Indexing Cost Sensitive Prediction

ABSTRACT

Predictive models are often used for real-time decision making. However, typical machine learning techniques ignore feature evaluation cost, and focus solely on the accuracy of the machine learning models obtained utilizing all the features available. We develop algorithms and indexes to support cost-sensitive prediction, i.e., making decisions using machine learning models taking feature evaluation cost into account. Given an item and a online computation cost (i.e., time) budget, we present two approaches to return an appropriately chosen machine learning model that will run within the specified time on the given item. The first approach returns the optimal machine learning model, i.e., one with the highest accuracy, that runs within the specified time, but requires significant up-front precomputation time. The second approach returns a possibly suboptimal machine learning model, but requires little up-front precomputation time. We study these two algorithms in detail and characterize the scenarios (using real and synthetic data) in which each performs well. Unlike prior work that focuses on a narrow domain or a specific algorithm, our techniques are *very general*: they apply to any cost-sensitive prediction scenario on any machine learning algorithm.

1. INTRODUCTION

Predictive models are ubiquitous in real-world applications: ad-networks predict which ad the user will most likely click on based on the user's web history, Netflix uses a user's viewing and voting history to pick movies to recommend, and content moderation services decide if an uploaded image is appropriate for young children. In these applications, the predictive model needs to process the input data and make a prediction within a bounded amount of time, or risk losing user engagement or revenue [4, 13, 34].

Unfortunately, traditional feature-based classifiers take a one-model-fits all approach when placed in production, behaving the same way regardless of input size or time budget. From the classifier's perspective, the features used to represent an input item have already been computed, and this computation process is external to the core task of classification. This approach isolates and simplifies the core task of machine learning, allowing theorists to focus on tasks like quick learning convergence and accuracy. But it leaves out many important aspects of production systems can be just as important as accuracy, such as tunable prediction speed.

In reality, the cost of computing features can easily dominate prediction time. For example, a content moderation application may use a support-vector machine to detect inappropriate images. At runtime, SVMs need only perform a single dot-product between a feature vector and pre-computed weight vector. But computing the feature vector may require several scans of an image which may take longer than an alotted time budget.

If feature computation is the dominating cost factor, one might intuitively accomodate time constraints by computing and using only a subset of features available. But selecting which subset to use at runtime—and guaranteeing that a model is available that was trained on that subet—is made challenging by a number of factors. Features vary in *predictive power*, e.g. skin tone colors might more accurately predict inappropriate images than image size. They also vary in *prediction cost*, e.g. looking up the image size is much faster than computing a color histogram over an entire image. This cost also varies with respect to input size—the size feature may be $O(1)$, stored in metadata, while the histogram may be $O(r \times c)$. Finally for any n features, 2^n distinct subsets are possible, each with their aggregate predictive power and cost, and each potentially requiring its own custom training run. As the number of potential features grows large, training a model for every possible subset is clearly prohibitively costly. All of these reasons highlight why deploying real-time prediction, while extremely important, is a particularly challenging problem.

Existing strategies of approaching this problem (see Section 6) tend to be either tightly coupled to a particular prediction task or to a particular mathematical model. While these approaches work for a particular problem, they are narrow in their applicability: if the domain (features) change or the machine learning model is swapped for new one (e.g., SVM for AdaBoost), the approach will no longer work.

In this paper, we develop a framework for *cost-sensitive real-time classification* as a *wrapper* over "off-the-shelf" feature-based classifiers. That is, given an item that needs to be classified or categorized in real time and a cost (i.e., time) budget for feature evaluation, our goal is to *identify features to compute in real time that are within the budget, identify the appropriate machine learning model that has been learned in advance, and apply the model on the extracted features.*

We take a systems approach by decoupling the problem of cost-sensitive prediction from the problem of machine learning in general. We present an algorithm for cost sensitive prediction that operates on any feature-based machine learning algorithm as a black box. The few assumptions it makes reasonably transfer between

Proceedings of the VLDB Endowment, Vol. 7, No. 4

different feature sets and algorithms (and are justified herein). This decoupled approach is attractive for the same reason that machine learning literature did not originally address such problems: it segments reasoning about the core tasks of learning and prediction from system concerns about operationalizing and scaling. Additionally, encapsulating the details of classification as we do ensures advances in machine learning algorithms and feature engineering can be integrated without change to the cost-sensitivity apparatus.

Thus, our focus in this paper is on *systems issues underlying this wrapper-based approach*, i.e., on intelligent indexing and pruning techniques to enable rapid online decision making and not on the machine learning algorithms themselves. Our contribution is two approaches to this problem as well as new techniques to mitigate the challenges of each approach. These two approaches represents two ends of a continuum of approaches to tackle the problem of model-agnostic cost sensitivity:

- Our POLY-DOM approach yields optimal solutions but requires significant offline pre-computation,
- Our GREEDY approach yields relatively good solutions but does not require significant offline pre-computation.

First, consider the GREEDY approach: GREEDY, and its two subvariants GREEDY-ACC and GREEDY-COST (described in Section 4), are all simple but effective techniques adapted from prior work by Xu *et al.* [45], wherein the technique only applied to a sub-class of SVMs [15]. Here, we generalize the techniques to apply to any machine learning classiciation algorithm as a black box. GREEDY is a "quick and dirty" technique that requires little precomputation, storage and retrieval, and works well in many settings.

Then, consider our POLY-DOM approach, which is necessary whenever accuracy is paramount, a typical scenario in critical applications like credit card fraud detection, system performance monitoring, and ad-serving systems. In this approach, we conceptually store, for each input size, a skyline of predictive models along the axes of total real-time computation cost vs. accuracy. Then, given an input of that size, we can simply pick the predictive model along the skyline within the total real-time computation cost budget, and then get the best possible accuracy.

However, there are many difficulties in implementing this skyline-based approach:

- Computing the skyline in a naive fashion requires us to compute for all $2^{|\mathcal{F}|}$, subsets of features (where $\mathcal{F}$ is the set of all features) the best machine learning algorithm for that set, and the total real-time computation time (or cost). If the number of features is large, say in the 100s or the 1000s, computing the skyline is impossible to do, even with an unlimited amount of time offline. How do we intelligently reduce the amount of precomputation required to find the skyline?
- The skyline, once computed, will require a lot of storage. How should this skyline be stored, and what index structures should we use to allow efficient retrieval of individual models on the skyline?
- Computing and storing the skyline for each input size is simply infeasible: an image, for instance, can vary between 0 to 70 Billion Pixels (the size of the largest photo on earth [1]), we simply cannot store or precompute this much information. What can we do in such a case?

To deal with the challenges above, POLY-DOM use a dual-pronged solution, with two precomputation steps:

- *Feature Set Pruning:* We develop a number of pruning techniques that enable us to minimize the number of feature sets for which we need to learn machine learning algorithms. Our lattice pruning techniques are *provably correct under some very reasonable assumptions*, i.e., they do not discard any feature sets if those feature sets could be potentially optimal under certain input conditions. We find that our pruning techniques often allow us to prune up to 90% of the feature sets.
- *Polydom Index:* Once we gather the collection of feature sets, we develop an index structure that allows us to represent the models learned using the feature sets in such a way that enables us to perform *efficient retrieval of the optimal machine learning model given constraints on cost and input size*. This index structure relies on reasoning about polynomials that represent cost characteristics of feature sets as a function of input size.

Overall, our approach offers a systems perspective to an increasingly important topic in the deployment of machine learning systems. The higher-level goal is to isolate and develop the mechanisms for storage and delivery of cost-sensitive prediction without having to break the encapsulation barrier that should surround the fundamental machinery of machine learning. Our techniques could be deployed alongside the existing algorithms in a variety of real-time prediction scenarios, including:

- An ad system needs to balance between per-user ad customization and latency on the small scale, and allocate computational resources between low-value and high-value ad viewers on the aggregate scale.
- Cloud-based financial software needs to run predictive models on portfolios of dramatically different input size and value. A maximum latency on results may be required, but the best possible model for each time and input size pairing is financially advantageous.
- An autopilot system in an airplane has a limited time to respond to an error. Or more broadly, system performance monitors in a variety of industrial systems have fixed time to decide whether to alert a human operator of a failure.
- A mobile sensor has limited resources to decide if an error needs to be flagged and sent to the central controller.

In the rest of the paper, we will first formally present our problem variants in Section 2, then describe our two-pronged POLY-DOM solution in Section 3 and our "quick and dirty" GREEDY solution in Section 4, and finally present our experiments on both synthetic and real-world datasets in Section 5.

2. PROBLEM DESCRIPTION

We begin by describing some notation that will apply to the rest of the paper, and then we will present the formal statement of the problems that we study.

Our goal is to classify an item (e.g., image, video, text) x during real-time. We assume that the size of x, denoted $|x|$, would be represented using a single number or dimension n, e.g., number of words in the text. Our techniques also apply to the scenario when the size can be represented using a vector of dimensions: for example, (length, breadth), for an image; however, for ease of exposition, we focus on the single dimension scenario. The entire set of features we can evaluate on x is $\mathcal{F}$; each individual feature is denoted f_i, while a non-empty set of features is denoted F_j.

We assume that we have some training data, denoted $\mathcal{T}$, wherein every single feature $f_i \in \mathcal{F}$ is evaluated for each item. Since training is done offline, it is not unreasonable to expect that we have the ability to compute all the features on each item. In addition, we have some testing data, denoted $\mathcal{T}'$, where once again every single feature is evaluated for each item. We use this test data to estimate the accuracy of the machine learning models we discover offline.

Cost Function: We assume that evaluating a feature on an item x depends only on the feature that is being computed, and the size of

the item $|x|$. We denote the cost of computing f_i on x as: $c(f_i, |x|)$. We can estimate $c(f_i, n)$ during pre-processing time by running the subroutine corresponding to feature evaluation f_i on varying input sizes. Our resulting expression for $c(f_i, n)$ could either be a constant (if it takes a fixed amount of time to evaluate the feature, no matter the size), or could be a function of n, e.g., $3n^2 + 50$, if evaluating a feature depends on n.

Then, the cost of computing a set of features F_i on x can be computed as follows:

$$c(F_i, |x|) = \sum_{f \in F_i} c(f, |x|) \quad (1)$$

We assume that each feature is computed independently, in sequential order. Although there may be cases where multiple features can be computed together (e.g., multiple features can share scans over an image simultaneously), we expect that users provide the features as "black-box" subroutines and do not want to place additional burden by asking users to provide subroutines for combinations of features as well. That said, our techniques will equally well apply to the scenario when our cost model is more general than Equation 1, or if users have provided subroutines for generating multiple feature values simultaneously (e.g., extracting a word frequency vector from a text document).

Accuracy Function: We model the machine learning algorithm (e.g., SVM, decision tree, naive-bayes) as a black box function supplied by the user. This algorithm takes as input the entire training data $\mathcal{T}$, as well as a set of features F_i, and outputs the best model $\mathcal{M}(F_i)$ learned using the set of features F_i. We denote the accuracy of $\mathcal{M}(F_i)$ inferred on the testing data $\mathcal{T}'$ as $a(F_i)$, possibly using k-fold cross-validation. We assume that the training data is representative of the items classified online (as is typical), so that the accuracy of the model $\mathcal{M}(F_i)$ is still $a(F_i)$ online.

Note that we are implicitly assuming that the accuracy of the classification model only depends on the set of features inferred during test time, and not on the size of the item. This assumption is typically true in practice: whether or not an image needs to be flagged for moderation is independent of the size of the image.

Characterizing a Feature Set: Since we will be dealing often with sets of features at a time, we now describe what we mean by characterizing a feature set F_i. Overall, given F_i, as discussed above, we have a black box that returns

- a machine learning model learned on some or all the features in F_i, represented as $\mathcal{M}(F_i)$.
- $a(F_i)$, i.e., an estimate of the accuracy of the model $\mathcal{M}(F_i)$ on test data $\mathcal{T}'$.

In addition, we can estimate $c(F_i, n)$, i.e., the cost of extracting the features to apply the model at test time as a function of the size of the item n. Note that unlike the last two quantities, this quantity will be expressed in symbolic form. For example, $c(F_i, n)$ could be an expression like $3n^2 + 4n \log n + 7n$.

Characterizing a feature set F_i thus involves learning all three quantities above for F_i: $\mathcal{M}, a, c$. For the rest of the paper, we will operate on feature sets, implicitly assuming that a feature set is characterized by the best machine learning model for that feature set, an accuracy value for that model, and a cost function.

Problem Statements: The most general version of the problem is when c (i.e., the cost or time constraint) and the size n of an item x are not provided to us in advance:

PROBLEM 1 (PROB-GENERAL). *Given $\mathcal{F}, \mathcal{T}, \mathcal{T}'$ at preprocessing time, compute classification models and indexes such that the following task can be completed at real-time:*

- *Given $x, |x| = n$, and a cost constraint c at real time, identify a set $F_i \in \mathcal{F}' = \{F \subseteq \mathcal{F} \mid c(F, n) \leq c\}$ such that $\forall_{F \in \mathcal{F}'} \alpha \times a(F_i) \geq a(F)$ and return $\mathcal{M}(F_i)(x)$.*

That is, our goal is to build classification models and indexes such that given a new item at real time, we select a feature set F_i and the corresponding machine learning model $\mathcal{M}(F_i)$ that both obeys the cost constraint, and is within $\alpha \geq 1\times$ of the best accuracy among all feature sets that obey the cost constraint. The reason we care about $\alpha > 1$ is that, in contrast to a hard cost constraint, a slightly lower accuracy is often acceptable as long as the amount of computation required for computing, storing, and retrieving the appropriate models is manageable. We will consider $\alpha = 1$, i.e., the absolute best accuracy, as a special case; however, for most of the paper, we will consider the more general variants.

There are two special cases of the general problem that we will consider. The first special case considers the scenario when the input size is provided up-front, e.g., when Yelp fixes the size of profile images uploaded to the website that need to be moderated.

PROBLEM 2 (PROB-n_0-FIXED). *Given $\mathcal{F}, \mathcal{T}, \mathcal{T}'$ and a fixed n_0 at preprocessing time, compute classification models and indexes such that the following task can be completed at real-time:*

- *Given $x, |x| = n_0$, and a cost constraint c at real time, identify the set $F_i \in \mathcal{F}' = \{F \subseteq \mathcal{F} \mid c(F, n_0) \leq c\}$ such that $\forall_{F \in \mathcal{F}'} \alpha \times a(F_i) \geq a(F)$ and return $\mathcal{M}(F_i)(x)$.*

We also consider the version where c_0 is provided in advance but the item size n is not, e.g., when an aircraft needs to respond to any signals within a fixed time.

PROBLEM 3 (PROB-c_0-FIXED). *Given $c_0, \mathcal{F}, \mathcal{T}, \mathcal{T}'$ at preprocessing time, compute classification models and indexes such that the following task can be completed at real-time:*

- *Given $x, |x| = n$, at real time, identify the set $F_i \in \mathcal{F}' = \{F \subseteq \mathcal{F} \mid c(F, n) \leq c_0\}$ such that $\forall_{F \in \mathcal{F}'} \alpha \times a(F_i) \geq a(F)$ and return $\mathcal{M}(F_i)(x)$.*

Reusing Skyline Computation is Incorrect: We now argue that it is not sufficient to simply compute the skyline of classification models for a fixed item size and employ that skyline for all n, c. For the following discussion, we focus on $\alpha = 1$; the general α case is similar. Given a fixed $n = n_0$, we define the n_0-*skyline* as the set of all feature sets that are *undominated* in terms of cost and accuracy (or equivalently, error, which is $1-$ accuracy). A feature set F_i is undominated if there is no feature set F_j, where $c(F_j, n_0) < c(F_i, n_0)$ and $a(F_i) \leq a(F_j)$, and no feature set F_j where $c(F_j, n_0) \leq c(F_i, n_0)$ and $a(F_i) < a(F_j)$[1]. A naive strategy is to enumerate each feature set $F \subseteq \mathcal{F}$, and characterize each F by its cross-validation accuracy and average extraction cost over the training dataset. Once the feature sets are charcterized, iterate through them by increasing cost, and keep the feature sets whose accuracy is greater than any of the feature sets preceeding it. The resulting set is the n_0-skyline. However, it is every expensive to enumerate and characterize all feature sets (especially when the number of features is large), and one of our key contributions will be to avoid this exhaustive enumeration. But for the purposes of discussion, let us assume that we have the n_0-skyline computed. Note that the skyline feature sets are precisely the ones we need to consider as possible solutions during real-time classification for $n = n_0$ for various values of c_0.

[1]Notice that the $\leq$ operator is placed in different clauses in the two statements.

Then, one approach to solving Problem 1 could be to simply reuse the n_0-skyline for other n. However, this approach is incorrect, as depicted by Figure 1. Figure 1(a) depicts the cost and error of each feature set and the error vs. cost skyline curve for $n = n_0$, while Figure 1(b) depicts what happens when n changes from n_0 to a larger value n_1 (the same holds when the value changed to a smaller value): as can be seen in the figure, different feature sets move by different amounts, based on the cost function $c(F_i, n)$. This is because different polynomials behave differently as n is varied. For instance, $2n^2$ is less than $3n + 32$ when n is small, however it is significantly larger for big values of n. As a result, a feature set which was on the skyline may now no longer be on the skyline, and another one that was dominated could suddenly become part of the skyline.

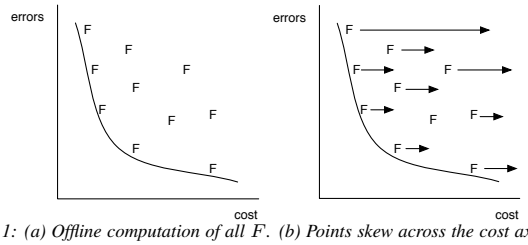

Figure 1: (a) Offline computation of all F. (b) Points skew across the cost axis as n changes.

Learning Algorithm Properties: We now describe a property about machine learning algorithms that we will leverage in subsequent sections. This property holds because even in the worst case, adding additional features simply gives us no new useful information that can help us in classification.

AXIOM 2.1 (INFORMATION-NEVER-HURTS). *If $F_i \subset F_j$, then $a(F_i) \leq a(F_j)$*

While this property is known by the machine learning community to be anecdotally true [11], we experimentally validate this in our experiments. In fact, even if this property is violated in a few cases, our POLY-DOM algorithm can be made more robust by taking that into account, as we will see in Section 3.1

3. POLY-DOM SOLUTION

Our solution follows three steps:

- *Feature Set Pruning:* First, we will start by constructing what we call as a *candidate set*, that is, the set of all feature sets (and corresponding machine learning models) that will be solutions to Problem 1. As a side effect, we will find a solution to Problem 2. The candidate set will be a carefully constructed superset of the skyline feature sets, so that we do not discard any feature sets that could be useful for any n. We will describe this in Section 3.1.
- *Polydom Index Construction:* In Section 3.2, we describe a new data-structure for Problem 1, called the *poly-dominance index*, which compactly represents the candidate set and allows it to be indexed into given a specific item size n and budget c during query time. In particular, we would like to organize the candidate set so that it can be efficiently probed even for large candidate sets size.
- *Online Retrieval:* Lastly, we describe how the *poly-dominance index* is accessed during query time in Section 3.3.

3.1 Offline: Constructing the Candidate Set

We will construct the candidate set using a bi-directional search on the lattice of all subsets of features, depicted in Figure 2.

When a sequence of features is listed, this sequence corresponds to the feature set containing those features. In the figure, feature

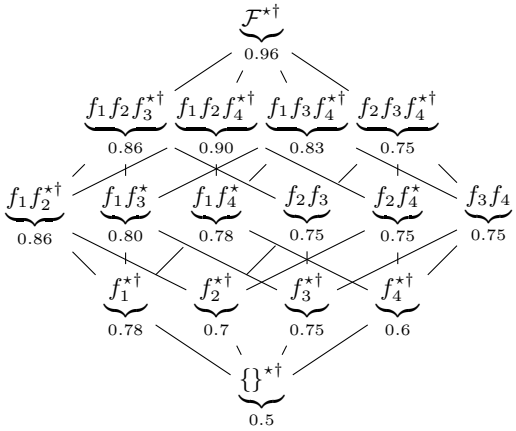

Figure 2: Lattice: $\star$ indicates the nodes that will be expanded for $\alpha = 1$; $\dagger$ indicates the nodes that will be expanded for $\alpha = 1.1$

sets are listed along with their accuracies (listed below the feature set)[2]. An edge connect two features sets that differ in one feature. For now, ignore the symbols $\star, \dagger$, we will describe their meaning subsequently. The feature set corresponding to $\mathcal{F}$ is depicted at the top of the lattice, while the feature set corresponding to the empty set is depicted at the bottom. The feature sets in between have 0 to $|\mathcal{F}|$ features. In the following we use feature sets and nodes in the lattice interchangably.

Bidirectional Search: At one extreme, we have the empty set $\{\}$, and at the other extreme, we have $\mathcal{F}$. We begin by learning and characterizing the best machine learning model for $F = \{\}$, and for $F = \mathcal{F}$: i.e., we learn the best machine learning model, represented as $\mathcal{M}(F)$, and learn the accuracy $a(F)$ (listed below the node) and $c(F, n)$ for the model[3] We call this step *expanding* a feature set, and a feature set thus operated on is called an *expanded* feature set.

At each round, we expand the feature sets in the next layer, in both directions. We stop once we have expanded all nodes. In our lattice in Figure 2, we expand the bottom and top layer each consisting of 1 node, following which, we expand the second-to-bottom layer consisting of 4 nodes, and the second-to-top layer again consisting of 4 nodes, and then we finally expand the middle layer consisting of 6 nodes.

However, notice that the total number of nodes in the lattice is $2^{|\mathcal{F}|}$, and even for relatively small $\mathcal{F}$, we simply cannot afford to expand all the nodes in the lattice. Therefore, we develop pruning conditions to avoid expanding all the nodes in the lattice. Note that all the pruning conditions we develop are guaranteed to return an accurate solution. That is, we do not make approximations at any point that take away the optimality of the solution.

Dominated Feature Sets: We now define what we mean for a feature set to dominate another.

DEFINITION 3.1 (DOMINANCE). *A feature set F_i dominates a feature set F_j if $\forall n, c(F_i, n) \leq c(F_j, n)$ and $\alpha \times a(F_i) \geq a(F_j)$*

As an example from Figure 2, consider node f_3f_4 and $f_2f_3f_4$ on the right-hand extreme of the lattice: the accuracies of both these feature sets is the same, while the cost of $f_2f_3f_4$ is definitely higher (since an additional feature f_2 is evaluated). Here, we will always prefer to use f_3f_4 over $f_2f_3f_4$, and as a result, $f_2f_3f_4$ is dominated by f_2f_3.

[2]We have chosen accuracy values that satisfy Axiom 2.1 in the previous section.

[3]Note that the cost of a model (i.e. featureset) is simply the sum of the individual features and can re-use previously computed costs.

Overall, a feature set F_j that is dominated is simply not under consideration for any n, because it is not going to be the solution to Problems 1, 2, or 3, given that F_i is a better solution. We formalize this as a theorem:

THEOREM 3.2. *A feature set that is dominated cannot be the solution to Problems 1, 2, or 3 for any n.*

Given the property above, we need to find domination rules that allow us to identify and discard dominated feature sets. In particular, in our lattice, this corresponds to not expanding feature sets.

Pruning Properties: Our first property dictates that we should not expand a feature set that is strictly "sandwiched between" two other feature sets. It can be shown that any such feature set is dominated, and therefore, using Theorem 3.2, can never be a solution to any of the problems listed in the previous section.

PROPERTY 3.3 (SANDWICH-PROPERTY). *If $F_i \subset F_k$, and $\alpha \times a(F_i) \geq a(F_k)$, then no F_j such that $F_i \subset F_j \subset F_k$, needs to be expanded.*

Intuitively, if there is a feature set F_i that dominates an F_j, while $F_i \subset F_j$, then all other feature sets betweeen F_i and F_j are also dominated. Consider Figure 2, with $\alpha = 1$, let $F_i = f_3$, and $F_k = f_2f_3f_4$, since $a(F_i) = a(F_k)$, feature sets F_j corresponding to f_2f_3 and f_3f_4 need not be expanded: in the figure, both these feature sets have precisely the same accuracy as f_3, but have a higher cost.

In Figure 2 once again for $\alpha = 1$, let us consider how many expansions the previous property saves us while doing bidirectional search: We first expand $\mathcal{F}$ and $\{\}$, and then we expand all nodes in the second-to-top layer and the second-to-bottom layer. Then, from the next layer, f_2f_3 and f_3f_4 will not be expanded (using the argument from the previous paragraph), while the rest are expanded. Thus, we save two expansions. The expanded nodes in the lattice are denoted using $\star$s.

Now, on changing α slightly, the number of evaluations goes down rapidly. The nodes expanded in this case are denoted using a $\dagger$. Let us consider $\alpha = 1.1$. Once again, nodes in the top two and bottom two layers are expanded. However, only f_1f_2 in in the middle layer needs to be expanded. This is because:

- f_1f_3 and f_1f_4 are sandwiched between f_1 and $f_1f_3f_4$
- f_2f_3 and f_2f_4 are sandwiched between f_2 and $f_2f_3f_4$
- f_2f_3 and f_3f_4 are sandwiched between f_3 and $f_2f_3f_4$

The previous property is hard apply directly (e.g., before expanding every feature set we need to verify if there exists a pair of feature sets that sandwich it). Next, we describe a property that specifies when it is safe to stop expanding all non-expanded ancestors of a specific node.

PROPERTY 3.4 (COVERING-PROPERTY). *If $F_i \subset F_{k_i}, \forall i \in 1 \ldots r$ such that $\cup_{i \in 1 \ldots r} F_{k_i} = \mathcal{F}$, and $\alpha \times a(F_i) \geq a(F_{k_i}), \forall i \in 1 \ldots r$, then no feature set sandwiched between F_i and F_{k_i} needs to be expanded.*

This property states if any set of feature sets F_{k_i} 1) contain F_i, 2) in aggregate covers all the features in $\mathcal{F}$ and 3) are dominated by F_i, then all feature sets between F_i and F_{k_i} do not need to be expanded. The inverse property for pruning descendents also holds.

We use this property to extend the bidirectional search with an additional pruning step. Let the top frontier $\mathcal{F}_{top}$ be the set of feature sets expanded from the top for which no child feature set has been expanded, and let $\mathcal{F}_{bot}$ be similarly defined from the bottom. By directly applying the *Covering-Property*, we can prune the parents of $F \in \mathcal{F}_{bot}$ if $\forall F' \in \mathcal{F}_{top}\ F \subseteq F'$, F dominates F'. We can similarly use the inverse of the property to prune feature sets in the top frontier.

Properties of Expanded Nodes: We have the following theorem, that is a straightforward consequence of Property 3.3:

THEOREM 3.5. *The set of expanded nodes form a superset of the skyline nodes for any n.*

In figure 2, the set of expanded nodes (denoted by $\star$ for $\alpha = 1$ and $\dagger$ for $\alpha = 1.1$) are the ones relevant for any n.

Candidate Nodes: Given the expanded set of nodes, two properties to allow us to prune away some of the expanded but dominated nodes to give the *candidate* nodes. Both these properties are straightforward consequences of the definition of dominance.

PROPERTY 3.6 (SUBSET PRUNING-PROPERTY). *If $F_i \subset F_k$, and $\alpha \times a(F_i) \leq a(F_k)$, then F_k does not need to be retained as a* candidate

For instance, even though $f_2f_3f_4$ and f_3 are both expanded, $f_2f_3f_4$ does not need to be retained as a candidate node when f_3 is present (for any α); also, $f_1f_3f_4$ does not need to be retained as a candidate node when f_1 is present for $\alpha = 1.1$.

The next property is a generalization of the previous, when we have a way of evaluating polynomial dominance.

PROPERTY 3.7 (POLY-DOM PRUNING-PROPERTY). *If $c(F_i, x) < c(F_k, x), \forall x \geq 0$, and $\alpha \times a(F_i) \leq a(F_k)$, then F_k does not need to be retained as a* candidate

The next theorem states that we have not made any incorrect decisions until this point, i.e., the set of candidate nodes includes all the nodes that are solutions to Problems 1, 2, 3 for all n.

THEOREM 3.8. *The set of candidate nodes form a superset of the skyline nodes for any n.*

Algorithm: The pseudocode for the algorithm can be found in the appendix split into: Algorithm 2 (wherein the lattice is traversed and the nodes are expanded) and Algorithm 1 (wherein the dominated expanded nodes are removed to give the candidate nodes).

In brief, Algorithm 2 maintains two collections: frontierTop and frontierBottom, which is the frontier (i.e., the boundary) of already expanded nodes from the top and bottom of the lattice respectively. The two collections activeTop and activeBottom contain the next set of nodes to be expanded. When a node is expanded, its children in the lattice are added to activeTop if the node is expanded "from the top", while its parents in the lattice are added to activeBottom if the node is expanded "from the bottom".

Note that there may be smart data structures we could use to check if a node is sandwiched or not, or when enumerating the candidate set. Unfortunately, the main cost is dominated by the cost for expanding a node (which involves training a machine learning model given a set of features and estimating its accuracy), thus these minor improvements do not improve the complexity much.

Discussion: When the number of features in $\mathcal{F}$ are in the thousands, the lattice would be massive. In such cases, even the number of expanded nodes can be in the millions. Expanding each of these nodes can take a significant time, since we would need to run our machine learning algorithm on each node (i.e., feature set). In such a scenario, we have two alternatives: (a) we apply a feature selection algorithm [32] that allows us to bring the number of features under consideration to a smaller, more manageable number, or; (b) we apply our pruning algorithm in a progressive modality. In this

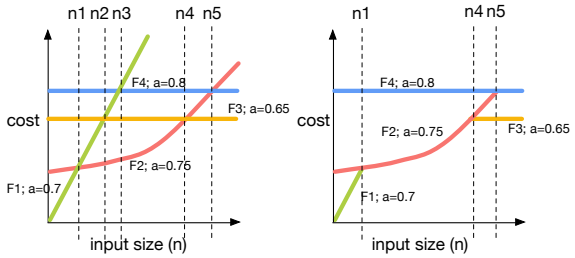

Figure 3: Incremental Poly-Dom Creation

modality, a user provides a precomputation pruning cost budget, and the pruning algorithm picks the "best α" to meet the precomputation cost budget (i.e., the smallest possible α for which we can apply the lattice pruning algorithm within the precomputation cost budget.) The approach is the following: we start with a large α (say 2), and run the lattice pruning algorithm. Once complete, we can reduce α by a small amount, and rerun the lattice pruning algorithm, and so on, until we run out of the precomputation cost budget. We can make use of the following property:

PROPERTY 3.9. *The nodes expanded for $\alpha_1, 1 \leq \alpha_1 < \alpha_2 \leq 2$ is a superset of the nodes expanded for α_2.*

Thus, no work that we do for larger αs are wasted for smaller αs: as a result, directly using the α that is the best for the precomputation cost budget would be equivalent to the above procedure, since the above procedure expands no more nodes than necessary.

Anti-Monotonicity: Note that there may be practical scenarios where the assumption of monotonicity, i.e., Axiom 2.1, does not hold, but instead, a relaxed version of monotonicity holds, that is,

AXIOM 3.10 (INFORMATION-NEVER-HURTS-RELAXED). *If $F_i \subset F_j$, then $a(F_i) \leq a(F_j) + e$*

Here, if F_i is a subset of F_j, then $a(F_i)$ cannot be larger than $a(F_j) + e$. Intuitively, the violations of monotonicity, if any are small—smaller than e (we call this the *error* in the monotonicity.) Note that when $e = 0$, we have Axiom 2.1 once again.

In such a scenario, only the lattice construction procedure is modified by ensuring that we do not prematurely prune away nodes (say, using the sandwich property) that can still be optimal. We use the following modified sandwich property:

PROPERTY 3.11 (SANDWICH-PROPERTY-RELAXED). *If $F_i \subset F_k$, and $\alpha \times (a(F_i) - e) \geq a(F_k)$, then no F_j such that $F_i \subset F_j \subset F_k$, needs to be expanded.*

With the above property, we have a more stringent condition, i.e., that $\alpha \times (a(F_i) - \delta)$ and not simply $\alpha \times a(F_i)$ has to be greater than $a(F_k)$. As a result, fewer pairs of nodes i, k qualify, and as a result, fewer nodes $j : F_i \subset F_j \subset F_k$ are pruned without expansion.

Subsequently, when deciding whether to remove some of the expanded nodes to give candidate nodes, we have the true accuracies of the expanded nodes, we no longer need to worry about the violations of monotonicity.

3.2 Offline: Constructing the Index

We begin by collecting the set of candidate nodes from the previous step. We denote the set of candidate nodes as $\mathcal{C}$, $|\mathcal{C}| = k$. We now describe how to construct the *poly-dom* index.

Alternate Visualization: Consider an alternate way of visualizing the set of candidate nodes, depicted in Figure 3(left). Here, we depict the cost $c(F_i, n)$ for each of the candidate nodes, as a function of n. Also labeled with each cost curve is the accuracy. Recall that unlike cost, the accuracy stays constant independent of the input size n. We call each of the curves corresponding to the candidate nodes as *candidate curves*. We depict in our figure four candidate curves, corresponding to feature sets F_1, F_2, F_3, F_4. In the figure, we depict five 'intersection points', where these candidate curves cross each other. We denote, in ascending order, the intersection points, as $n_1 < \ldots < n_r$. In Figure 3(left), $r = 5$. It is easy to see that the following holds:

LEMMA 3.12. *For all intersection points $n_1 < \ldots < n_r$ between candidate curves, storing the skyline for the following ranges $(-\infty, n_1), [n_1, n_2), \ldots, [n_{r-1}, n_r), [n_r, \infty)$ is sufficient for Problems 1,2, 3, since for any such range, the skyline is fixed.*

For $(-\infty, n_1)$, F_4 has accuracy 0.8, F_3 has accuracy 0.65, F_2 has accuracy 0.76, and F_1 has accuracy 0.7. The skyline of these four candidate sets for $n < n_1$ is F_4, F_2, F_1; F_3 is dominated by F_2 and F_1 both of which have lower cost and higher accuracy.

The lemma above describes the obvious fact that the relationships between candidate curves (and therefore nodes) do not change between the intersection points, and therefore, we only need to record what changes at each intersection point. Unfortunately, with r candidate curves, there can be as many as r^2 intersection points.

Thus, we have a naive approach to compute the index that allows us to retrieve the optimal candidate curve for each value of n_0, c_0:

- for each range, we compute the skyline of candidate nodes, and maintain it ordered on cost
- when n_0, c_0 values are provided at query time, we perform a binary search to identify the appropriate range for n_0, and do a binary search to identify the candidate node that that respects the condition on cost.

Our goal, next, is to identify ways to prune the number of intersection points so that we do not need to index and maintain the skyline of candidate nodes for many intersection points.

Traversing Intersection Points: Our approach is the following: We start with $n = 0$, and order the curves at that point in terms of cost. We maintain the set of curves in an ordered fashion throughout. Note that the first intersection point after the origin between these curves (i.e., the one that has smallest $n, n > 0$) has to be an intersection of two curves that are next to each other in the ordering at $n = 0$. (To see this, if two other curves intersected that were not adjacent, then at least one of them would have had to intersect with a curve that is adjacent.) So, we compute the intersection points for all pairs of adjacent curves and maintain them in a priority queue (there are at most k intersection points).

We pop out the smallest intersection point from this priority queue. If the intersection point satisfies certain conditions (described below), then we store the skyline for that intersection point. We call such an intersection point an *interesting intersection point*. If the point is not interesting, we do not need to store the skyline for that point. Either way, when we have finished processing this intersection point, we do the following: we first remove the intersection point from the priority queue. We then add two more intersection points to the priority queue, corresponding to the intersection points with the new neighbors of the two curves that intersected with each other. Subsequently, we may exploit the property that the next intersection point has to be one from the priority queue of intersection points of adjacent curves. We once again pop the next intersection point from the priority queue and the process continues.

The pseudocode for our procedure is listed in Algorithm 3 in the appendix. The array sortedCurves records the candidate curves sorted on cost, while the priority queue intPoints contains the intersection points of all currently adjacent curves. As long as intPoints is not empty, we keep popping intersection points from it, update sortedCurves to ensure that the ordering is updated, and add the

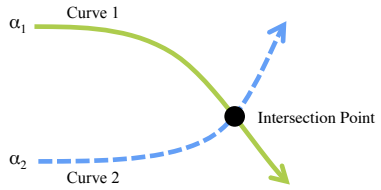

Figure 4: Reasoning about Intersection Points

point to the list of skyline recomputation points if the point is an interesting intersection point. Lastly, we add the two new intersections points of the curves that intersected at the current point.

Pruning Intersection Points: Given a candidate intersection point, we need to determine if we need to store the skyline for that intersection point. We now describe two mechanisms we use to prune away "uninteresting" intersection points. First, we have the following theorem, which uses Figure 4:

THEOREM 3.13. *We assume that for no two candidate nodes, the accuracy is same. The only intersection points (depicted in Figure 4, where we need to recompute the skyline are the following:*

- *Scenario 1: Curve 1 and 2 are both on the skyline, and $\alpha_1 > \alpha_2$. In this case, the skyline definitely changes, and therefore the point is an interesting intersection point.*
- *Scenario 2: Curve 1 is not on the skyline while Curve 2 is. Here, we have two cases: if $\alpha_1 > \alpha_2$ then the skyline definitely changes, and if $\alpha_1 < \alpha_2$, then the skyline changes iff there is no curve below Curve 2, whose accuracy is greater than α_2.*

As an example of how we can use the above theorem, consider Figure 3, specifically, intersection point n_1 and n_2. Before n_1, the skyline was $[F_1, F_2, F_4]$, with F_3 being dominated by F_1, F_2. At n_1, F_1 and F_2 intersect. Now, based on Theorem 3.13, since the lower curve (based on cost) before the intersection has lower accuracy, the curve corresponding to F_2 now starts to dominate the curve corresponding to F_1, and as a result, the skyline changes. Thus, this intersection point is indeed interesting.

Between n_1 and n_2, the skyline was $[F_2, F_4]$, since F_3 and F_1 are both dominated by F_2 (lower cost and higher accuracy). Now, at intersection point n_2, curves F_3 and F_1 intersect. Note that neither of these curves are on the skyline. Then, based on Theorem 3.13, we do not need to recompute the skyline for n_2.

Recall that we didn't use α approximation at all. Since we already used α to prune candidate nodes in the first step, we do not use it again to prune potential curves or intersection points, since that may lead to incorrect results. In our experience, the lattice pruning step is more time-consuming (since we need to train a machine learning model for each expanded node), so it is more beneficial to use α in that step. We leave determining how to best integrate α into the learning algorithms as future work. Finally, the user can easily integrate domain knowledge, such as the distribution of item sizes, into the algorithm to further avoid computing and indexing intemediate intersection points.

Determining the Skyline: As we are determining the set of interesting intersection points, it is also easy to determine and maintain the skyline for each of these points. We simply walk up the list of candidate nodes at that point, sorted by cost, and keep all points that have not been dominated by previous points. (We keep track of the highest accuracy seen so far.)

Index Construction: Thus, our polydom indexing structure is effectively is a two-dimensional sorted array, where we store the skyline for different values of n. We have, first, a sorted array corresponding to the sizes of the input. Attached to each of these locations is an array containing the skyline of candidate curves.

For the intersection curve depicted in Figure 3, the index that we construct is depicted in Figure 5.

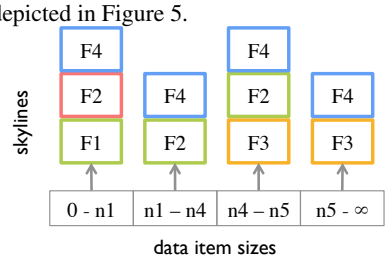

Figure 5: Poly-Dom Index for the feature sets in Figure 3

3.3 Online: Searching the Index

Given an instance of Problem 1 at real-time classification time, finding the optimal model is simple, and involves two steps. We describe these steps as it relates to Figure 5.

- We perform a binary search on the n_i ranges, i.e., horizontally on the bottom most array, to identify the range within which the input item size n_0 lies.
- We then perform a binary search on the candidate nodes, i.e., vertically for the node identified in the previous step, to find the candidate node for which the cost is the largest cost is less than the target cost c. Note that we can perform binary search because this array is sorted in terms of cost. We then return the model corresponding to the given candidate node.

Thus, the complexity of searching for the optimal machine learning model is: $O(\log t_{int} + \log t_{cand})$: where t_{int} is the number of interesting intersection points, while t_{cand} is the number of candidate nodes on the skyline.

4. GREEDY SOLUTION

The second solution we propose, called GREEDY, is a simple adaptation of the technique from Xu *et al.* [45]. Note that Xu *et al.*'s technique does not apply to generic machine learning algorithms, and only works with a specific class of SVMs; hence we had to adapt it to apply to all machine learning algorithms as a black box. Further, this algorithm (depicted in Algorithm 4 in the appendix) only works with a single size; multiple sizes are handed as described subsequently. For now, we assume that the following procedure is performed with the median size of items.

Expansion: Offline, the algorithm works as follows: for a range of values of $\lambda \in \mathcal{L}$, the algorithm does the following. For each λ, the algorithm considers adding one feature at a time to the current set of features that improves the most the function

$$\text{gain} = (\text{increase in accuracy}) — \lambda\times\ (\text{increase in cost})$$

This is done by considering adding one feature at a time, *expanding* the new set of features, and estimating its accuracy and cost. (The latter is a number rather than a polynomial — recall that the procedure works on a single size of item.) Once the best feature is added, the corresponding machine learning model for that set of features is recorded. This procedure is repeated until all the features are added, and then repeated for different λs. The intuition here is that the λ dictates the priority order of addition of features: a large λ means a higher preference for cost, and a smaller λ means a higher preference for accuracy. Overall, these sequences (one corresponding to every $\lambda \in \mathcal{L}$) correspond to a number of *depth-first* explorations of the lattice, as opposed to POLY-DOM, which explored the lattice in a *breadth-first* manner.

Indexing: We maintain two indexes, one which keeps track of the sequence of feature sets expanded for each λ, and one which keeps

the skyline of accuracy vs. cost for the entire set of feature sets. The latter is sorted by cost. Notice that since we focus on a single size, for the latter index, we do not need to worry about cost functions, we simply use the cost values for that size.

Retrieval: Online, when an item is provided, the algorithm performs a binary search on the skyline, picks the desired feature set that would fit within the cost budget. Then, we look up the λ corresponding to that model, and then add features starting from the first feature, computing one feature at a time, until the cost budget is exhausted for the given item. Note that we may end up at a point where we have evaluated more or less features than the feature set we started off with, because the item need not be of the same size as the item size used for offline indexing. Even if the size is different, since we have the entire sequence of expanded feature sets recorded for each λ, we can, when the size is larger, add a subset of features and still get to a feature set (and therefore a machine learning model) that is good, or when the size is smaller, add a superset of features (and therefore a model) and get to an even better model.

Comparison: This algorithm has some advantages compared to POLY-DOM:

- The number of models expanded is simply $|\mathcal{L}| \times |\mathcal{F}|^2$, unlike POLY-DOM, whose number of expanded models could grow exponentially in $\mathcal{F}$ in the worst case.
- Given the number of models stored is small (proportional to $|\mathcal{F}| \times |\mathcal{L}|$), the lookup can be simple and yet effective.
- The algorithm is any-time; for whatever reason if a feature evaluation cost is not as predicted, it can still terminate early with a good model, or terminate later with an even better model.

GREEDY also has some disadvantages compared to POLY-DOM:

- It does not provide any guarantees of optimality.
- Often, GREEDY returns models that are worse than POLY-DOM. Thus, in cases where accuracy is crucial, we need to use POLY-DOM.
- The λ values we iterate over, i.e., $\mathcal{L}$, requires hand-tuning, and may not be easy to set. Our results are very sensitive to this cost function.
- Since GREEDY uses a fixed size, for items that are of a very different size, it may not perform so well.

We will study the advantages and disadvantages in our experiments.

Special Cases: We now describe two special cases of GREEDY that merit attention: we will consider these algorithms in our experiments as well.

- GREEDY-ACC: This algorithm is simply GREEDY where $\mathcal{L} = \{-\infty\}$; that is, this algorithm adds one at a time, the feature with the smallest cost at the median size.
- GREEDY-COST: This algorithm is simply GREEDY where $\mathcal{L} = \{\infty\}$; that is, this algorithm adds one at a time, the feature that adds the most accuracy with no regard to cost.

Note that these two algorithms get rid of one of the disadvantages of GREEDY, i.e., specifying a suitable $\mathcal{L}$.

5. EXPERIMENTS

Online prediction depends on two separate phases—an offline phase to precompute machine learning models and data structures, and an online phase to make the most accurate prediction within a time budget. To this end, the goals of our evaluation are threefold: First, we study how effectively the POLY-DOM and GREEDY-based algorithms can prune the feature set lattice and thus reduce the number of models that need to be trained and indexed during offline pre-computation. Second, we study how these algorithms affect the latency and accuracy of the models that are retrieved online. Lastly, we verify the extent to which our anti-monotonicity assumption holds in real-world datasets.

To this end, we first run extensive simulations to understand the regimes when each algorithm performs well (Section 5.2). We then evaluate how our algorithms perform on a real-world image classification task (Section 5.3), and empirically study anti-monotonicity (Section B) and finally evaluate our algorithms on the real-wold classification task.

5.1 Experimental Setup

Metrics: We study multiple metrics in our experiments:

- *Offline Feature Set Expansion (Metric 1)*: Here, we measure the number of feature sets "expanded" by our algorithms, which represents the amount of training required by our algorithm.
- *Offline Index (Metric 2)*: Here, we measure the total size of the index necessary to look up the appropriate machine learning model given a new item and budget constraints.
- *Online Index lookup time (Metric 3)*: Here, we measure the amount of time taken to consult the index.
- *Online Accuracy (Metric 4)*: Here, we measure the accuracy of the algorithm on classifying items from the test set.

In the offline case, the reason why we study these two metrics (1 and 2) separately is because in practice we may be bottlenecked in some cases by the machine learning algorithm (i.e., the first metric is more important), and in some cases by how many machine learning models we can store on a parameter server (i.e., the second metric is more important). The reason behind studying the two metrics in the online case is similar.

Our Algorithms: We consider the following algorithms that we have either developed or adapted from prior work against each other:

- POLY-DOM: The optimal algorithm, which requires more storage and precomputation.
- GREEDY: The algorithm adapted from Xu *et al* [45], which requires less storage and precomputation than POLY-DOM, but may expand a sub-optimal set of feature sets that result in lower accuracy when the item sizes change.
- GREEDY-ACC: This algorithm involves a single sequence of GREEDY (as opposed to multiple sequences) prioritizing for the features contributing most to accuracy.
- GREEDY-COST: This algorithm involves a single sequence of GREEDY prioritizing for the features that have least cost.

Comparison Points: We developed variations of our algorithms to serve as baseline comparisons for the lattice exploration and indexing portions of an online prediction task:

- *Lattice Exploration*: NAIVE-EXPAND-ALL expands the complete feature set lattice.
- *Indexing*: While POLY-DOM only indexes points where the dominance relationship changes, POLY-DOM-INDEX-ALL indexes every intersection point between all pairs of candidate feature sets. Alternatively, NAIVE-LOOKUP does not create an index and instead scans all candidate feature sets online.

5.2 Synthetic Prediction Experiments

Our synthetic experiments explore how feature extraction costs, individual feature accuracies, interactions between feature accuracies, and item size variance affect our algorithms along each of our four metrics.

Synthetic Prediction: Our first setup uses a synthetic dataset whose

item sizes vary between 1 and 500. To explore the impact of non-constant feature extraction costs, we use a training set whose sizes are all 1, and vary the item sizes in the test dataset. For the feature sets, we vary four key parameters:

- *Number of features* f_i: We vary the number of features in our experiments from 1 to 15. The default value in our experiments is 12.
- *Feature extraction cost* $c(f_i, n)$: We randomly assign the cost function to ensure a high degree of intersection points. Each function is a polynomial of the form $c = a_0 + a_1 n + a_2 n^2$ where the coefficients are picked as follows: $a_0 \in [0, 100]$, $a_1 \in [0, \frac{100-a_0}{10}]$, $a_2 \in [0, \frac{100-a_0-a_1}{4}]$. (The reason why typically $a_0 < a_1 < a_2$ is that a_1 is multiplied by n, while a_2 is multiplied by n^2.) We expect that typical cost functions are bounded by degree n^2 and found that this is consistent with the cost functions from the real-world task. Note that POLY-DOM is insensitive to the exact cost functions, only the intersection points.
- *Single feature accuracy* $a(\{f_i\})$: Each feature's accuracy is sampled to be either *helpful* with probability p or *not helpful* with probability $1 - p$. If a feature is helpful, then its accuracy is sampled uniformly from within $[0.7, 0.8]$ and within $[0.5, 0.6]$ if it is not.
- *Feature interactions* $a(F_i)$: We control how the accuracy of a feature set F_i depends on the accuracy of its individual features using a parameterized *combiner* function:

$$a_k(F_i) = 1 - \Pi_{f_j \in F_i^k}(1 - a(f_j))$$

where F_i^k are the top k most accurate features in F_i. Thus when $k = 1$, F_i's accuracy is equal to its most accurate single feature. When $k = \infty$, F_i's accuracy increases as more features are added to the set. We will explore $k = 1$ and ∞ as two extremes of this combiner function. We denote the combiner function for a specific k value as cf_k. Note that for any k, the accuracy values are indeed monotone. We will explore the ramifications of non-monotone functions in the real-world experiments.

We use the following parameters to specify a specific synthetic configuration: the number of features n, the parameter p, and k, the amount the features interact with each other. In each run, we use these parameters to generate the specific features, cost functions and accuracies that are used for all of the algorithms. Unless specified, the default value of α is 1.2. For GREEDY, the $\mathcal{L} = \{0, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.5, 1, 5, 10\}$. Although we vary $\mathcal{L}$ over a large range, in practice the majority give rise to identical sequences because the dominating set of feature sets is fixed for a given item size (as assumed by GREEDY).

Feature Set Expansions (Metric 1): We begin by comparing the number of models that POLY-DOM, GREEDY and NAIVE-EXPAND-ALL train as a function of the combiner function and the number of features. This is simply the total number of unique feature For POLY-DOM and NAIVE-EXPAND-ALL, the number of feature sets is simply the number of nodes in the lattice that are expanded, while for GREEDY this is simply the number of unique feature sets expanded.

Metric 1 Summary: On the synthetic dataset, for cf_1, the number of feature sets (i.e., lattice nodes) expanded by POLY-DOM's lattice pruning phase for $\alpha = 1.2$ is at least an order-of-magnitude (10×) smaller than NAIVE-EXPAND-ALL, and is similar to GREEDY.

While POLY-DOM with $\alpha = 1$ expands similar to $\alpha = 1.2$ for cf_1, it expands all features for cf_∞. This is not surprising at all: cf_∞ is designed to be a scenario where every intermediate feature set is "useful", i.e., none of them get sandwiched between other feature sets.

In Figure 6(a) and Figure 6(b), we depict the number of feature sets expanded (in log scale) as a function of the number of features in the dataset along the x axis, for cf_1 and cf_∞ respectively. p is set to 0.6. The plots for other values are similar.

For both combiner functions, the total number of possible feature sets (depicted as NAIVE-EXPAND-ALL) scales very rapidly, as expected. On the other hand, the number of feature sets expanded by POLY-DOM for $\alpha = 1.2$ grows at a much slower rate for both graphs, because POLY-DOM's pruning rules allow it to "sandwich" a lot of potential feature sets and avoid expanding them. Consider first combiner function (i.e., Figure 6(a)) For 10 features, NAIVE-EXPAND-ALL expands around 1000 feature sets, while POLY-DOM for $\alpha = 1$ expands about 50, and $\alpha = 1.2$ expands about 30. We find that the ability to effectively prune the lattice of feature sets depends quite a bit on the combiner function. While POLY-DOM with $\alpha = 1.2$ continues to perform similarly. POLY-DOM $\alpha = 1$ expands as much as NAIVE-EXPAND-ALL; this is not surprising given that all intermediate feature sets have accuracy values strictly greater than their immediate children. In comparison, GREEDY expands about as many features as POLY-DOM with $\alpha = 1.2$ but with a slower growth rate as can be seen from both cf_1 and cf_∞; this is not surprising because in the worst case GREEDY expands $|\mathcal{L}| \times |\mathcal{F}|^2$.

Indexing Size and Retrieval (Metric 2 and 3): Here, we measure the indexing and retrieval time of the POLY-DOM algorithms, which use a more complex indexing scheme than the GREEDY algorithms.

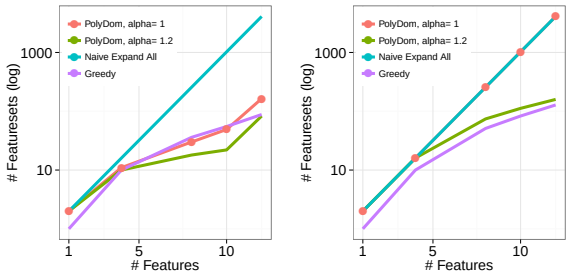

Figure 6: # of feature sets expanded vs # features for (a) cf_1 (b) cf_∞

Metric 2 and 3 Summary: On the synthetic dataset, especially for larger numbers of features, the size of the POLY-DOM index is significantly smaller than the size of the POLY-DOM-INDEX-ALL index, and almost as small as the NAIVE-LOOKUP index. However, while NAIVE-LOOKUP has a smaller index size, NAIVE-LOOKUP's retrieval time is much larger than POLY-DOM (for multiple values of α), making it an unsuitable candidate.

In Figure 7(a) and Figure 7(b), we plot, for the two combiner functions the total size of the poly-dom index as the number of features is increased (for $\alpha = 1.2, p = 0.6$.) Consider the case when the number of features is 10 for cf_1: here, POLY-DOM and NAIVE-LOOKUP's index size are both less than 200, POLY-DOM-INDEX-ALL's index size is at the 1000 mark, and rapidly increases to 6000 for 12 features, making it an unsuitable candidate for large numbers of features. The reason why POLY-DOM's index size is smaller than POLY-DOM-INDEX-ALL is because POLY-DOM only indexes those points where the dominance relationship changes,

while POLY-DOM-INDEX-ALL indexes all intersection points between candidate feature sets. NAIVE-LOOKUP, on the other hand, for both cf_1 and cf_∞ only needs to record the set of candidate sets, and therefore grows slowly as well.

On the other hand, for retrieval time, depicted in Figures 8(a) and 8(b), we find that the POLY-DOM's clever indexing scheme does much better than NAIVE-LOOKUP, since we have organized the feature sets in such a way that it is quick to retrieve the appropriate feature set given a cost budget On the other hand, NAIVE-LOOKUP does significantly worse than POLY-DOM, since it linearly scans all candidate feature sets to pick the best one — especially as the number of features increases.

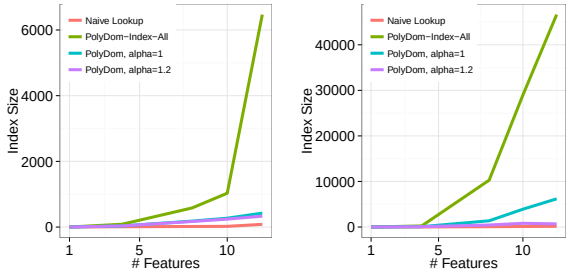

Figure 7: Index Size for (a) cf_1 (b) cf_∞

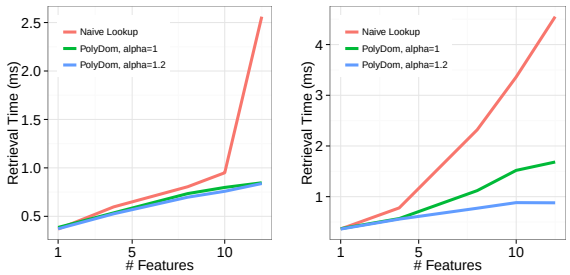

Figure 8: retrieval time for (a) cf_1 (b) cf_∞

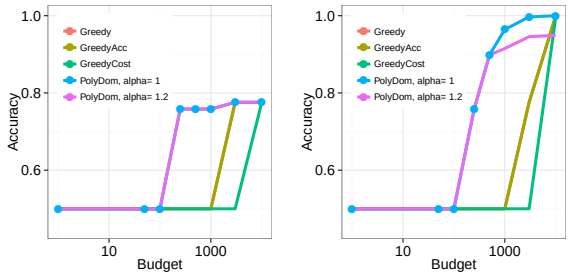

Figure 9: Accuracy as a function of budget (a) cf_1 (b) cf_∞

Real-Time Accuracy (Metric 4): We now test the accuracy of the eventual model recommended by our algorithm.

> **Metric 4 Summary:** On synthetic datasets, for cf_1 and cf_∞, over a range of budgets, POLY-DOM (with both $\alpha = 1$ or 1.3), returns models with greater accuracies than GREEDY and GREEDY-ACC, which returns models with greater accuracies than GREEDY-COST. Often the accuracy difference (for certain budgets) between POLY-DOM and GREEDY, or between GREEDY and GREEDY-COST can be as high as 20%.

In figure 9(a) and 9(b), we plot the accuracy as a function of budget for POLY-DOM with $\alpha = 1$ and 1.2, and for GREEDY, GREEDY-ACC and GREEDY-COST. For space constraints, we fix the item size to 50 and use 12 features. $\alpha = 1$ and 1.2 is almost always better than GREEDY. For instance, consider budget 1000 for cf_1 POLY-DOM with $\alpha = 1\&1.2$ has an accuracy of about 80%, while GREEDY, GREEDY-ACC and GREEDY-COST all have an accuracy of about 50%; as another example, consider budget 1000 for cf_∞, where POLY-DOM with $\alpha = 1\&1.2$ has an accuracy of more than 90%, while GREEDY, GREEDY-ACC and GREEDY-COST all have accuracies of about 50%. In this particular case, this may be because GREEDY, GREEDY-ACC, and GREEDY-COST all explore small portions of the lattice and may get stuck in local optima. That said, apart from "glitches" in the mid-tier budget range, all algorithms achieve optimality for the large budgets, and are no better than random for the low budgets.

Further, as can be seen in the figure GREEDY does better than GREEDY-COST, and similar to GREEDY-ACC. We have in fact also seen other instances where GREEDY does better than GREEDY-ACC, and similar to GREEDY-COST. Often, the performance of GREEDY is similar to one of GREEDY-ACC or GREEDY-COST.

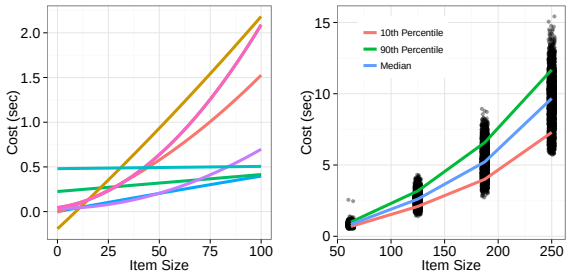

Figure 10: Varying size for (a) many features (b) one feature, but plotting quantiles

5.3 Real Dataset Experiments

Real Dataset: This subsection describes our experiments using a real image classification dataset [21]. The experiment is a multi-classification task to identify each image as one out of 15 possible scenes. There are 4485 labeled 250×250 pixel images in the original dataset. To test the how our algorithms perform on varying image sizes, we rescale them to 65×65, 125×125 and 187×187 pixel sizes. Thus in total, our dataset contains 17900 images. We use 8000 images as training and the rest as test images.

The task uses 13 image classification features (e.g., SIFT and GIST features) that vary significantly in cost and accuracy. In Figure 10(a), we plot the cost functions of eight representative features as a function of n, the item size, that we have learned using least squares curve fitting to the median cost at each training image size. As can be seen in the figure, there are some features whose cost functions are relatively flat (e.g., gist), while there are others that are increasing linearly (e.g., geo_map8×8) and super-linearly (e.g., texton).

However, note that due to variance in feature evaluation time, we may have cases where the real evaluation cost does not exactly match the predicted or expected cost. In Figure 10(a), we depict the 10% and 90% percentile of the cost given the item size for a single feature. As can be seen in the figure, there is significant variance — especially on larger image sizes.

To compensate for this variation, we compute the cost functions using the worst-case extraction costs rather than the median. In this way, we ensure that the predicted models in the experiment are always within budget. Note that GREEDY does not need to do this since it can seamlessly scale up/down the number of features evaluated as it traverses the sequence corresponding to a given λ. We did not consider this dynamic approach for the POLY-DOM algorithm.

The "black box" machine learning algorithm we use is a Linear classifier using stochastic gradient descent learning with hinge loss and L1 penalty. We first train the model over the training images for all possible combinations of features and cache the resulting

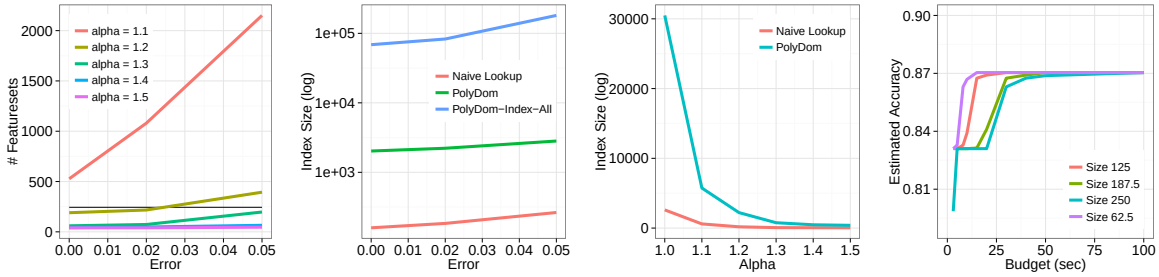

Figure 11: (a) Feature sets expanded (b) Index Size on varying e (c) Index size on varying α (d) Accuracy vs. Budget

models and cross-validation (i.e., estimated) accuracies. The rest of the experiments can look up the cached models rather than retrain the models for each execution.

For this dataset, our default values for α, e are $1.2, 0$, respectively As we will see in the following, the impact of e is small, even though our experiments described appendix B find that $e \geq 0$.

Feature Set Expansions (Metric 1):

Metric 1 Summary: On the real dataset, the number of feature sets expanded by POLY-DOM's offline lattice pruning phase for $\alpha = 1.2$ is $20\times$ smaller than NAIVE-EXPAND-ALL, with the order of magnitude increasing as α increases, and as e decreases. GREEDY expands a similar number of feature sets as POLY-DOM with $\alpha = 1.2$.

In Figure 11(a), we depict the number of feature sets expanded (in log scale) as a function of the tolerance to non-monotonicity e along the x axis, for POLY-DOM with values of $\alpha = 1.1, \ldots, 1.5$, and for GREEDY.

As can be seen in the figure, while the the total number of possible feature sets is close to 8200 (which is what NAIVE-EXPAND-ALL would expand), the number of feature sets by POLY-DOM is always less than 400 for $\alpha = 1.2$ or greater, and is even smaller for larger αs (the more relaxed variant induces fewer feature set expansions). GREEDY (depicted as a flat black line) expands a similar number of feature sets as POLY-DOM with $\alpha = 1.2$. On the other hand, for $\alpha = 1.1$, more than 1/4th of the feature sets are expanded.

The number of feature sets expanded also increases as e increases (assuming violations of monotonicity are more frequent leads to more feature set expansions).

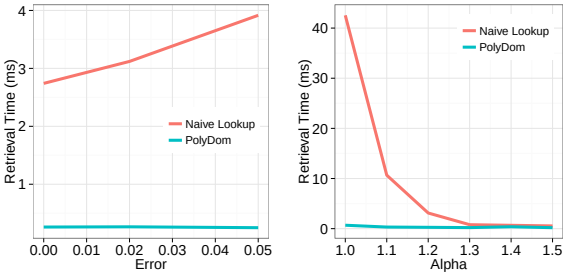

Figure 12: Retrieval time on varying (a) e (b) α

Indexing Size and Retrieval (Metric 2 and 3):

Metric 2 and 3 Summary: On the real dataset the size of the index for POLY-DOM is two orders of magnitude smaller than POLY-DOM-INDEX-ALL, while NAIVE-LOOKUP is one order of magnitude smaller than that. The index size increases as e increases and decreases as α increases. However, the retrieval time for POLY-DOM is minuscule compared to the retrieval time for NAIVE-LOOKUP.

In Figure 11(b), we plot the total index size as the tolerance to non-monotonicity is increased (for $\alpha = 1.3$.) As can be seen in the figure, the index size for POLY-DOM grows slowly as compared to POLY-DOM-INDEX-ALL, while NAIVE-LOOKUP grows even slower. Then, in Figure 11(c), we display the total index size that decreases rapidly as α is increased.

On the other hand, if we look at retrieval time, depicted in Figures 12(a) and 12(b) (on varying e and on varying α respectively), we find that NAIVE-LOOKUP is much worse than POLY-DOM—POLY-DOM's indexes lead to near-zero retrieval times, while NAIVE-LOOKUP's retrieval time is significant, at least an order of magnitude larger. Overall, we find that as the number of candidate sets under consideration grows (i.e., as α decreases, or e increases), we find that POLY-DOM does much better relative to POLY-DOM-INDEX-ALL in terms of space considerations, and does much better relative to NAIVE-LOOKUP in terms of time considerations. This is not surprising: the clever indexing scheme used by POLY-DOM pays richer dividends when the number of candidate sets is large.

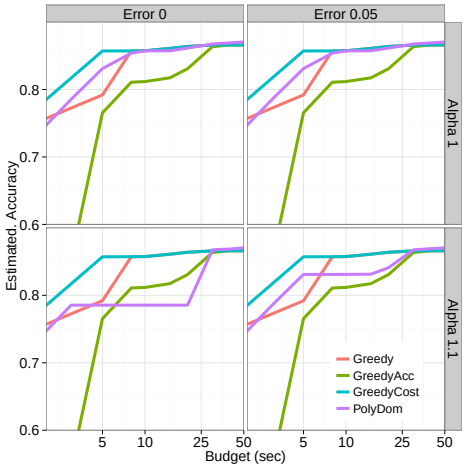

Figure 13: Estimated accuracy vs budget on varying α and e

Real-Time Accuracy (Metric 4):

Metric 4 Summary: On the real dataset, we find that while POLY-DOM still performs well compared to other algorithms for small α, some GREEDY-based algorithms are competitive, and in a few cases, somewhat surprisingly, better than POLY-DOM. All other things being fixed, the accuracy increases as the item size decreases, the budget increases, the α decreases, and e increases.

In Figure 13(a), we plot the average estimated accuracy of the model retrieved given a budget for various values of α and e for POLY-DOM, GREEDY, GREEDY-ACC, and GREEDY-COST across a range of image sizes. As can be seen in the figure, POLY-DOM dominates GREEDY and GREEDY-ACC apart from the case when $\alpha = 1.1$. Even for $\alpha = 1.1$, POLY-DOM dominates GREEDY-ACC when $e = 0.05$: here, we see that a higher e leads to better performance, which we did not see in other cases.

Perhaps the most surprising aspect of this dataset is that GREEDY-COST dominates all the others overall. While the reader may be surprised that a GREEDY-based algorithm can outperform POLY-DOM with $\alpha = 1$, recall that the GREEDY algorithms are any-time algorithms that can adapt to high variance in the feature evaluation cost, as opposed to POLY-DOM, which provisions for the worst-case and does not adapt to the variance. In future work, we plan to explore any-time variants of POLY-DOM, or hybrid variants of POLY-DOM with GREEDY-based algorithms.

In Figure 11(d), we plot the estimated accuracy of the model retrieved as a function of the budget for various image sizes (across a number of images of the same size). As can be seen in the figure, the estimated accuracy is higher for the same size of image, as budget increases. Also, the estimated accuracy is higher for the same budget, as image size decreases (as the image size decreases, the same budget allows us to evaluate more features and use a more powerful model.)

6. RELATED WORK

Despite its importance in applications, cost-sensitive real-time classification is not a particularly well-studied problem: typically, a feature selection algorithm [32] is used to identify a set of inexpensive features that are used with an inexpensive machine learning model (applied to all items, large or small), and there are no dynamic decisions enabling us to use a more expensive set of features if the input parameters allow it. This approach ends up giving us a classifier that is sub-optimal given the problem parameters. This approach has been used for real-time classification in a variety of scenarios, including: sensor-network processing [28, 8], object and people tracking [22, 36, 17], understanding gestures [29, 35], face recognition, speech understanding [3, 9], sound understanding [33, 39] scientific studies [16, 23], and medical analysis [31, 20]. All of these applications could benefit from the algorithms and indexing structures outlined in this paper.

Our techniques are designed using a wrapper-based approach [19] that is agnostic to the specific machine learning model, budget metric, and features to extract. For this reason, our approach can be applied in conjunction with a variety of machine learning classification or regression techniques, including SVMs [15], decision trees [27], linear or ridge regression [14], among others [12]. In addition, the budget can be defined in terms of systems resources, monetary metrics, time or a combination thereof.

There has been some work on adapting traditional algorithms to incorporate some notion of joint optimization with resource constraints, often motivated by a transition of these algorithms out of the research space and into industry. A few examples of this approach have been developed by the information retrieval and document ranking communities. In these papers the setup is typically described as an additive cascade of classifiers intermingled with pruning decisions. Wang *et al* notes that if these classifiers are independent the constrained selection problem is essentially the knapsack problem. In practice, members of an ensemble do not independently contribute to ensemble performance, posing a potentially exponential selection problem. For the ranking domain, Wang *et al* apply an algorithm that attempts to identify and remove redundant features [42], assemble a cascade of ranking and pruning functions [41], and develop a set of metrics to describe the efficienty-effectiveness tradeoff for these functions [40]. Other work focuses specifically on input-sensitive pruning aggresiveness [38] and early cascade termination strategies [6]. These approaches are similar in spirit to ours but tightly coupled to the IR domain. For example, redundant feature removal relies on knowledge of shared information between features (e.g., unigrams and bigrams), and the structure of the cascade (cycles of pruning and ranking) is particular to this particular problem. Further, these approaches are tuned to the ranking application, and do not directly apply to classification.

Xu's classifier cascade work [7, 44, 43, 45] considers the problem of post-processing classifiers for cost sensitivity. Their approach results in similar benefits to our own (e.g., expensive features may be chosen first if the gains outweigh a combination of cheap features), but it is tailored to binary classification environments with low positive classification rate and does not dynamically factor in runtime input size. Others apply markov decision processes to navigate the exponential space of feature combinations [18], terminate feature computation once a test point surpasses a certain similarity to training points [24], or greedily order feature computation [25], but none of these formalize the notion of budget or input size into the runtime model, making it difficult to know whether high-cost high-reward features can be justified up front or if they should be forgone for an ensemble of lower-cost features. That said, our GREEDY algorithm (along with its variants, GREEDY-ACC and GREEDY-COST) are adapted from these prior papers [45, 25].

Our POLY-DOM algorithms are also related to prior work on the broad literature on frequent itemset mining [2], specifically [5, 26], that has a notion of a lattice of sets of items (or *market baskets*) that is explored incrementally. Further, portions of the lattice that are dominated are simply not explored. Our POLY-DOM algorithm is also related to skyline computation [37], since we are implicitly maintaining a skyline at all "interesting" item sizes where the skyline changes drastically.

Anytime algorithms are a concept from planning literature that describe algorithms which always produce *some* answer and continuously refine that answer give more time [10]. Our GREEDY-family of algorithms are certainly anytime algorithms.

7. CONCLUSION AND FUTURE WORK

In this paper, we designed machine-learning model-agnostic cost-sensitive prediction schemes. We developed two core approaches (coupled with indexing techniques), titled POLY-DOM and GREEDY, representing two extremes in terms of how this cost-sensitive prediction wrapper can be architected. We found that POLY-DOM's optimization schemes allow us to maintain optimality guarantees while ensuring significant performance gains on various parameters relative to POLY-DOM-INDEX-ALL, NAIVE-LOOKUP, and NAIVE-EXPAND-ALL, and many times GREEDY, GREEDY-ACC, and GREEDY-COST as well. We found that GREEDY, along with the GREEDY-ACC and GREEDY-COST variants enable "quick and dirty" solutions that are often close to optimal in many settings.

In our work, we've taken a purely black-box approach towards how features are extracted, the machine learning algorithms, and the structure of the datasets. In future work, we plan to investigate how knowledge about the model, or correlations between features can help us avoid expanding even more nodes in the lattice.

Furthermore, in this paper, we simply used the size of the image as a signal to indicate how long a feature would take to get evaluated: we found that this often leads to estimates with high variance (see Figure 10(b)), due to which we had to provision for the worst-case instead of the average case. We plan to investigate the use of other "cheap" indicators of an item (like the size) that allow us to infer how much a feature evaluation would cost. Additionally, in this paper, our focus was on classifying a single point. If our goal was to evaluate an entire dataset within a time budget, to find the item with the highest likelihood of being in a special class, we would need very different techniques.

8. REFERENCES

[1] Largest image. **http://70gigapixel.cloudapp.net**.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM, 1993.

[3] J. N. Bailenson, E. D. Pontikakis, I. B. Mauss, J. J. Gross, M. E. Jabon, C. A. Hutcherson, C. Nass, and O. John. Real-time classification of evoked emotions using facial feature tracking and physiological responses. *International journal of human-computer studies*, 66(5):303–317, 2008.

[4] J. Brutlag. Speed matters for google web search. **http://googleresearch.blogspot.com/2009/06/speed-matters.html#!/2009/06/speed-matters.html**. Published: 2009.

[5] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 443–452. IEEE, 2001.

[6] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, pages 411–420, New York, NY, USA, 2010. ACM.

[7] M. Chen, Z. E. Xu, K. Q. Weinberger, O. Chapelle, and D. Kedem. Classifier cascade for minimizing feature evaluation cost. In *AISTATS*, pages 218–226, 2012.

[8] D. Comaniciu, V. Ramesh, and P. Meer. Real-time tracking of non-rigid objects using mean shift. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 2, pages 142–149. IEEE, 2000.

[9] B. Crawford, K. Miller, P. Shenoy, and R. Rao. Real-time classification of electromyographic signals for robotic control. In *AAAI*, volume 5, pages 523–528, 2005.

[10] T. Dean and M. Boddy. Time-dependent planning. AAAI '88, pages 49–54, 1988.

[11] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[12] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, 2012.

[13] J. Hamilton. The cost of latency. **http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx**.

[14] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[15] M. A. Hearst, S. Dumais, E. Osman, J. Platt, and B. Scholkopf. Support vector machines. *Intelligent Systems and their Applications, IEEE*, 13(4):18–28, 1998.

[16] Q. A. Holmes, D. R. Nuesch, and R. A. Shuchman. Textural analysis and real-time classification of sea-ice types using digital sar data. *Geoscience and Remote Sensing, IEEE Transactions on*, (2):113–120, 1984.

[17] D. M. Karantonis, M. R. Narayanan, M. Mathie, N. H. Lovell, and B. G. Celler. Implementation of a real-time human movement classifier using a triaxial accelerometer for ambulatory monitoring. *Information Technology in Biomedicine, IEEE Transactions on*, 10(1):156–167, 2006.

[18] S. Karayev, M. J. Fritz, and T. Darrell. Dynamic feature selection for classification on a budget. In *International Conference on Machine Learning (ICML): Workshop on Prediction with Sequential Models*, 2013.

[19] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, Dec. 1997.

[20] S. M. LaConte, S. J. Peltier, and X. P. Hu. Real-time fmri using brain-state classification. *Human brain mapping*, 28(10):1033–1044, 2007.

[21] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2169–2178. IEEE, 2006.

[22] A. J. Lipton, H. Fujiyoshi, and R. S. Patil. Moving target classification and tracking from real-time video. In *Applications of Computer Vision, 1998. WACV'98. Proceedings., Fourth IEEE Workshop on*, pages 8–14. IEEE, 1998.

[23] A. Mahabal, S. Djorgovski, R. Williams, A. Drake, C. Donalek, M. Graham, B. Moghaddam, M. Turmon, J. Jewell, A. Khosla, et al. Towards real-time classification of astronomical transients. *arXiv preprint arXiv:0810.4527*, 2008.

[24] F. Nan, J. Wang, K. Trapeznikov, and V. Saligrama. Fast margin-based cost-sensitive classification. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 2952–2956. IEEE, 2014.

[25] P. Naula, A. Airola, T. Salakoski, and T. Pahikkala. Multi-label learning under feature extraction budgets. *Pattern Recognition Letters*, 40:56–65, 2014.

[26] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Information systems*, 24(1):25–46, 1999.

[27] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[28] R. Rad and M. Jamzad. Real time classification and tracking of multiple vehicles in highways. *Pattern Recognition Letters*, 26(10):1597–1607, 2005.

[29] M. Raptis, D. Kirovski, and H. Hoppe. Real-time classification of dance gestures from skeleton animation. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 147–156. ACM, 2011.

[30] V. C. Raykar, B. Krishnapuram, and S. Yu. Designing efficient cascaded classifiers: tradeoff between accuracy and cost. In *KDD*, pages 853–860, 2010.

[31] J. Rodriguez, A. Goni, and A. Illarramendi. Real-time classification of ecgs on a pda. *Information Technology in Biomedicine, IEEE Transactions on*, 9(1):23–34, 2005.

[32] Y. Saeys, I. n. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517, Sept. 2007.

[33] J. Saunders. Real-time discrimination of broadcast speech/music. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96 Vol 2. Conference Proceedings., 1996 IEEE International Conference on*, volume 2, pages 993–996. IEEE, 1996.

[34] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. **http://velocityconf.com/velocity2009/public/schedule/detail/8523**, 2009.

[35] J. Shotton, T. Sharp, A. Kipman, A. Fitzgibbon, M. Finocchio, A. Blake, M. Cook, and R. Moore. Real-time human pose recognition in parts from single depth images.

Communications of the ACM, 56(1):116–124, 2013.

[36] C. Stauffer and W. E. L. Grimson. Learning patterns of activity using real-time tracking. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):747–757, 2000.

[37] K.-L. Tan, P.-K. Eng, B. C. Ooi, et al. Efficient progressive skyline computation. In *VLDB*, volume 1, pages 301–310, 2001.

[38] N. Tonellotto, C. Macdonald, and I. Ounis. Efficient and effective retrieval using selective pruning. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 63–72, New York, NY, USA, 2013. ACM.

[39] G. Tzanetakis and P. Cook. Musical genre classification of audio signals. *Speech and Audio Processing, IEEE transactions on*, 10(5):293–302, 2002.

[40] L. Wang, J. Lin, and D. Metzler. Learning to efficiently rank. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 138–145, New York, NY, USA, 2010. ACM.

[41] L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 105–114, New York, NY, USA, 2011. ACM.

[42] L. Wang, D. Metzler, and J. Lin. Ranking under temporal constraints. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 79–88, New York, NY, USA, 2010. ACM.

[43] Z. E. Xu, M. J. Kusner, G. Huang, and K. Q. Weinberger. Anytime representation learning. In *ICML (3)*, pages 1076–1084, 2013.

[44] Z. E. Xu, M. J. Kusner, K. Q. Weinberger, and M. Chen. Cost-sensitive tree of classifiers. In *ICML (1)*, pages 133–141, 2013.

[45] Z. E. Xu, K. Q. Weinberger, and O. Chapelle. The greedy miser: Learning under test-time budgets. In *ICML*, 2012.

APPENDIX

A. PROOF OF THEOREM 3.13

The following a proof of the poly-dom index construction algorithm described in Section 3.2.

PROOF The proof follows a case analysis:

- Scenario 1: Curve 1 and 2 both on skyline and $\alpha_1 > \alpha_2$
- Scenario 2: Curve 1 is not on the skyline while Curve 2 is
- Scenario 3: Curve 1 and 2 are both not on skyline
- Scenario 4: Curve 1 is on the skyline and Curve 2 is not, and $\alpha_1 > \alpha_2$

The argument is that for Scenario 3 and 4, the skyline will not change: in scenario 3, Curve 1 and 2 will still not be on the skyline, while in Scenario 4, since Curve 1 gets even better, it will still be on the skyline, while Curve 2 will be dominated by Curve 1 and therefore will not be on the skyline.

On the other hand, for Scenario 1, Curve 1 will start dominating Curve 2, and so Curve 2 now is removed from the skyline. For Scenario 2, which is a little tricky, Curve 1, which is not on the skyline because of high cost, may move into the skyline if there is no other curve that dominates it (i.e., has lower cost and higher accuracy).

The other scenarios cannot happen:

- Other half of Scenario 1: Curve 1 and 2 are both on skyline, and $\alpha_1 < \alpha_2$ cannot happen, because then Curve 2 would dominate Curve 1
- Other half of Scenario 4: Curve 1 is on the skyline while Curve 2 is not on the skyline and $\alpha_1 < \alpha_2$ cannot happen since Curve 1 is dominated by Curve 2 □

B. ASSUMPTION VALIDATION

In addition to the above performance metrics, we evaluated the sources of three types of variation that deviate from our assumptions.

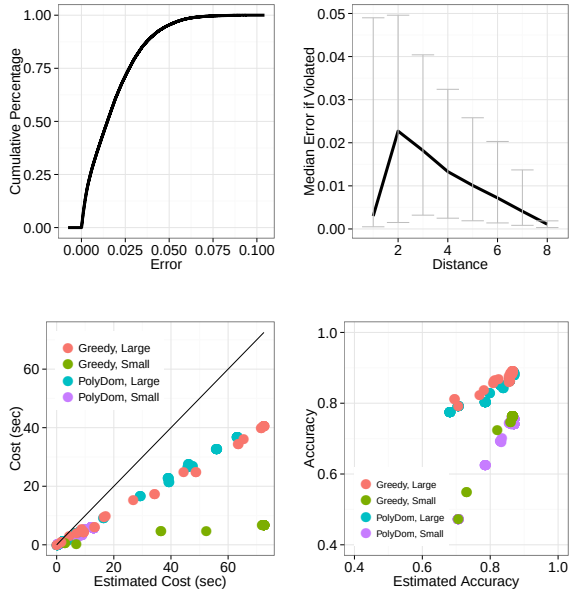

Figure 14: (a) CDF of anti-monotonicity (b) anti-monotonicity as a function of distance (c) Cost vs. Estimated Cost (d) Accuracy vs. Estimated Accuracy

Anti-monotonicity: First, we focus on the monotonicity assumption that we made on the real dataset, which we used to define the e values used in the experiments. Although the monotonicity

principle is anecdotally known to be true among machine learning practitioners, we were unable to find a reference for the practical evaluation of monotonicity on a real dataset. We view this as an additional contribution of our paper.

Figure 14(a) plots the cumulative distribution of the violations of monotonicity between pairs of ancestor-descendant feature sets in the lattice. Most of the violations are small: close to 95% of the violations are below an error of 5%, while all violations are within 7.5%. Thus, we believe the monotonicity principle is largely true, even on a real dataset. Note that we found fimor devations between the quality of the retrieved models as we decreased e to 0, so the assumptions that we made in the paper do not hurt us in a significant way.

Next, we would like to evaluate where in fact these violations of monotonicity exist in the lattice. Figure 14(b) evaluates the distribution of violations as a function of distance[4] between feature set pairs. The black line is the median e (with grey error bars) of all violations as a function of distance. As can be seen in the figure, the highest median e as well as the largest variance is at distance 2 and both quickly decrease to close to 0 at 8. This is a good sign: violations of monotonicity, if any, are local rather than global, with almost no violations that are between pairs of feature sets that are far away from each other. These results suggest that the Skyline algorithm is not likely to falsely prune a feature set early on due to a violation in monotonicity. Furthermore, a modest e value can compensate for the majority of violations.

Estimated Versus Actual: Next, we compare the estimated cost and accuracies of the real-world model with the true values for large and small image sizes.

Figure 14(c) plots the estimated versusactual cost. We find that the cost function tends to over estimate the actual cost because the cost functions are trained on the worst, rather than average case. We chose this because if we did provision for the mean cost, the poly-dom index may return models whose true costs exceed the time budget. The costs for GREEDY are similar, however because it ignores item size during the offline phase, it severely underestimates the cost of the small images, in contrast to POLY-DOM.

Figure 14(d) plots the estimated and true accuracy of the models retrieved. We find the the estimated accuracy is indeed linearly correlated with the true accuracy. However the model consistently overestimates the accuracy because the small images are downsampled, so the features are correspondingly less accurate. Overall, this suggests that optimizing for estimated accuracy is a reliable proxy for the quality of predictions at test time.

C. ALGORITHM PSEUDOCODE

Algorithm 1: CANDIDATE-SET-CONSTRUCTION

Data: $\mathcal{F}, n_0, \alpha$
Result: candidateSet
expandedNodes = EXPAND-ENUMERATE $(\mathcal{F}, n_0, \alpha)$;
toRemove = $\emptyset$;
for *s, r* $\in$ expandedNodes **do**
  if *s is dominated by r* **then**
    add s to toRemove;
candidateSet = expandedNodes − toRemove;

Algorithm 2: EXPAND-ENUMERATE

Data: $\mathcal{F}, n_0, \alpha$
Result: expandedNodes
activeTop = $\{\mathcal{F}\}$;
activeBottom = $\{\{\}\}$;
frontierTop = frontierBottom = expandedNodes = $\emptyset$;
while activeTop *or* activeBottom *is non-empty* **do**
  activeTop2 = activeBottom2 = $\emptyset$;
  for *s in* activeTop **do**
    if *s is not sandwiched between* frontierTop, frontierBottom *AND has not been expanded* **then**
      expand s and add to expandedNodes;
      add s's children to activeTop2;
      add s to frontierTop;
      remove s's parents from frontierTop;
    remove s from activeTop;
  for *s in* activeBottom **do**
    if *s is not sandwiched between* frontierTop, frontierBottom *AND has not been expanded* **then**
      expand s and add to expandedNodes;
      add s's parents to activeBottom2;
      add s to frontierBottom;
      remove s's children from frontierBottom;
    remove s from activeBottom;
  activeTop = activeTop2;
  activeBottom = activeBottom2;

Algorithm 3: POLYDOMINTERSECTIONS

Data: $\mathcal{C}, \alpha$
Result: Poly − Dom
candCurves = curves corresponding to $\mathcal{C}$;
sortedCurves = sorted candCurves for $n < 1$ on cost;
intPoints = p.queue of int. pts. of neighboring curves;
while IntPoints *is not empty* **do**
  singlePt = intPoints.$pop()$;
  update sortedCurves for singlePt;
  a, b = curves that intersect at singlePt;
  if interesting(singlePt, sortedCurves) **then**
    ptsSoFar.add(singlePt);
  intPoints.add(new intersections of a, b with neighbors in sortedCurves);
return ptsSoFar;

Algorithm 4: GREEDY CANDIDATE-SEQUENCES

Data: $\mathcal{F}, n_0$
Result: candidateSeq
for $\lambda \in \mathcal{L}$ **do**
  curSet = $\emptyset$;
  remSet = $\mathcal{F}$;
  while remSet $\neq \emptyset$ **do**
    add best feature $f \in$ remSet to curSet;
    candidateSeq(λ).append(model corresponding to curSet);
    remove f from remSet;
return candidateSeq;

[4]the difference in number of features between the ancestor and descendant featuresets