. INTRODUCTION

Analytics on large collections of data is a topic of vast interest in recent years. Although analysis of data was always central in the data management community, the prevalence of various machine learning and statistical systems/packages has corroborated to the interest. As a result several recent lines of research across communities aim to engineer popular machine learning techniques both at the algorithmic as well as the systems level to scale in large data collections [2, 13, 21, 16].

Data analytics tasks however, are rarely run in isolation. Typically an analysis workload consists of applying an algorithm (e.g., machine learning algorithm or statistical operation) on a large data set building a model and subsequently refine the operation based on the results of previous steps. For example consider building a model (e.g., regression operation) on a data set produced for the first two weeks of a month (e.g., sales data as it relates to various traffic parameters and promotions activities on a web site). Based on the results of the operation (e.g., regression parameters, error, etc) one decides to run an additional regression operation for the data set representing the entire month. Alternatively during a data exploration task, one creates a data model for a year worth of data collected for a service, only to decide to drill down and build a model for the second month of the year that seems to present an anomaly for the given model fit.

It is evident that analysis tasks can be part of an analysis workload and rarely run in isolation. Moreover, exploratory tasks, may involve extending or refining previously completed tasks. As a result, this behavior reveals certain dependencies among the steps of an analysis workload. Such dependencies expose opportunities for work sharing across tasks. For example one may be able to reuse the model for the first two weeks of the month instead of building the model for the entire month from scratch. Such reuse could be achieved by incrementally updating the current model with additional data. Alternatively if the model for the subsequent two weeks of the month is available, the desired model for the month could be build by combining the two models as opposed building it from scratch. Such an option is advantageous as the models are already build and one simply derives a new one without the need to access possibly large collections of data. In a similar fashion we may be able to reuse the model build for a month to derive the model for the first two weeks of the month by removing the last two weeks worth of data from the model, instead of building the desired model from scratch.

These examples reveal two basic observations that we explore further in this paper. First analysis workloads consisting of multiple modelling tasks are amenable to work sharing across tasks. In particular one may be able to reuse models previously build on a data set in order to derive new models on demand. Second, incremental updates (inserting or deleting data) is an operation that may aid to derive a new model from an existing one. It is natural to expect that some models would enable work sharing easier than others. Some models for example may allow us to derive a new model by ”extending” (with new data) or ”shrinking” (removing data) the current model and still derive the exact same model we would have derived by building it from scratch utilizing base data. Some other models could allow us to do this only approximately. At the same time from a performance standpoint it may not always be beneficial to utilize an existing model and derive a new one by adding or deleting data from it. We expect that in some cases utilizing an existing model to derive a new one may be beneficial (we may be able to build the model much faster) but in some other cases, building the model from scratch is the best (faster) option.

Currently, systems that enjoy vast attention and are utilized for data analysis tasks (e.g., R [7]) do not take advantage of such dependencies and inherent relationships across operations of a data analytics workload. An analyst has to be aware of work sharing opportunities as well as optimization opportunities and express them (in code) explicitly which is not an ideal solution.

In this paper we initiate a study to explore these possibilities. We introduce *model materialization* and *incremental model reuse* as first class citizens in the execution of an analytical workload. By model materialization we mean that a model can be stored after it is build in order to be considered when generating other models. Since a model requires some space to store it, we incur a storage cost but we aim to offset such costs with increased performance in executing subsequent operations. By incremental model reuse we mean that during the decision to build a model required by an analyst, we consider models previously build as candidates to generate the model. Thus, we decide whether we should reuse existing models and/or adjust them incrementally or build the model from scratch. The decision is typically based on performance and we aim to make the choice that results in building the model fastest. Towards this goal we adopt a cost model that aids in this decision; we develop the suitable optimization frameworks that decide which models to use and the suitable action to take with the objective of producing the resulting model with the smallest cost.

More specifically in this paper we make the following contributions:

- We introduce *model materialization* and *incremental model reuse* as frameworks to be considered during the execution of an analysis workload.
- Using linear regression and Naive Bayes as examples, we demonstrate how these common models can be casted in our framework. More specifically we establish that incremental model reuse and model materialization offer large performance benefits, while guarantying that models are constructed without loss of accuracy.
- We introduce an algorithm that given a collection of materialized linear regression/naive bayes models, chooses the best models to reuse and also the suitable operations in order to modify them deriving the desired target model with minimal cost.
- Using logistic regression as an example, we demonstrate that incremental model reuse and model materialization offer large performance benefits while guarantying that models are constructed with quantifiable loss in accuracy.
- We introduce an algorithm that given a collection of logistic regression models, chooses the best models to reuse and the suitable operations in order to modify them deriving the desired target model with minimal cost.
- We present the results of an extensive performance comparison demonstrating the performance benefits of our approach under varying parameters of interest.

This paper is organized as follows: Section 2 presents introductory material and basic notation. Section 3 demonstrates incremental manipulation of linear regression and naive bayes models, followed by Section 4 that treats the case of logistic regression models. Section 5 introduces our optimization framework followed by Section 6 that details and empirical evaluation of the proposal. Section 7 discusses related work and Section 8 concludes the paper.

2. BACKGROUND

We provide basic notation and a brief introduction to the techniques we adopt to showcase our overall approach. A more detailed description of the algorithms is available elsewhere [8, 17]

2.1 Linear Regression

Linear regression is modelling the relationship between a scalar dependent variable and one or more independent variables. Consider a data set of n records; each record x is a d-dimensional feature vector of independent variables denoted by $\mathbf{x_i}$ and a target dependent variable $y_i \in \mathbb{R}$. Generally, a linear regression takes the following form :

$$y_i = w^T \mathbf{x_i} + \epsilon_i$$

where w is the weight vector which is estimated and ϵ_i is an error term. Usually, the weight parameters are learned by minimizing sum of squared errors. A L_2-regularization term is added to avoid over-fitting of the model. The solution thus obtained has a closed form and is represented as :

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}(\mathbf{X}^T\mathbf{y}) \tag{1}$$

$\mathbf{X}$ is a $n \times d$ matrix of the input vectors, $\mathbf{y}$ is a $n \times 1$ matrix of the target values and λ is the regularization parameter.

2.2 Naive Bayes Classifier

Naive Bayes classifiers are simple probabilistic models assuming pair-wise independence of features given the class label. Albeit simple, Naive Bayes models perform very well in classification problems[20]. Given a class variable Y and a set of predictor variables $x_1, ..., x_d$ Bayes theorem states that

$$P(Y = c|x_1, ..., x_d) = \frac{P(Y = c).P(x_1, ..., x_d|Y = c)}{P(x_1, ..., x_d)}$$

Under the naive assumption and given that $P(x_1, ..., x_d)$ is constant for a particular training set we can conclude that

$$P(Y = c|x_1, ..., x_d) \propto P(Y = c).\prod_{i=1}^{d} P(x_i|Y = c)$$

$P(Y = c)$ can be calculated from training data by maximum likelihood estimation. The class probability $P(Y = c)$ is simply the relative frequency of class c in the training set, $P(Y = c) = N_c/N$ where N_c is number of training example which have class c and N is the total number of training examples.

Depending upon the choice of distribution for the conditional density $P(x_1, ..., x_d|Y = c)$ we have variations of the Naive Bayes classifier. A popular choice in the case of real valued features is the Gaussian distribution.

$$P(x_1, ..., x_d|Y = c) = \prod_{j=1}^{d} \mathcal{N}(x_j|\mu_{jc}, \sigma_{jc}^2)$$

where μ_{jc} is the mean of feature j in samples with class label as c and σ_{jc}^2 is its variance. This is often referred to as *Gaussian Naive Bayes*. In case of categorical features the multinomial distribution is a preferred choice for conditional density. The distribution is parametrized by vectors $\theta_c = (\theta_{c1}, ..., \theta_{cd})$ for each class, d is the dimension of the feature vector and θ_{ci} is the probability $P(x_i|c)$ of feature i appearing in sample belonging to class c.

$$P(x_1, ..., x_d|Y = c) = (\sum_{i}^{d} x_i)! \prod_{i=1}^{d} \frac{\theta_{ci}^{x_i}}{x_i!}$$

θ_{ci} can be calculated by a smoothed version of maximum likelihood estimation.

$$\theta_{ci} = \frac{N_{ci} + 1}{N_c + d}$$

where $N_{ci} = \sum_{j=1}^{n} x_i^{(j)}[Y = c]$, $N_c = \sum_{i=1}^{d} \sum_{j=1}^{n} x_i^{(j)}[Y = c]$ and n is the total number of points in the training set. These counters are computed for each class in the training data.

2.3 Logistic Regression

Logistic regression is a linear classifier belonging to the family of Generalized Linear Models [8]. Let y denote a class variable and x represent a feature vector, then Logistic Regression can be formally represented as an optimization problem minimizing a loss function to identify the model parameters. The loss function has the following form

$$F(w) = \frac{1}{n} \sum_{i=1}^{n} L(w; x^{(i)}, y^{(i)}) + \lambda R(w) \qquad (2)$$

A very common choice for function L in logistic regression is the cross entropy loss function :

$$L(w; x^{(i)}, y^{(i)}) = y^{(i)} log h_w(x^{(i)}) + (1 - y^{(i)}) log(1 - h_w(x^{(i)}))$$

and regularization function $R(w) = \|w\|^2$. Here $h_w(x)$ is the logistic function $h_w(x) = \frac{1}{1+e^{-w^T x}}$.

The Stochastic Gradient Descent(SGD) algorithm [17] is used to optimize the loss function to determine the model parameters. SGD initializes the model parameter w to some w_0 and then updates the parameter as

$$w \leftarrow w - \alpha \nabla F_i(w)$$

where α is the learning rate and $\nabla F_i(w)$ is the gradient of the convex loss function just using the $i^t h$ sample. Stochastic gradient descent requires a single pass on the data to converge.

3. AN INCREMENTAL APPROACH

We now demonstrate how model materialization and incremental model reuse can be supported in each of the types of models we consider. We discuss how one can combine two models on different data sets to produce a new model on the union of the data sets. We also discuss how an existing model can be manipulated (by adding or removing data) to produce a new one. Formally, let M_1 be a model on data set D_1 and M_2 is the model on data set D_2. We assume that the data sets D_1 and D_2 have the same properties. We discuss two machine learning models described in the previous section, Linear Regression and Naive Bayes.

3.1 Model Materialization

A typical machine learning model is characterized by its parameters. In order to support incremental updates to a given model extra information has to be maintained depending on the model. We show that while materializing a model we can also materialize extra information that would be sufficient in supporting incremental updates. This information varies across different types of models as discussed further in this section.

3.1.1 Linear Regression

Let D be a data set of n points and let M represent a machine learning model build on this data set.

Parameters for a linear regression are provided by Equation 3. The equation can be considered as a combination of two terms $A = X^T X$ and $B = X^T y$. Simplifying the terms

$$X^T X = \begin{bmatrix} \sum_{j=1}^{n} x_1^{(j)} x_1^{(j)} & \dots & \sum_{j=1}^{n} x_1^{(j)} x_d^{(j)} \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^{n} x_d^{(j)} x_1^{(j)} & \dots & \sum_{j=1}^{n} x_d^{(j)} x_d^{(j)} \end{bmatrix}$$

$$X^T Y = \begin{bmatrix} \sum_{j=1}^{n} x_1^{(j)} y^{(j)} \\ \vdots \\ \sum_{j=1}^{n} x_d^{(j)} y^{(j)} \end{bmatrix}$$

where A is a $d \times d$ matrix and each term is the sum product of any two features of the feature vector over the n training samples. $X^T y$ is a $d \times 1$ matrix where each term is the sum product of the features and the target values. We will maintain matrix A and B, along with the model parameters while building a model. Thus we end up maintaining $d^2 + d$ extra values. It is important to note that the amount of extra information we have to maintain is independent of the number of training samples (n). Given that we have both the components A and B we can compute the model parameters at any point using equation 2. Later on we will show how we can support incremental updates to Linear Regression model utilizing this information.

3.1.2 Naive Bayes

As discussed in section 2.2 *Gaussian Naive Bayes* is parametrized by the following variables: the class prior probabilities $P(Y = c) = \frac{N_c}{N}$, μ_{jc} and σ_{jc}^2 the parameters explaining the conditional density distribution. These parameters can be computed as shown below

$$N_c = \sum_{i=1}^{n} [Y^{(j)} = c]$$

$$\mu_{jc} = \frac{\sum_{i=1}^{n} x_j^{(i)} [Y^{(j)} = c]}{N_c}$$

$$\sigma_{jc}^2 = \frac{\sum_{i=1}^{n}(x_j^{(i)}[Y^{(j)}=c])^2}{N_c} - \left(\frac{\sum_{i=1}^{n} x_j^{(i)}[Y^{(j)}=c]}{N_c}\right)^2$$

We maintain N_c for each class in the data set, which is the number of samples belonging to each class. In order to calculate μ_{jc} we maintain the sum of feature j over the samples in class c, represented by S_{jc}. Similarly for σ_{jc} we maintain the sum of squares of the values of feature j in class c, represented by SS_{jc}. Maintaining the statistics above we calculate all the parameters of the model. Assuming we have C classes in total in the data set, we need to maintain $O(d \times C)$ values. This is again independent of the number of training examples (n).

The multinomial Naive Bayes model also has the same class prior probabilities $P(Y = c) = \frac{N_c}{N}$. In addition we have to maintain θ_{ci} for which we need to also store N_{ci} and N_c. These parameters are expressed as sum of feature values across the classes. For the case of the multinomial model, we need to maintain $O(d \times C)$ number of parameters for the model.

3.2 Incremental Model Updates

In this section we demonstrate how incremental changes (data additions or deletions) can be supported by the two models considered. Formally, let M be a model build on data set D consisting of points n. We will demonstrate the incremental changes by considering adding point $(p_1, ..., p_d, y)$ to the data set D, where d is the dimension of the data. We wish to find the parameters of the new model M' for data set $D' = D \cup (p_1 \ldots p_d, y)$ of size $n + 1$.

3.2.1 Linear Regression

For the linear regression model M we have already computed matrix A and B on data set D. We will calculate the A' and B' on D' by operating on A and B and updating them to reflect the new point. The equations below show how to update matrix A and B:

$$A'_{ij} = \sum_{j=1}^{n} x_i^{(j)} x_j^{(j)} + p_i p_j$$

$$B'_{i1} = \sum_{j=1}^{n} x_i^{(j)} y^{(j)} + p_i y$$

Deletions are handled similarly. Larger collections of points can be added/deleted in a similar fashion. Other statistics computed while building regression models like ANOVA table, AIC etc. which explain the goodness of fit of the model can also be incrementally maintained in a similar fashion. Details have been omitted for brevity.

3.2.2 Naive Bayes Classifier

For the Naive Bayes model M we have computed N_c , S_{jc} and SS_{jc} on D. We can update these statistics for D' according to the equations below

$$N'_c = N_c + [y = c]$$

$$S'_{jc} = S_{jc} + p_j[y = c]$$

$$SS'_{jc} = SS_{jc} + p_j^2[y = c]$$

Given that we have the updated statistics we can compute the parameters of the updated model M'. Similar observations hold for deleting data as well as operating on collections of points.

3.3 Combining Models

Let D be the underlying data set of n points. Assume that points in D are associated with a unique identifier, namely a point $p \in D$ is represented as $p = (id, y, \mathbf{x})$, where id is the identifier, y the dependent (class) variable and $\mathbf{x}$ the feature vector as before. To simplify notation for the remainder of the paper, we assume, without loss of generality that the unique identifier imposes a natural ordering in D. For example id could be a time-stamp associated with the point (indicating the time it was generated). Casting our entire framework for the case where the points of the underlying data set D do not have a unique ordering is indeed possible. It requires however a different methodology and we defer description of this case in our subsequent future work. Also for brevity we will denote as D_i both the model and the data set (subset of D) for which we wish to build a model on. A sequence of these data point identifiers determines a *model descriptor* which is a range of points in D. Let D_1 and D_2 be data sets represented by model descriptors $d(D_1) = [a_1, b_1]$ and $d(D_2) = [a_2, b_2]$. Our aim is to compute the model $D_c = D_1 \cup D_2$

We discuss the linear regression case. Naive Bayes models are handled similarly so we omit the description for brevity. Let D_1 and D_2 be two linear regression models. For each model we maintain the associated matrices $A = X^T X$ and $B = X^T y$ along with the model descriptor signifying the data set on which it was calculated. Computing the regression model $D_c = D_1 \cup D_2$, involves considering two cases: *Case 1*: The two data sets do not have any points in common i.e. $D_1 \cap D_2 = \phi$; this case can be easily identified by comparing the model descriptors of the two data sets. A specific entry in the matrix $X^T X$ for model D_1 looks like $\sum_{j}^{D_1} x_a^{(j)} x_b^{(j)}$, where a and b are any two features. Thus, it can be seen that the corresponding matrix A on data set D_c can be computed as

$$\sum_{j}^{D_c} x_a^{(j)} x_b^{(j)} = \sum_{j}^{D_1} x_a^{(j)} x_b^{(j)} + \sum_{j}^{D_2} x_a^{(j)} x_b^{(j)}$$

which is essentially adding the corresponding elements of matrix A of the two models directly.

Case 2: The two data sets have points in common i.e $D_1 \cap D_2 \neq \phi$; in this case the points common to both data sets can be determined from the corresponding model descriptors. If we directly operate on the two models the points which are common will be accounted for twice. Thus, we need to exclude points represented in both model and make sure we account for them once in the final model. We compute matrix A on data set D_c as follows:

$$\sum_{j}^{D_c} x_a^{(j)} x_b^{(j)} = \sum_{j}^{D_1} x_a^{(j)} x_b^{(j)} + \sum_{j}^{D_2} x_a^{(j)} x_b^{(j)} - \sum_{j}^{D_1 \cap D_2} x_a^{(j)} x_b^{(j)}$$

$$\sum_{j}^{D_c} x_a^{(j)} x_b^{(j)} = \sum_{j}^{D_1} x_a^{(j)} x_b^{(j)} + \sum_{j}^{D_2 - D_1} x_a^{(j)} x_b^{(j)}$$

$$\sum_{j}^{D_c} x_a^{(j)} x_b^{(j)} = \sum_{j}^{D_2} x_a^{(j)} x_b^{(j)} + \sum_{j}^{D_1-D_2} x_a^{(j)} x_b^{(j)}$$

The matrix $X^T y$ for D_c can be computed in a similar fashion. Notice that in this case we need to retrieve a few extra points from D_1, D_2. This incurs an IO cost that needs to be accounted for (see section 5).

4. INCREMENTAL LOGISTIC REGRESSION MODELS

Stochastic Gradient Descent(SGD) is a popular optimization framework for estimating parameters of a Logistic Regression model. SGD is a sequential algorithm that updates weight parameters at each iteration until convergence. A typical drawback of SGD is its poor scalability on large data sets. Recognizing the importance of analytical tasks on massive data sets, recent work has established methodologies to scale SGD into realistic data sets [16, 21]. We adopt such methodologies and extend them to fit our framework.

A generic loss function for the Logistic Regression model is given in Equation 2. SGD is applied to identify the model parameters w which minimize the loss function. We describe a variant of the SGD algorithm called Mixture Weight Methods [16]. Let us consider a sample $S = (S_1, ..., S_p)$ of pm points formed by p sub-samples of m points each drawn i.i.d, $S_1, ...S_p$. Algorithm 1 outlines the steps for executing Mixture Weight Method. Notice that the outer-loop of the algorithm can be executed in parallel and as a result the approach can easily utilize multiple processors if required.

Algorithm 1 Mixture Weight Method

```
for all $i \in \{1, ...p\}$ do
    $\mathbf{w_i} \leftarrow 0$
    for $t \leftarrow 1$ to $T$ do
        $\nabla F_{S_i}(w) \leftarrow$ GRADIENT$(F_{S_i}(w))$
        $w_i \leftarrow w_i + \lambda(\nabla F_{S_i}(w))$
    end for
end for
Aggregate all $w_\mu = \sum_{k=1}^{p} \mu_k w_k$
```

Where F_{S_i} is the optimization function for sample S_i and T is the number of iteration required to converge. Thus, algorithm 1 computes the model parameters on subsets of data and then averages the parameters across all the subsets to compute the parameter for the complete set of data. In [16] it is shown that Algorithm 1 has good convergence properties and under certain assumptions establishes a relationship between the w_μ estimated and the values computed executing SGD on the entire data set.

We extend this idea in our framework as well. Let D be an underlying data-set of size n and a point $p \in D$ is represented as $p = (id, y, \mathbf{x})$, where id is the identifier, y the dependent (class) variable and $\mathbf{x}$ the feature vector as before.

A request to create a logistic regression model on data set D_q (the query set), is represented by a range of id values $[a, b]$ over D such that $b - a = |D_q| + 1$. The query data set is segmented into smaller chunks of equal size l with the obvious assumption that $l \le |D_q|/2$. This results into $\lfloor \frac{|D_q|}{l} \rfloor$ number of chunks of equal size. These chunks are created in the increasing order of ID values. A chunk S_i is given by the following range

$$S_i = [a + (i-1) * l, a + i * l]$$

and $i \in \{1, ..., \lfloor \frac{|D_q|}{l} \rfloor\}$. Assuming that the logistic regression models for each chunk are available, they are combined in the spirit of algorithm 1 and produce the model for D_q. Assuming that none of the chunks is available, a request to build the model for D_q can utilize the base data to build the logistic regression model. At the same time, the chunks are generated for D_q, the logistic regression model build for each of them, and the result is materialized in order to benefit future model creation requests.

Any request to build a logistic regression model for a data set $D_q^{'}$ first tests whether $D_q^{'}$ contains any of the chunks for which a model has already been materialized. If it does we can readily utilize its parameters and save computation time. Any parts of $D_q^{'}$ that are not currently "covered" by existing chunks have to be computed from the base data set. Thus, we retrieve the parts of $D_q^{'}$ for which we don't have the model, generate chunks of size l and compute the model parameters for them. Finally we average all parameters from all chunks to compute the model. Algorithm 2 presents our overall approach.

Algorithm 2 Incremental Logistic Regression

```
 1: procedure INCREMENTAL LOGISTIC REGRESSION($D_q$)
 2:     $S \leftarrow$ ranges in $D_q$ for which a model already exists
 3:     $P \leftarrow \{\}$
 4:     for all the ranges $r \in S$ do
 5:         $D_q \leftarrow D_q - r$
 6:         $P_i \leftarrow$ Linear Regression parameters for r
 7:         $P \leftarrow P \cup P_i$
 8:     end for
 9:     Sort $D_q$ in increasing order of $ID$ values
10:     Create chunks of size $l$ from $D_q$
11:     Compute Linear Regression parameters on each
        chunk $l$ and add to $P$
12:     Average all parameters in $P$
13: end procedure
```

Theorem 1 establishes a relationship between the outcome of Algorithm 2 on $D_q^{'}$ and that computed by applying SGD directly on $D_q^{'}$.

THEOREM 1. *Let w_μ denote the mixture of weight vector obtained by applying Algorithm 2 on a model query D_q and μ_{SGD} be the weight vector computed by applying SGD on D_q. Then, for any $\delta > 0$, with probability at least $1 - \delta$, the following inequality holds:*

$$\|w_\mu - w_{SGD}\| \le \frac{R\sqrt{2}}{\lambda}\left(\frac{1}{\sqrt{l}} + \frac{1}{\sqrt{|D_q|}}\right) + \frac{2\sqrt{2}R}{\lambda\sqrt{pl}}\sqrt{log1/\delta}$$

where R is the bound for the norm of feature vectors, λ is the regularization constant, $p = \lfloor \frac{|D_q|}{l} \rfloor$ is the number of chunks of D_q created in step 10 of Algorithm 2, l is the size of each chunk and $1 - \delta$ represents the probability with which this inequality holds. The proof of 1 follows the methodology presented in [16] and is available in the full version of the paper [5].

Note that in contrast to the discussion of section 3.2, for logistic regression models, this framework supports adding points to an existing model not deleting them. Thus we can construct new models only by adding points to existing models (combining existing chunks). This is inherent to the nature of the approximation of the logistic regression. As a result the space of all possible options to consider when creating a new model considers addition of points to an existing model, not deletions.

5. OPTIMIZATION CONSIDERATIONS

Given a collection of materialized models over a data set D, it is evident that a request to create a new model D_q can readily utilize existing models. We seek to understand the trade offs involved while building the new model D_q. Several options are available including building D_q by manipulating data from D or utilizing materialized models directly and/or suitably adjusting them using data from D.

Consider Figure 1a. It depicts data set D and four materialized models (D_1, D_2, D_3, D_4). A request to build model D_q is faced with numerous options. Using the materialized models to generate model D_q , Equations 3, 4 and 5 show different ways in which this can be achieved

$$D_q = D_3 + D_4 - [b, c] - [e, f]. \tag{3}$$

$$D_q = D_3 + D_4 - (D_1 - D_2) - [e, f]. \tag{4}$$

$$D_q = [c, d] + D_4 - [e, f]. \tag{5}$$

Equation 3 represents an execution strategy which will fetch models D_3 and D_4 combine them, then remove all points in the range of $[b, c]$ and $[e, f]$ (this constitutes incrementally updating, removing these points, from the combined model). This step consists of accessing D and retrieving all points between $[b, c]$ and $[e, f]$. In equation 4 instead of retrieving $[b, c]$ from D, we compute that operation by manipulating (subtracting) models D_2 and D_1. If the model allows (e.g., linear regression) we can subtract D_2 from D_1 and compute the model for $[b, c]$ directly. Similarly, Equation 5 represents another execution strategy which involves retrieving D_4 along with data points between $[c, d]$ and $[e, f]$ and manipulating them (incrementally updating, adding and removing points) to complete the model construction. Other choices are also possible including retrieving all points between $[c, e]$ from D and computing the model directly from base data. In order to be able to quantify the merits of each choice, as is typical in cost based query optimization [10] we need to a) assess all possible choices efficiently and b) quantify the cost of each option in order to determine the least cost way to build the model.

The specifics of the cost model are orthogonal to our approach. The cost depends on the type of model and also the model descriptor which may or may not involve disk access. In addition retrieving data from D typically involves disk access. The only requirement we impose in the cost model adopted is to be monotonic. This means that all things being equal, the cost of retrieving a certain number of data points from disk should be at least as costly as the cost of retrieving less points. For the remainder of the paper we assume a cost model C that is monotonic. To facilitate notation the cost of using a materialized model D_i is denoted as $C(D_i)$. The cost of retrieving n data points from disk is denoted as $F(n)$.

Let S be a collection of materialized models on data set D. For a model D_q, let $d(D_q) = [l_q, u_q]$ be a model descriptor on which a new model has to be computed. l_q and u_q in this case express a range of data points on D. We wish to identify the minimum cost collection of materialized models and/or data points from D that would be used to construct the model for $d(D_q)$, D_q.

Definition 1. Let $d(D_q) = [l_q, u_q]$ represent a model descriptor for model D_q which we wish to construct and S be the set of available materialized models. Then the set $S_R \subseteq S$ of *relevant models* for D_q is defined as follows :

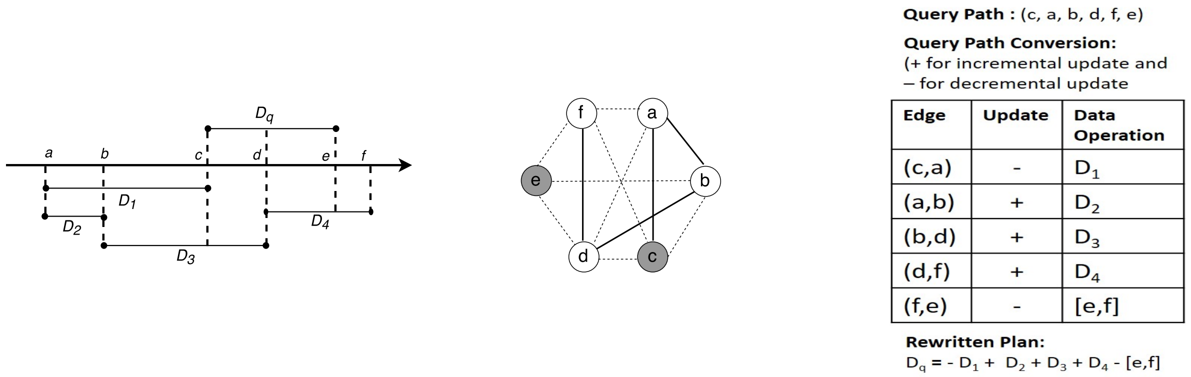
1. If for a materialized model $S_i \in S$, $d(S_i) \cap q \neq \varnothing$, then $S_i \in S_R$.
2. $\forall S'_i \in S$ such that $\exists S_j \in S_R$ with $d(S'_i) \cap d(S_j) \neq \varnothing$ then $S'_i \in S_R$.

Intuitively the models in S_R are *relevant models* because they either contain common data points with the ones of interest to D_q and/or they are models that can be manipulated (by combinations of models or incremental updates of models) to produce models that assist in computing D_q. As we can see in Figure 1a materialized models D_3, D_4 contain data points common with D_q while D_1 and D_2 can be manipulated along with D_3 to produce models relevant to the computation of D_q. While computing D_q, only relevant models will be part of S_R.

Algorithm 3 PreprocessDescriptors (S)

```
enhancedDescriptors ← mapping of descriptors and the corresponding materialized models
descriptor ← a model descriptor represented by $[l, u]$
arrayDescriptors ← array of descriptors
Sort S in increasing order of $l$ values
descriptor[0] ← $l$ value of first descriptor in S
descriptor[1] ← $u$ value of first descriptor in S
arrayDescriptors ← append first descriptor in S
for each descriptor $r \in S$ do
    if $r$ overlaps descriptor then
        descriptor[1] ← max(descriptor[1], $u$ value of $r$)
        arrayDescriptors ← append $r$
    else
        enhancedDescriptors.put(descriptor,arrayDescriptors)
        arrayDescriptors ← {}
        arrayDescriptors ← append $r$
        descriptor[0] ← $l$ value of $r$
        descriptor[1] ← $u$ value of $r$
    end if
end for
return enhancedDescriptors
```

The set of relevant models S_R is important since it accurately reflects the set of models to be considered during the computation of D_q. Instead of assessing all relevant models every time a new request for a model D_q arises, we pre-process the collection of all materialized models S to facilitate the derivation of S_R for a given D_q. Thus given S we pre-process it to facilitate the computation of relevant models. Algorithm 3 presents the overall approach. The basic idea is to pre-process S and create *enhanced* descriptors that are the union of multiple model descriptors. Such enhanced descriptors can facilitate quick search for relevant models.



Query Path : (c, a, b, d, f, e)

Query Path Conversion:
(+ for incremental update and
– for decremental update

Edge	Update	Data Operation
(c,a)	-	D_1
(a,b)	+	D_2
(b,d)	+	D_3
(d,f)	+	D_4
(f,e)	-	[e,f]

Rewritten Plan:
D_q = - D_1 + D_2 + D_3 + D_4 - [e,f]

(a) Materialized model state

(b) Query Graph using cost model

(c) Query Path Conversion

Figure 1: Graph modeling to find optimal execution strategy for query interval I_Q

Running algorithm 3 in the example of Figure 1a will produce two enhanced descriptors namely $[a, d]$ formed by combining descriptors for models $\{D_1, D_2, D_3\}$ and $[d, f]$ which constitutes the descriptor of model $\{D_4\}$.

Maintaining *enhancedDescriptors* makes it easier to compute the set S_R. When the descriptor of a model D_q is provided, we compare it against the *enhancedDescriptors*. If a descriptor intersects any of the descriptors in *enhancedDescriptors* all the materialized models mapped to that descriptor become part of S_R.

Algorithm 3 will produce the set S_R of all models that should be considered in deriving model D_q. Using the descriptors in S_R we create a complete undirected graph $G(V, E)$ where each node $v \in V$ corresponds to the l or u values of the model descriptions in S_R. As for our running example the set S_R contains models D_1 to D_4. Thus we add the l and u values of the descriptors of these materialized models. As we can see in figure 1b it contains a to f as nodes. An edge $\epsilon \in E$ corresponds to the cost of building a model for the data set specified by the two nodes adjacent to ϵ. If materialized model M exists for the data descriptor specified by the nodes adjacent to the edge ϵ then the cost of the edge is the cost of using model $M, C(M)$. If a model does not exist for that data set the cost of that edge is determined by the number of points in the range. In our example the solid edges in our graph represent the materialized models D_1 to D_4. For all the other edges the cost is given by $F(n)$, where n is the number of points in the interval represented by the edge. Given D_q and $d(D_q) = [l_q, u_q]$ values l_q, u_q represent the source and destination respectively. These are shown as grey nodes in Figure 1b.

Algorithm 4 Identify Optimal Execution Path

```
procedure GenerateGraph($S_R, D_q, C(M), F(n)$)
    initialize Graph $G(V, E)$
    for each descriptor $r \in S_R$ do
        G ← add vertices corresponding to $l$ and $u$ values of $r$
        G ← add an edge between two new vertices with weight $C(D_r)$
    end for
    for each vertex $v \in G$ do
        for each vertex $u \in G$ do
            if (no edge between $u$ and $v$) & $u \neq v$ then
                G ← add an edge b/w $u$ & $v$ with weight $F(|u - v|)$
            end if
        end for
    end for
    return $G(u, v)$
end procedure
procedure OptimalPath($S_R, D_q, C(M), F(n)$)
    Identify $S_R$ using algorithm PreprocessDescriptors
    G ← GenerateGraph($S_R, D_q, C(M), F(n)$)
    Apply Dijkstra's Algorithm using $d(D_q)$ $l$ and $u$ values as source/destination
    Return the shortest path
end procedure
```

Every path from source node to destination represents an execution strategy to construct model D_q. Figure 1c illustrates how to convert a path on the graph to a set of operations that compute the model. Consider a path on the graph represented by the following sequence of nodes (c, a, b, d, f, e). We fetch four materialized models D_1, D_2, D_3 and D_4 for the edges $(c, a), (a, b), (b, d)$ and (d, f) respectively. The edge (f, e) does not correspond to any materialized model , thus cost of that edge is equivalent to fetching the corresponding data points from disk. The decision whether to manipulate an existing model by adding or removing data points from it is decided by the nodes of the edge. If we traverse the edge (i, j) from i to j and $i > j$ then we remove points from the model otherwise we add data points. In our example edge (c, a) $c > a$ (as indicated in Figure 1a) and that constitutes removing points. The total cost of a query path is given by

$$C(D_q) = \sum_{i}^{k} cost(e_i) + (k - 1) * c_{merge}$$

where $cost(e_i)$ is cost of each edge and c_{merge} is cost of merging two materialized models. The cost c_{merge} depends on the type of model under consideration. For example for linear regression the cost is outlined in section 3.3. It involves (after retrieving the model parameters) a simple manipulation of corresponding model representations. It is expected that the cost of merging two materialized models is much less than the cost of fetching models or the cost of fetching data points from the disk ($c_{merge} \ll e_i$). Depending on how the model descriptors and model parameters are stored, retrieving them may not require any disk access. For example in the case of a linear regression model, the model descriptors would be just a range of values and the model parameters would be as outlined in Section 3.1.1.

It is evident that by construction the problem of identifying the minimum cost to construct the model D_q is equivalent to identifying the shortest path from a single source in a weighted graph. Dijkstra's algorithm can be used to identify the optimal solution in $O(|E|log|V|)$, $|E|$ is the number of edges and $|V|$ is the number of vertices in the graph.

We presented the entire solution for the case of models that support addition and removal of points to derive new models, as is the case of models such as linear regression and Naive Bayes. For the case of logistic regression removal of points is not supported in the model we utilize to approximate the regression. In this case we have to modify slightly the algorithm to enable optimization of logistic regression models as well. The changes are as follows:

- During identification of the set S_R we will include models such that their descriptors are fully contained in the descriptor $d(D_q)$.
- The graph G constructed will only contain directed edges from nodes i to j such that $i < j$.

These two changes will enable algorithm 4 to operate on logistic regression models and yield the least cost options to construct such models as well.

6. EXPERIMENTS

In this section we present a detailed performance comparison of our entire approach and proposal compared to alternate approaches. We utilize materialized models to save processing costs, while building new models for an incoming (model construction) query D_q as described in section 5. The natural alternative is not to materialize models, but instead build the new model directly from the raw data. We compare our approach against this baseline. Our aim from these experiments is three-fold : (a) Highlight the factors that affect performance for our materialization framework and associated trade-offs. (b) Detail the impact of our optimization framework in terms of its overheads and benefits and (c) analyze the accuracy of logistic regression materialization framework. Note that for the case of the linear regression and naive Bayes models, the models we construct are exactly the same as those constructed by the baseline, so there are no accuracy trade offs in these cases.

Data. We test our framework utilizing synthetically generated data. Two different data set are generated for regression and classification problem. The choice of synthetic data allows us to change various parameters during experimentation. In addition experiments are focused on performance while scaling the size of the model and performance does not depend on quality of data but is governed by the size and type of data. The data is generated using publicly available synthesizers [18]. A random noise and interdependency among features is added while synthesizing data to simulate real world scenarios. In this section we present results using data sets up to 5 millions points with 10 features in each point. We tested all algorithms with synthetically generated data sets of larger sizes but the trends observed in our experiments were nearly the same. In addition we utilized popular real data sets from UCI Machine learning repository [3] in our experiments and in all cases the results are consistent with those presented herein for synthetic data sets.

Experimental Setup. All our experiments were prototyped on top of MySQL(version 5.5.44) in a single node RDBMS setting. The model materialization framework code has been written in Python. The experiments were carried out on a PC running Linux Kernel Version 3.13.0-43-generic. The machine has a 3.40GHz Intel Core i7-3770 CPU with 16 GB of main memory.

Our framework is naturally parametrized by the size of the materialized models (l) and the size of the incoming model construction query (D_q). Another important parameter which is implicit in our discussion is the amount of data covered by the materialized models. Materialized models can be spread uniformly across the data set or may be concentrated on a few data points. To quantify the coverage we compute the number of unique data points covered by the materialized models and express it as a percentage of the total size of the data set. Formally let $D_1, ..., D_n$ be the collection of models materialized at a given stage in the framework. For the data set, D, coverage is defined as follows :

$$Coverage(\%) = \frac{|D_1 \cup D_2 ... \cup D_n|}{|D|} \times 100$$

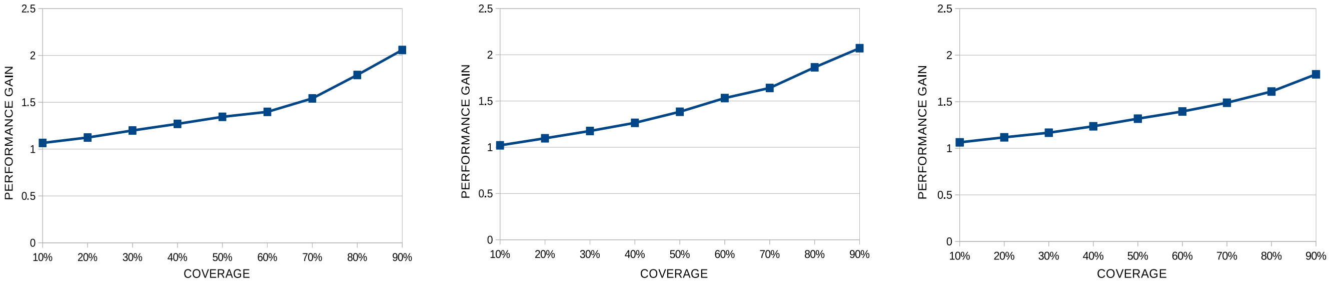
These parameters are varied across our experiments to understand their impact on performance gain. Let D_q be a model construction query. Our optimization framework identifies the optimal way to build model D_q. Let the overall time taken by our framework to build the model be T (including the optimization and model construction time). Let the time taken by the baseline be T_0. Then the performance gain is calculated as follows

$$Performance\ Gain(PG) = \frac{T}{T_0}$$

In all experiments we report expected numbers. A query set S containing one thousand queries is generated for each experiment. The query size is chosen from a uniform or normal distribution as explained in individual sections. These queries can represent a range of data points which is positioned anywhere across the underlying data. Similarly the materialized model size (l) is also chosen from a uniform distribution, normal distribution or a fixed size. We create a set of materialized models M on the data set with a given coverage as required in the experimental setting. The models are materialized before executing the query set S.

6.1 Analyzing Performance

We assess the overall performance gain attained by our approach as compared to the baseline. Experiments were run for all three machine leaning models Linear Regression, Naive Bayes and Logistic Regression. The sizes of the sets M and S are chosen from the same normal distribution, $\mathcal{N}(50K, 12.5K)$. The x-axis depicts the percentage of data covered by materialized models. We execute the queries in set S and report the performance gain. Figure 2a and 2b show that we were able to achieve a performance gain of **2x** as the coverage reaches 90%. The increase in coverage implies a higher probability of identifying *relevant models* for

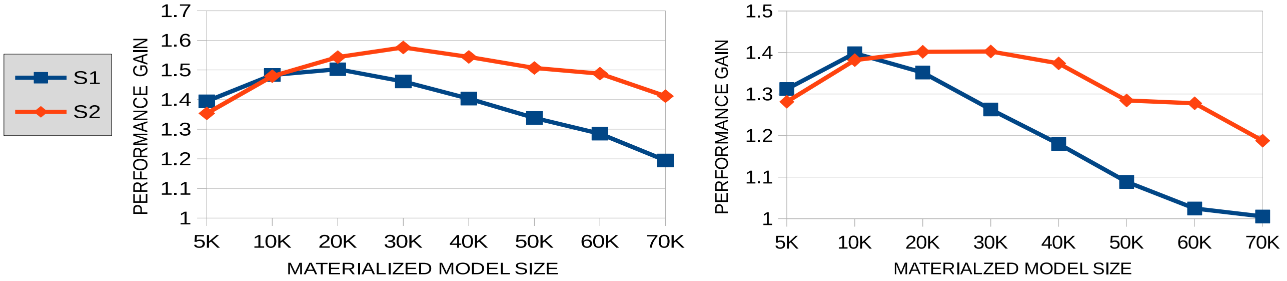


(a) Naive Bayes Model

(b) Linear Regression Model

(c) Logistic Regression

Figure 2: Performance gain against coverage percentage



(a) Naive Bayes Model

(b) Logistic Regression Model

Figure 3: Performance gain against materialized model size

the query. Thus the expected performance gain improves as the coverage increases. The performance gain for Logistic regression is shown in Figure 2c. The maximum performance gain achieved in logistic regression is **1.8x** which is slightly lower than the earlier two models. This can be explained by the fact that for Logistic Regression our framework supports only incremental updates to materialized models (section 4). Thus, it eliminates certain execution strategies which would have been faster in the presence of decremental updates.

Coverage	Model Sizes (MB)
20%	1.5
40%	1.8
60%	2.5
80%	3.5
90%	4.5

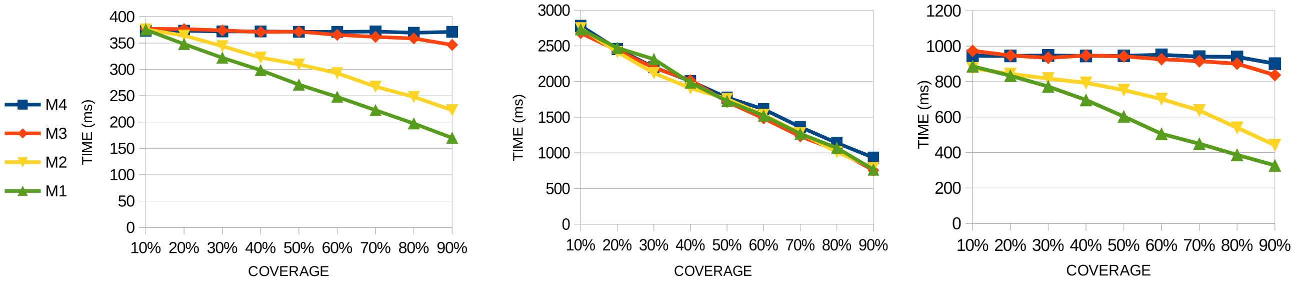
Table 1: Disk space occupied by materialized models for various coverage(%)

The previous experiment demonstrates that utilizing materialized models can have a profound effect on performance when constructing new. However materializing a model comes at a cost, namely that of storing the model descriptors as well as the model details (e.g., regression parameters and meta-data in the case of linear regression as defined in section 3). Table 1 depicts the space occupied by the materialized linear regression models for each value of coverage. The size of the materialized model is fixed at 5K points. The base data set size is 350MB containing 5M points with 10 features. As it is visible from the table, the overheads in storage imposed by the materialized models is around 1.2% of the original data. Similar trends hold for the other models of interest in our study. It is evident that the minor storage overheads are heavily compensated in light of the performance benefits.

6.2 Materialized Model Size and Performance Gain

The size of materialized models is an important parameter in our framework. With the next set of experiments we wish to understand the impact of the size of materialized models on performance. Two test query sets S1 and S2 of size 50K and 100k points are used as shown in the figure 3. On the x-axis we represent different materialized model sets of fixed size of coverage fixed to 50%. The size of the materialized model sets is varied from 5K points to 70K points as shown in the Figure 3a and 3b. We present results for Naive Bayes (supports both incremental and decremental updates) and Logistic Regression (supports only incremental updates) as similar trends hold for linear regression as well. Figure 3a, 3b present results for Naives Bayes and Logistic Regression respectively. We observe that for a fixed query size D_q and fixed coverage there is an optimum size of materialized models which results in maximum performance gain. We achieve a maximum performance gain for S1 at materialized model size of 20K for Naive Bayes. Similarly, for Logistic Regression we achieve the maximum performance gain at 10K materialized model size. As the size of the query increases the optimal materialized model size also increases. As shown in the graphs the query set S2 has its maximum at 30K and 20K for Naive Bayes and Logistic Regression respectively, which is larger than the maximum for S1. The exact position of the maximum on the graph depends on the size of the specific query (or query workload for multiple queries) for a given cost model.

6.3 Materialized Model and Query Size



(a) Small Size Model Query

(b) Large Size Model Query

(c) Small Size Model Query on Real Data

Figure 4: Time take by large and small queries for various materialized model sizes

We conducted experiments to quantify performance while scaling to larger input queries and materialized models sizes. The model chosen for these experiment was Naive Bayes, although linear regression also shows the same trends. Figure 4 shows four sizes of materialized models under consideration $M1$ to $M4$. $M1$ represents materialized models with their size chosen from a uniform distribution represented by U(25k,50k). Thus M1 is the scenario in which all the materialized models have a size uniformly distributed between 25K to 50K. Similarly M2,M3 and M4 are represented following a uniform distribution U(75k,100k), U(150K,200k) and U(250K,500K). Figure 4a shows the time taken to execute queries of small sizes represented by U(50K,100K). As depicted in the graph for M1 and M2 the time taken to execute the model queries decreases linearly as coverage increases. However for M3 and M4 which correspond to considerably larger materialized model sizes, the performance improvement becomes significant after 70% coverage. As coverage increases there is a higher probability to find two materialized models which can be subtracted in order to create a smaller model. Figure 4c shows similar trend for small queries on a real world data set from the UCI machine learning repository representing physical activity data of 3M points, consisting of 31 attributes and 13 classes. It is evident that the main trends are the same as in the case of synthetic data set as is the case in all of our experiments. Figure 4b is the graph for larger query sizes represented by distribution U(500K,750K). Since the query size is much larger we can observe that all four cases materialized models are utilized to generate the model for the input query. For M1, small models can be combined to generate the models for larger data sets. While for M4 a large materialized model which has the maximum overlap with the incoming model construction query is manipulated to generate the new model. It is evident that the relationship of the query size to the materialized model size is important in our setting. When the query workload has a much smaller size than the materialized model sizes (correspondingly when the query workload has much larger size than the materialized model sizes) employing our framework does not result in large performance benefits. It is evident however that enabling our framework in these cases does not impose an overhead either.

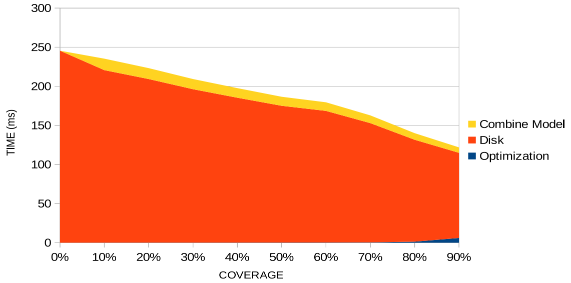
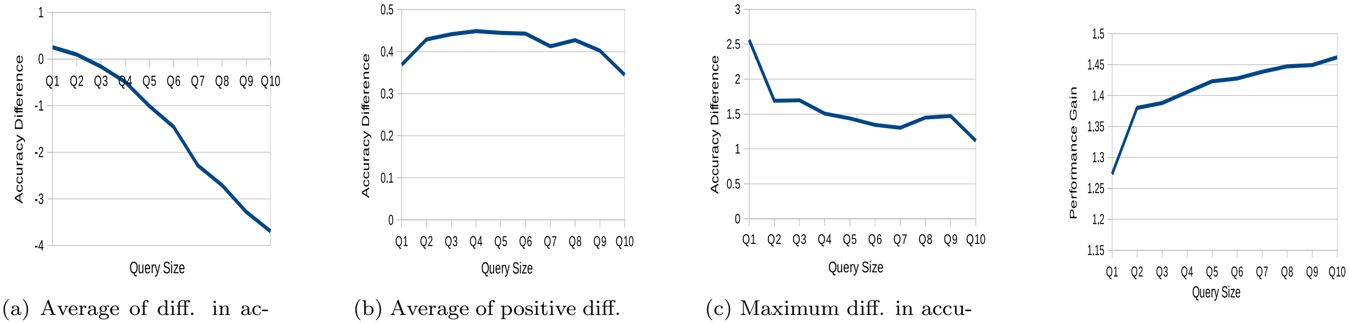


Figure 5: Distribution of time across various I/O and computation tasks

6.4 Optimization and I/O Time

As mentioned in section 5 the cost of merging models is considerably smaller as compared to disk access time. We measure the time taken by the three major components of our framework namely optimizer time, disk access time (including both fetching materialized model and/or fetching direct data points) and model combination time. The optimizer time refers to the time taken to run algorithm 4. The time spend in fetching any information from MySQL is referred to as I/O time. The time remaining in our computations which cannot be attributed to the above cases is the time taken to merge the models. Experiments were run on a test set of a thousand queries. The size of the model to be generated is chosen from the normal distribution $\mathcal{N}(50K, 12.5K)$.

The expected time for each component is reported as shown in graph 5. As can be observed the majority of time to create models is spent while fetching data from disk. Model combination time is fairly constant and is much smaller as compared to disk time. Optimizer time is insignificant for small coverage and only becomes visible (but still negligible) on the graph when coverage is close to 80% and above. As coverage increases the number of possible execution plans become considerably larger thus the optimizer takes much longer to build the graph and determine the shortest paths in the graph. This graph reveals that the overhead of running the optimization is minimal. Since the potential benefits of considering materialized models are significant, it is evident that if one chooses to materialize models, the performance overhead of the optimizer is negligible. Thus, running the optimizer, even if the decision is to employ the baseline, imposes minimal penalty in the query performance. In the graph the baseline is represented by the x-axis value at zero percent coverage. It can be seen that disk time reduces from 250 ms to 110 ms, while the optimizer time and model combination time are roughly 10ms. Thus, when the coverage is low, the overhead of the optimizer is so small that even when no materialized model can be utilized and the



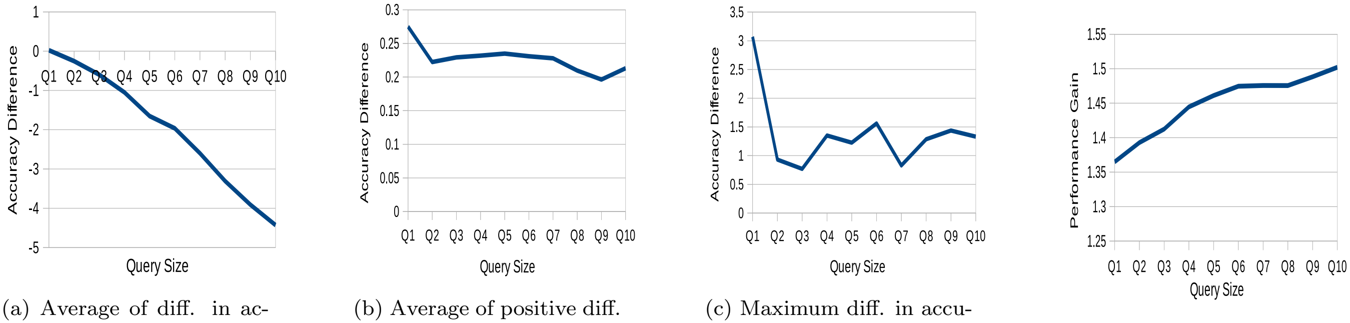
(a) Average of diff. in accuracy

(b) Average of positive diff. in accuracy

(c) Maximum diff. in accuracy

(d) Performance Gain

Figure 6: Accuracy and Performance statistics for Logistic Regression with materialized model size of 10K



(a) Average of diff. in accuracy

(b) Average of positive diff. in accuracy

(c) Maximum diff. in accuracy

(d) Performance Gain

Figure 7: Accuracy and Performance statistics for Logistic Regression with materialized model size of 20K

model has to be constructed from the baseline, the impact of the optimizer to the overall performance is immaterial as evident in Figure 5. At high coverage, the chances of utilizing materialized models are much higher. In that case, the small overhead of the optimizer is clearly compensated by the large savings in model construction time.

6.5 Accuracy

In this section we analyze the accuracy of our framework for the logistic regression models presented in section 4. We quantify the accuracy of the overall approach.

Synthetically generated classification data with 10 features and 2 classes were used to run test experiments. Similar trends hold when the number of classes increases, so we omit these experiments for brevity. We ran experiments on a test set S of a thousand queries. For each of these queries the model was built using our framework and also by applying SGD. We compare the accuracy on training data for both models by computing their difference. Let A refer to the accuracy of the model built by our framework and A_0 refer to accuracy of SGD algorithm, the accuracy difference can be represented as $A_0 - A$. Various statistics are reported on this difference. Figure 6a and 7a presents the average of the accuracy difference between the model constructed by our approach and the model constructed by SGD directly. The x-axis represents queries in increasing order of size. The graphs show negative average values which means that on average the model generated by our framework outperforms the model developed by SGD on training data. Also as the query size increases the expected performance of our model improves. Figure 6b and 7b presents the average difference in accuracy for the cases where $(A_0 - A) > 0$. It can be seen that the average positive difference lies within 0.5%. It is evident that the overall approach is highly accurate. Across the materialized model sizes we observe that larger size has better accuracy as compared to smaller sizes. Finally Figures 6c and 7c present the maximum difference across various query sizes. The graph shows that as the query size increases the maximum difference between the model computed by our framework and that computed by SGD decreases. It is visible from the graph that $max(A_0 - A) < 3\%$. The last set of graphs presents the trade off between accuracy and the corresponding performance gains achieved by our framework. As figures 6d and 7d suggest we experience a performance gain of 1.5x while we compromise accuracy by 3% in the worst case. Similar results were observed on real world data sets including the PAMAP2 publicly available data set [3]. Since they are consistent with what has been presented these results are omitted for brevity.

7. RELATED WORK

There has been an ever increasing interest to integrate statistical and machine learning capabilities to data management systems. Several efforts have been made in academia and industry to address this demand. Major database vendors now support analytical capabilities on top their database engines : IBM's SystemML [12] , Oracle's ORE [4], SAP HANA [6]. However the integration is loose and does not support notions of model persistence or incremental computations. In the open source community one can observe similar trends with MADLib [13] library support for Postgres. Other data platforms like Spark and Hadoop also support machine libraries as an external layer on top of their data processing system with MLLib [2] and Mahout [1] respectively. Such approaches either utilize an existing data management platform and deploy its extensions to provide analytics capabilities or represent systems that can execute machine learning and statistical packages. See [11] for a general overview of systems support for machine learning and statistical operations. Haloop [9] and Dryad [14] are examples of systems that utilize a form of persistence in their operations to improve the execution of a graph data flow. Although related in spirit, the approach and goal of these systems is to improve the performance of specific iterative graph data flow computations; they do not address the case of synthesizing a new model by extending and/or combining past models which is central in our approach.

Recent work [15] focused on pushing machine learning primitives inside a relational database engine. Our work is intended as a middle layer between the data processing engine and the analytical computing language layer. We require awareness of previous computations by collecting them and explore materialized models to build new models for the data. Our goal is to explore natural work sharing opportunities that exist in a typical data analysis workload.

Materializing portion of computations with the intention of reuse has also been explored in the domain of feature selection [19] for machine learning tasks. Our work however explores the incremental updates and reuse of model to build new models.

8. CONCLUSIONS

In this paper we presented an approach that utilizes model materialization and incremental model reuse as a first class citizen while processing data analytics workloads. Utilizing popular machine learning models we demonstrated their incremental aspects and detailed an optimization methodology that determines the best way (in terms of performance) to build a given new model. We demonstrated that our apporach can achieve significant savings in performance for new model construction while only imposing modest overheads in storage.

The work opens several avenues for future work. First there is a plethora of other models that are important and can be considered in conjunction with our framework. Studying their incremental aspects and embedding them into the same optimization framework is an interesting direction for future work. Incremental model reuse for analytics is an important direction of research that blends nicely with the way current data management systems build integrations to existing analytical packages. Our framework can be easily injected between the analytical package and the RDBMS and recognize as well as handle all opportunities for improved performance. We are currently building such as system based on the ideas presented herein in which we will report soon.

Finally, our focus in this paper has been in the case that a total ordering exists in the underlying data set. An interesting case is when such an ordering does not exist. In that case the model descriptors will be different as well as the associated optimizations. Indeed our entire framework can be extended for this case as well and we will be reporting on such extensions in our future work.

9. REFERENCES

[1] Apache Mahout. http://mahout.apache.org/.

[2] Apache Spark MLLib. http://spark.apache.org/.

[3] Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets.html.

[4] Oracle R Enterprise. https://docs.oracle.com/cd/E36939_01/doc.13/e36761.pdf.

[5] Processing Analytical Workloads Incrementally. http://www.cs.toronto.edu/~priyank/incremental_analytics.pdf.

[6] SAP HANA and R. http://help.sap.com/hana/sap_hana_r_integration_guide_en.pdf.

[7] The R Project for Statistical Computing. https://www.r-project.org/.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[10] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. *PODS*, 1998.

[11] T. Condie, P. Mineiro, N. Polyzotis, and M. Weimer. Machine learning on big data (sigmod tutorial). In *SIGMOD Conference*, pages 939–942. 2013.

[12] A. Ghoting and et al. SystemML: Declarative Machine Learning on MapReduce. *ICDE*, 2009.

[13] J. M. Hellerstein, C. Re, F. Schoppmann, and D. Z. Wang. The MADlib Analytics Library, 2012.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[15] A. Kumar, J. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. *SIGMOD*, 2014.

[16] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficinet Large-Scale Distributed Training of Conditional Maximum Entropy Models. *Advances in Neural Information Processing Systems*, 2009.

[17] K. P. Murphy. *Machine Learning a Probablistic Perspective*. MIT Press, 2012.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] C. Zhang, A. Kumar, and C. Re. Materialization Optimizations for Feature Selection Workloads. *SIGMOD*, 2015.

[20] H. Zhang. The Optimality of Naive Bayes. *American Association for Artificial Intelligence*, 2004.

[21] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*, 15(5):795–825, 2010.

APPENDIX

A. COMBINING NAIVE BAYES MODELS

Let D be the underlying data set of n points. A point $p \in D$ is represented as $p = (id, y, x)$, where id is the identifier, y the dependent (class) variable and x the feature vector as defined before. Let D_1 and D_2 be two Naive Bayes model with corresponding model descriptors represented by $d(D1) = [a_1, b_1]$ and $d(D2) = [a_2, b_2]$. Our aim is to compute the Naive Bayes model for $D_u = D_1 \cup D_2$. For each materialized models we maintain the following information N_c, S_{jc} and SS_{jc} as discussed in section 3.1.2. Combining the two Naive Bayes model D_1 and D_2 involves considering two cases : *Case 1:* The two datasets do not overlap i.e. $D_1 \cap D_2 = \phi$; which can be easily identified by comparing model descriptors for D_1 and D_2. The model D_u can be materialized using the following equations

$$N_c^u = N_c^1 + N_c^2$$

$$S_{jc}^u = S_{jc}^1 + S_{jc}^2$$

$$SS_{jc}^u = SS_{jc}^1 + SS_{jc}^2$$

Case 2: The two data sets have points in common i.e. $D_1 \cap D_2 \neq \phi$; the points common to both data sets can be determined from the corresponding model descriptors. We can compute the materialized model D_u along same lines as case 2 in section 3.3

$$N_c^u = N_c^1 + N_c^2 - \sum_{i}^{D_1 \cap D_2} [y^{(i)} = c]$$

$$N_c^u = N_c^1 + \sum_{i}^{D_2 - D_1} [y^{(i)} = c]$$

$$N_c^u = N_c^2 + \sum_{i}^{D_1 - D_2} [y^{(i)} = c]$$

S_{jc}^u and SS_{jc}^u can also be computed in a similar fashion.

B. BOUND FOR PSGD

In this section we wish to establish a bound for the model parameter calculated by Algorithm 1 . The bound obtained below holds true for any model with convex and differentiable loss function. Let X denote the feature vector space and Y the output space, and $\Phi : X \times Y \rightarrow H$ a (feature) mapping to a Hilbert space H, which in many practical settings coincides with $\mathbb{R}^N, N = dim(H) < \infty$. The norm induced by the inner product associated with the hilbert space H is represented by $\| \ \|$. Let $S = ((x_1, y_1), ..., (x_m, y_m))$ be a training sample of m tuples in $X \times Y$. A conditional maximum entropy model has a conditional probability of the form $p_w[y|x] = \frac{1}{Z(x)} exp(w\Phi(x, y))$ with $Z(x) = \sum_{yY} exp(w\Phi(x, y))$, where the weight or parameter vector $w \in H$ is the solution of the following optimization problem:

$$w = \underset{w \in H}{\operatorname{argmin}} \ F_S(w) = \underset{w \in H}{\operatorname{argmin}} \ \lambda \|w\|^2 - \frac{1}{m} \sum_{i=1}^{m} log \ p_w[y_i | x_i]$$

where λ is a regularization parameter. Let $z = (x, y) \in X \times Y$ denote a training sample and $L_z(w) = -log p_w[y|x]$ is the negative log-likelihood. Let S and $S^{'}$ be two training sample of size m which differ at one point, $S = (z_1, ..., z_{m1}, z_m)$ and $S^{'} = (z_1, ..., z_{m1}, z_m^{'})$. Let w and $w^{'}$ be the parameter vector obtained after training on sample S and $S^{'}$ respectively. Let Δw be denoted as $w^{'} - w$. We assume that the feature vectors are bounded, $\exists R > 0$ such that $\forall (x, y) \in X \times Y$, $\|\Phi(x, y)\| \leq R$.

Theorem 2. *Let $S^{'}$ and S be two arbitrary samples of size m differing only by one point. Then, the following stability bound holds for the weight vector returned by a conditional maxent model:*

$$\|\Delta\| \leq \frac{2R}{\lambda m}$$

the above theorem can be proved using constructs of Bregman divergence and applying Cauchy-Schwarz inequality. Let D denote the true distribution according to which training and test points are drawn. Let F^* be the associated loss function and $w^* = \underset{w \in H}{\operatorname{argmin}} \ F^*(w)$.

Theorem 3. *Let $w \in H$ be the weight vector returned by conditional maximum entropy when trained on a sample S of size m. Then, for any $\delta > 0$, with probability at least $1 - \delta$, the following inequality holds:*

$$\|w - w^*\| \leq \frac{2R}{\lambda\sqrt{m/2}}(1 + \sqrt{log 1/\delta})$$

The above result is independent of the dimension of the feature space and depends only on R the radius of the sphere containing the feature vectors. Consider now a sample $S = (S_1, ..., S_{pm})$ of pm points formed by p subsamples of size m points drawn i.i.d. and let w_μ denote the parameter derived using Algorithm 1. The following theorem gives a bound for w_μ.

Theorem 4. *For any $\mu \in \Delta_p$, let $w_{mu} \in H$ denote the mixture weight vector obtained from a sample of size pm by combining the p weight vectors w_k, $k[1, p]$, each returned by conditional maximum entropy when trained on the sample S_k of size m. Then, for any $\delta > 0$, with probability at least $1 - \delta$, the following inequality holds:*

$$\|\mathbf{w}_\mu - \mathbf{w}^*\| \leq E[\|\mathbf{w}_\mu - \mathbf{w}^*\|] + \frac{R\|\mu\|}{\lambda\sqrt{m/2}}\sqrt{log 1/\delta}$$

For a uniform mixture the norm is given by $\|\mu\| = \frac{1}{\sqrt{p}}$. We can bound the term $E[\|\mathbf{w}_\mu - \mathbf{w}^*\|]$ by applying the triangle inequality.

$$E[\|w_\mu - w^*\|] = E[\|\frac{1}{p}\sum_{k=1}^{p}(w_k - w^*)\|] \leq E[\|w_1 - w^*\|]$$

Thus using Theorems 2 and 3 we can compare the parameter w_{pm} obtained by training on a sample of size pm versus the mixture of weight parameter w_μ for the same sample.