Materialization Optimizations for Feature Selection Workloads

Ce Zhang[†‡] Arun Kumar[†] Christopher Ré[‡]
[†]University of Wisconsin-Madison, USA
[‡]Stanford University, USA
{czhang,chrismre}@cs.stanford.edu, arun@cs.wisc.edu

May 31, 2014

Abstract

There is an arms race in the data management industry to support analytics, in which one critical step is feature selection, the process of selecting a feature set that will be used to build a statistical model. Analytics is one of the biggest topics in data management, and feature selection is widely regarded as the most critical step of 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 analog in traditional SQL systems, but they may be interesting for array and scientific database applications. On a diverse set of data sets 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 tradeoff 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.

1 Introduction

One of the most critical stages in the data 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 their 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 [20, 25],

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 [6]. Nevertheless, the feature selection process has structure: analysts often use domain-specific cookbooks that outline best practices for feature selection [2, 7, 20].

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 [20]). Over the last 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 [17], 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 data set. As a result, it is difficult for analysts to pick the correct combination of materialization optimizations.

To study these tradeoffs, we introduce Columbus, an R language extension and execution framework designed for feature selection. To use Columbus, a user writes a 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 [20] 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 Def. 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 data set (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

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

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

Figure 1: Summary of Tradeoffs in COLUMBUS.

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. In addition, we identify three novel classes of optimizations, study the tradeoffs 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 the 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 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 tradeoff for sampling-based materialization strategies. Of course, sampling strategies have the ability to improve performance by an order of magnitude. On a real data set, called Census, we found that d was 1000x smaller than N, as well as that using a coreset outperforms a baseline approach by 89x, while still getting a solution that is within 1% of the loss of the solution on the entire dataset.

Transformation Materialization Linear algebra has a variety of decompositions that are analogous to sophisticated materialized views. One such decomposition, called 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, QR can outperform a baseline by more than 10X; QR can also be applied together with coresets, which can result in 5x more speed up. Of course, there is a cost-benefit tradeoff 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 with the default approach in R (initializing with a random start point or all 0's), our heuristic provides a 13x speedup; compared with a simple strategy that selects a random model in the cache, our heuristic achieves a 6x 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 tradeoffs. Of course, the correct choice also depends on computational constraints, notably parallelism. We describe a series of experiments to validate this tradeoff 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 above materialization strategies. We validate that our heuristic optimizer has performance within 10% of the optimal optimization strategy (found offline by brute force) on all our workloads. In the full version, 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 we pursue apply beyond feature selection to areas like array and scientific databases and tuning machine learning.

Outline The rest of this paper is organized as follows. In Section 2, we provide an overview of the Columbus system. In Section 3, we describe the tradeoff 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, including ORE, Revolution Analytics, etc. 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). Figure 5 shows all ROPs that are used in Columbus.

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

Figure 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

Figure 3: Summary of Operators in COLUMBUS.

2 System Overview

2.1 Columbus Programs

In COLUMBUS, a user expresses their 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.[2] 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 *data set*, which is a relational table $R(A_1, \ldots, A_d)$.[3] A *feature set* F for a dataset $R(A_1, \ldots, A_d)$ is a subset of the attributes $F \subseteq \{A_1, \ldots, A_d\}$. 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 data sets; (2) *Evaluate Operations*, which evaluate data sets and models; (3) *Regression Operations*, which produce a model given a feature set; and (4) *Explore Operations*, which produce new feature sets:

(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;

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

[3]Note that the table itself can be a view; this is allowed in COLUMBUS, and the tradeoffs for materialization are standard, so we omit the discussion of them in the paper.

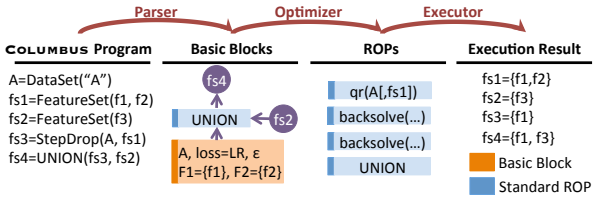

Figure 4: Architecture of COLUMBUS.

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 Akaike Information Criterion (AIC) [20]. 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 produces 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 data set 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 et al. [20].

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 [19]).

Definition 2.1. *A* task *is a tuple* $\mathtt{t} = (A, b, \ell, \epsilon, F, R)$ *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:*

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

Here Π_F is the axis-aligned projection that selects the columns or feature sets specified by F.[4] *Denote an optimal solution of the task $x_*(\mathsf{t})$ defined as*

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

Our goal is to find an $x(\mathsf{t})$ that satisfies the error[5]

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

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 non-linear 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 2.1. *Consider the 6^{th} line in Figure 2, which specifies a 5-fold cross validation operator with least squares over data set d_1 and feature set s_1.* COLUMBUS *will generate a basic block B with 5 tasks, one for each fold. Let $t_i = (A, b, l, \epsilon, F, R)$. Then, A and b are defined by the data set d_1 and $l(x, b) = (x - b)^2$. The error tolerance ϵ is given by the user in the 1^{st} 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 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. Figure 5 summarizes the host-level operators that COLUMBUS uses, and we observe that these operators are present in both R and ORE. Our focus is 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 4: (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

[4] 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.

[5] We allow termination criteria via a user-defined function or the number of iterations. The latter simplifies reuse calculations in Section 3, while 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 as it brings out interesting tradeoffs.

(a) ROPs Used in Columbus and Their Running Times

ROPs	Semantic	Cost
A <- B[1:n,1:d]	matrix sub-selection	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 E: n×d V: n×1 W: d×d
The cost of an op. with **black** font face dominates the cost of any green op. for that ROP

Figure 5: (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 sub-selection ($a \leftarrow B[1:n, 1:d]$) is only dn. In (b), **Materialization ROPs** are the set of ROPs used in the materialization phase for an operator, while **Execution ROPs** are those ROPs 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 ROPs they produce in (a).

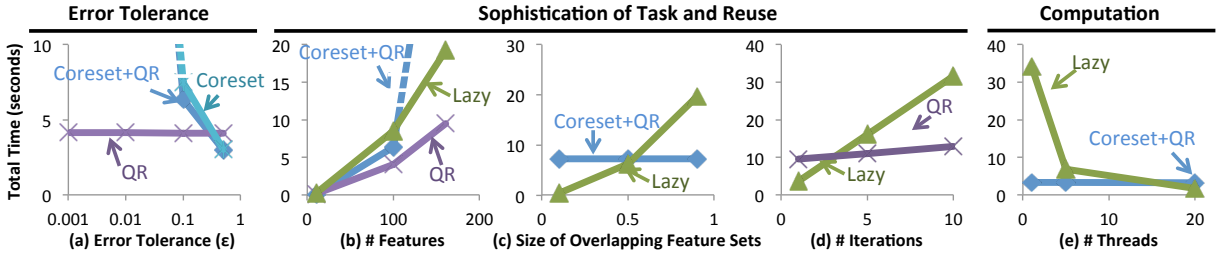

Figure 6: An Illustration of the Tradeoff Space of Columbus, which is defined in Section 3.

edges indicate data flow dependency. The optimizer is responsible for generating a "physical plan." This plan defines which algorithms and materialization stategies 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, which is 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.

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 data set 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:[6]

(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), the number of feature sets or rows selected, and their degree of overlap. In Figure 6, we set the number of features as $\{10, 100, 161\}$ and the number of tasks in each block as $\{1, 10, 20, 50\}$.

(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[7] and the number of available threads that COLUMBUS uses, which we set here to $\{1, 5, 10, 20\}$.[8]

We discuss these graphs in paragraphs marked *Tradeoff* 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 last two of these leverage the fact that we are solving several regression problems. Each of 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 5 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.[9]

To simplify our presentation, in this subsection, 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 subsection. Our basic block can be simplified to $\mathsf{B} = (A, b, \bar{F}, \bar{R}, \varepsilon)$, for which we compute:

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

Our goal is to compile the basic block into a set of ROPs. We explain the optimizations that we identify below.

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—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

[6] For each combination of parameters below, 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 paper. Any omitted data point is dominated by a shown data point.

[7] 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.

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

[9] 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. In the full version of this paper, we show that the cost we estimated for one operator is within 15% of the actual execution time.

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 save I/Os versus *Lazy*.

Tradeoff Not surprisingly, *Eager* has a higher materialization cost than *Lazy*, while *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 tradeoffs 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. In the full paper, 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* [11, 27]; we describe the tradeoffs these methods provide.

Naïve Sampling Naïve random sampling is widely used, and in fact, analysts ask for it by name. In naïve random sampling, one selects some fraction of the data set. 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.

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:

Proposition 3.1 (Boutsidis et al. [11]). *For $A \in \mathbb{R}^{N \times d}$ and $b \in \mathbb{R}^N$. Define $s(i) = a_i^T \left(A^T A\right)^{-1} a_i$. 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*

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

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

This guarantee is strong.[10] 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 Prop 3.1 does not use the value b. COLUMBUS uses this fact later to optimize basic blocks with changing right-hand sides.

Tradeoff Looking at Figure 6, one can see a clear tradeoff 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 cross-over 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.

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 [18]. 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 3.1 (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.*

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 can 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 $R\Pi_F$.[11] This simple observation is critical for feature selection. Thus, if there are several different row selectors, COLUMBUS creates a separate QR factorization for each.

Tradeoff As summarized in Figure 5, QR's materialization cost is similar to importance sampling. In terms of execution time, Figure 6 shows that QR can be much faster than coresets: solving the

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

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

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 tradeoff 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 Tradeoff Space

Figure 6 shows the crossover points for the tradeoffs we described in this section for the CENSUS dataset. We describe why we assert that each of the following aspects affects the tradeoff 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 we need is even larger than the original data set, and if we force COLUMBUS to run on this large coreset, it would be more than 12x slower than QR. For tolerance 0.1, coreset is 1.82x slower than QR. We look into the breakdown of materialization time and execution time, and we find that materialization time contributes to more than 1.8x of this difference. When error tolerance is 0.5, Coreset+QR is 1.4x faster than QR. We ignore the curve for Lazy and Eager because they are insensitive to noises and are more than 1.2x slower than QR.

Sophistication One measure of sophistication is the number of features the analyst is considering. When the number of features in a basic block is much smaller than the data set size, coresets create much smaller but essentially equivalent data sets. 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 Figure6(b), as the number of features grows, CoreSet+QR slows down. With 161 features, the coreset will be larger than the original data set. 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 1.7x 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 data set, hence, the 1.7x 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. Figure6(c) shows the result. When the size of the overlapping feature sets is small, Lazy is 15x 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 2.5x 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, so mechanisms that reuse the computation may be optimal. As shown by Figure 6(e), when the number of threads is large, Lazy is 1.9x 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 11x faster than Lazy.

3.2 A Single, Non-linear Basic Block

We extend our methods to non-linear loss functions. The same tradeoffs from the previous section apply, but there are two additional techniques we can use. We describe them below.

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: stochastic gradient descent (SGD) [8, 10, 29], and iterative reweighted least squares (IRLS), which is implemented in R's generalized linear model package.[12] We describe an optimization, warmstarting, that applies to such models as well as to 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 we explore is one of the most popular, called *the Alternating Direction Method of Multipliers (ADMM)* [13], 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:

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

The constant $\rho \in (0, 2)$ is a step size parameter that we set by a grid search over 5 values.

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

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

(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 iteration. 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,

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

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 5. 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.

Tradeoffs In COLUMBUS, ADMM is our default solver for non-linear 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 hyper-parameters, like ρ above or regularization parameters; in this case, ADMM enables opportunities for high degrees of sharing. We cover these more complex cases in the full version of this paper.

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: (1) We downsample the data, learn a model on the sample, and then train a model on the original data. (2) We perform stepwise removal of a feature in feature selection, and the "parent" model with all features is already trained. (3) 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 13x 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 we need to do across blocks: (1) We need to decide on how coarse or fine to make a basic block, and (2) we need to execute the sequence of basic blocks across the backend.

	Data Set			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	

Figure 7: Dataset and Program Statistics. LS refers to Least Sqaures. LR refers to Logistic Regression.

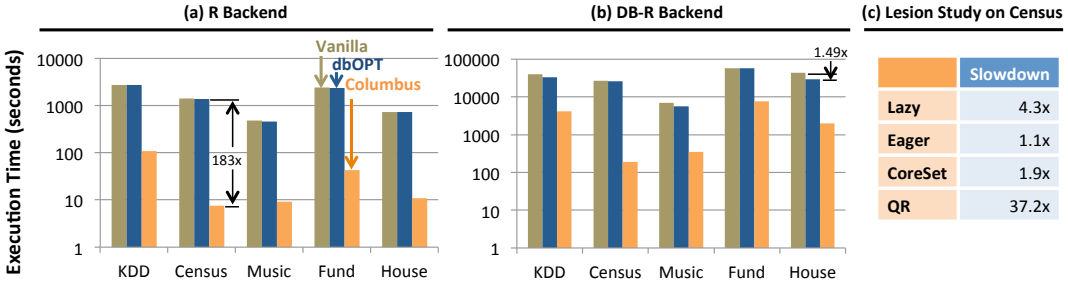

Figure 8: End-to-end Performance of COLUMBUS. All approaches return a loss within 1% optimal loss.

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, for example, merged into a single basic block. In COLUMBUS, we greedily improve the cost using the obvious estimates from Figure 5. The problem of deciding the optimal partitioning of many feature sets is NP-hard by a reduction to WEIGHTEDSETCOVER, which we explain in the full version of the paper. The intuition is clear, as one must cover all 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.

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 tradeoff 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). Since this particular set of optimizations did not have a dramatic effect on the runtime for any of our data sets, we report them only in the full version of this paper.

4 Experiments

Using the materialization tradeoffs 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 tradeoff 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 data sets: (1) **Census**, (2) **House**, (3) **KDD**, (4) **Music**, and (5) **Fund**.[13] These data sets have different sizes, and we show the statistics in Figure 7. We categorize them by the number of features in each data set.

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 data sets, 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 (that selects a few sparse features), manual intervention (moving around the feature space), and heavy use of cross-validation techniques.[14] **Census** is a dataset for the task of predicting 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.[15] This is the reason that we select this task for our running example. **Music** is similar to **Census**, and both programs contain both linear models (least squares) and non-linear 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*, 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,

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

[14] The **KDD** program contains six basic blocks, each of which is a 10-fold cross-validation. These 6 different basic blocks work on non-overlappings set of features specified by the user manually.

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

244 GB RAM, and 2x120GB SSD and runs Ubuntu 12.04.[16]

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 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 8 shows the result of running these systems over all five data sets 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 data sets. This is because for data sets that contain fewer rows and more columns, QR dominates CoreSet-based approaches, as described in the previous Section. One reason that 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 tradeoff space, compare the efficiency of dbOPT with 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 1.5x 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. While 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 tradeoff 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 tradeoffs that we identified affect the efficiency of Columbus. In Section 3, we designed experiments to understand the tradeoff 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 cross-over 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.

[16]We also run experiments on other dedicated machines. The tradeoff space is similar to what we report in this paper.

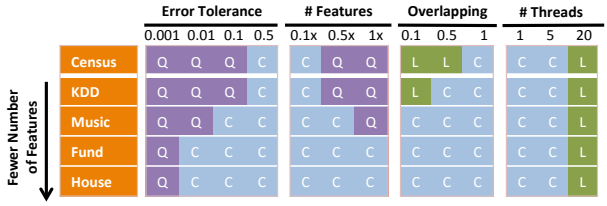

Figure 9: Robustness of Materialization Tradeoffs Across Datasets. For each parameter setting (one column in the table), we report the materialization strategy that has the fastest execution time given the parameter setting. Q refers to QR, C refers to CoreSet+QR, and L refers to Lazy. The protocol is the same as 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 8(c) in main memory on Census. We see that Lazy, QR, and CoreSet all have significant impacts on quality, ranging from 1.9x to 37x. This means that if we drop any of them from Columbus, one would expect a 1.9x to 37x 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 (1.5x slowdown).

We validate our claim that the high-level principles of the tradeoffs remain the same across datasets, but we contend that the tradeoff points change across data sets. Thus, our work provides a guideline about these tradeoffs, 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 9 shows that across different data sets, the same pattern holds, but with different crossover points. Consider the *error tolerance*. On all data sets, 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, while on Music, only for the lowest two error tolerance is QR faster than CoreSet+QR. While on Fund and House, for all error tolerances except the lowest one, CoreSet+QR is faster than QR. Thus, the cross-over point changes.

4.4 Non-Linear Basic Blocks with ADMM

Columbus uses ADMM as the default non-linear solver, which requires that one solves a least-squares problem that we studied in linear basic blocks. Compared with linear basic blocks, one key twist with ADMM is that it is iterative–thus, it has an additional parameter, the number of iterations to run. We validate that tradeoffs similar to the linear case still apply to non-linear basic blocks, and we describe how convergence impacts the tradeoff space. For each data set, 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 2.24x 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 3.8x faster than Lazy. This is not surprising based on our study for linear cases–by running more iterations, the opportunities for reuse increase. We expect an even larger speedup if we run more iterations.

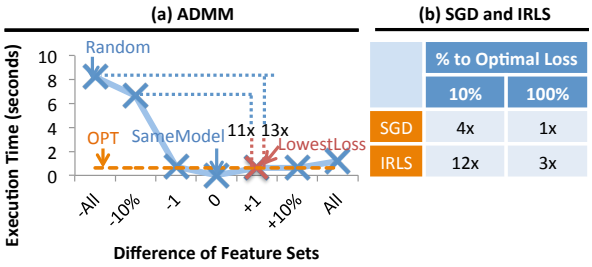

Figure 10: (a) The Impact of Model Caching for ADMM. The execution time is measured by the time needed to converge to 1% optimal loss. (b) The Speed-up of Model Caching. Different iterative approaches are compared to a given loss with a random initial model.

Warmstart Using Model Caching We validate the model-caching tradeoff for different iterative models, including SGD, ADMM, and IRLS. We use logistic regression as an example throughout (as it is the most common non-linear 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}, $F^{+10\%}$, F^{+All}. Where F^{-All} mimics the cold start problem when the analyst has not trained any models for this data set, we initialize with a random data point (this is RANDOM); F^{+All} means that the analyst has a "super parent" model trained for all features and has selected some features for warmstart; $F^{-10\%}$ (resp. $F^{+10\%}$) is when we warmstart with a model different from F by removing (resp. adding) $10\%|F|$ of randomly picked features; F^{+1} and F^{-1} mimic models generated by StepDrop or StepAdd, which are 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 data set, 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 2x of the optimal loss for F. Our heuristic, 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 10(a) shows the result. We see that LOWESTLOSS is 13x faster than RANDOM. Also, if we disable our heuristic and just pick a random feature set to use for warmstart, we could be 11x slower in the worst case. Compared with OPT, LOWESTLOSS is 1.03x 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 with RANDOM. As shown in Figure 10(b), we see that both SGD and IRLS are able to take advantage of warmstarting to speed up their convergence by up to 12x.

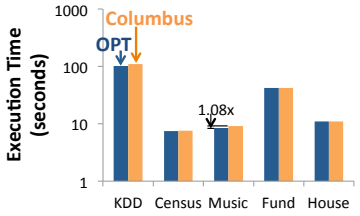

Figure 11: Greedy versus optimal optimizer

4.5 Multiblock Optimization

We validate that our greedy optimizer for multi-block optimization has comparable performance to the optimal. 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.[17] We call the optimal physical plan OPT and show the result in Figure 11.

As shown in Figure 11, 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.

5.1 Feature Selection

Algorithms for feature selection have been studied in the statistical and machine learning literature for decades [12, 20–22, 24]. 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 the features are used for. Wrappers are meta-algorithms that score feature sets using a statistical model. Embedded methods wire feature selection into a statistical model [21]. Our conversations with analysts revealed that feature selection is often a data-rich process with the analyst in the loop, not 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 [3]. Singh et al. parallelize forward selection for logistic regression on MapReduce/Hadoop [30]. In contrast, our focus is on building a generic framework for the feature selection processes, rather than for specific algorithms and platforms.

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

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, which integrate specific algorithms with an RDBMS [3, 23]. Similar efforts exist for other data platforms [1]. 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 [4], IBM's SystemML [17], SAP HANA [5], and the RIOT project [32]. 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 et al. [28], who studied how to efficiently organize *multidimensional arrays* in an RDBMS. Since then, there has been a recent resurgence in arrays as first-class citizens [14, 15, 31]. For example, Stonebraker et al. [31] recently envisioned the idea of using carefully optimized C++ code, e.g., ScaLAPACK, in array databases for matrix calculations. Our Columbus system is complementary to these efforts, as we focus on how to optimize the execution of multiple operations to facilitate reuse. The materialization tradeoffs we explore are (largely) orthogonal to these lower-level tradeoffs.

There has been an intense effort to scale up individual linear algebra operations in data processing systems [9, 16, 32]. Constantine et al. [16] propose a distributed algorithm to calculate QR decomposition using MapReduce, while ScaLAPACK [9] uses a distributed main memory system to scale up linear algebra. The RIOT [32] 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 [32] and SystemML [17]. RIOT-DB optimizes I/O by rearranging page accesses for specific loop constructs in an R program [32]. SystemML [17] converts R-style programs to workflows of MapReduce jobs. They describe an optimization called piggybacking, which enables sharing of data access by jobs that follow each other.

In a similar spirit, declarative machine learning systems, e.g., MLBase [26], provide the end users a high-level language to specify a machine learning task. Compared with 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 tradeoffs that we study can be applied in declarative machine learning systems.

6 Conclusion

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 last two 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.

Acknowledgment. We gratefully acknowledge the support of Oracle, the Office of Naval Research under awards No. N000141210041 and No. N000141310129, the National Science Foundation CAREER Award under No. IIS-1054009, Sloan Research Fellowship, and American Family Insurance. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of ONR, NSF, or the US government.

References

[1] Apache Mahout. `mahout.apache.org`.

[2] Feature Selection and Dimension Reduction Techniques in SAS. `nesug.org/Proceedings/nesug11/sa/sa08.pdf`.

[3] Oracle Data Mining. `oracle.com/technetwork/database/options/advanced-analytics/odm`.

[4] Oracle R Enterprise. `docs.oracle.com/cd/E27988_01/doc/doc.112/e26499.pdf`.

[5] SAP HANA and R. `help.sap.com/hana/hana_dev_r_emb_en.pdf`.

[6] SAS Report on Analytics. `sas.com/reg/wp/corp/23876`.

[7] Variable Selection in the Credit Card Industry. `nesug.org/proceedings/nesug06/an/da23.pdf`.

[8] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[9] L. S. Blackford and et al. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. In *SuperComputing*, 1996.

[10] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *NIPS*, 2007.

[11] C. Boutsidis and et al. Near-optimal coresets for least-squares regression. *IEEE Transactions on Information Theory*, 2013.

[12] D. Boyce and et al. *Optimal Subset Selection*. Springer, 1974.

[13] S. Boyd and et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 2011.

[14] P. G. Brown. Overview of sciDB: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[15] J. Cohen and et al. MAD skills: New analysis practices for big data. *PVLDB*, 2009.

[16] P. G. Constantine and D. F. Gleich. Tall and skinny qr factorizations in mapreduce architectures. In *MapReduce*, 2011.

[17] A. Ghoting and et al. SystemML: Declarative machine learning on MapReduce. In *ICDE*, 2011.

[18] G. Golub. Numerical methods for solving linear least squares problems. *Numerische Mathematik*, 1965.

[19] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[20] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *JMLR*, 2003.

[21] I. Guyon and et al. *Feature Extraction: Foundations and Applications*. New York: Springer-Verlag, 2001.

[22] T. Hastie and et al. *The Elements of Statistical Learning: Data mining, inference, and prediction*. Springer, 2001.

[23] J. Hellerstein and et al. The MADlib analytics library or MAD skills, the SQL. In *PVLDB*, 2012.

[24] G. H. John and et al. Irrelevant features and the subset selection problem. In *ICML*, 1994.

[25] S. Kandel and et al. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 2012.

[26] T. Kraska and et al. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[27] M. Langberg and L. J. Schulman. Universal ϵ-approximators for integrals. In *SODA*, 2010.

[28] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *ICDE*, 1994.

[29] S. Shalev-Shwartz and N. Srebro. SVM optimization: Inverse dependence on training set size. In *ICML*, 2008.

[30] S. Singh and et al. Parallel large scale feature selection for logistic regression. In *SDM*, 2009.

[31] M. Stonebraker and et al. Intel "big data" science and technology center vision and execution plan. *SIGMOD Rec.*, 2013.

[32] Y. Zhang and et al. I/O-efficient statistical computing with RIOT. In *ICDE*, 2010.

A Batching Optimization

Recall that the parser of COLUMBUS outputs a directed acyclic graph (DAG) of operations. Denote this DAG by $G(V, E)$, where E imposes a partial order (based on data flow dependency) $\leq$ on the nodes (operations) in V. The batching optimization packs the nodes of the DAG $G(V, E)$ to output a *chain* of nodes $G'(V', E')$. In general, we need to consider a memory-constrained setting, where we cannot batch nodes indefinitely because each node (operation) has a certain memory footprint, typically linear in the size of its input feature set. More precisely, we are given a system memory budget M, and memory footprints of each node in G as $f : V \rightarrow \mathbf{R}^+$ (we assume $f(v) \leq M, \forall v \in V$). Our goal is now to pack the nodes of G into a sequence of batches such that we minimize the number of batches.

Theorem A.1. *Given $G(V, E)$, a memory limit M, and the memory footprints of all nodes in V, batching optimization is* NP*-Hard in the width of the partial order imposed by E on V.*

Proof: The problem can easily be reduced from BINPACKING. Given a bin size B, objects to pack $\{x_i\}$ with sizes $\{s_i\}$, we construct an instance of our problem as follows: set $M = B$. Create $G(V, E)$ such that V corresponds exactly to $\{x_i\}$, and $E = \phi$. Set the memory footprints of the nodes in V as the the corresponding sizes in $\{s_i\}$ The problem instances are now equivalent, and thus an optimal solution to our instance will be an optimal solution to the given instance of BINPACKING. □

First-Fit Heuristic: Consider nodes from V that form an *antichain*, i.e., no pair among them is comparable with respect to the partial order imposed by E on V. There are no dependencies among the nodes in an antichain and so they can all go into the same batch. Note that the length of the longest such antichain is by definition the width of the partial order on G. Our heuristic algorithm is given in Algorithm 1. Since E imposes a partial order on V, our DAG $G(V, E)$ we will have at least as many antichains as the height of this partial order. We simply apply Algorithm 1 to each antichain individually. We construct the antichains in time $O(|V| + |E|)$ using a topological sort of $G(V, E)$. Since the antichains need not be unique, we use the following technique to obtain a unqiue set of antichains – We simply assign a node in V to an antichain k if k is the earliest possible occurrence of that node in any topological order on V (i.e., how "far" is that node from the "start" of the DAG).

B Materialization Optimization

We now state and prove the hardness of the materialization optimization. We first recall the setup of Eager vs Lazy optimization.

Problem Statement Fix dataset $R(\mathcal{F})$. We are given a set of accesses of subsets of $\mathcal{F}$, labeled $\mathcal{S} = \{F_1, \ldots, F_N\}$, where $F_i \subseteq \mathcal{F}, \forall i = 1 \ldots N$. 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 $\mathcal{S}' \subseteq \mathcal{S}$ 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:

Algorithm 1: First-Fit Heuristic used for Batching under Memory Constraints

Data: Antichain of nodes $W \subseteq V$, Memory footprints $f : W \to \mathbf{R}^+$, Memory limit M
Result: Number of batches X, and allocation $a : W \to \{0, 1, \ldots, X\}$
$X \leftarrow 1$
$a(w) \leftarrow 0, \forall w \in W$ //initialization
foreach $w \in W$ **do**
  for $x = 1$ *to* X **do**
    if $f(w) + \sum_{u \in W : a(u) = x} f(u) \leq M$ **then**
      $a(w) \leftarrow x$
      break
    end
  end
  //w is still not alloted to any batch
  if $a(w) = 0$ **then**
    $X \leftarrow X + 1$
    $a(w) = X$
  end
end

MatOpt: $\min_{\mathcal{S}' \subseteq \mathcal{S}}$ Cost($\mathcal{S}'$),

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

In other words, we materialize the feature sets in $\mathcal{S}'$ and use the "thinnest" among them (or $R(\mathcal{F})$) to serve each feature set.

Theorem B.1. MatOpt *is* NP*-Hard in the width of the lattice on* $\mathcal{S}$*, the set of input feature sets.*

Proof: We prove by reduction from the NP-Hard problem of SetCover-EC We are given a universe of elements $\mathcal{F}$, and a family $\mathcal{S}$ of subsets of $\mathcal{F}$ that covers $\mathcal{F}$, i.e., $\cup_{s \in \mathcal{S}} s = \mathcal{F}$, such that $s \in \mathcal{S}$ has equal cardinality L. The optimization version of SetCover-EC asks for a sub family $S' \subseteq \mathcal{S}$ of minimum $|S'|$ that covers $\mathcal{F}$. We reduce an instance of SetCover-EC to an instance of our problem as follows:

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

Firstly, make CPU costs negligible so that data access (scan) times are linear only in the number of features in the dataset. It is given by $Scan(F_j) = \alpha |F_j|$, where α is a free parameter. Note that it does not depend on F_i, which is the input for computations.

Secondly, materialization costs are also linear, and given by $Store(F_i) = Scan(\mathcal{F}) + Write(F_i) = \alpha|\mathcal{F}| + \beta|F_i|$, where β is also a free parameter. It is reasonable to assume that only $\mathcal{F}$ is used for materialization since we can always make $\mathcal{S}$ an antichain by dropping a set contained in another within $\mathcal{S}$ in the given SetCover-EC instance.

Thirdly, 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 since their individual materialization costs are higher than just accessing $\mathcal{F}$, or any member of $\mathcal{S}$. So, we set the parameters of our problem instance in such a way that our optimal solution will materialize a subset of $\mathcal{S}$ 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 3 free parameters – α, β, and γ. The given instance of SetCover-EC gives us $\mathcal{F}$ (the universe of features), $\mathcal{S}$ (the family of subsets of $\mathcal{F}$), N ($= |\mathcal{S}|$), and L (cardinality of each member of $\mathcal{S}$). Our problem's objective function ($S' \subseteq \mathcal{S}$)is given by:

$$
\begin{aligned}
Cost(S') = & \sum_{s \in S'} (Store(s) + \gamma Scan(F_s)) + \sum_{s \in \mathcal{S} \setminus S'} \gamma Scan(\mathcal{F}) \\
& + \sum_{x \in X : \exists s \in S', x \subseteq s} Scan(F_s) + \sum_{x \in X : \forall s \in S', x \not\subseteq s} Scan(\mathcal{F}) \\
& = |S'|((\alpha|\mathcal{F}| + \beta L) + \gamma\alpha L) + (|\mathcal{S}| - |S'|)\gamma\alpha|\mathcal{F}| \\
& + \alpha L(\#x_j \textit{ covered by } S') + \alpha|\mathcal{F}|(\#x_j \textit{ not covered by } S') \\
& = |\mathcal{S}|\gamma\alpha|\mathcal{F}| \ + \ |S'|(-\gamma\alpha|\mathcal{F}| + \alpha|\mathcal{F}| + \beta L + \gamma\alpha L) \\
& + \alpha L|X| \ + \ \alpha(|\mathcal{F}| - L)(\#x_j \textit{ not covered by } S')
\end{aligned}
$$

The terms $|\mathcal{S}|\gamma\alpha|\mathcal{F}|$ and $\alpha L|X|$ above are constants independent of S', and so we can drop them from the optimization. Thus, we rewrite as :

$$Cost(S') = |S'|(\alpha|\mathcal{F}| + \beta L - \gamma\alpha(|\mathcal{F}| - L)) + \alpha(|\mathcal{F}| - L)(\#x_j \textit{ not covered by } S')$$

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

Firstly, we want the coefficient of $|S'|$ above (call it p) to be positive, i.e., $p = \alpha|\mathcal{F}| + \beta L - \gamma\alpha(|\mathcal{F}| - L) > 0$. 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 s.t. $\beta > \alpha(\gamma(|\mathcal{F}| - L) - |\mathcal{F}|)/L$. Call the RHS d (say),

i.e., $d := \alpha(\gamma(|\mathcal{F}| - L) - |\mathcal{F}|)/L$, and $\beta > d$.

Secondly, we need to see if the maximum value of the first term (which is Np, when $S' = \mathcal{S}$) 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 $Np < \alpha(|\mathcal{F}| - L)$. Expanding p, it becomes if $N(\alpha|\mathcal{F}| + \beta L - \gamma\alpha(|\mathcal{F}| - L)) < \alpha(|\mathcal{F}| - L)$. Rearranging in terms of β, it becomes if $\beta < \alpha(\gamma(|\mathcal{F}| - L) - |\mathcal{F}|)/L + \alpha(|\mathcal{F}| - L)/(LN)$, i.e., if $\beta < d + \alpha(|\mathcal{F}| - L)/(LN)$. Since the second term is positive (as $|\mathcal{F}| > L$), we only have to choose β such that $d < \beta < d + \alpha(|\mathcal{F}| - L)/(LN)$.

Since not covering even one x_j raises the cost more than choosing all the sets, a minimum cost solution to our problem instance will always cover all x_j, and will minimize among the chosen subsets of S. Thus, an optimal solution to our problem instance as constructed above will be an optimal solution to the given instance of SetCover-EC. □

C Overall Materialization Optimization

Following the results of Section B, it is clear that choosing the optimal materialization strategy (irrespective of batching) is also NP-Hard, in general.

D Scheduling Optimization for Multiple Threads

TODO: Ce

E Justification of Cost Model

We assume the following parameters given a basic block $B = (A, b, \bar{F}, \bar{R})$.

1. N: number of rows in A
2. D: number of colums in A
3. We assume that there are r distinct elements in $\bar{R}$, each of them corresponds to f distinct elements in $\bar{F}$.
4. We assume each $F \in \bar{F}$ has the number of features d.
5. We assume each $R \in \bar{R}$ has the number of features n.
6. Let $\cup_i F_i = k_f d$, and $\cup_i R_i = k_r n$.

For simplicity, we define the notation $[...]_k$ which means that there are k runs with the same complexity that can be executed in parallel.

The most detailed cost model is as follows (Simplify later).

1. Lazy
   (a) MAT: 0

(b) EXE: $[DN + k_f dn + N + n + k_f^2 d^2 n + k_f dn]_r + [\frac{2d^3}{3}]_{rf}$

2. Eager

(a) MAT: $[DN + k_r k_f dn + N + k_r n]_1$

(b) EXE: $[k_f k_r dn + k_f dn + k_r n + n + k_f^2 d^2 n + k_f dn]_r + [\frac{2d^3}{3}]_{rf}$

3. QR

(a) MAT: $[DN + k_f dn + N + n + 2k_f^2 d^2 n + k_f dn]_r$

(b) EXE: $[d^2/2]_{rf}$

4. CoreSet

(a) MAT: $[DN+k_f dn+N+n+k_f^2 d^2 n+\frac{2k_f^3 d^3}{3}+k_f^2 d^2 n+2k_f dn+n+k_f dn+2k_f d^2/\epsilon^2+n+2d/\epsilon^2]_r$

(b) EXE: $[2k_f^2 d^3/\epsilon^2 + 2k_f d^2/\epsilon^2]_r + [\frac{2d^3}{3}]_{rf}$

5. NaïveSampling

(a) MAT: $[n + DN + \rho Dn + N + \rho n]_r$

(b) EXE: $[\rho Dn + \rho k_f dn + k_f^2 d^2 n + k_f dn]_r + [\frac{2d^3}{3}]_{rf}$

6. Bootstrap

(a) MAT: $[n + n + DN + \rho Dn + N + \rho n]_{Kr}$

(b) EXE: $[\rho Dn + \rho k_f dn + \rho k_f^2 d^2 n + \rho k_f dn]_{Kr} + [\frac{2d^3}{3}]_{Krf}$

We make the following approximation. Let $D \sim D + 1$ and $k_f d \sim k_f d + 1$.

1. Lazy

(a) MAT: 0

(b) EXE: $[DN + k_f dn + k_f^2 d^2 n]_r + [\frac{2d^3}{3}]_{rf}$

2. Eager

(a) MAT: $[DN + k_r k_f dn]_1$

(b) EXE: $[k_f k_r dn + k_f dn + k_f^2 d^2 n]_r + [\frac{2d^3}{3}]_{rf}$

3. QR

(a) MAT: $[DN + k_f dn + 2k_f^2 d^2 n]_r$

(b) EXE: $[d^2/2]_{rf}$

4. CoreSet

(a) MAT: $[DN + 2k_f dn + 2k_f^2 d^2 n + \frac{2k_f^3 d^3}{3} + n + 2k_f d^2/\epsilon^2]_r$

(b) EXE: $[2k_f^2 d^3/\epsilon^2]_r + [\frac{2d^3}{3}]_{rf}$

Algorithm 2: Lazy using ROPs

Data: Data Matrix A and b. Set of Feature Sets $\bar{F}$, Set of Example Sets $\bar{R}$, Error Tolerance ϵ.

Procedure `Materialization`(A, b, $\bar{F}$, $\bar{R}$, ϵ)

$M \leftarrow$ **list**() // Materialized Information

return M

Procedure `Execution`(M, A, b, $\bar{F}$, $\bar{R}$, ϵ)

$\tilde{R} = \{R_i \in \bar{R}\}$

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

5. NaïveSampling
   (a) MAT: $[DN + \rho Dn]_r$
   (b) EXE: $[\rho Dn + \rho k_f^2 d^2 n]_r + [\frac{2d^3}{3}]_{rf}$
6. Bootstrap
   (a) MAT: $[DN + \rho Dn]_{Kr}$
   (b) EXE: $[\rho Dn + \rho k_f^2 d^2 n]_{Kr} + [\frac{2d^3}{3}]_{Krf}$

We use the following cost of ROPs. ($A \in \mathbb{R}^{N \times d}$, $B, R \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$.

1. **solve**(B, b): $\frac{2d^3}{3}$. R calls DGESV in LAPACK, which has this complexity according to `http://www.netlib.org/lapack/lug/node71.html`.
2. **solve**(B). Same to above for matrix inverse.
3. **qr**(A). $2d^2N$. R calls DQRDC in LINPACK, which uses householder transformation to calculate QR. Complexity is obtained from pseudo-code in `http://tel.archives-ouvertes.fr/docs/00/83/33/56/PDF/VD2_KHABOU_AMAL_11022013.pdf`.
4. **backsolve**(R, b) R calls dtrsm in BLAS, which is a level-3 operator. So we implemented our own version of **backsolve** with complexity $O(d^2/2)$.

Algorithm 3: Eager using ROPs

Data: Data Matrix A and b. Set of Feature Sets $\bar{F}$, Set of Example Sets $\bar{R}$ Error Tolerance ϵ.

Procedure `Materialization`(A, b, $\bar{F}$, $\bar{R}$, ϵ)

$F \leftarrow UNION(\bar{F})$

$R \leftarrow UNION(\bar{R})$

$M = \textbf{list}(\text{"}A\text{"} = A[R, F], \text{"}b\text{"} = b[R])$ // $DN + k_r k_f dn + N + k_r n$

return M

Procedure `Execution`(M, A, b, $\bar{F}$, $\bar{R}$, ϵ)

$\tilde{R} = \{R_i \in \bar{R}\}$

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**

    $rs \leftarrow \textbf{solve}(AA[F_i, F_i], bb[F_i])$ // $2d^3/3$

  end

end

Algorithm 4: QR using ROPs

Data: Data Matrix A and b. Set of Feature Sets $\bar{F}$, Set of Example Sets $\bar{R}$ Error Tolerance ϵ.

Procedure `Materialization`(A, b, $\bar{F}$, $\bar{R}$, ϵ)

$M = \textbf{list}()$
$\tilde{R} = \{R_i \in \bar{R}\}$
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, $\bar{F}$, $\bar{R}$, ϵ)

$\tilde{R} = \{R_i \in \bar{R}\}$
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

Algorithm 5: 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 ϵ.

Procedure `Materialization(`A, b, $\bar{F}$, $\bar{R}$, ϵ`)`

$M = \textbf{list}()$
$\tilde{R} = \{R_i \in \bar{R}\}$
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, $\bar{F}$, $\bar{R}$, ϵ`)`

$\tilde{R} = \{R_i \in \bar{R}\}$
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

Algorithm 6: Naïve Sampling using ROPs

Data: Data Matrix A and b. Set of Feature Sets $\bar{F}$, Set of Example Sets $\bar{R}$, Error Tolerance ϵ, selectivity of sampling ρ

Procedure `Materialization`(A, b, $\bar{F}$, $\bar{R}$, ϵ, ρ)

$M \leftarrow$ **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, $\bar{F}$, $\bar{R}$, ϵ)

$\tilde{R} = \{R_i \in \bar{R}\}$
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

Algorithm 7: Bootstrap using ROPs

Data: Data Matrix A and b. Set of Feature Sets $\bar{F}$, Set of Example Sets $\bar{R}$, Error Tolerance ϵ, selectivity of sampling ρ, number of bags of bootstrap K

Procedure `Materialization`(A, b, $\bar{F}$, $\bar{R}$, ϵ, ρ, K)

$M \leftarrow$ **list**()
$\tilde{R} = \{R_i \in \bar{R}\}$
for $i \in 1 : K$ **do**
  for $R \in \tilde{R}$ **do**
    $sampled \leftarrow$ **sample**$(1 : |R|, \rho|R|)$ // n
    $sampled \leftarrow$ **sample**$(sampled, |R|)$ // n
    $MA = A[sampled,]$ // $DN + \rho Dn$ ($sampled$ has only ρn distinct rows, so this subselection is cheaper by maintain a count.)
    $Mb = b[sampled]$ // $N + \rho n$
    $M[(R, i)] =$ **list**("A" $= MA$,"b" $= Mb$)
  end
end
return M

Procedure `Execution`(M, A, b, $\bar{F}$, $\bar{R}$, ϵ)

$\tilde{R} = \{R_i \in \bar{R}\}$
for $R \in \tilde{R}$ **do**
  for $i \in 1 : K$ **do**
    $\tilde{F} = \{F_i \in \bar{F} : R_i = R\}$
    $F = UNION(\tilde{F})$
    $MA \leftarrow M[(R, i)]\$A[, F]$ // $\rho Dn + \rho k_f dn$
    $Mb \leftarrow M[(R, i)]\$b$
    $AA \leftarrow MA^T\% * \%MA$ // $\rho k_f^2 d^2 n$
    $bb \leftarrow MA^T\% * \%Mb$ // $\rho k_f dn$
    for $F_i \in F$ **do**
      $rs \leftarrow$ **solve**$(AA[F_i, F_i], bb[F_i])$ // $2d^3/3$
    end
  end
end