Materialization Optimizations for Feature Selection Workloads

CE ZHANG, Stanford University
ARUN KUMAR, University of Wisconsin–Madison
CHRISTOPHER RÉ, Stanford University

There is an arms race in the data management industry to support statistical analytics. Feature selection, the process of selecting a feature set that will be used to build a statistical model, is widely regarded as the most critical step of statistical analytics. Thus, we argue that managing the feature selection process is a pressing data management challenge. We study this challenge by describing a feature selection language and a supporting prototype system that builds on top of current industrial R-integration layers. From our interactions with analysts, we learned that feature selection is an interactive human-in-the-loop process, which means that feature selection workloads are rife with reuse opportunities. Thus, we study how to materialize portions of this computation using not only classical database materialization optimizations but also methods that have not previously been used in database optimization, including structural decomposition methods (like QR factorization) and warmstart. These new methods have no analogue in traditional SQL systems, but they may be interesting for array and scientific database applications. On a diverse set of datasets and programs, we find that traditional database-style approaches that ignore these new opportunities are more than two orders of magnitude slower than an optimal plan in this new trade-off space across multiple R backends. Furthermore, we show that it is possible to build a simple cost-based optimizer to automatically select a near-optimal execution plan for feature selection.

Categories and Subject Descriptors: H.2.4 [**Information Systems**]: Database Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Feature selection, statistical analytics, machine learning, materialization, optimization, declarative language, R

ACM Reference Format:
Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization optimizations for feature selection workloads. ACM Trans. Database Syst. 41, 1, Article 2 (February 2016), 32 pages.
DOI: http://dx.doi.org/10.1145/2877204

We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) XDATA program award FA8750-12-2-0335 and DEFT program award FA8750-13-2-0039, DARPA's MEMEX program and SIMPLEX program, the National Science Foundation (NSF) CAREER award IIS-1353606, the Office of Naval Research (ONR) awards N000141210041 and N000141310129, the National Institutes of Health (NIH) grant U54EB020405 awarded by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) through funds provided by the trans-NIH Big Data to Knowledge (BD2K, http://www.bd2k.nih.gov) initiative, the Sloan Research Fellowship, the Moore Foundation, American Family Insurance, Google, and Toshiba. The second author is supported by a grant from the Microsoft Jim Gray Systems Lab. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, AFRL, NSF, ONR, NIH, or the U.S. government, or any of the other sponsors.
Authors' addresses: C. Zhang and C. Ré, Department of Computer Science, Stanford University, 353 Serra Mall, Stanford, CA 94305-9025; emails: {czhang, chrismre}@cs.stanford.edu; A. Kumar, Department of Computer Sciences, 1210 West Dayton Street, Madison, WI-53706; email: arun@cs.wisc.edu.

DOI: http://dx.doi.org/10.1145/2877204

1. INTRODUCTION

One of the most critical stages in the statistical analytics process is *feature selection*; in feature selection, an analyst selects the inputs or features of a model to help improve modeling accuracy or to help an analyst understand and explore his or her data. With the increased interest in data analytics, a pressing challenge is to improve the efficiency of the feature selection process. In this work, we propose COLUMBUS, the first data processing system designed to support the enterprise feature selection process.

To understand the practice of feature selection, we interviewed analysts in enterprise settings. This included an insurance company, a consulting firm, a major database vendor's analytics customer, and a major e-commerce firm. Uniformly, analysts agreed that they spend the bulk of their time on the feature selection process. Confirming the literature on feature selection [Guyon and Elisseeff 2003; Kandel et al. 2012], we found that features are selected (or not) for many reasons: their statistical performance, their real-world explanatory power, legal reasons,[1] or for some combination of reasons. Thus, feature selection is practiced as an interactive process with an analyst in the loop. Analysts use feature selection algorithms, data statistics, and data manipulations as a dialogue that is often specific to their application domain.[2] Nevertheless, the feature selection process has structure: analysts often use domain-specific cookbooks that outline best practices for feature selection from both industry[3] and academia [Guyon and Elisseeff 2003].

Although feature selection cookbooks are widely used, the analyst must still write low-level code, increasingly in R, to perform the subtasks in the cookbook that comprise a feature selection task. In particular, we have observed that such users are forced to write their own custom R libraries to implement simple routine operations in the feature selection literature (e.g., stepwise addition or deletion [Guyon and Elisseeff 2003]). Over the past few years, database vendors have taken notice of this trend, and now virtually every major database engine ships a product with some R extension: Oracle's ORE,[4] IBM's SystemML [Ghoting et al. 2011], SAP HANA,[5] and Revolution Analytics on Hadoop and Teradata. These R-extension layers (RELs) transparently scale operations, such as matrix-vector multiplication or the determinant, to larger sets of data across a variety of backends, including multicore main memory, database engines, and Hadoop. We call these REL operations *ROPs*. Scaling ROPs is actively worked on in industry.

However, we observed that one major source of inefficiency in analysts' code is not addressed by ROP optimization: *missed opportunities for reuse and materialization across ROPs*. Our first contribution is to demonstrate a handful of materialization optimizations that can improve performance by orders of magnitude. Selecting the optimal materialization strategy is difficult for an analyst, as the optimal strategy depends on the reuse opportunities of the feature selection task; the error the analyst is willing to tolerate; and properties of the data and compute node, such as parallelism and data size. Thus, an optimal materialization strategy for an R script for one dataset may not be the optimal strategy for the same task on another dataset. As a result, it is difficult for analysts to pick the correct combination of materialization optimizations.

To study these trade-offs, we introduce COLUMBUS, an R language extension and execution framework designed for feature selection. To use COLUMBUS, a user writes a

[1] Using credit score as a feature is considered a discriminatory practice by the insurance commissions in both California and Massachusetts.

[2] sas.com/reg/wp/corp/23876.

[3] http://www.lexjansen.com/nesug/nesug11/sa/sa08.pdf; http://www.lexjansen.com/nesug/nesug06/an/da23.pdf.

[4] docs.oracle.com/cd/E27988_01/doc.112/e26499.pdf.

[5] help.sap.com/hana/hana_dev_r_emb_en.pdf.

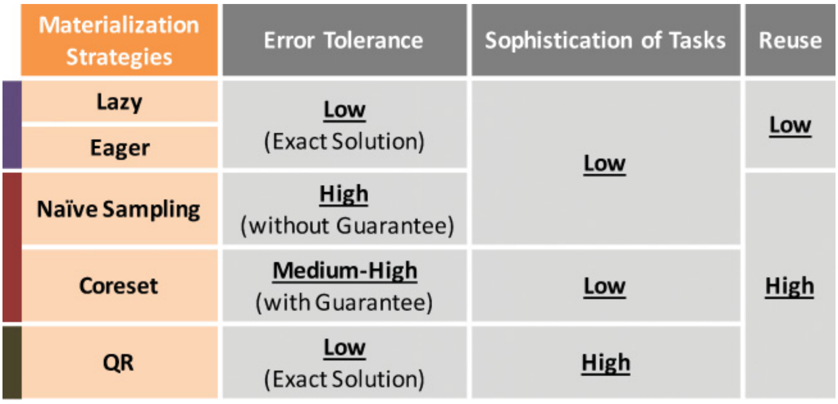
standard R program. COLUMBUS provides a library of several common feature selection operations, such as *stepwise addition*, i.e., *"add each feature to the current feature set and solve."* This library mirrors the most common operations in the feature selection literature [Guyon and Elisseeff 2003] and what we observed in analysts' programs. COLUMBUS's optimizer uses these higher-level declarative constructs to recognize opportunities for data and computation reuse. To describe the optimization techniques that COLUMBUS employs, we introduce the notion of a *basic block*.

A basic block is COLUMBUS's main unit of optimization. A basic block captures a feature selection task for *generalized linear models*, which captures models like linear and logistic regression, support vector machines, lasso, and many more (see Definition 2.1). Roughly, a basic block B consists of a data matrix $A \in \mathbb{R}^{N \times d}$, where N is the number of examples and d is the number of features; a target $b \in \mathbb{R}^{N}$; several feature sets (subsets of the columns of A); and a (convex) loss function. A basic block defines a set of regression problems on the same dataset (with one regression problem for each feature set). COLUMBUS compiles programs into a sequence of basic blocks, which are optimized and then transformed into ROPs. Our focus is not on improving the performance of ROPs but on how to use widely available ROPs to improve the performance of feature selection workloads.

We describe the opportunities for reuse and materialization that COLUMBUS considers in a basic block. As a baseline, we implement classical batching and materialization optimizations. We implement two approaches, namely Lazy and Eager, based on the classical database literature on materialized views (see Section 3.1.1). In addition, we identify three novel classes of optimizations, study the trade-offs each presents, and then describe a cost model that allows COLUMBUS to choose between them. These optimizations are novel in that they have not been considered in traditional SQL-style analytics (but all optimizations have been implemented in other areas).

Subsampling. Analysts employ subsampling to reduce the amount of data the system needs to process to improve runtime or reduce overfitting. These techniques are a natural choice for analytics, as both the underlying data collection process and solution procedures are only reliable up to some tolerance. Popular sampling techniques include naïve random sampling and importance sampling (coresets). Coresets is a relatively recent importance-sampling technique; when $d \ll N$, coresets allow one to create a sample whose size depends on d (the number of features)—as opposed to N (the number of examples)—and that can achieve strong approximation results: essentially, the loss is preserved on the sample for *any* model. In enterprise workloads (as opposed to Web workloads), we found that the overdetermined problems ($d \ll N$), well-studied in classical statistics, are common. Thus, we can use a coreset to optimize the result with a provably small error. However, computing a coreset requires computing importance scores that are more expensive than a naïve random sample. We study the cost-benefit trade-off for sampling-based materialization strategies. Of course, sampling strategies have the ability to improve performance by an order of magnitude. On a real dataset, called *Census* (Section 4.1), we found that d was 1,000× smaller than N, as well as that using a coreset outperforms a baseline approach by 89×, while still getting a solution that is within 1% of the loss of the solution on the entire data set.

Transformation materialization. Linear algebra has a variety of decompositions that are analogous to sophisticated materialized views. One such decomposition, referred to as a (thin) QR decomposition, is widely used to optimize regression problems. Essentially, after some preprocessing, a QR decomposition allows one to solve a class of regression problems in a single scan over the matrix. In feature selection, one has to solve *many* closely related regression problems, e.g., for various subsets of features (columns of A). We show how to adapt QR to this scenario as well. When applicable,



Materialization Strategies	Error Tolerance	Sophistication of Tasks	Reuse
Lazy	Low (Exact Solution)	Low	Low
Eager			
Naïve Sampling	High (without Guarantee)		High
Coreset	Medium-High (with Guarantee)	Low	
QR	Low (Exact Solution)	High	

Fig. 1. Summary of trade-offs in Columbus. Error tolerance, sophistication of tasks, and reuse are defined in Section 3. As an example, "Low" in the Error Tolerance column means the corresponding approach can *support* the workload in which the user has a low tolerance for error (i.e., requires an exact solution).

QR can outperform a baseline by more than 10X; QR can also be applied together with coresets, which can result in 5× more speedup. Of course, there is a cost-benefit trade-off that one must make when materializing QR, and Columbus develops a simple cost model for this choice.

Model caching. Feature selection workloads require that analysts solve many similar problems. Intuitively, it should be possible to reuse these partial results to "warmstart" a model and improve its convergence behavior. We propose to cache several models, and we develop a technique that chooses which model to use for a warmstart. The challenge is to be able to find "nearby" models, and we introduce a simple heuristic for model caching. Compared to the default approach in R (initializing with a random start point or all 0's), our heuristic provides a 13× speedup; compared to a simple strategy that selects a random model in the cache, our heuristic achieves a 6× speedup. Thus, the cache and the heuristic contribute to our improved runtime.

We tease apart the optimization space along three related axes: *error tolerance*, the *sophistication* of the task, and the amount of *reuse* (see Section 3). Figure 1 summarizes the relationship between these axes and the trade-offs. Of course, the correct choice also depends on computational constraints, notably parallelism. We describe a series of experiments to validate this trade-off space and find that no one strategy dominates another. Thus, we develop a cost-based optimizer that attempts to select an optimal combination of the preceding materialization strategies. We validate that our heuristic optimizer has performance within 10% of the optimal optimization strategy (found offline by brute force) on all of our workloads. We establish that many of the subproblems of the optimizer are classically NP-hard, justifying heuristic optimizers.

Contributions. This work makes three contributions: (1) we propose Columbus, which is the first data processing system designed to support the feature selection dialogue; (2) we are the first to identify and study both existing and novel optimizations for feature selection workloads as data management problems; and (3) we use the insights from (2) to develop a novel cost-based optimizer. We validate our results on several real-world programs and datasets patterned after our conversations with analysts. Additionally, we validate Columbus across two backends from main memory and REL for an RDBMS. We argue that these results suggest that feature selection is a promising area for future data management research. Additionally, we are optimistic that the technical optimizations that we pursue apply beyond feature selection to areas like array and scientific databases and tuning machine learning.

Outline. The rest of this article is organized as follows. In Section 2, we provide an overview of the COLUMBUS system. In Section 3, we describe the trade-off space for executing a feature selection program and our cost-based optimizer. We describe experimental results in Section 4. We discuss related work in Section 5 and conclude in Section 6.

The key task of COLUMBUS is to compile and optimize an extension of the R language for feature selection. We compile this language into a set of *REL operations*, which are R-language constructs implemented by today's language extenders, such as ORE and Revolution Analytics. One key design decision in COLUMBUS is *not* to optimize the execution of these REL operators; these have already been studied intensively and are the subjects of major ongoing engineering efforts. Instead, we focus on how to compile our language into the most common of these REL operations (ROPs). Later, Figure 4 shows all ROPs used in COLUMBUS.

A shorter version of this article was published at the ACM SIGMOD 2014 conference [Zhang et al. 2014]. In that paper, we introduced the COLUMBUS system, explained the optimization axes, and outlined the materialization strategies. Compared to that paper, here we provide a more detailed explanation of our cost model, including how we derived the costs. We also explain the materialization strategies in greater detail and provide the exact algorithms using the ROPs. Finally, we significantly expand the discussion of the multiblock optimization case, specifically the Lazy and Eager materializations. We prove that the materialization problem is NP-hard in general but can be solved in polynomial time in two prominent special cases. We provide dynamic programming algorithms for the special cases.

2. SYSTEM OVERVIEW

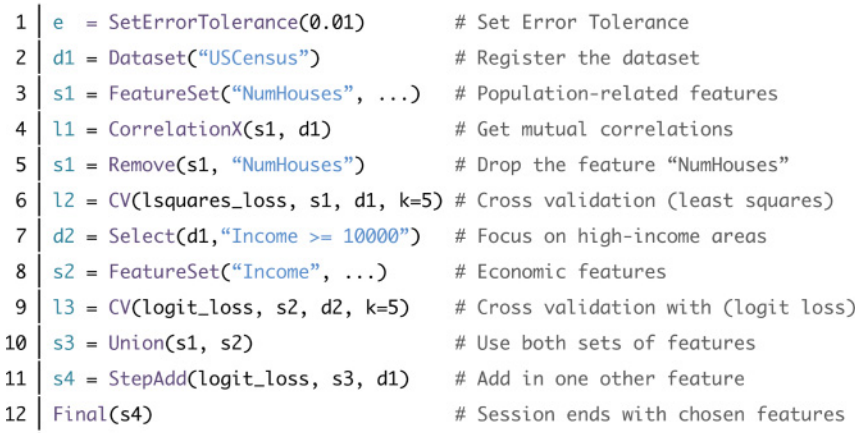
2.1. COLUMBUS Programs

In COLUMBUS, a user expresses his or her feature selection program against a set of high-level constructs that form a domain-specific language for feature selection. We describe these constructs next, and we selected these constructs by talking to a diverse set of analysts and following the state-of-the-art literature in feature selection. COLUMBUS's language is a strict superset of R, so the user still has access to the full power of R.[6] We found that this flexibility was a requirement for most of the analysts surveyed. Figure 2 shows an example snippet of a COLUMBUS program. For example, the 9^{th} line of the program executes logistic regression and reports its score using cross validation.

COLUMBUS has three major datatypes: a *dataset* is a relational table $R(A_1, \ldots, A_d)$,[7] a *feature set* F for a dataset $R(A_1, \ldots, A_d)$ is a subset of the attributes $F \subseteq \{A_1, \ldots, A_d\}$, and a *model* for a feature set is a vector that assigns each feature a real-valued weight. As shown in Figure 3, COLUMBUS supports several operations. We classify these operators based on what types of output an operator produces and order the classes in roughly increasing order of the sophistication of optimization that COLUMBUS is able to perform for such operations (see Figure 3 for examples): (1) *data transformation operations*, which produce new datasets; (2) *evaluate operations*, which evaluate datasets and models; (3) *regression operations*, which produce a model given a feature set; and (4) *explore operations*, which produce new feature sets:

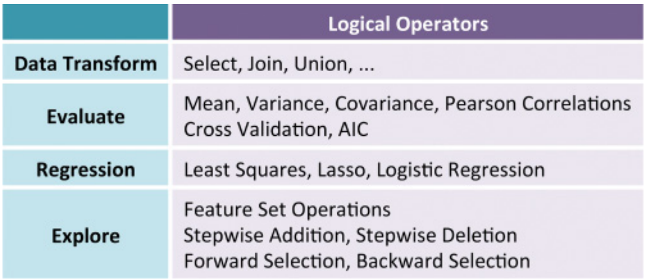
[6] We also have expressed the same language over Python, but for simplicity, we stick to the R model in this article.

[7] Note that the table itself can be a view; this is allowed in COLUMBUS, and the trade-offs for materialization are standard, so we omit the discussion of them in the article.



```
e  = SetErrorTolerance(0.01)        # Set Error Tolerance
d1 = Dataset(“USCensus”)            # Register the dataset
s1 = FeatureSet(“NumHouses”, ...)   # Population-related features
l1 = CorrelationX(s1, d1)           # Get mutual correlations
s1 = Remove(s1, “NumHouses”)        # Drop the feature “NumHouses”
l2 = CV(lsquares_loss, s1, d1, k=5) # Cross validation (least squares)
d2 = Select(d1,“Income >= 10000”)   # Focus on high-income areas
s2 = FeatureSet(“Income”, ...)      # Economic features
l3 = CV(logit_loss, s2, d2, k=5)    # Cross validation with (logit loss)
s3 = Union(s1, s2)                  # Use both sets of features
s4 = StepAdd(logit_loss, s3, d1)    # Add in one other feature
Final(s4)                           # Session ends with chosen features
```

Fig. 2. Example snippet of a COLUMBUS program.



	Logical Operators
Data Transform	Select, Join, Union, ...
Evaluate	Mean, Variance, Covariance, Pearson Correlations Cross Validation, AIC
Regression	Least Squares, Lasso, Logistic Regression
Explore	Feature Set Operations Stepwise Addition, Stepwise Deletion Forward Selection, Backward Selection

Fig. 3. Summary of operators in COLUMBUS.

(1) *Data transform*: These operations are standard data manipulations to slice and dice the dataset. In COLUMBUS, we are aware only of the schema and cardinality of these operations; these operations are executed and optimized directly using a standard RDBMS or main-memory engine. In R, the frames can be interpreted either as a table or an array in the obvious way. We map between these two representations freely.

(2) *Evaluate*: These operations obtain various numeric scores given a feature set, including descriptive scores for the input feature set, e.g., mean, variance, or Pearson correlations, and scores computed after regression, e.g., cross-validation error (e.g., of logistic regression), and the Akaike information criterion (AIC) [Guyon and Elisseeff 2003]. COLUMBUS can optimize these calculations by batching several together.

(3) *Regression*: These operations obtain a model given a feature set and data, e.g., models trained by using logistic regression or linear regression. The result of a regression operation is often used by downstream *explore operations*, which produce a new feature set based on how the previous feature set performs. These operations also take a termination criterion (as they do in R): either the number of iterations or until an error criterion is met. COLUMBUS supports either of these conditions and can perform optimizations based on the type of model (as we discuss).

(4) *Explore*: These operations enable an analyst to traverse the space of feature sets. Typically, these operations result in training many models. For example, a STEPDROP operator takes as input a dataset and a feature set and outputs a new feature set that removes one feature from the input by training a model on each candidate

feature set. Our most sophisticated optimizations leverage the fact that these operations operate on features in *bulk*. The other major operation is StepAdd. Both are used in many workloads and are described in Guyon and Elisseeff [2003].

Columbus is not intended to be comprehensive. However, it does capture the workloads of several analysts that we observed, so we argue that it serves as a reasonable starting point to study feature selection workloads.

2.2. Basic Blocks

In Columbus, we compile a user's program into a directed acyclic dataflow graph with nodes of two types: R functions and an intermediate representation called a *basic block*. The R functions are opaque to Columbus, and the central unit of optimization is the basic block (extensible optimizers [Graefe and McKenna 1993]).

Definition 2.1. A *task* is a tuple $\mathsf{t} = (A, b, \ell, \epsilon, F, R)$, where $A \in \mathbb{R}^{N\times d}$ is a data matrix, $b \in \mathbb{R}^N$ is a label (or target), $\ell : \mathbb{R}^2 \to \mathbb{R}^+$ is a loss function, $\epsilon > 0$ is an error tolerance, $F \subseteq [d]$ is a feature set, and $R \subseteq [N]$ is a subset of rows. A task specifies a regression problem of the form

$$L_{\mathsf{t}}(x) = \sum_{i\in R} \ell(z_i, b_i) \text{ s.t. } z = A\Pi_F x.$$

Here, Π_F is the axis-aligned projection that selects the columns or feature sets specified by F.[8] Denote an optimal solution of the task $x_*(\mathsf{t})$ defined as

$$x_*(\mathsf{t}) = \underset{x\in\mathbb{R}^d}{\operatorname{argmin}}\, L_{\mathsf{t}}(x).$$

Our goal is to find an $x(\mathsf{t})$ that satisfies the error[9]

$$\|L_{\mathsf{t}}(x(\mathsf{t})) - L_{\mathsf{t}}(x_*(\mathsf{t}))\|_2 \le \epsilon.$$

A *basic block*, B, is a set of tasks with common data (A, b) but with possibly different feature sets $\bar{F}$ and subsets of rows $\bar{R}$.

Columbus supports a family of popular nonlinear models, including support vector machines, (sparse and dense) logistic regression, ℓ_p regression, lasso, and elastic net regularization. We give an example to help clarify the definition.

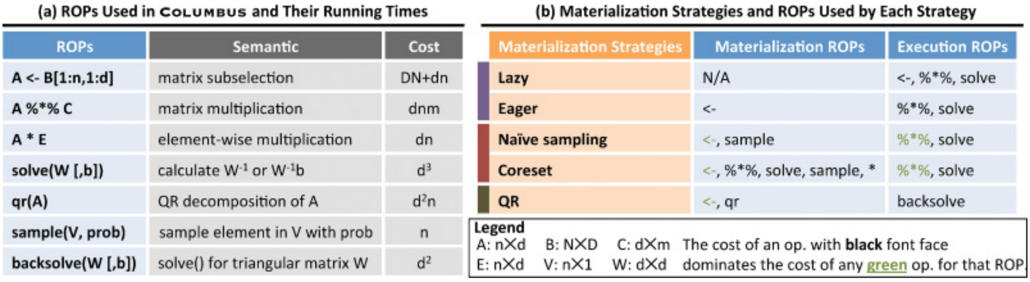
Example 2.2. Consider the sixth line in Figure 2, which specifies a fivefold cross-validation operator with least squares over dataset d_1 and feature set s_1. Columbus will generate a basic block B with five tasks, one for each fold. Let $t_i = (A, b, l, \epsilon, F, R)$. Then, A and b are defined by the dataset d_1 and $l(x, b) = (x - b)^2$. The error tolerance ϵ is given by the user in the first line. The projection of features $F = s_1$ is found by a simple static analysis. Finally, R corresponds to the set of examples that will be used by the i^{th} fold.

The basic block is the unit of Columbus's optimization. Our design choice is to combine several operations on the same data at a high enough level to facilitate bulk optimization, which is our focus in the next section.

Columbus's compilation process creates a task for each regression or classification operator in the program; each of these specifies all of the required information. To

[8] For $F \subseteq [d]$, $\Pi_F \in \mathbb{R}^{d\times d}$, where $(\Pi_F)_{ii} = 1$ if $i \in F$ and all other entries are 0.

[9] We allow termination criteria via a user-defined function or the number of iterations. The latter simplifies reuse calculations in Section 3, whereas arbitrary code is difficult to analyze (we must resort to heuristics to estimate reuse). We present the latter as the termination criterion to simplify the discussion and because it brings out interesting trade-offs.



(a) ROPs Used in COLUMBUS and Their Running Times

ROPs	Semantic	Cost
A <- B[1:n,1:d]	matrix subselection	DN+dn
A %*% C	matrix multiplication	dnm
A * E	element-wise multiplication	dn
solve(W [,b])	calculate W^{-1} or $W^{-1}b$	d^3
qr(A)	QR decomposition of A	d^2n
sample(V, prob)	sample element in V with prob	n
backsolve(W [,b])	solve() for triangular matrix W	d^2

(b) Materialization Strategies and ROPs Used by Each Strategy

Materialization Strategies	Materialization ROPs	Execution ROPs
Lazy	N/A	<-, %*%, solve
Eager	<-	%*%, solve
Naïve sampling	<-, sample	%*%, solve
Coreset	<-, %*%, solve, sample, *	%*%, solve
QR	<-, qr	backsolve

Legend
A: n×d B: N×D C: d×m The cost of an op. with **black** font face
E: n×d V: n×1 W: d×d dominates the cost of any green op. for that ROP.

Fig. 4. (a) ROPs used in COLUMBUS and their costs. (b) Cost model of materializations in COLUMBUS. When data are stored in main memory, the cost for matrix subselection ($a \leftarrow B[1:n, 1:d]$) is only dn. In (b), materialization ROPs are the ROPs used in the materialization phase for an operator, whereas execution ROPs are those used during execution (i.e., while solving the model). The materialization cost and execution cost for each strategy can be estimated by summing the cost of each ROP that they produce in (a).

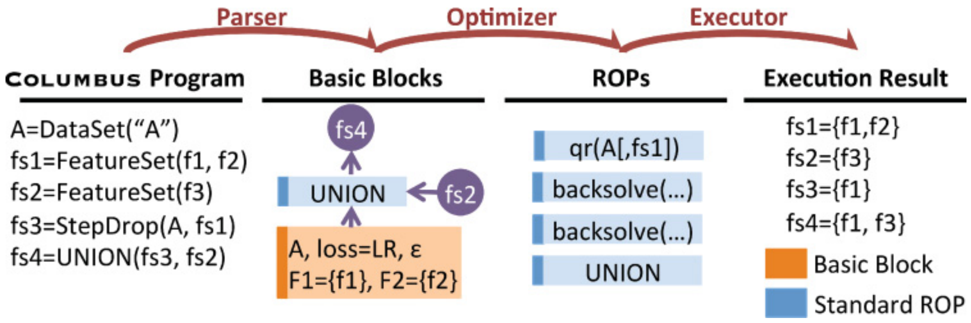


Fig. 5. Architecture of COLUMBUS.

enable arbitrary R code, we allow black-box code in this work flow, which is simply executed. Selecting how to both optimize and construct basic blocks that will execute efficiently is the subject of Section 3.

REL operations. To execute a program, we compile it into a sequence of REL operations (ROPs). These are operators that are provided by the R runtime, e.g., R and ORE. Later, Figure 4 summarizes the host-level operators that COLUMBUS uses, and we observe that these operators are present in both R and ORE. Our focus is on how to optimize the compilation of language operators into ROPs.

2.3. Executing a COLUMBUS Program

To execute a COLUMBUS program, our prototype contains three standard components, as shown in Figure 5: (1) parser, (2) optimizer, and (3) executor. At a high level, these three steps are similar to the existing architecture of any data processing system. The output of the parser can be viewed as a directed acyclic graph, in which the nodes are either basic blocks or standard ROPs, and the edges indicate data flow dependency. The optimizer is responsible for generating a "physical plan." This plan defines which algorithms and materialization strategies are used for each basic block; the relevant decisions are described in Sections 3.1 and 3.2. The optimizer may also merge basic blocks together, called *multiblock optimization*, which is described in Section 3.4. Finally, there is a standard executor that manages the interaction with the REL and issues concurrent requests.

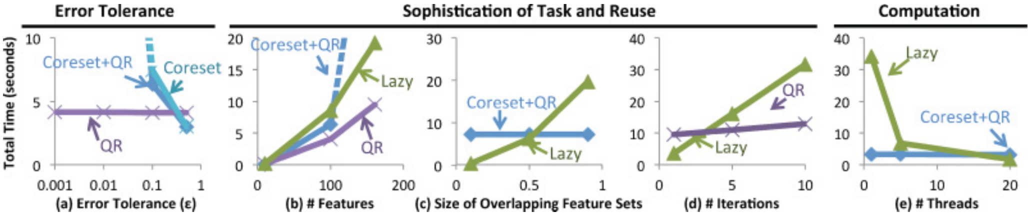


Fig. 6. An illustration of the trade-off space of COLUMBUS, which is defined in Section 3.

3. THE COLUMBUS OPTIMIZER

We begin with optimizations for a basic block that has a least-squares cost, which is the simplest setting in which COLUMBUS's optimizations apply. We then describe how to extend these ideas to basic blocks that contain nonlinear loss functions and then describe a simple technique called *model caching*.

Optimization axes. To help understand the optimization space, we present experimental results on the CENSUS dataset using COLUMBUS programs modeled after our experience with insurance analysts. Figure 6 illustrates the crossover points for each optimization opportunity along three axes that we will refer to throughout this section:[10]

(1) *Error tolerance* depends on the analyst and task. For intuition, we think of different types of error tolerances, with two extremes: *error tolerant* $\epsilon = 0.5$ and *high quality*$\epsilon = 10^{-3}$. In Figure 6, we show $\epsilon \in \{0.001, 0.01, 0.1, 0.5\}$.
(2) *Sophistication* of the feature selection task, namely the loss function (linear or not) and the number of feature sets or rows selected. In Figure 6, we set the number of features as $\{10, 100, 161\}$ and the number of tasks in each block as $\{1, 10, 20, 50\}$. Intuitively, sophistication measures how many regression tasks (e.g., number of folds in cross validation or number of runs inside a StepAdd operator) are in each basic block and how many features each task needs to solve. The reuse opportunities across each task are measured by the Reuse axes as defined later.
(3) *Reuse* is the degree to which we can reuse computation (and that it is helpful to do so). The key factors are the amount of overlap in the feature sets in the workloads[11] and the number of available threads that COLUMBUS uses, which we set here to $\{1, 5, 10, 20\}$.[12]

We discuss these graphs in paragraphs marked *Trade-off* and in Section 3.1.4.

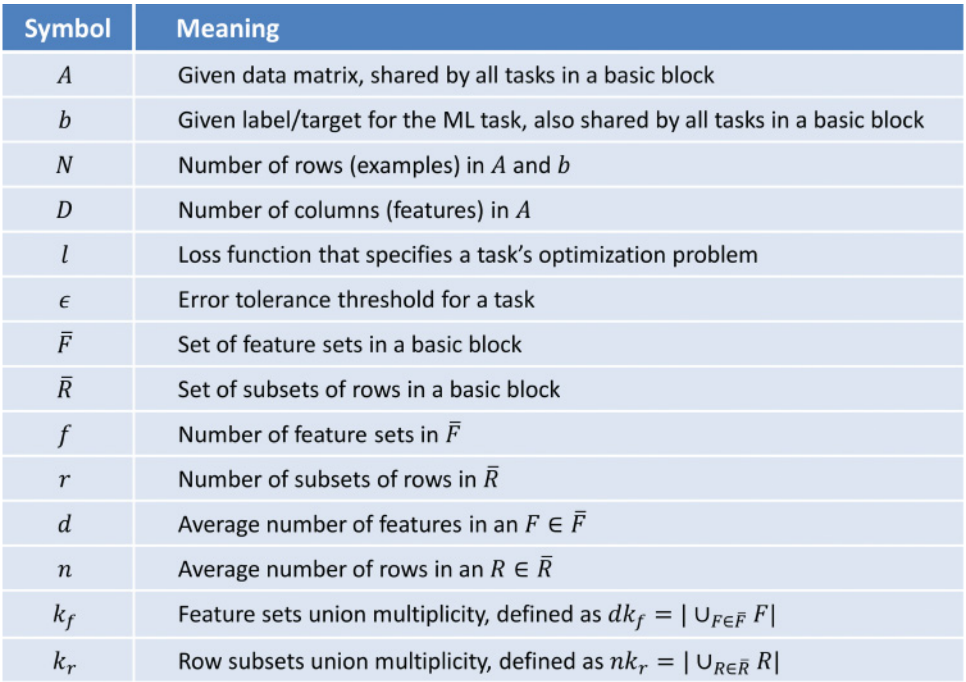
3.1. A Single, Linear Basic Block

We consider three families of optimizations: (1) classical database optimizations, (2) sampling-based optimizations, and (3) transformation-based optimizations. The first optimization is essentially unaware of the feature selection process; in contrast, the other two leverage the fact that we are solving several regression problems. Each of

[10] For each combination of parameters, we execute COLUMBUS and record the total execution time in a main memory R backend. This gives us about 40K data points, and we only summarize the best results in this article. Any omitted data point is dominated by a shown data point.

[11] Let $G = (\cup_{F \in \bar{F}} F, E)$ be a graph, in which each node corresponds to a feature. An edge $(f_1, f_2) \in E$ if there exists $F \in \bar{F}$ such that $f_1, f_2 \in F$. We use the size of the largest connected component in G as a proxy for overlap.

[12] Note that COLUMBUS supports two execution models, namely batch mode and interactive mode.



Symbol	Meaning
A	Given data matrix, shared by all tasks in a basic block
b	Given label/target for the ML task, also shared by all tasks in a basic block
N	Number of rows (examples) in A and b
D	Number of columns (features) in A
l	Loss function that specifies a task's optimization problem
ϵ	Error tolerance threshold for a task
$\bar{F}$	Set of feature sets in a basic block
$\bar{R}$	Set of subsets of rows in a basic block
f	Number of feature sets in $\bar{F}$
r	Number of subsets of rows in $\bar{R}$
d	Average number of features in an $F \in \bar{F}$
n	Average number of rows in an $R \in \bar{R}$
k_f	Feature sets union multiplicity, defined as $dk_f = \lvert \cup_{F\in\bar{F}} F \rvert$
k_r	Row subsets union multiplicity, defined as $nk_r = \lvert \cup_{R\in\bar{R}} R \rvert$

Fig. 7. Notation used throughout Section 3.

these optimizations can be viewed as a form of precomputation (materialization). Thus, we describe the mechanics of each optimization, the cost it incurs in materialization, and its cost at runtime. Figure 4 summarizes the cost of each ROP and the dominant ROP in each optimization. Because each ROP is executed once, one can estimate the cost of each materialization from this figure.[13]

To simplify our presentation, in this section we let $\ell(x, b) = (x - b)^2$, i.e., the least-squares loss, and suppose that all tasks have a single error ε. We return to the more general case in the next section. Our basic block can be simplified to $\mathsf{B} = (A, b, \bar{F}, \bar{R}, \varepsilon)$, for which we compute

$$x(R, F) = \underset{x\in\mathbb{R}^d}{\operatorname{argmin}} \, \|\Pi_R\,(A\Pi_F x - b)\,\|_2^2 \text{ where } R \in \bar{R}, F \in \bar{F}.$$

Our goal is to compile the basic block into a set of ROPs. We explain the optimizations that we identify next. Figure 7 lists the notation used throughout this section.

3.1.1. Classical Database Optimizations. We consider classical Eager and Lazy view materialization schemes. Denote $F^{\cup} = \cup_{F\in\bar{F}} F$ and $R^{\cup} = \cup_{R\in\bar{R}}$ in the basic block. It may happen that A contains more columns than $F^{\cup}$ and more rows than $R^{\cup}$. In this case, one can project away these extra rows and columns, which are analogous to materialized views of queries that contain selections and projections. As a result, classical database materialized view optimizations apply. Specially, Columbus implements two strategies, namely *Lazy* and *Eager*. The *Lazy* strategy will compute these projections at execution time, and *Eager* will compute these projections at materialization time and use them directly at execution time. When data are stored on disk, e.g., as in ORE, *Eager* could

[13] We ran experiments on three different types of machines to validate that the cost we estimated for each operator is close to the actual running time.

ALGORITHM 1: Lazy Materialization Using ROPs

```
Data: Data matrix $A$ and $b$. Set of feature sets $\bar{F}$, set of example sets $\bar{R}$, error tolerance $\epsilon$.
Procedure Materialization($A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$)
    $M \leftarrow$ list() // Materialized information
    return $M$
Procedure Execution($M$, $A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$)
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
        $F = UNION(\tilde{F})$
        $MA \leftarrow A[R, F]$   // $DN + k_f dn$
        $Mb \leftarrow b[R]$   // $N + n$
        $AA \leftarrow MA^T \% * \% MA$   // $k_f^2 d^2 n$
        $bb \leftarrow MA^T \% * \% Mb$   // $k_f dn$
        for $F_i \in F$ do
            $rs \leftarrow$ solve($AA[F_i, F_i], bb[F_i]$) // $2d^3/3$
        end
    end
```

ALGORITHM 2: Eager Materialization Using ROPs

```
Data: Data matrix $A$ and $b$. Set of feature sets $\bar{F}$, set of example sets $\bar{R}$, error tolerance $\epsilon$.
Procedure Materialization($A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$)
    $F \leftarrow UNION(\bar{F})$
    $R \leftarrow UNION(\bar{R})$
    $M =$ list(“$A''$ $= A[R, F]$, “$b''$ $= b[R]$) // $DN + k_r k_f dn + N + k_r n$
    return $M$
Procedure Execution($M$, $A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$)
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
        $F = UNION(\tilde{F})$
        $MA \leftarrow M\$A[R, F]$   // $k_f k_r dn + k_f dn$
        $Mb \leftarrow M\$b[R]$   // $k_r n + n$
        $AA \leftarrow MA^T \% * \% MA$ // $k_f^2 d^2 n$
        $bb \leftarrow MA^T \% * \% Mb$ // $k_f dn$
        for $F_i \in \tilde{F}$ do
            $rs \leftarrow$ solve($AA[F_i, F_i], bb[F_i]$) // $2d^3/3$
        end
    end
```

save I/Os versus *Lazy*. Algorithms 1 and 2 provide the exact procedures for these two materialization approaches using ROPs.

Trade-off. Not surprisingly, *Eager* has a higher materialization cost than *Lazy*, whereas *Lazy* has a slightly higher execution cost than *Eager* as one must subselect the data. Note that if there is ample parallelism (at least as many threads as feature sets), then Lazy dominates. The standard trade-offs apply, and COLUMBUS selects between these two techniques in a cost-based way. If there are disjoint feature sets $F_1 \cap F_2 = \emptyset$, then it may be more efficient to materialize these two views separately. Later, in

ALGORITHM 3: Naïve Sampling Using ROPs

```
Data: Data matrix $A$ and $b$. Set of feature sets $\bar{F}$, set of example sets $\bar{R}$, error tolerance $\epsilon$,
      selectivity of sampling $\rho$.
Procedure Materialization($A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$, $\rho$)
    $M \leftarrow$ list()
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $sampled \leftarrow$ sample$(1 : |R|, \rho|R|)$   // $n$
        $MA = A[sampled, ]$       // $DN + \rho Dn$
        $Mb = b[sampled]$       // $N + \rho n$
        $M[R] =$ list$(“A'' = MA, “b'' = Mb)$
    end
    return $M$
Procedure Execution($M$, $A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$)
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
        $F = UNION(\tilde{F})$
        $MA \leftarrow M[R]\$A[, F]$   // $\rho Dn + \rho k_f dn$
        $Mb \leftarrow M[R]\$b$
        $AA \leftarrow MA^T \% * \% MA$   // $\rho k_f^2 d^2 n$
        $bb \leftarrow MA^T \% * \% Mb$   // $\rho k_f dn$
        for $F_i \in F$ do
            $rs \leftarrow$ solve$(AA[F_i, F_i], bb[F_i])$ // $2d^3/3$
        end
    end
```

Section 3.4, we show that the general problem of selecting an optimal way to split a basic block to minimize cost is essentially a weighted set cover, which is NP-hard. As a result, we use a simple heuristic: split disjoint feature sets. With a feature selection workload, we may know the number of times a particular view will be reused, which Columbus can use to more intelligently chose between Lazy and Eager (rather than not having this information). These methods are insensitive to error and the underlying loss function, which will be major concerns for our remaining feature-selection-aware methods.

3.1.2. Sampling-Based Optimizations. Subsampling is a popular method to cope with large data and long runtimes. This optimization saves time simply because one is operating on a smaller dataset. This optimization can be modeled by adding a subset selection ($R \in \bar{R}$) to a basic block. In this section, we describe two popular methods: *naïve random sampling* and a more sophisticated importance-sampling method called *coresets* [Boutsidis et al. 2013; Langberg and Schulman 2010]; we describe the trade-offs that these methods provide.

Naïve sampling. Naïve random sampling is widely used. In fact, analysts ask for it by name. In naïve random sampling, one selects some fraction of the dataset. Recall that A has N rows and d columns; in naïve sampling, one selects some fraction of the N rows (say 10%). The cost model for both materialization and its savings of random sampling is straightforward, as one performs the same solve—only on a smaller matrix. We perform this sampling using the ROP sample. Algorithm 3 provides the exact procedure using ROPs.

Coresets. A recent technique called *coresets* allows one to sample from an overdetermined system $A \in \mathbb{R}^{N \times d}$ with a sample size that is proportional to d and independent of N with much stronger guarantees than naïve sampling. In some enterprise settings, d (the number of features) is often small (say 40), but the number of data points is much higher, say millions, and coresets can be a large savings. We give one such result in the following proposition.

PROPOSITION 3.1 ([BOUTSIDIS ET AL. 2013]). *For $A \in \mathbb{R}^{N\times d}$ and $b \in \mathbb{R}^N$. Define $s(i) = a_i^T (A^T A)^{-1} a_i$. Let $\tilde{A}$ and $\tilde{b}$ be the result of sampling m rows, where row i is selected with probability proportional to $s(i)$. Then, for all $x \in \mathbb{R}^d$, we have the following,*

$$\Pr\left[\left| \|Ax - b\|_2^2 - \frac{N}{m}\|\tilde{A}x - \tilde{b}\|_2^2 \right| < \varepsilon \|Ax - b\|_2^2\right] > \frac{1}{2},$$

as long as $m > 2\varepsilon^{-2} d \log d$.

This guarantee is strong.[14] To understand how strong, observe that naïve sampling cannot meet this type of guarantee without sampling proportional to N. To see this, consider the case in which a feature occurs in only a single example, e.g., $A_{1,1} = 1$ and $A_{i,1} = 0$ for $i = 2, \ldots, N$. It is not hard to see that the only way to achieve a similar guarantee is to make sure the first row is selected. Hence, naïve sampling will need roughly N samples to guarantee this is selected. Note that Proposition 3.1 does not use the value b. COLUMBUS uses this fact later to optimize basic blocks with changing right-hand sides. Algorithm 4 provides the exact procedure using ROPs.

Trade-off. Looking at Figure 6, one can see a clear trade-off space: coresets require two passes over the data to compute the sensitivity scores. However, the smaller sample size (proportional to d) can be a large savings when the error ϵ is moderately large. But as either d or ϵ^{-1} grows, coresets become less effective, and there is a crossover point. In fact, coresets are useless when $N = d$. One benefit of coresets is that they have explicit error guarantees in terms of ϵ. Thus, our current implementation of COLUMBUS will not automatically apply *naïve sampling* unless the user requests it. With optimal settings for sample size, naïve sampling can get better performance than coresets. Nevertheless, recent analysis has shown that one could leverage randomized linear algebra techniques to construct a new importance-sampling method that can be used as a preconditioner for SGD to obtain the best of both worlds—good performance as well as theoretical guarantees on convergence [Yang et al. 2016]. We leave it to future work to integrate this new performance–accuracy trade-off into COLUMBUS.

3.1.3. Transformation-Based Optimizations. In linear algebra, there are decomposition methods to solve (repeated) least squares efficiently; the most popular of these is called the *QR decomposition* [Golub 1965]. At a high level, we use the QR decomposition to transform the data matrix A so that *many* least-squares problems can be solved efficiently. Typically, one uses a QR to solve for many different values of b (e.g., in Kalman filter updates). However, in feature selection, we use a different property of the QR: that it can be used across many *different feature sets*. We define the QR factorization (technically the thin QR), describe how COLUMBUS uses it, and then describe its cost model.

Definition 3.2 (*Thin QR Factorization*). The QR decomposition of a matrix $A \in R^{N\times d}$ is a pair of matrices (Q, R), where $Q \in \mathbb{R}^{N\times d}$, $R \in \mathbb{R}^{d\times d}$, and A = QR. Q is an orthogonal matrix, i.e., $Q^T Q = I$ and R is upper triangular.

[14] Note that one can use $\log_2 \delta^{-1}$ independent trials to boost the probability of success to $1 - \delta$.

ALGORITHM 4: CoreSet Using ROPs (Assume $\forall R_i, R_j \in \bar{R}, R_i \cap R_j = \emptyset$)

```
Data: Data matrix $A$ and $b$. Set of feature sets $\bar{F}$, set of example sets $\bar{R}$, error tolerance $\epsilon$.
Procedure Materialization($A, b, \bar{F}, \bar{R}, \epsilon$)
    $M = \textbf{list}()$
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
        $F = UNION(\tilde{F})$
        $MA \leftarrow A[R, F]$   // $DN + k_f dn$
        $Mb \leftarrow b[R]$   // $N + n$
        $N \leftarrow \textbf{NROW}(MA)$
        $d \leftarrow \textbf{NCOL}(MA)$
        $AA \leftarrow t(MA)\% * \%MA$   //$k_f^2 d^2 n$
        $inv \leftarrow \textbf{solve}(AA)$   // $2k_f^3 d^3/3$
        $sensitivity \leftarrow \textbf{rowSums}((MA\ \% * \%\ inv) * MA)$   //$k_f^2 d^2 n + 2k_f dn$
        $sampled \leftarrow \textbf{sample}(1 : N, 2d/\epsilon^2, prob = sensitivity)$   //$n$
        $MA \leftarrow MA[sampled, ]$   // $k_f dn + 2k_f d^2/\epsilon^2$
        $Mb \leftarrow Mb[sampled]$   // $n + 2d/\epsilon^2$
        M[(R, F)] = list("A"=MA, "b"=Mb)
    end
    return $M$
Procedure Execution($M, A, b, \bar{F}, \bar{R}, \epsilon$)
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
        $F = UNION(\tilde{F})$
        $MA \leftarrow M[(R, F)]$
        $Mb \leftarrow b[R]$
        $AA \leftarrow MA^T \% * \%MA$   // $2k_f^2 d^3/\epsilon^2$
        $bb \leftarrow MA^T \% * \%Mb$   // $2k_f d^2/\epsilon^2$
        for $F_i \in F$ do
            $rs \leftarrow \textbf{solve}(AA[F_i, F_i], bb[F_i])$   // $d^3$
        end
    end
```

We observe that since $Q^{-1} = Q^T$ and R is upper triangular, one can solve $Ax = b$ by setting $QRx = b$ and multiplying through by the transpose of Q so that $Rx = Q^T b$. Since R is upper triangular, one can solve this equation with back substitution; back substitution does not require computing the inverse of R, and its running time is linear in the number of entries of R, i.e., $O(d^2)$.

Columbus leverages a simple property of the QR factorization: upper triangular matrices are closed under multiplication, i.e., if U is upper triangular, then so is RU. Since Π_F is upper triangular, we can compute *many* QR factorizations by simply reading off the inverse of $R\Pi_F$.[15] This simple observation is critical for feature selection. Thus, if there are several different row selectors, Columbus creates a separate QR factorization for each. Algorithm 5 provides the exact procedure using ROPs.

Trade-off. As summarized in Figure 4, QR's materialization cost is similar to importance sampling. In terms of execution time, Figure 6 shows that QR can be much faster

[15] Notice that $\Pi_R Q$ is not necessarily orthogonal, so $\Pi_R Q$ may be expensive to invert.

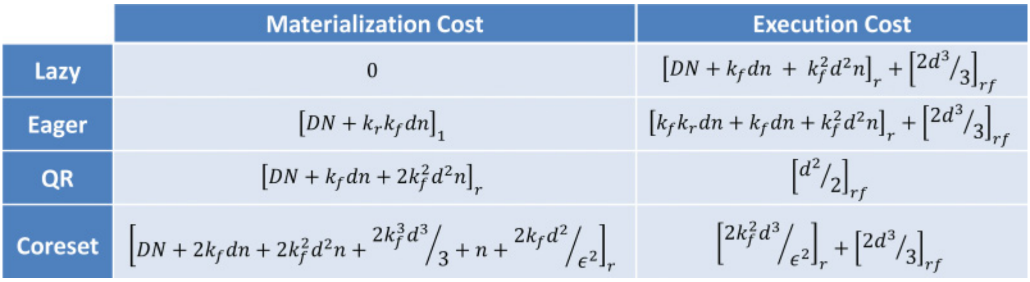
ALGORITHM 5: QR Decomposition Using ROPs

```
Data: Data matrix $A$ and $b$. Set of feature sets $\bar{F}$, set of example sets $\bar{R}$, error tolerance $\epsilon$.
Procedure Materialization($A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$)
    $M = \textbf{list}()$
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
        $F = UNION(\tilde{F})$
        $MA \leftarrow A[R, F]$   // $DN + k_f dn$
        $Mb \leftarrow b[R]$   // $N + n$
        $qrrs = \textbf{qr}(MA)$
        $q \leftarrow \textbf{qr}.\textbf{Q}(qrrs)$
        $r \leftarrow \textbf{qr}.\textbf{R}(qrrs)$   // $2k_f^2 d^2 n$
        $Mb \leftarrow q^T \% * \% Mb$   // $k_f dn$
        M[(R, F)] = list(“r”=r, “b”=Mb)
    end
    return $M$
Procedure Execution($M$, $A$, $b$, $\bar{F}$, $\bar{R}$, $\epsilon$)
    $\tilde{R} = \{R_i \in \bar{R}\}$
    for $R \in \tilde{R}$ do
        $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
        $F = UNION(\tilde{F})$
        $r \leftarrow M[(R, F)]\$r$
        $Mb \leftarrow Mb[(R, F)]\$b$
        for $F_i \in \tilde{F}$ do
            $rs \leftarrow \textbf{backsolve}(r[, F_i], Mb)$   // $d^2/2$
        end
    end
```

than coresets: solving the linear system is quadratic in the number of features for QR but cubic for coresets (without QR). When there are a large number of feature sets and they overlap, QR can be a substantial win (this is precisely the case when coresets are ineffective). These techniques can also be combined, which further modifies the optimal trade-off point. An additional point is that QR does not introduce error (and is often used to improve numerical stability), which means that QR is applicable in error tolerance regimes when sampling methods cannot be used.

3.1.4. Discussion of Trade-Off Space. Figure 6 shows the crossover points for the trade-offs that we described in this section for the Census dataset. We describe why we assert that each of the following aspects affects the trade-off space.

Error For error-tolerant computation, naïve random sampling provides dramatic performance improvements. However, when low error is required, then one must use classical database optimizations or the QR optimization. In between, there are many combinations of QR, coresets, and sampling that can be optimal. As we can see in Figure 6(a), when the error tolerance is small, coresets are significantly slower than QR. When the tolerance is 0.01, the coreset that we need is even larger than the original dataset, and if we force Columbus to run on this large coreset, it would be more than 12× slower than QR. For tolerance 0.1, coreset is 1.82× slower than QR. We look into the breakdown of materialization time and execution time, and we find that materialization time contributes to more than 1.8× of this difference. When error tolerance is 0.5, Coreset+QR is 1.4× faster than QR. We ignore the curve for Lazy and Eager because they are insensitive to noises and are more than 1.2× slower than QR.



	Materialization Cost	Execution Cost
Lazy	0	$[DN + k_f dn + k_f^2 d^2 n]_r + [2d^3/3]_{rf}$
Eager	$[DN + k_r k_f dn]_1$	$[k_f k_r dn + k_f dn + k_f^2 d^2 n]_r + [2d^3/3]_{rf}$
QR	$[DN + k_f dn + 2k_f^2 d^2 n]_r$	$[d^2/2]_{rf}$
Coreset	$[DN + 2k_f dn + 2k_f^2 d^2 n + 2k_f^3 d^3/3 + n + 2k_f d^2/\epsilon^2]_r$	$[2k_f^2 d^3/\epsilon^2]_r + [2d^3/3]_{rf}$

Fig. 8. Cost model. We list the costs of different materialization strategies. We use $[a]_b$ to denote that b runs with the cost a can be executed in parallel. Given a basic block $B = (A, b, \bar{F}, \bar{R})$, N and D are the number of rows and columns in A, and r and f are the distinct number of elements in $\bar{R}$ and $\bar{F}$, respectively; we assume that d and n are the number of elements of $F \in \bar{F}$ and $R \in \bar{R}$ and $|\cup_i F_i| = k_f d$ and $|\cup_i R_i| = k_r n$.

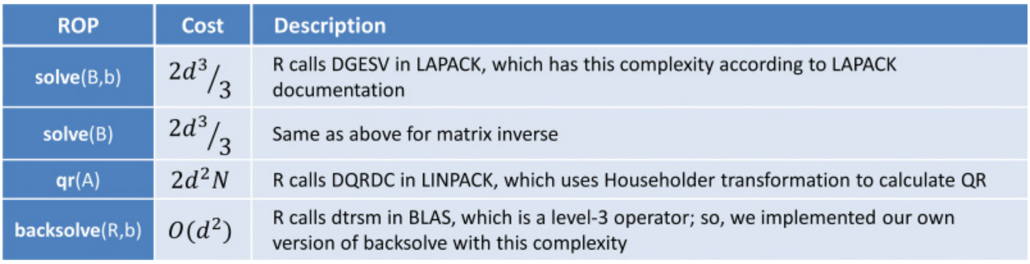
Sophistication. One measure of sophistication is the number of features that the analyst is considering. When the number of features in a basic block is much smaller than the dataset size, coresets create much smaller but essentially equivalent datasets. As the number of features, d, increases, or the error decreases, coresets become less effective. On the other hand, optimizations like QR become more effective in this regime: although materialization for QR is quadratic in d, it reduces the cost to compute an inverse from roughly d^3 to d^2.

As shown in Figure 6(b), as the number of features grows, CoreSet+QR slows down. With 161 features, the coreset will be larger than the original dataset. However, when the number of features is small, the gap between CoreSet+QR and QR will be smaller. When the number of features is 10, CoreSet+QR is 1.7× faster than QR. When the number of feature is small, the time it takes to run a QR decomposition over the coreset could be smaller than over the original dataset, hence the 1.7× speedup of CoreSet+QR over QR.

Reuse. In linear models, the amount of overlap in the feature sets correlates with the amount of reuse. We randomly select features but vary the size of overlapping feature sets. Figure 6(c) shows the result. When the size of the overlapping feature sets is small, Lazy is 15× faster than CoreSet+QR. This is because CoreSet wastes time in materializing for a large feature set. Instead, Lazy will solve these problems independently. On the other hand, when the overlap is large, CoreSet+QR is 2.5× faster than Lazy. Here, CoreSet+QR is able to amortize the materialization cost by reusing it on different models.

Available parallelism. If there is a large amount of parallelism and one needs to scan the data only once, then a Lazy materialization strategy is optimal. However, in feature selection workloads where one is considering hundreds of models or repeatedly iterating over data, parallelism may be limited and mechanisms that reuse the computation may be optimal. As shown by Figure 6(e), when the number of threads is large, Lazy is 1.9× faster than CoreSet+QR. The reason is that although the reuse between models is high, all of these models could be run in parallel in Lazy. Thus, although CoreSet+QR does save computation, it does not improve the wall-clock time. On the other hand, when the number of threads is small, CoreSet+QR is 11× faster than Lazy.

3.1.5. Cost Model. The preceding performance trade-offs between materialization and execution can be captured using a simple cost model that considers the cost for each ROP in each strategy. Figure 8 presents the detailed costs of materialization and execution for one basic block. The cost of each ROP either comes from the manual of R



ROP	Cost	Description
solve(B,b)	$2d^3/3$	R calls DGESV in LAPACK, which has this complexity according to LAPACK documentation
solve(B)	$2d^3/3$	Same as above for matrix inverse
qr(A)	$2d^2N$	R calls DQRDC in LINPACK, which uses Householder transformation to calculate QR
backsolve(R,b)	$O(d^2)$	R calls dtrsm in BLAS, which is a level-3 operator; so, we implemented our own version of backsolve with this complexity

Fig. 9. Costs of the major ROPs used to obtain the cost model for the various materialization strategies. We also describe how we obtained these costs. The complexity of *solve* and *qr* are obtained from LAPACK and LINPACK, respectively.[16]

or the corresponding BLAS function that R invokes. We list the costs of the ROPs (the nonobvious ones) in Figure 9.

3.2. A Single, Nonlinear Basic Block

We extend our methods to nonlinear loss functions. The same trade-offs from the previous section apply, but there are two additional techniques that we can use. We describe them next.

Recall that a task solves the problem

$$\min_{x \in R^d} \sum_{i=1}^{N} \ell(z_i, b_i) \text{ subject to } z = Ax,$$

where $\ell : \mathbb{R}^2 \to \mathbb{R}^+$ is a convex function.

Iterative methods. We select two methods: first, stochastic gradient descent (SGD) [Bertsekas 1999; Shalev-Shwartz and Srebro 2008; Bottou and Bousquet 2007], and second, iterative reweighted least squares (IRLS), which is implemented in R's generalized linear model package.[17] We describe an optimization—warmstarting—that applies to such models as well as to the alternating direction method of multipliers (ADMM).

ADMM. There is a classical general-purpose method that allows one to *decompose* such a problem into a least-squares problem and a second simple problem. The method that we explore—ADMM-is one of the most popular [Boyd et al. 2011], which has been widely used since the 1970s. We explain the details of this method to highlight a key property that allows us to reuse the optimizations from the previous section.

ADMM is iterative and defines a sequence of triples (x^k, z^k, u^k) for $k = 0, 1, 2, \ldots$. It starts by randomly initializing the three variables (x^0, z^0, u^0), which are then updated by the following equations:

$$\begin{aligned} x^{(k+1)} &= \operatorname*{argmin}_x \frac{\rho}{2} ||Ax - z^{(k)} + u^{(k)}||_2^2 \\ z^{(k+1)} &= \operatorname*{argmin}_z \sum_{i=1}^{N} l(z_i, b_i) + \frac{\rho}{2} ||Ax^{(k+1)} - z + u^{(k)}||_2^2 \\ u^{(k+1)} &= u^{(k)} + Ax^{(k+1)} - z^{(k+1).} \end{aligned}$$

[16] http://www.netlib.org/lapack/lug/node71.html; http://tel.archives-ouvertes.fr/docs/00/83/33/56/PDF/VD2_KHABOU_AMAL_11022013.pdf.

[17] stat.ethz.ch/R-manual/R-patched/library/stats/html/glm.html.

The constant $\rho \in (0, 2)$ is a step-size parameter that we set by a grid search over five values.

There are two key properties of ADMM equations that are critical for feature selection applications:

(1) *Repeated least squares*: The solve for $x^{(k+1)}$ is a linear basic block from the previous section since z and u are fixed and the A matrix is *unchanged* across iterations. In nonlinear basic blocks, we solve multiple feature sets concurrently, so we can reuse the transformation optimizations of the previous section for each such update. To take advantage of this, Columbus logically rewrites ADMM into a sequence of linear basic blocks with custom R functions.

(2) *One-dimensional z*: We can rewrite the update for z into a series of independent, one-dimensional problems—that is,

$$z_i^{(k+1)} = \operatorname*{argmin}_{z_i} l(z_i, b_i) + \frac{\rho}{2}(q_i - z_i)^2, \text{ where } q = Ax^{(k+1)} + u^{(k).}$$

This one-dimensional minimization can be solved by fast methods, such as bisection or Newton. To update $x^{(k+1)}$, the bottleneck is the ROP "solve," whose cost is in Figure 4. The cost of updating z and u is linear in the number of rows in A and can be decomposed into N problems that may be solved independently.

Trade-offs. In Columbus, ADMM is our default solver for nonlinear basic blocks. Empirically, on all of our applications in our experiments, if one first materializes the QR computation for the least-squares subproblem, then we find that ADMM converges faster than SGD to the same loss. Moreover, there is sharing *across* feature sets that can be leveraged by Columbus in ADMM (using our earlier optimization about QR). One more advanced case for reuse is when we must fit hyperparameters, like ρ above or regularization parameters; in this case, ADMM enables opportunities for high degrees of sharing.

3.3. Warmstarting by Model Caching

In feature selection workloads, our goal is to solve a model after having solved many similar models. For iterative methods like gradient descent or ADMM, we should be able to partially reuse these similar models. We identify three situations in which such reuse occurs in feature selection workloads. First, we downsample the data, learn a model on the sample, and then train a model on the original data. Second, we perform stepwise removal of a feature in feature selection, and the "parent" model with all features is already trained. Third, we examine several nearby feature sets interactively. In each case, we should be able to reuse the previous models, but it would be difficult for an analyst to implement effectively in all but the simplest cases. In contrast, Columbus can use warmstart to achieve up to 13× performance improvement for iterative methods without user intervention.

Given a cache of models, we need to choose a model. We observe that computing the loss of each model on the cache on a sample of the data is inexpensive. Thus, we select the model with the lowest sampled loss. To choose models to evict, we simply use an LRU strategy. In our workloads, the cache does not become full, so we do not discuss it. However, if one imagines several analysts running workloads on similar data, the cache could become a source of challenges and optimizations.

3.4. Multiblock Optimization

There are two tasks that we need to do across blocks: (1) decide on how coarse or fine to make a basic block and (2) execute the sequence of basic blocks across the backend.

ALGORITHM 6: Dynamic Program to Solve MatOpt for a Chain of Feature Sets $\bar{F}$

```
$T(i, 0) = r(F_i)\,\text{Read}(F) + T(i-1, 0)$ //$i > 0$, use original dataset
$T(i, j) = r(F_i)\,\text{Read}(F_j) + T(i-1, j)$ //$i > j > 0$, use the last materialization
$T(i, i) = \min_{j=1\ldots i-1}(\text{Store}(F_j, F_i) + T(i-1, j)) + r(F_i)\,\text{Read}(F_i)$//$i > 0$, mat. costs on diagonal
```

Multiblock logical optimization. Given a sequence of basic blocks from the parser, Columbus must first decide how coarse or fine to create individual blocks. Cross validation is, e.g., merged into a single basic block. In Columbus, we greedily improve the cost using the obvious estimates from Figure 4. The problem of deciding the optimal partitioning of many feature sets is NP-hard in general. The intuition is clear, as one must cover all of the different features with as few basic blocks as possible. However, the heuristic merging can have large wins, as operations like cross validation and grid searching parameters allow one to find opportunities for reuse. We now formalize the problem of materializing different subsets of feature sets (Eager vs. Lazy) and prove its hardness. Note that the different feature sets can be from the same basic block or from different basic blocks that share the data and other inputs.

Problem statement. Fix dataset (A, b) with a full set of features F. We are given a set of accesses of subsets of F, labeled $\bar{F} = \{F_1, \ldots, F_N\}$, wherein $F_i \subseteq F, \forall i = 1 \ldots N$. The subsets are accessed with *repetitions* $\{r(F_i)\}_{i=1}^{N}$, i.e., F_j is accessed $r(F_j)$ times. A *materialization plan* is a subset $\bar{F}' \subseteq \bar{F}$ whose feature sets are materialized and used for data accesses. We want to obtain a materialization plan with minimum cost. We formulate it as an optimization problem:

$$\text{MatOpt: } \min_{\bar{\mathcal{F}}' \subseteq \bar{F}} \text{Cost}(\bar{\mathcal{F}}'),$$
$$\text{where Cost}(\bar{\mathcal{F}}') = \sum_{F \in \bar{\mathcal{F}}'} \min_{F' \in \bar{\mathcal{F}}' \cup \{F\} \setminus \{F\} : F' \supseteq F} \text{Store}(F', F)$$
$$+ \sum_{F \in \bar{F}} r(F) \min_{F' \in \bar{\mathcal{F}}' \cup \{F\} : F' \supseteq F} \text{Read}(F').$$

In other words, we materialize the feature sets in $\bar{\mathcal{F}}'$ and use the "thinnest" among them (or the whole dataset) for each feature set. In the objective function of MatOpt, we use generic costs (considering both I/O and CPU) for reading and writing data with a feature subset materialized. Essentially, *Read*(F') is the cost of reading the dataset with features $F' \subseteq F$. Additionally, *Store*(F', F) is the cost of materializing (writing) a dataset with features F' by reading the data set with features $F \supseteq F'$. A more complex formulation could consider reading a smaller data set instead. We first consider two restrictions on $\bar{F}$ based on real usage that we observed—first, when $\bar{F}$ is a *chain*, i.e., there is a containment hierarchy among the feature sets, and second, when the width of the lattice of $\bar{F}$ is fixed. We provide polynomial-time algorithms for these two special cases. After that, we consider the general set of sets and prove that it is NP-hard.

Theorem 3.3. *For $\bar{F}$ that is a chain,* MatOpt *can be solved in $O(N^2)$ time and $O(N^2)$ space by Algorithm 6.*

Proof. Algorithm 6 presents a dynamic programming-based solution for MatOpt on chain inputs. With N input feature sets, the dynamic program fills the lower triangular part of the $N \times N$ memo (excluding diagonal elements) in time $\Theta(N(N-1)/2)$. Filling the diagonal elements requires referring to each row above (up to the diagonal element). This is in time $\Theta(N(N-1)/2 + N)$. Reading the backpointers is in time $\Theta(N)$. Thus, the overall algorithm is in time $O(N^2)$. The memo of time costs is in space $\Theta(N(N-1)/2+N)$, whereas the backpointers are in space $\Theta(N)$. Thus, the overall space complexity is also $O(N^2)$. □

ALGORITHM 7: Dynamic Program to Solve MatOpt for a set of Features Sets $\bar{F}$ That Exist in a k-Width Lattice

```
$//i \geq 0:$
$T(i+1, \mathbf{0}) = T(i, \mathbf{0}) + \sum_{j=1\dots k} r(F(i+1, j))\,\mathrm{Read}(\mathcal{F})$
$//i \geq 0; \mathbf{v}\ s.t.\ \forall j = 1\dots k, v_j \leq i:$
$T(i+1, \mathbf{v}) = T(i, \mathbf{v}) + [\sum_{j=1\dots k} r(F(i+1, j))] \min_{l=1\dots k} \mathrm{Read}(F(v_l, l))$
$//i \geq 1; \mathbf{v}\ s.t.\ \forall j = 1\dots k, v_j \leq i,\ and\ \exists j = 1\dots k, s.t.\ v_j = i:$
$T(i, \mathbf{v}) = \sum_{j:v_j=i}(r(F(i, j))\,\mathrm{Read}(F(i, j))) + [\sum_{j:v_j<i} r(F(i, j))] \min_{l=1\dots k} \mathrm{Read}(F(v_l, l)) +$
        $\min_{\mathbf{w}:w_j<i, \forall j:v_j=i \wedge w_j=v_j, \forall j:v_j<i} [T(i-1, \mathbf{w}) + \sum_{j:v_j=i} \min_{l=1\dots k} \mathrm{Store}(F(i, j), F(w_l, l))]$
```

Next, we analyze MatOpt for the input feature sets existing in a bounded-width lattice. The feature sets in $\bar{F}$ have a lattice ordering $L : (\bar{F}, \subseteq)$. Let the height of L be N (i.e., we have N *levels* $L_1 \dots L_N$). The width of L is a given constant k (i.e., k sets at each level). Sets in $L_1 \dots L_{N-1}$ have k children each, and sets in $L_2 \dots L_N$ have k parents each. In this case, our dynamic program's memo will become richer—its has k entries per slot, corresponding to the width of the lattice. In this scenario, we can still compute an optimum in time polynomial in N (but exponential in the constant k).

We introduce a convenient "grid" notation for the feature sets in $\bar{F}$ that we will use in the rest of this section. We write $\bar{F} = \{F(i, j)\}, i \in \{1, \dots, N\}$ and $j \in \{1, \dots, k\}$. A level is thus a set of feature sets with the same i, whereas we call a set of feature sets with the same j as a "column." Our memo is changed to $T(i, \mathbf{v})$, which is the cumulative cost up to level i, and $\mathbf{v} = (v_1, \dots, v_k)$ is a k-length vector of indices, each of which refers to the level at which the last materialization was done in its respective column ($0 \leq v_j \leq N, \forall j = 1 \dots k$). We define $T(i, \mathbf{v})$ recursively (considering only scan and materialization costs) as given in Algorithm 7.

Theorem 3.4. *For $\bar{F}$ that is a k-width lattice,* MatOpt *can be solved in $O(k^2 2^k N^{k+1})$ time with $O(N^{k+1})$ space by Algorithm 7.*

Proof. The proof is along the lines of that for Theorem 3.3. In our memo $T(i, \mathbf{v})$, i can take N values, and $\mathbf{v}$ is a k-length vector in which each entry v_j can take one of N values. Thus, the size of the memo is $O(N \times N^k)$. The backpointers require a space of only $\Theta(N)$, making the overall space complexity $O(N^{k+1})$.

Fixing i, there is 1 entry of the form $T(i, \mathbf{0})$, computing which is in time $\Theta(k)$. Therefore, we get total time $O(Nk)$ here.

Fixing i, there are $i^k - 1$ entries of the form $T(i, \mathbf{v})$, s.t. $\forall j, v_j \leq i - 1$, and $\mathbf{v} \neq \mathbf{0}$. Computing each is in time $\Theta(k)$. Thus, we get total time $k \sum_{i=1}^{N}(i^k - 1) = O(N^{k+1})$ here.

Finally, fixing i, the entries of the form $T(i, \mathbf{v})$, s.t. $\forall j, v_j \leq i$, and $\exists j$, s.t. $v_j = i$ and their total time can be computed using a z to count how many v_j exist such that $v_j = i$. Therefore, the time spent for a fixed i is

$$
\begin{aligned}
t(i) &= \sum_{z=1}^{k} \binom{k}{z} i^{k-z}(z + (k-z)k + i^z zk) \\
&= ki^k \sum_{z=1}^{k} \binom{k}{z} z + k^2 \sum_{z=1}^{k} \binom{k}{z} i^{k-z} - (k-1) \sum_{z=1}^{k} \binom{k}{z} i^{k-z} z \\
&\leq ki^k(k2^{k-1}) + k^2 i^k[(1 + i^{-1})^k - 1] \\
&\leq k^2 2^k i^k + k^2[(i+1)^k - i^k].
\end{aligned}
$$

Thus, the total time that we get here is $\leq \sum_{i=1}^{N} t(i) \leq k^2 2^k N^{k+1} + k^2(N+1)^k \leq ck^2 2^k N^{k+1}$, for some constant c. Therefore, the time here is $O(k^2 2^k N^{k+1})$, so the overall time is also $O(k^2 2^k N^{k+1})$. □

THEOREM 3.5. MATOPT *is* NP*-hard in the width of the lattice on* $\bar{F}$*, the set of input feature sets.*

PROOF. We prove by reduction from the NP-hard problem of SETCOVER-EC [Garey and Johnson 1979]. We are given a universe of elements U, and a family $\bar{U}$ of subsets of U that covers U, i.e., $\cup_{u \in \bar{U}} u = U$, such that all $u \in \bar{U}$ have equal cardinality l. The optimization version of SETCOVER-EC asks for a subfamily $\bar{U}' \subseteq \bar{U}$ of minimum $|\bar{U}'|$ that covers U. We reduce an instance of SETCOVER-EC to an instance of our problem as follows.

U is treated as the full set of features F. Let $|\bar{U}| = N$. Create N feature set inputs $\{F_s | F_s \subseteq F\}$, one for each $s \in \bar{U}$. We use $\bar{F}$ for $\{F_s\}$ as well. All F_s equal cardinality l. Create $|F|$ additional feature singleton set inputs $\{x_j\}$, representing singleton feature sets from F. We use X to denote $\{x_j\}$. We now construct other inputs to our problem conforming to our cost model.

First, make CPU costs negligible so that data access (read) times are linear only in the number of features in the dataset. It is given by $Read(F_j) = \alpha |F_j|$, where α is a free parameter. Note that it does not depend on F_i, which is the input for computations. Second, materialization costs are also linear, and given by $Store(F, F_i) = Read(F) + Write(F_i) = \alpha|F| + \beta|F_i|$, where β is also a free parameter. Note that $Write(F_i)$ is the cost of writing alone. It is reasonable to assume that only F is used for materialization, as we can always make $\bar{F}$ an antichain by dropping a set contained in another within $\bar{F}$ in the given SETCOVER-EC instance. Third, set the repetitions of all F_s to be equal to γ, another free parameter. Set the repetitions of every x_j to 1.

The idea is that members of X will never be materialized, as their individual materialization costs are higher than just accessing F, or any member of $\bar{F}$. Thus, we set the parameters of our problem instance in such a way that our optimal solution will materialize a subset of $\bar{F}$ to use both for serving those feature sets covers all of X. This optimal materialized subset will be an optimum for the given SETCOVER-EC instance.

Recall that we have three free parameters—α, β, and γ. The given instance of SETCOVER-EC gives us F (the universe of features), $\bar{F}$ (the family of subsets of F), $N (= |\bar{F}|)$, and l (cardinality of each member of $\bar{F}$). Thus, the objective function ($\bar{F}' \subseteq \bar{F}$) can be rewritten as follows:

$$\begin{aligned} Cost(\bar{F}') = & \sum_{s \in \bar{F}'} (Store(F, s) + \gamma Read(F_s)) + \sum_{s \in \bar{F} \setminus \bar{F}'} \gamma Read(F) \\ & + \sum_{x \in X : \exists s \in \bar{F}', x \subseteq s} Read(F_s) + \sum_{x \in X : \forall s \in \bar{F}', x \not\subseteq s} Read(F) \end{aligned}$$

$$\begin{aligned} = & |\bar{F}'|((\alpha|F| + \beta l) + \gamma\alpha l) + (|\bar{F}| - |\bar{F}'|)\gamma\alpha|F| \\ & + \alpha l(\#x_j \textit{ covered by } \bar{F}') + \alpha|F|(\#x_j \textit{ not covered by } \bar{F}') \end{aligned}$$

$$\begin{aligned} = & |\bar{F}|\gamma\alpha|F| + |\bar{F}'|(-\gamma\alpha|F| + \alpha|F| + \beta l + \gamma\alpha l) \\ & + \alpha l|X| + \alpha(|F| - l)(\#x_j \textit{ not covered by } \bar{F}'). \end{aligned}$$

The terms $|\bar{F}|\gamma\alpha|F|$ and $\alpha l|X|$ shown earlier are constants independent of $\bar{F}'$ and so we can drop them from the optimization. Thus, we rewrite as

$$Cost(\bar{F}') = |\bar{F}'|(\alpha|F| + \beta l - \gamma\alpha(|F| - l)) + \alpha(|F| - l)(\#x_j \textit{ not covered by } \bar{F}').$$

Now, $|F|, l$, and $N(= |\bar{F}|)$ are given by the instance. We demonstrate a choice of α, β, γ such that the optimal solution to the preceding gives the optimal to the SetCover-EC instance.

First,we want the coefficient of $|\bar{F}'|$ shown earlier (call it p) to be positive, i.e., $p = \alpha|F| + \beta l - \gamma\alpha(|F| - l) > 0$. This can be done by fixing any positive value for α. Then fix any positive integer value for γ. Then, we choose a positive β subject to $\beta > \alpha(\gamma(|F| - l) - |F|)/L$. Call the RHS d (say), i.e., $d := \alpha(\gamma(|F| - l) - |F|)/l$, and $\beta > d$. Second, we need to see if the maximum value of the first term (which is Np, when $\bar{F}' = \bar{F}$) is outweighed by an occurrence of the second term, i.e., when at least one x_j is not covered. In other words, we check if $Np < \alpha(|F| - l)$. Expanding p, it becomes if $N(\alpha|F| + \beta l - \gamma\alpha(|F| - l)) < \alpha(|F| - l)$. Rearranging in terms of β, it becomes if $\beta < \alpha(\gamma(|F| - l) - |F|)/l + \alpha(|F| - l)/(lN)$, i.e., if $\beta < d + \alpha(|F| - l)/(lN)$. Since the second term is positive (as $|F| > l$), we only have to choose β such that $d < \beta < d + \alpha(|F| - l)/(lN)$.

Since not covering even one x_j raises the cost more than choosing all the sets, a minimum cost solution to the precedint instance of MatOpt will always cover all x_j, and it will minimize among the chosen subsets of $\bar{F}$. Thus, an optimal solution to the instance of MatOpt as constructed earlier will be an optimal solution to the given instance of SetCover-EC. □

Cost-based execution. Recall that the executor of Columbus executes ROPs by calling the required database or main-memory backend. The executor is responsible for executing and coordinating multiple ROPs that can be executed in parallel; Columbus executor simply creates one thread to manage each of these ROPs. The actual execution of each physical operator is performed by the backend statistical framework, e.g., R or ORE. Nevertheless, we need to decide how to schedule these ROPs for a given program. We experimented with the trade-off of how coarsely or finely to batch the execution. Many of the straightforward formulations of the scheduling problems are, not surprisingly, NP-hard. Nevertheless, we found that a simple greedy strategy (to batch as many operators as possible, i.e., operators that do not share data flow dependencies) was within 10% of the optimal schedule obtained by a brute-force search. After digging into this detail, we found that many of the host-level substrates already provide sophisticated data processing optimizations, e.g., sharing scans.

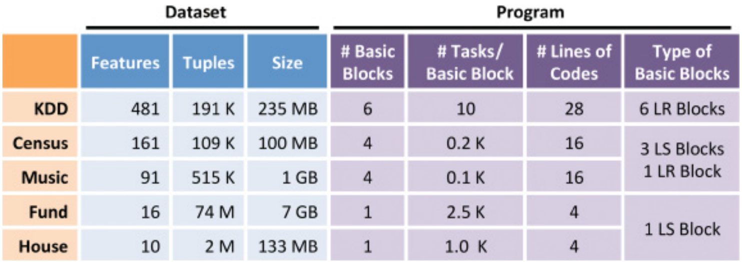
4. EXPERIMENTS

Using the materialization trade-offs that we have outlined, we validate that Columbus is able to speed up the execution of feature selection programs by orders of magnitude compared to straightforward implementations in state-of-the-art statistical analytics frameworks across two different backends: R (in-memory) and a commercial RDBMS. We validate the details of our technical claims about the trade-off space of materialization and our (preliminary) multiblock optimizer.

4.1. Experiment Setting

Based on conversations with analysts, we selected a handful of datasets and created programs that use these datasets to mimic analysts' tasks in different domains. We describe these programs and other experimental details.

Datasets and programs. To compare the efficiency of Columbus with baseline systems, we select five publicly available datasets: (1) Census, (2) House, (3) KDD, (4) Music, and



	Dataset			Program			
	Features	Tuples	Size	# Basic Blocks	# Tasks/ Basic Block	# Lines of Codes	Type of Basic Blocks
KDD	481	191 K	235 MB	6	10	28	6 LR Blocks
Census	161	109 K	100 MB	4	0.2 K	16	3 LS Blocks 1 LR Block
Music	91	515 K	1 GB	4	0.1 K	16	
Fund	16	74 M	7 GB	1	2.5 K	4	1 LS Block
House	10	2 M	133 MB	1	1.0 K	4	

Fig. 10. Dataset and program statistics. LS refers to least squares. LR refers to logistic regression.

(5) Fund.[18] These datasets have different sizes, and we show the statistics in Figure 10. We categorize them by the number of features in each dataset.

Both House, a dataset for predicting household electronic usage, and Fund, a dataset for predicting the donation that a given agency will receive each year, have a small number of features (fewer than 20). In these datasets, it is feasible to simply try and score almost all combinations of features. We mimic this scenario by having a large basic block that regresses a least-squares model on feature sets of sizes larger than 5 on House and 13 on Fund and then scores the results using AIC. These models reflect a common scenario in current enterprise analytics systems.

At the other extreme, KDD has a large number of features (481), and it is infeasible to try many combinations. In this scenario, the analyst is guided by automatic algorithms, like lasso (which selects a few sparse features), manual intervention (moving around the feature space), and heavy use of cross-validation techniques.[19] Census is a dataset for the task of predicting the mail responsiveness of people in different Census blocks, each of which contains a moderate number of features (161). In this example, analysts use a mix of automatic and manual specification tasks that are interleaved.[20] This is the reason we select this task for our running example. Music is similar to Census, and both programs contain both linear models (least squares) and nonlinear models (logistic regression) to mimic the scenario in which an analyst jointly explores the feature set to select and the model to use.

R backends. We implemented COLUMBUS on multiple backends and report on two: (1) *R*, which is the standard main-memory R, and (2) *DB-R*, the commercial R implementation over RDBMS. We use R 2.15.2 and the most recent available versions of the commercial systems.

For all operators, we use the result of the corresponding main memory R function as the gold standard. All experiments are run on instances on Amazon EC2 (cr1.8xlarge), which has 32 vCPU, 244GB RAM, and 2×120GB SSD and runs Ubuntu 12.04.[21]

4.2. End-to-End Efficiency

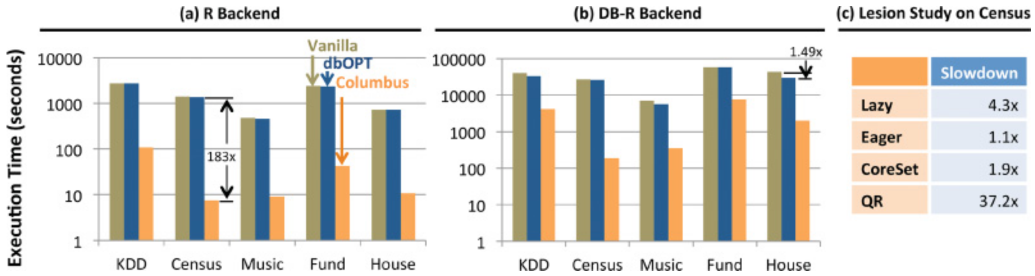
We validate that COLUMBUS improves the end-to-end performance of feature selection programs. We construct two families of competitor systems (one for each backend): VANILLA and DBOPT. VANILLA is a baseline system that is a straightforward

[18] These datasets are publicly available on Kaggle (www.kaggle.com/) or the UCI Machine Learning Repository (archive.ics.uci.edu/ml/).

[19] The KDD program contains six basic blocks, each of which is a 10-fold cross validation. These six different basic blocks work on a nonoverlappings set of features specified by the user manually.

[20] The Census program contains four basic blocks, each of which is a STEPDROP operation on the feature set output by the previous basic block.

[21] We also ran experiments on other dedicated machines. The trade-off space is similar to what we report in this article.



	Slowdown
Lazy	4.3x
Eager	1.1x
CoreSet	1.9x
QR	37.2x

Fig. 11. End-to-end performance of COLUMBUS. All approaches return a loss within 1% optimal loss.

implementation of the corresponding feature selection problem using the ROPs; thus, it has the standard optimizations. DBOPT is COLUMBUS, but we enable only the optimizations that have appeared in classical database literature, i.e., Lazy, Eager, and batching. DBOPT and COLUMBUS perform scheduling in the same way to improve parallelism to isolate the contributions of the materialization. Figure 11 shows the result of running these systems over all five datasets with error tolerance ϵ set to 0.01.

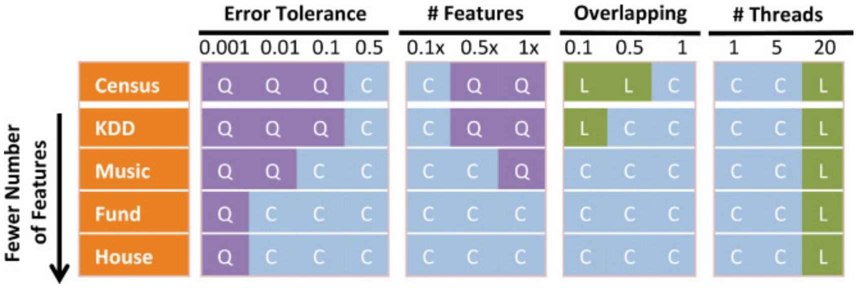
On the R-based backend, COLUMBUS executes the same program using less time than R on all datasets. On Census, COLUMBUS is two orders of magnitude faster, and on Music and Fund, COLUMBUS is one order of magnitude faster. On Fund and House, COLUMBUS chooses to use CoreSet+QR as the materialization strategy for all basic blocks and chooses to use QR for other datasets. This is because for datasets that contain fewer rows and more columns, QR dominates CoreSet-based approaches, as described in the previous section. One reason COLUMBUS improves more on Census than on Music and Fund is that Census has more features than Music and Fund; therefore, operations like StepDrop produce more opportunities for reuse than Census.

To understand the classical points in the trade-off space, compare the efficiency of DBOPT to the baseline system, VANILLA. When we use R as a backend, the difference between DBOPT and R is less than 5%. The reason is that R holds all data in memory, and accessing a specific portion of the data does not incur any IO cost. In contrast, we observe that when we use the DB backend, DBOPT is 1.5× faster than VANILLA on House. However, this is because the underlying database is a row store, so the time difference is due to IO and deserialization of database tuples.

We can also see that the new forms of reuse we outline are significant. If we compare the execution time of Census and Music, we see a difference between the approaches. Whereas Census is smaller than Music, baseline systems, e.g., VANILLA, are slower on Census than on Music. In contrast, COLUMBUS is faster on Census than on Music. This is because Census contains more features than Music; therefore, the time that VANILLA spent on executing complex operators like STEPDROP is larger in CENSUS. In contrast, by exploiting the new trade-off space of materialization, COLUMBUS is able to reuse computation more efficiently for feature selection workloads.

4.3. Linear Basic Blocks

We validate that all materialization trade-offs that we identified affect the efficiency of COLUMBUS. In Section 3, we designed experiments to understand the trade-off between different materialization strategies with respect to three axes, i.e., error tolerance, sophistication of tasks and reuse, and computation. Here, we validate that each optimization contributes to the final results in a full program (on Census). We then validate our claim that the crossover points for optimizations change based on the dataset but that the space essentially stays the same. We only show results on the main-memory backend.



	Error Tolerance				# Features			Overlapping			# Threads		
	0.001	0.01	0.1	0.5	0.1x	0.5x	1x	0.1	0.5	1	1	5	20
Census	Q	Q	Q	C	C	Q	Q	L	L	C	C	C	L
KDD	Q	Q	Q	C	C	Q	Q	L	C	C	C	C	L
Music	Q	Q	C	C	C	C	Q	C	C	C	C	C	L
Fund	Q	C	C	C	C	C	C	C	C	C	C	C	L
House	Q	C	C	C	C	C	C	C	C	C	C	C	L

Fewer Number of Features

Fig. 12. Robustness of materialization trade-offs across datasets. For each parameter setting (one column in the table), we report the materialization strategy that has the fastest execution time given the parameter setting. Q refers to QR, C refers to CoreSet+QR, and L refers to Lazy. The protocol is the same as for Figure 6 in Section 3.

Lesion study. We validate that each materialization strategy has an impact on the performance of Columbus. For each parameter setting used to create Figure 6, we remove a materialization strategy. Then we measure the maximum slowdown of an execution with that optimization removed. We report the maximum slowdown across all parameters in Figure 11(c) in main memory on Census. We see that Lazy, QR, and CoreSet all have significant impacts on quality, ranging from 1.9× to 37×. This means that if we drop any of them from Columbus, one would expect a 1.9× to 37× slowdown on the whole Columbus system. Similar observations hold for other backends. The only major difference is that our DB backend is a row store, and Eager has a larger impact (1.5× slowdown).

We validate our claim that the high-level principles of the trade-offs remain the same across datasets, but we contend that the trade-off points change across datasets. Thus, our work provides a guideline about these trade-offs, but it is still difficult for an analyst to choose the optimal point. In particular, for each parameter setting, we report the name of the materialization strategy that has the fastest execution time. Figure 12 shows that across different datasets, the same pattern holds but with different crossover points. Consider the *error tolerance*. On all datasets, for high error tolerance, CoreSet+QR is always faster than QR. On Census and KDD, for the lowest three error tolerances, QR is faster than CoreSet+QR, whereas on Music, only for the lowest two error tolerance is QR faster than CoreSet+QR. On Fund and House, for all error tolerances except the lowest one, CoreSet+QR is faster than QR. Thus, the crossover point changes.

We now validate that the high-level principles of the trade-offs do not change even when we vary the number of examples and features. Note that most of our real datasets have a relatively small number of examples and features even if they resemble the real use cases of analysts that we found in practice. Thus, we use a new dataset with 100K examples and 2,000 features.[22] We drop features and examples (or add synthetic features) to generate a series of datasets with the number of examples in $\{1000, 10000, 100000\}$, the number of features in $\{100, 200, 400, 800, 1000, 2000, 4000\}$, and the number of features smaller than the number of examples. We run each dataset with one thread and the first 100 tasks in a StepDrop operator. Figure 13(a) shows the relative execution time speedup of QR over Lazy. We see that even with very large number of examples and features, QR consistently outperforms Lazy, as indicated by our cost model. The largest speedup is achieved when the number of examples is large but the number of features is small. Given the same number of features, the more

[22] http://largescale.ml.tu-berlin.de/instructions/.

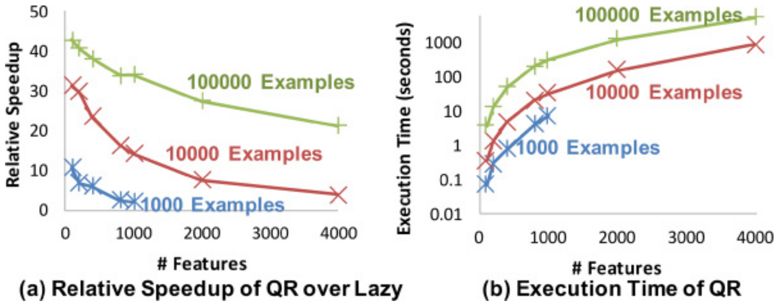


Fig. 13. Varying the number of features and number of examples. (a) Speedup of QR over Lazy. (b) Actual execution time of QR. We only consider datasets where the the number of examples exceeds the number of features.

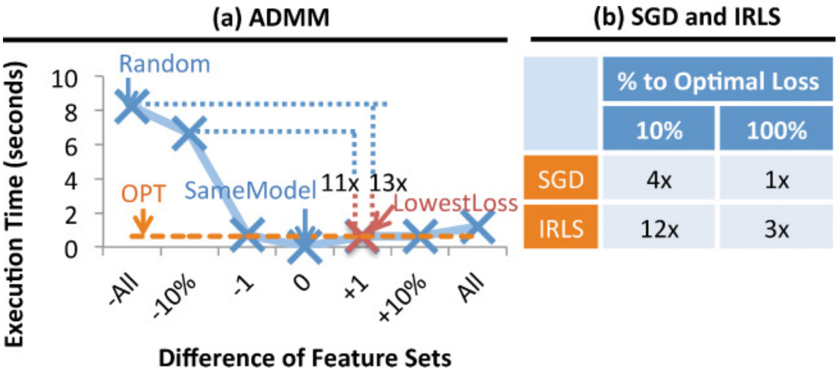
examples a dataset has, the larger the speedup. This is because the execution time of QR is only quadratic in the number of features, as shown in Figure 4. Figure 13(b) shows the absolute execution time of QR for reference. For the largest dataset, it takes about 1 hour to finish both materialization and execution. Of course, improving the efficiency of each ROP could reduce the total execution time even more. This issue is orthogonal to our work.

4.4. Nonlinear Basic Blocks with ADMM

Columbus uses ADMM as the default nonlinear solver, which requires that one solves a least-squares problem that we studied in linear basic blocks. Compared to linear basic blocks, one key twist with ADMM is that it is iterative and thus has an additional parameter: the number of iterations to run. We validate that trade-offs similar to the linear case still apply to nonlinear basic blocks, and we describe how convergence impacts the trade-off space. For each dataset, we vary the number of iterations to run for ADMM and try different materialization strategies. For CoreSet-based approaches, we grid search the error tolerance, as we did for the linear case. As shown in Figure 6(d), when the number of iterations is small, QR is 2.24× slower than Lazy. Because there is only one iteration, the least-squares problem is only solved once. Thus, Lazy is the faster strategy compared with QR. However, when the number of iterations grows to 10, QR is 3.8× faster than Lazy. This is not surprising based on our study for linear cases—by running more iterations, the opportunities for reuse increase. We would expect an even larger speedup if we ran more iterations.

Warmstart using model caching. We validate the model-caching trade-off for different iterative models, including SGD, ADMM, and IRLS. We use logistic regression as an example throughout (as it is the most common nonlinear loss).

Given a feature set F, we construct seven feature sets to mimic different scenarios of model caching, and we call them F^{-All}, $F^{-10\%}$, F^{-1}, F^{0} F^{+1}, $F^{+10\%}$, and F^{+All}. Where F^{-All} mimics the cold start problem when the analyst has not trained any models for this dataset, we initialize with a random data point (this is Random); F^{+All} means that the analyst has a "superparent" model trained for all features and has selected some features for warmstart; $F^{-10\%}$ (respectively, $F^{+10\%}$) is when we warmstart with a model different from F by removing (respectively, adding) $10\%|F|$ of randomly picked features; F^{+1} and F^{-1} mimic models generated by StepDrop or StepAdd, which are made different from F by removing or adding only a single feature. F^{0} is when we have the same F (we call it SameModel). For each dataset, we run ADMM with the initial model set to the model obtained from training different Fs. Then we measure the time needed to converge to 1%, 10%, and 2× of the optimal loss for F. Our heuristic



	% to Optimal Loss	
	10%	100%
SGD	4x	1x
IRLS	12x	3x

Fig. 14. (a) The impact of model caching for ADMM. The execution time is measured by the time needed to converge to 1% optimal loss. (b) The speedup of model caching. Different iterative approaches are compared to a given loss with a random initial model.

LowestLoss chooses a single model based on which has the lowest loss, and OPT chooses the lowest execution time among all Fs, except F^0. Because the process relies on randomly selecting features to add or remove, we randomly select 10 feature sets and train a model for each them and report the average.

We first consider this for ADMM, and Figure 14(a) shows the result. We see that LowestLoss is 13× faster than Random. In addition, if we disable our heuristic and just pick a random feature set to use for warmstart, we could be 11× slower in the worst case. Compared to OPT, LowestLoss is 1.03× slower. This validates the effectiveness of our warmstart heuristic. Another observation is that when we use a feature set that is a superset of F (F^{+1}, $F^{+10\%}$), we can usually expect better performance than using a subset (F^{-1}, $F^{-10\%}$). This is not surprising because the superset contains more information relevant to all features in F.

Other iterative approaches. We validate that warmstart also works for other iterative approaches, namely SGD and IRLS (the default in R). We run all approaches and report the speedup using our proposed lowest-loss heuristic compared to Random. As shown in Figure 14(b), we see that both SGD and IRLS are able to take advantage of warmstarting to speed up their convergence by up to 12×.

4.5. Multiblock Optimization

We validate that our greedy optimizer for multiblock optimization has comparable performance to the optimal optimizer. We compare the plan that is picked by the Columbus optimizer, which uses a greedy algorithm to choose between different physical plans and the optimal physical plan that we find by brute-force search.[23] We call the optimal physical plan OPT and show the result in Figure 15.

As shown in Figure 15, Columbus's greedy optimizer is slower than OPT on KDD, Census, and Music; however, it is slower within a range of less than 10%. For Fund and House, OPT has the same performance as Columbus's greedy optimizer, because there is only one basic block, and the optimal strategy is picked by Columbus's greedy optimizer. When OPT selects a better plan than our greedy optimizer, we found that the greedy optimizer does not accurately estimate the amount of reuse.

5. RELATED WORK

We consider work in feature selection and analytics.

[23] For each physical plan, we terminate the plan if it runs longer than the plan picked by the greedy scheduler.

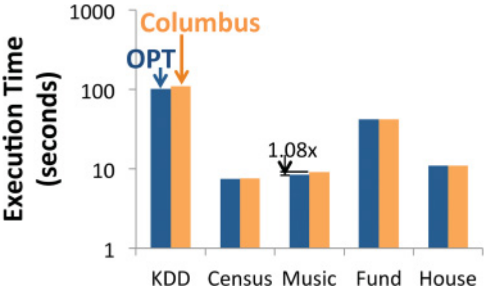


Fig. 15. Greedy versus optimal optimizer.

5.1. Feature Selection

Algorithms for feature selection have been studied in the statistical and machine learning literature for decades [Boyce 1974; John et al. 1994; Hastie et al. 2001; Guyon and Elisseeff 2003; Guyon et al. 2006]. A typical formalization is to obtain one subset of the features of a given dataset, subject to some optimization criteria. Feature selection algorithms are categorized into *Filters*, *Wrappers*, and *Embedded* methods. Filters assign scores to features independent of what statistical model for which the features are used. Wrappers are meta-algorithms that score feature sets using a statistical model. Embedded methods wire feature selection into a statistical model [Guyon et al. 2006]. Our conversations with analysts revealed that feature selection is often a data-rich process with the analyst in the loop rather than a one-shot algorithm. Our goal is to take a step toward managing the process of feature selection using data management ideas. We aim to leverage popular selection algorithms, not design new ones.

Scaling individual feature selection algorithms to larger data has received attention in the past for specific platforms. Oracle Data Mining offers three popular feature selection algorithms over in-RDBMS data.[24] Singh et al. [2009] parallelize forward selection for logistic regression on MapReduce/Hadoop. In contrast, our focus is on building a generic framework for the feature selection processes rather than for specific algorithms and platforms.

5.2. Analytics Systems

Systems that deal with data management for statistical and machine learning techniques have been developed in both industry and academia. These include data mining toolkits from major RDBMS vendors that integrate specific algorithms with an RDBMS [Hellerstein et al. 2012] and systems that aim to make it easier to implement machine learning in an RDBMS [Feng et al. 2012]. Similar efforts exist for other data platforms.[25] The second stream includes recent products from enterprise analytics vendors that aim to support statistical computing languages like R over data residing in data platforms, e.g., Oracle's ORE,[26] IBM's SystemML [Ghoting et al. 2011], SAP HANA,[27] and the RIOT project [Zhang et al. 2010]. Our work focuses on the data management issues in the process of feature selection, and our ideas can be integrated into these systems.

Array databases were initiated by Sarawagi and Stonebraker [1994], who studied how to efficiently organize *multidimensional arrays* in an RDBMS. Since then, there

[24] oracle.com/technetwork/database/options/advanced-analytics/odm.

[25] mahout.apache.org.

[26] https://docs.oracle.com/cd/E27988_01/doc.112/e26499.pdf.

[27] help.sap.com/hana/hana_dev_r_emb_en.pdf.

has been a recent resurgence in arrays as first-class citizens [Cohen et al. 2009; Brown 2010; Hellerstein et al. 2012; Stonebraker et al. 2013]. For example, Stonebraker et al. [2013] recently envisioned the idea of using carefully optimized C++ code, e.g., ScaLAPACK, in array databases for matrix calculations. COLUMBUS is complementary to these efforts, as we focus on how to optimize the execution of multiple operations to facilitate reuse. The materialization trade-offs that we explore are (largely) orthogonal to these lower-level trade-offs. However, since linear algebra operations arise in both R and array databases, we hope that our proposed techniques can also be applied in the realm of array databases.

There has been an intense effort to scale up individual linear algebra operations in data processing systems [Benson et al. 2013; Zhang et al. 2010; Blackford et al. 1996]. Benson et al. [2013] propose a distributed algorithm to calculate QR decomposition using MapReduce, whereas ScaLAPACK [Blackford et al. 1996] uses a distributed main memory system to scale up linear algebra. The RIOT [Zhang et al. 2010] system optimizes the I/O costs incurred during matrix calculations. Similar to array databases, COLUMBUS directly takes advantage these techniques to speed up the execution of each ROP.

Our focus on performance optimizations across full programs was inspired by similar efforts in RIOT-DB [Zhang et al. 2010] and SystemML [Ghoting et al. 2011]. RIOT-DB optimizes I/O by rearranging page accesses for specific loop constructs in an R program [Zhang et al. 2010]. SystemML [Ghoting et al. 2011] converts R-style programs to workflows of MapReduce jobs. They describe an optimization called *piggybacking* that enables sharing of data access by jobs that follow each other.

In a similar spirit, declarative machine learning systems, e.g., MLBase [Kraska et al. 2013], provide a high-level language to end users to specify a machine learning task and compare multiple learning algorithms. Compared to these systems, COLUMBUS focuses on providing a high-level language for feature selection as opposed to algorithms. The conventional wisdom is that most improvement comes through good features as opposed to different algorithms. We are hopeful that the materialization trade-offs that we study can be applied in declarative machine learning systems.

Finally, the Hazy project highlighted usability and development issues in statistical analytics (in addition to performance and scalability) that require more research attention [Kumar et al. 2013]. In particular, feature engineering, which is the process of designing and managing features for machine learning, is increasingly being recognized by the database community as a critical bottleneck in this regard. For example, Brainwash envisions a framework to support an iterative, human-in-the-loop approach to feature engineering [Anderson et al. 2013]. We hope that our work on COLUMBUS contributes to more research in this direction.

6. CONCLUSION AND FUTURE WORK

COLUMBUS is the first system to treat the feature selection dialogue as a database systems problem. Our first contribution is a declarative language for feature selection, informed by conversations with analysts over the past 2 years. We observed that there are reuse opportunities in analysts' workloads that are not addressed by today's R backends. To demonstrate our point, we showed that simple materialization operations could yield orders of magnitude performance improvements on feature selection workloads. As analytics grows in importance, we believe that feature selection will become a pressing data management problem. There are several directions for future work.

First, we could better support the human-in-the-loop process of feature selection by building an REPL-like environment for feature selection that automatically performs materializations even as analysts browse and analyze their datasets. To support this type of application, we expect the trade-off space studied in COLUMBUS to be useful in guiding which materialization strategies should be executed. One question in this context is whether it is possible to predict the future workloads of the analysts based on their queries to a given point in time. This could enable new, speculative materialization opportunities. Recent work in modeling SQL workloads [Ganapathi et al. 2010] might be relevant for this problem, but it is still an open question as to how their techniques can be adapted to statistical analytics.

Second, the current formalization of basic blocks in COLUMBUS is motivated by feature selection workloads. It is an open question as to whether it is possible to extend our ideas to a larger subset of the R language. This would require us to consider a richer set of techniques for materialization beyond just QR and coresets such as partial and incremental model computations. More generally, new opportunities to improve performance and usability arise when we consider Explore, Evaluate, Regression, and other learning operations in conjunction with data transform operations. For example, recognizing that the joins that typically precede learning over normalized data might introduce redundancy enables one to simply avoid such redundancy by "learning *over* joins" rather than *after* joins [Kumar et al. 2015]. It is future work to extend the COLUMBUS optimizer and cost model to integrate these new techniques and support such new workloads.

Finally, another direction is to apply the approach of COLUMBUS to less-structured data sources such as text, scanned documents, and images. For these sources, features are often generated by a separate feature engineering phase, which is also a human-in-the-loop process [Shin et al. 2015] that requires domain experts to iteratively write "extractors" to produce features. As future work, we could study how these two human-in-the-loop processes interact with each other. For example, will a fast feature selection subsystem provide intuitive guidelines and hints on promising new features one has not yet extracted? By jointly conducting feature selection and feature engineering in a single framework, it might be possible to further facilitate the development of feature-centric statistical analytics systems.

REFERENCES

Michael Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. 2013. Brainwash: A data system for feature engineering. In *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*. http://web.eecs.umich.edu/~michjc/papers/mythical_man.pdf.

Austin R. Benson, David F. Gleich, and James Demmel. 2013. Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. In *Proceedings of the 2013 IEEE International Conference on Big Data*. 264–272. DOI: http://dx.doi.org/10.1109/BigData.2013.6691583

D. P. Bertsekas. 1999. *Nonlinear Programming*. Athena Scientific.

L. Susan Blackford, Jaeyoung Choi, Andrew J. Cleary, James Demmel, Inderjit S. Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clinton Whaley. 1996. ScaLAPACK: A portable linear algebra library for distributed memory computers—design issues and performance. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. 5. DOI: http://dx.doi.org/10.1109/SC.1996.41

Léon Bottou and Olivier Bousquet. 2007. The tradeoffs of large scale learning. In *Proceedings of the 21st Annual Conference on Neural Information Processing Systems (NIPS'07)*. 161–168. http://papers.nips.cc/paper/3323-the-tradeoffs-of-large-scale-learning.

Christos Boutsidis, Petros Drineas, and Malik Magdon-Ismail. 2013. Near-optimal coresets for least-squares regression. *IEEE Transactions on Information Theory* 59, 10, 6880–6892. DOI: http://dx.doi.org/10.1109/TIT.2013.2272457

David E. Boyce. 1974. *Optimal Subset Selection: Multiple Regression, Interdependence, and Optimal Network Algorithms*. Springer-Verlag.

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning* 3, 1, 1–122.

Paul G. Brown. 2010. Overview of SciDB: Large scale array storage, processing and analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 963–968. DOI:http://dx.doi.org/10.1145/1807167.1807271

Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. 2009. MAD skills: New analysis practices for big data. *Proceedings of the VLDB Endowment* 2, 2, 1481–1492. DOI:http://dx.doi.org/10.14778/1687553.1687576

Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. 325–336. DOI:http://dx.doi.org/10.1145/2213836.2213874

Archana Ganapathi, Yanpei Chen, Armando Fox, Randy H. Katz, and David A. Patterson. 2010. Statistics-driven workload modeling for the cloud. In *Proceedings of the Workshops of the IEEE International Conference on Data Engineering (ICDE'10)*. 87–92. DOI:http://dx.doi.org/10.1109/ICDEW.2010.5452742

M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'11)*. 231–242. DOI:http://dx.doi.org/10.1109/ICDE.2011.5767930

G. Golub. 1965. Numerical methods for solving linear least squares problems. *Numerische Mathematik* 7, 3, 206–216.

Goetz Graefe and William J. McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'93)*. 209–218. DOI:http://dx.doi.org/10.1109/ICDE.1993.344061

Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *Journal of Machine Learning Research* 3, 1157–1182. http://www.jmlr.org/papers/v3/guyon03a.html.

Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lotfi A. Zadeh. 2006. *Feature Extraction: Foundations and Applications*. Springer-Verlag, New York, NY.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib analytics library or MAD skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12, 1700–1711. http://vldb.org/pvldb/vol5/p1700_joehellerstein_vldb2012.pdf.

George H. John, Ron Kohavi, and Karl Pfleger. 1994. Irrelevant features and the subset selection problem. In *Proceedings of the 11th International Conference on Machine Learning*. 121–129.

Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12, 2917–2926. DOI:http://dx.doi.org/10.1109/TVCG.2012.219

Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. 2013. MLbase: A distributed machine-learning system. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*. http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper118.pdf.

Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning generalized linear models over normalized data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. 1969–1984. DOI:http://dx.doi.org/10.1145/2723372.2723713

Arun Kumar, Feng Niu, and Christopher Ré. 2013. Hazy: Making it easier to build and maintain big-data analytics. *Communications of the ACM* 56, 3, 40–49. DOI:http://dx.doi.org/10.1145/2428556.2428570

Michael Langberg and Leonard J. Schulman. 2010. Universal epsilon-approximators for integrals. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*. 598–607. DOI:http://dx.doi.org/10.1137/1.9781611973075.50

Sunita Sarawagi and Michael Stonebraker. 1994. Efficient organization of large multidimensional arrays. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'94)*. 328–336. DOI:http://dx.doi.org/10.1109/ICDE.1994.283048

Shai Shalev-Shwartz and Nathan Srebro. 2008. SVM optimization: Inverse dependence on training set size. In *Machine Learning: Proceedings of the 25th International Conference*. 928–935. DOI:http://dx.doi.org/10.1145/1390156.1390273

Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. 2015. Incremental knowledge base construction using deepdive. *Proceedings of the VLDB Endowment* 8, 11, 1310–1321. DOI:http://dx.doi.org/10.14778/2809974.2809991

Sameer Singh, Jeremy Kubica, Scott Larsen, and Daria Sorokina. 2009. Parallel large scale feature selection for logistic regression. In *Proceedings of the SIAM International Conference on Data Mining (SDM'09)*. 1172–1183. DOI:http://dx.doi.org/10.1137/1.9781611972795.100

Michael Stonebraker, Sam Madden, and Pradeep Dubey. 2013. Intel "big data" science and technology center vision and execution plan. *ACM SIGMOD Record* 42, 1, 44–49. DOI:http://dx.doi.org/10.1145/2481528.2481537

Jiyan Yang, Yin-Lam Chow, Christopher Ré, and Michael Mahoney. 2016. Weighted SGD for lp regression with randomized preconditioning. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'16)*.

Ce Zhang, Arun Kumar, and Christopher Ré. 2014. Materialization optimizations for feature selection workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. 265–276. DOI:http://dx.doi.org/10.1145/2588555.2593678

Yi Zhang, Weiping Zhang, and Jun Yang. 2010. I/O-efficient statistical computing with RIOT. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'10)*. 1157–1160. DOI:http://dx.doi.org/10.1109/ICDE.2010.5447819

Received February 2015; revised September 2015; accepted November 2015