Accelerating Machine Learning Inference with Probabilistic Predicates

Yao Lu[1,3], Aakanksha Chowdhery[2,3], Srikanth Kandula[3], Surajit Chaudhuri[3]
[1]UW, [2]Princeton, [3]Microsoft

ABSTRACT

Classic query optimization techniques, including predicate pushdown, are of limited use for machine learning inference queries, because the user-defined functions (UDFs) which extract relational columns from unstructured inputs are often very expensive; query predicates will remain stuck behind these UDFs if they happen to require relational columns that are generated by the UDFs. In this work, we demonstrate constructing and applying probabilistic predicates to filter data blobs that do not satisfy the query predicate; such filtering is parametrized to different target accuracies. Furthermore, to support complex predicates and to avoid per-query training, we augment a cost-based query optimizer to choose plans with appropriate combinations of simpler probabilistic predicates. Experiments with several machine learning workloads on a big-data cluster show that query processing improves by as much as 10×.

KEYWORDS

Query processing, user defined functions, probabilistic predicates, machine learning, inference.

ACM Reference Format:
Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* https://doi.org/10.1145/3183713.3183751

1 INTRODUCTION

Relational data platforms are increasingly being used to analyze data blobs such as unstructured text, images or videos [5, 11, 36, 48]. Queries in these systems begin by applying user-defined functions (UDFs) to extract relational columns from blobs. Consider the following example which finds *red SUVs* from city-wide surveillance cameras:

```
SELECT cameraID, frameID,
C1(F1(vehBox)) AS vehType, C2(F2(vehBox)) AS vehColor
FROM (PROCESS inputVideo
      PRODUCE cameraID, frameID, vehBox
      USING VehDetector)
WHERE vehType = SUV ∧ vehColor = red;
```

Here, $VehDetector$ extracts vehicle bounding boxes from each video frame. $\mathcal{F}_1$ and $\mathcal{F}_2$ extract relevant features from each bounding box, and finally $\mathcal{C}_1, \mathcal{C}_2$ are classifiers that identify the vehicle type and color using the extracted features.

How can we execute such machine learning inference queries efficiently? Clearly, traditional query optimization techniques such as predicate pushdown are not useful here, because they will not push predicates below the UDFs that generate the predicate columns. In the above example, `vehType` and `vehColor` are available only after $VehDetector$, $\mathcal{C}$ and $\mathcal{F}$ have been executed. Even when the predicate has low selectivity (perhaps 1-in-100 images have red SUVs), every video frame has to be processed by all the UDFs. Figure 1 shows a typical query plan for this query.

$$\texttt{Input} \rightarrow \underline{VehDetector \rightarrow \mathcal{F}_1, \mathcal{F}_2 \rightarrow \mathcal{C}_1, \mathcal{C}_2} \rightarrow \sigma_{SUV} \wedge \sigma_{red} \rightarrow \texttt{Result}$$

Figure 1: The query plan to retrieve red SUVs from traffic surveillance videos. Materializing the `vehType` and the `vehColor` columns (underlined) takes 99.8% of the query cost.

$$\texttt{Input} \rightarrow \texttt{PP}_{SUV}, \texttt{PP}_{red} \rightarrow \underline{VehDetector \rightarrow \mathcal{F}_1, \mathcal{F}_2 \rightarrow \mathcal{C}_1, \mathcal{C}_2} \rightarrow \sigma_{SUV} \wedge \sigma_{red} \rightarrow \texttt{Result}$$

Figure 2: We construct and apply probabilistic predicates (PPs) to filter data blobs that do not satisfy the predicates.

It is tempting to simplify the problem by separating the machine-learning components from the relational portion. For example, some component exogenous to the data platform may pre-process the blobs and materialize all the necessary columns; a traditional query optimizer is then applied on the remaining query. This approach may be feasible in certain cases but is, in general, infeasible. In many workloads, the queries are complex and use many different types of feature extractors and classifiers; pre-computing all possible options would be expensive. Moreover, pre-computing will be wasteful for ad-hoc queries since many of the columns with extracted features may never be used. In surveillance scenarios, for example, ad-hoc queries typically obtain retroactive video evidence for traffic incidents. While some videos and columns may be accessed by many queries, some may not be accessed at all. Finally, for online queries (e.g., queries on live newscasts or broadcast games), it could be faster to execute the queries and ML components directly on the live data.

In this work, our goal is to accelerate machine learning inference queries with expensive UDFs. Specifically, we propose the notion of probabilistic predicates (PPs). PPs are binary classifiers on the unstructured input which shortcut subsequent UDFs for those data blobs that will not pass the query predicate; the query cost is therefore reduced. As shown in Figure 2, if the query predicate has a small selectivity and the PP is able to discard half of the frames that do not have red SUVs, the query may speed up by 2×.

Furthermore, whereas conventional predicate pushdown produces deterministic filtering results, filtering with PPs is parametric over a precision-recall curve; different filtering rates (and hence speed-ups)

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183751

are achievable based on the desired accuracy. Notice that we have departed from the strict boolean semantics of a predicate. However, machine learning queries are inherently tolerant to error because even the unmodified queries have machine learning UDFs with some false positives and false negatives. We show that injecting PPs does not change the false positive rate but can increase the false negative rate. We develop a mechanism to bound the query-wide accuracy loss by choosing which PPs to use and how to combine them. Our experiments show sizable speed-ups with negligibly small accuracy loss on a variety of queries and datasets.

We find that different techniques to construct PPs are appropriate for different inputs and predicates (e.g., based on input sparsity, the number of dimensions and whether subsets of the input that pass and fail the predicate are linearly separable). We use several PP construction techniques (e.g., linear SVMs, kernel density estimators, DNNs) and use model selection to pick an appropriate technique that has high execution efficiency, high data reduction rate and low false negatives.

We also propose new query optimization techniques to support complex predicates and ad-hoc queries. We show how to integrate PPs into queries that have selects, projects and foreign-key joins. These techniques reduce the number of PPs that have to be trained. Our system only trains PPs for simple predicates and relies on the query optimizer to choose, for a complex or ad-hoc predicate, appropriate combinations of available PPs based not just on the selectivity of the PPs but also on their accuracy.

We have prototyped probabilistic predicates in a large production data-parallel query processing cluster at Microsoft [11]. We demonstrate the usefulness of PPs on various commonly occurring machine learning inference tasks over different large-scale datasets such as document classification on LSHTC [40], image labeling on ImageNet [31], COCO [35] and SUNAttributes [41] and video activity recognition on UCF101 [46]. We also show how to run more complex queries on the traffic video feeds from tens of cameras. Our experiments indicate that running online/batch machine learning inference with PPs achieves as much as 10× speedup with different predicates compared with executing the queries as-is.

To summarize, our key contributions are:

- A simple but broadly applicable design which incorporates a variety of PP construction techniques to accelerate online and batch machine learning inference queries.
- A query optimizer extension that matches complex predicates with available PPs and determines their parameters to meet the desired accuracy.
- Implementation and experiments on several real-world machine learning queries and datasets.

2 MACHINE LEARNING INFERENCE

We consider the problem of querying non-relational input such as videos, audios, images, unstructured text etc. This problem is crucial to many applications and services.

Consider for example the analysis of surveillance video [14]; recently, there have been city-wide deployments with over thousands of cameras [2], body cameras worn by police [4] and security cameras deployed at homes. Some example inference queries include:

Q1: Find cars with speed ≥ 80 mph on a highway.
Q2: What is the average car volume on each lane?
Q3: Find a black SUV with license plate 'ABC123'.
Q4: Find cars seen in camera C_1 and then in C_2.
Q5: Send text to phone if any external door is opened.
Q6: Alert police control room if shots are fired.

To answer such queries, multiple machine learning UDFs such as feature extractors, classifiers etc. are applied on the input. The subsequent rowsets are filtered, sometimes implicitly (e.g., video frames without vehicles are dropped in Q2). Queries may also contain grouping, aggregation (e.g., Q2) and joins (e.g., Q4).

It is easy to see that the *materialization cost*, i.e., time and resources used to execute the machine learning UDFs, would dominate in processing these queries. It is also easy to see that materialization is query-specific; while there is some commonality, in general, different queries invoke different feature extractors, regressors, classifiers etc. Considering all the possible queries that may be supported by a system, the number of distinct UDFs on the input is vast. Hence, a priori application of all UDFs on the input has a high cost. Furthermore, the query predicates may be rather complex, and the queries can be both online and offline. Security alerts, such as Q5 and Q6, are time-sensitive. Moreover, Q2 may be executed online to update driving directions [23] or to vary the toll price of express lanes in realtime [10].

Beyond surveillance analytics, many applications share the above three aspects: large materialization cost, diverse body of machine learning UDFs, latency and/or cost sensitivity. We review a few such applications in Table 1. The materialization cost in these systems ranges from milliseconds to seconds per input data item, which can be significant when millions of data blobs are generated in a short period of time in, say, a video streaming system. Since queries use many different UDFs, offline systems would need large amounts of compute and storage resources to pre-materialize the outputs of all possible UDFs. Online systems which often require rapid responses can also become bottlenecked by the latency to pre-materialize UDFs.

Recently, many systems support triggers over live video streams (newscasts, sportscasts etc.) [1]; the user specifies a trigger such as "music concert" and the system finds matching video feeds by analyzing a large corpus of live video feeds (e.g., from youtube live or periscope). These systems also satisfy the three aspects above: the space of possible triggers that a user can specify is quite large, applying machine learning functions on live feeds dominates the query cost, and query selectivity is small if only a small number of feeds match the trigger and the query is latency-sensitive because users expect a quick answer.

3 IDEAS AND CHALLENGES

To reduce the execution cost and latency of the machine learning queries, suppose we can apply a filter directly on the raw input which discards input data that will not pass the original query predicate. Cost decreases because the UDFs following the filter only have to process inputs that pass the filter; a higher *data reduction rate* r of the filter leads to larger possible performance improvement. Let the cost of applying the filter and the UDF be c and u respectively; then the gains from early filtering will be $\frac{1}{1-r+(c/u)}\times$. Hence, the more efficient the early filter is relative to the UDFs (small c/u), the larger the gains will be. Moreover, the query performance can become worse (instead of improving) if $r \leq c/u$, i.e., the early filter has a smaller

	System	Features	Classifiers/Regressors	Materialization Cost (sec)	Query predicate	Selectivity
Online	Ads recommendations [42]	Bag-of-words	Collaborative Regressor	$10^{-2} - 10^{-1}$	1 binary	1-in-hundreds
	Video recommendations [16]	Browse history	Bayesian Regressor	$10^{-1} - 10^{1}$	1 binary	1-in-thousands
	Credit card fraud [47]	Physical loc. etc.	Neural Network	$10^{-2} - 10^{-1}$	1 binary	1-in-thousands
Offline	Video tagging [24]	Keypoints	SVM w/ RBF kernel	$10^{-1} - 10^{1}$	n categorical	1-in-thousands
	Spam filtering [6]	Bag-of-word	Naive Bayes Classifier	$10^{-2} - 10^{-1}$	1 binary	1-in-several
	Image tagging [37, 55]	Keypoints	Collaborative Regressor	$10^{-1} - 10^{1}$	n categorical	1-in-thousands

Table 1: We examine queries from a few machine learning systems and list the features and classifiers that were used. Typical materialization costs are shown for each data item. We also list characteristics of typical predicates (number and type of clauses, selectivity).

data reduction relative to its additional cost; hence, only filters that have a large data reduction rate will speedup the query.

Another important consideration is the *accuracy* of the early filter; since the original UDFs and query predicate will process input that is passed by the early filter, the false positive rate of the query is unaffected. However, the filter may drop input data that would pass the original query predicate, i.e., can increase false negatives. Unlike queries on relational data, machine learning applications have an in-built tolerance for error since the original UDFs in the query also have some false positive and false negative rate. Hence, it is feasible, in our experience, to ask the users to specify a desired accuracy threshold a. Some queries, such as Q1 and Q2 in §2, tolerate a known amount of inaccuracy.

Challenges. To achieve sizable query speedup with desired accuracy, the following questions become important. First, how to construct these early filters? Since the raw input does not have the columns required by the original query predicate, constructing early filters is not akin to predicate pushdown [34] and is not the same as ordering predicates based on their cost and data reduction [17]. Instead, we propose to train binary classifiers that group the input blobs into those that *disagree* and those that *may agree* with the query predicate. The former are discarded, and the latter are passed to the original query plan. We call these classifiers *probabilistic predicates* (PPs), because each PP has associated values for the tuple ⟨data reduction rate, cost, accuracy⟩; it is possible to train PPs with different tuple values.

Next, how to construct probabilistic predicates that are *useful*, i.e., those that have a good trade-off between data reduction rate, cost and accuracy? Success in partitioning the data into two classes, a class that passes the original query predicate and the other that does not, depends on the underlying data distributions. A predicate can be thought of as a decision boundary separating the two classes. Intuitively, any classifier that can identify inputs far away from this decision boundary can be a useful PP. However, the nature of the inputs and the decision boundary affects which classifiers are effective at separating the two classes. We use different techniques to build PPs– linear support vector machine (SVM) [50] for linearly separable cases, and kernel density estimator (KDE) [43] and deep neural networks [33] for non-linearly separable cases. We note that PPs can also be created using any other classifier technique (e.g., [9]). To handle data blobs with high dimensionality, we utilize sampling, principal component analysis (PCA) [28] and feature hashing [53]. We apply model selection to choose appropriate classification and dimensionality reduction techniques.

A third question is how to support complex predicates and ad-hoc queries? Since query predicates can be diverse, trivially constructing a PP for each query is unlikely to scale. Consider the example in Figure 2, a PP trained for $red \wedge SUV$ cannot be applied to $red \wedge car$ or $blue \wedge SUV$. Moreover, ad-hoc queries with previously unseen predicates cannot be supported. To generalize, we propose to only build PPs per simple clause and have the query optimizer, at query compilation time, assemble an appropriate combination of PPs that (1) has the lowest cost, (2) is within the accuracy target and (3) is semantically implied by the original query predicate; i.e., the PP combination has to be a *necessary* condition of the query predicate (since we use PPs to drop blobs that are unlikely to satisfy the predicate). We will show in §6 how we extend a standard cost-based predicate exploration procedure to generate various possible plans that use one or more of the available PPs and stay within the given accuracy threshold; our QO then picks the lowest cost plan from among these alternatives.

Scope, limitations, and connections. More precisely, we build probabilistic predicates for clauses of the form $f\,(g_i(b), \ldots)\ \phi\ v$, where f, g_i are functions, b is an input blob, ϕ is an operator that can be $=, \neq, <, \leq, >, \geq$ and v is a constant. As noted above, we build PPs using a diverse set of techniques and only for clauses that have useful ⟨data-reduction, accuracy, cost⟩. Using these PPs, our QO can support predicates that contain arbitrary conjunctions, disjunctions or negations of the above clauses. Furthermore, we show in Appendix A.4 how to inject PPs into queries that have selections, projections and foreign-key joins.

Some important limitations are worth noting. Predicates that do not decompose onto individual inputs are not supported; for example `SELECT * FROM` T_1, T_2 `WHERE` $\mathcal{F}$`(`T_1`.a, `T_2`.b) > 0` and $\mathcal{F}$ is not a separable function. UDFs that are not deterministic (e.g., those that have random components or adapt to the input) are also not supported because the mapping from the input to the predicate outcome, which the PPs learn and use, will also have to adapt along with the UDF.

The basic intuition behind probabilistic predicates is akin to that of cascaded classifiers in machine learning [51, 52]; a more efficient but inaccurate classifier can be used in front of an expensive classifier to lower the overall cost. Typical cascades, however, use classifiers that have equivalent functionality (e.g., all are object detectors). In contrast, PPs are not equivalent to the UDFs that they bypass; agnostic to the functionality of the UDFs that are bypassed, PPs are always binary predicate-specific classifiers. Without this specialization (reduction in functionality), it may be impossible to obtain a classifier that executes over raw input and still achieves good data reduction without losing accuracy. Furthermore, typical cascades accept and reject input anywhere in the pipeline; while this could work for selection queries whose output is simply a subset of the input, it will not easily extend to queries having projections, joins or aggregations. In general, our PPs apply directly on an input and reject irrelevant blobs; the rest of the input is passed to the actual query.

a) Unmodified System b) Constructing prob. predicates c) Modified System

Figure 3: Comparing the unmodified system on the left with the proposed system on the right. Key changes are in the training and use of probabilistic predicates (PPs). See § 4 for details.

Our technical advances are in identifying and building useful PP classifiers (§5) and a deep integration with the QO (§6); the former involves careful model selection and the latter generalizes applicability to complex predicates and adhoc queries. A related system [27] identifies correlations between input columns and a user-defined predicate and then learns a probabilistic selection method which accepts or rejects inputs, based on the value of the identified correlated input columns, without evaluating the user-defined predicate. A contemporaneous system [29] uses a specialized DNN and video-specific filtering techniques such as background subtraction to speed up object detection on videos. Probabilistic predicates have broader applicability and offer comparable or more gains as we show empirically in §8. Both the above systems accept blobs early and hence do not easily extend beyond selection queries. Furthermore, the probabilistic selection method used in [27] maintains state per distinct value of the correlated input columns; the proposed extension to handle multiple predicates and joins substantially increases the state needed (exponential in # of predicates and per distinct combined value of the correlated columns and join columns [26]). The above systems also pick a specialized pipeline per query, i.e., need training for each query. Since we train PPs for simple predicates and use the QO to generalize, our approach can help many more queries, even those that are previously unseen, at lower training and runtime costs. Empirical comparisons and some more details are in §8.

4 SYSTEM DESIGN

Language support for UDFs: Similar to recent query languages that support user defined functions (UDFs) [11, 12, 36], our query language offers some new templates for UDFs; a developer can implement a UDF by inheriting from the appropriate UDF template. The *processor* template, which we saw earlier in §1, encapsulates row manipulators; they produce one or more output rows per input row. Processors are typically used to ingest data and per-blob ML operations such as feature extraction. *Reducers* encapsulate operations over groups of related items. Context-based ML operations, such as object tracking which uses an ordered sequence of frames from a camera, are built as reducers. On the query plan, reducers may translate to a partition-shuffle-aggregate. *Combiners* encapsulate custom joins, i.e., operations over multiple groups of related items. Similar to a join, they can be implemented in a few different ways, e.g., broadcast join, hash join etc. More details can be found in our previous work [36].

System inputs and outputs: With the above background, the inputs to our system are queries that may optionally have one or more user functions defined using the offered templates. The outputs are query results. As shown in Figure 3(a), the baseline system computes a query plan using a cascades-style cost based query optimizer. Our proposed architecture, shown on the right, extends the baseline system in two ways: it trains and injects probabilistic predicates into query plans. The architecture has slight differences based on whether it is used in an online or batch context as we will describe next.

Constructing PPs: The basic task of constructing a probabilistic predicate uses binary labeled input data. The labels specify whether an input blob passes or fails the predicate. The output is a PP annotated with the predicate clause that it corresponds to, the cost of execution, and the predicted data reduction vs. accuracy curve. Further details are in §5.

The "outer loop" of deciding which clauses to train PPs for and how to acquire labeled input, shown in Figure 3(b), is as follows. In a batch system, we use historical queries to infer the simple clauses (defined in §3) that appear frequently in the queries. To train probabilistic predicates for these clauses, we find that labeled input data is sometimes already available because a similar corpus was used to build the original UDFs (e.g., training the classifiers). Alternatively, we can generate the labeled corpus by annotating the query plans; i.e., the first query to use a certain clause will output labeled input in addition to its query results. In an online system, the above process runs contemporaneously with the query execution. That is, at cold start when no PP is available, the query plans output labeled inputs for relevant clauses; periodically or when enough labeled input is available, the PPs are trained and subsequent runs of the query use query plans containing the trained PPs.

Applying PPs: Our modified query optimizer, shown in Figure 3(c) takes two additional inputs compared to the baseline QO: a list of trained probabilistic predicates and a desired accuracy threshold for the query. As described in §6, the modified query optimizer injects appropriate combinations of PPs for each query based on the accuracy threshold; the PPs, shown in the figure as green dotted circles, execute directly on the raw inputs and the remaining query plan is semantically equivalent to the original query plan.

5 TRAINING INDIVIDUAL PPS

In this section, we describe the details of building a probabilistic predicate (PP). A PP for predicate clause p is uniquely characterized by the triple $\mathsf{PP}_p = \{\mathcal{D}, m, r[a]\}$ where:

Training Set $\mathcal{D}$ is the portion of data blobs on which PP_p is constructed. Each blob $x \in \mathcal{D}$ has an associated label $\ell(x)$ which is $+1$ for blobs that agree with p, and -1 for those that disagree with p.

Approach m is the filtering strategy picked by our model selection scheme, indicating which classification $f(\cdot)$ and dimension reduction $\psi(\cdot)$ algorithms to use. The cost of the PP can be read from Table 2 for different approaches.

Data reduction rate $r[a]$ is the portion of data blobs filtered by PP_p given the above settings. $a \in [0, 1]$ is the target accuracy, e.g., 1.0 or 0.95. We will train PPs that are parametrized with a target accuracy.

5.1 PP classifier 1: linear SVM

To identify data blobs that disagree with p, we consider linear support vector machines (SVMs) [50] which are well-known binary classifiers. A linear SVM classifier has the form of:

$$f_{\text{lsvm}}(\psi(\mathbf{x})) = \mathbf{w}^T \cdot \psi(\mathbf{x}) + b, \tag{1}$$

Approach		Space complexity (per n input)		Computational complexity				Applicability		
				Training (per n input)		Testing (per input)				
Dim. reduction $\psi(\cdot)$	Classifier $f(\cdot)$	ψ cost	f cost	ψ cost	f cost	ψ cost	f cost	Nonlinear	Dense	High-dim
None, $\psi(\mathbf{x}) = \mathbf{x}$	Linear SVM	-	$O(d)$	-	$O(\max(n,d)\min(n,d)^2)$	-	$O(d)$	X	✓	✓
	KDE	-	$O(nd)$	-	$O(n \log d)$	-	$O(n' \log d)$	✓	✓	X
	DNN	-	$O(d_m)$	-	$O(bn(c_f + c_b + c_u))$	-	$O(c_f)$	✓	✓	✓
PCA, $\psi(\mathbf{x}) = \mathbf{x}P$	Linear SVM	$O(dd_r)$	$O(d_r)$	$O(\min(n^2 d, nd^2))$	$O(nd_r^2)$	$O(d)$	$O(d_r)$	X	✓	✓
	KDE	$O(dd_r)$	$O(nd_r)$	$O(\min(n^2 d, nd^2))$	$O(n \log d_r)$	$O(d)$	$O(n' \log d_r)$	✓	✓	✓
Feature Hashing, $\psi(\mathbf{x}) = \sum_j \eta(j) \cdot \mathbf{x}_j$	Linear SVM	-	$O(d_r)$	$O(nd)$	$O(nd_r^2)$	$O(d)$	$O(d_r)$	X	X	✓
	KDE	-	$O(nd_r)$	$O(nd)$	$O(n \log d_r)$	$O(d)$	$O(n' \log d_r)$	✓	X	✓

Table 2: Complexity of different PP approaches for different dimension reduction ψ and classifier f techniques. n is the number of data items in the (sampled) training set; d (d_r) is the number of dimensions in vector $\mathbf{x}$ (that remain after dimensionality reduction); n' is the number of neighbor nodes in the k-d tree; d_m is the number of parameters in the DNN model; b is the number of epochs; $c_f/c_b/c_u$ are the forward/backward propagation/update costs. We assume $d_r \ll n$.

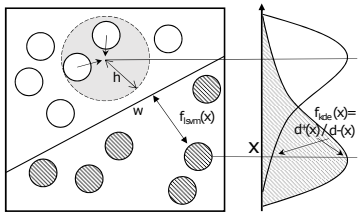


Figure 4: Demonstration of computing $f(x)$ by the PP classifiers. Left: SVM-based PP tries to find the decision boundary w. Right: 1-D visualization of the +1/-1 densities (dark circles for +1 and white circles for -1). KDE-based PP measures $f_{kde}(x) = d^+(x)/d^-(x)$ where $d^\cdot$ is estimated with a neighborhood of h.

where $\psi(\mathbf{x})$ denotes a dimension reduction technique to project the input blob $\mathbf{x}$ onto fewer dimensions. We will discuss different dimension reduction techniques later in §5.4. $\mathbf{w}$ is a weight matrix and b is a bias term; the training fits $f(\cdot)$ to the labels $\ell(\cdot)$ of the blobs in the training set $\mathcal{D}$ [25].

Constructing the PP: Equation 1 can be interpreted as a hyperplane that separates the labeled inputs into two classes as shown in Figure 4 (left). Perfect separation may not always be possible and hence we use the following decision function to predict the labels:

$$\mathsf{PP}(\mathbf{x}) = \begin{cases} +1 & \text{if } f(\psi(\mathbf{x})) > th[a] \\ -1 & \text{otherwise} \end{cases} \tag{2}$$

where $th[a]$ is a decision threshold under the desired filtering accuracy a. It is easy to see that different values of $th[a]$ will produce different accuracy and reduction ratio. For example, with $th = -\infty$ all blobs will be predicted to pass the predicate ($\mathsf{PP}(\mathbf{x}) = +1$), leading to zero reduction and perfect accuracy $a = 1$. We choose the parametric threshold $th[a]$ as follows:

$$th[a] = \max th \text{ s. t. } \frac{|\{x \in \mathcal{D} : f(\psi(\mathbf{x})) > th\}|}{|\{x \in \mathcal{D} : \ell(x) = +1\}|} \geq a. \tag{3}$$

It is useful to note that since the decision function is deterministic regardless of the $th[a]$ value, a PP parametrized for different accuracy thresholds can be built without retraining the SVM classifier. Figure 5 and Appendix B show examples of choosing $th[a]$. Finally, the reduction ratio achieved by the PP can be computed as:

$$r[a] = 1 - \frac{|\{x \in \mathcal{D} : f(\psi(\mathbf{x})) > th[a]\}|}{|\mathcal{D}|} \tag{4}$$

Usage notes: Linear SVMs have pros and cons. They can be trained efficiently (see Table 2) and have small cost at test. However, linear SVMs yield a poor PP if (a) the input blobs are not linearly separable or (b) meeting the desired filtering accuracy results in a small data

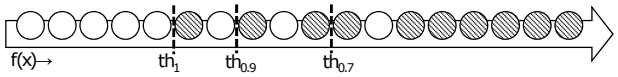


Figure 5: Data rows are ranked in ascending order according to their $f(x)$ values. Dark and white circles represent data blobs with +1 and -1 labels respectively. Threshold $th[a]$ is chosen to be the largest threshold value that correctly identifies an a portion of the +1 data points.

reduction. Using non-linear SVM kernels (e.g., RBF kernel [50]) is a potential fix; however, the computational complexity significantly increases for both training and inference, resulting in practically ineffective PPs. We introduce an alternate classification method below that is effective even when the problem is not linearly-separable.

5.2 PP classifier 2: KDE

Machine learning blobs such as images and videos are high dimensional and not always linearly-separable. Here, we construct a non-parametric PP classifier that does not assume any underlying data distribution. Intuitively, a set of labeled blobs can be translated into a density function such that the density at any location x indicates the likelihood of its belonging to the set. Consider the density functions in Figure 4 (right). We propose to compute two density functions for the blobs in the training set according to their labels; let $d^+(\psi(\mathbf{x}))$ and $d^-(\psi(\mathbf{x}))$ be the density (likelihood) that $\psi(\mathbf{x})$ has a +1 or -1 label, respectively. As shown in the figure the density functions may overlap. As before, $\psi(\mathbf{x})$ denotes a dimension reduction technique. We then have the following kernel density estimator:

$$f_{\text{kde}}(\psi(\mathbf{x})) = d^+(\psi(\mathbf{x}))/d^-(\psi(\mathbf{x})). \tag{5}$$

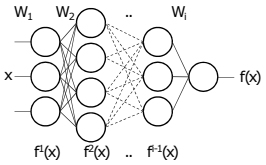
Intuitively, data points $\mathbf{x}$ with a true label of +1 should have a higher value on $d^+(\psi(\mathbf{x}))$ than $d^-(\psi(\mathbf{x}))$, leading to a high f_{kde} value; similarly if $\mathbf{x}$ has a true label of -1, f_{kde} should be low.

To build the density functions d^+ and d^-, we leverage kernel density estimation (KDE) [43]. $d^+(\psi(\mathbf{x}))$, the density of points with +1 labels, is defined as

$$d_h^+(\psi(\mathbf{x})) = \sum_{i=0, \ell_i=+1}^{n} K\Big(\frac{\psi(\mathbf{x}) - \psi(\mathbf{x_i})}{h}\Big) \tag{6}$$

where h is a fixed parameter indicating the size of $\psi(x)$'s neighborhood that we should look into. K is the kernel function to normalize $\psi(\mathbf{x})$'s neighborhood and we use a Gaussian kernel which yields smooth density estimations. $d^-(\psi(\mathbf{x}))$ is defined similarly over data blobs having -1 labels. We choose h using cross-validation; Silverman's rule of thumb [45] can also be used to pick an initial h.

Constructing the PP: To complete the construction of a probabilistic predicate using the KDE method, we note that Equations 2, 3, 4 can



$$f^i_{\text{fcn}}(\mathbf{x}) = g_i(W_i \cdot f^{i-1}_{\text{fcn}}(\mathbf{x}) + b_i),$$
$$g_i(x) = \text{ReLU}(x) = \max(0, x) \text{ or...}$$

Figure 6: Left: structure of a fully connected neural network. W_i are different fully connected layers. Right: formula at layer i. The input $f^0_{\text{fcn}}(\mathbf{x}) = \mathbf{x}$.

be applied by using f_{kde} in place of f_{lsvm}. In particular, as with the case of the linear SVM PP, we can parametrize the KDE PP without retraining the classifier.

Usage notes: Probabilistic predicates using the KDE method are effective even when the underlying data is not linearly separable; however this comes with some additional cost during test as noted in Table 2. In particular, applying the KDE PP at test time may require a pass through the entire training set because the densities d^+ and d^- are computed based on the distance between the test point $\mathbf{x}$ and each of the training points. To avoid this, we use k-d tree [8], a data structure that partitions the data by its dimensions. Similar data points are assigned to the same or nearby tree nodes. With a k-d tree, the density of an input blob $\mathbf{x}$ is approximately computed by applying Eq. 6 only on $\psi(\mathbf{x})$'s neighbors retrieved from the k-d tree (i.e., n' nodes as shown in Table 2 where $n' \ll n$, the number of training samples). The retrieval complexity is (on average) logarithmic in the feature length of the input blob.

5.3 PP classifier 3: DNN

To demonstrate how the core classification methods can be extended, we consider the case of building a PP using a deep neural network (DNN) [33]. As shown in Figure 6, the classifier can have multiple fully connected layers interpreted as multiplying an input blob with different weight matrices sequentially. The function g_i, implemented as ReLU, sigmoid etc., is a non-linear activation applied after each fully connected layer, introducing non-linearity to the whole model.

Constructing the PPs: We argue that the PP design can incorporate any classifier that can be cast as a real-valued function with a threshold (i.e., as f in Eq. 2); the applicability of the classifier, of course, depends on the data distribution, predicates and classifier costs. In particular, DNNs also fit this requirement and we build DNN PPs by using f_{fcn} from Fig. 6 in equations 2, 3 and 4.

Usage notes: DNNs have shown promising classification performance in various ML applications [31, 32]. However, the number of parameters to train is much larger (e.g., weight matrices) than the other classifiers we have discussed. Hence, training a DNN requires more data and the training cost is significant. In practice, PPs built using DNNs are appropriate for queries and predicates that (a) have very expensive UDFs (e.g., a large DNN), (b) have a large training corpus or (c) repeat frequently to justify higher training costs.

5.4 Dimension reduction $\psi(\cdot)$

In practice, input blobs have many dimensions; for example, in videos, each pixel in a frame or an 8x8 patch of pixels can be construed as a dimension. In bag-of-words representations of natural language text, each distinct word is a dimension and the vector $\mathbf{x}$ for a document is the frequency of its words. When the dimensionality increases, the Euclidean distances used to compute $\mathbf{w} \cdot \mathbf{x}$ and $\mathbf{x} - \mathbf{x_i}$ lose discriminative power. Our overall approach to address this concern is to apply dimension reduction techniques before the classifier. However, this is optional, i.e., $\psi(\mathbf{x})$ can be $\mathbf{x}$.

Principal Component Analysis (PCA) [28] is a popular technique for dimension reduction. The input $\mathbf{x}$ is projected using $\psi(\mathbf{x}) = \mathbf{x}P$, where P is the linear basis extracted from the training data. We note two aspects. First, even when the underlying data is not linearly separable, applying PCAs does not prevent the subsequent classifier from identifying blobs that are away from the decision boundary. Second, computing the PCA basis using singular value decomposition is quadratic in either the number of blobs in the training set or in the number of dimensions $O(\min(n^2 d, nd^2))$ [19]. To speed this up further, we compute PCA over a small sampled subset of the training data $\mathcal{D}$, trading off reduction rate for speed. Note the formulas in Table 2 where n can be either the full training set or the sampled subset.

Feature Hashing (FH). Feature hashing [53] is another popular dimension reduction technique which can be thought of as a simplified form of PCA that requires no training and is well-suited for sparse features. It uses two hash functions h and η as follows:

$$\forall i = 1 \ldots d_r, \quad \psi^{(h,\eta)}_i(\mathbf{x}) = \sum_{j=1}^{d} \mathbf{1}_{h(j)=i} \cdot \eta(j) \cdot \mathbf{x}_j, \tag{7}$$

where the first hash function $h(\cdot)$ projects each original dimension index ($j = 1 \ldots d$) into exactly one of d_r dimensions and the second hash function $\eta(\cdot)$ projects each original dimension index into ± 1, indicating the sign of that feature value. Thus the feature vector is reduced from d to d_r dimensions. It is easy to see that feature hashing is inexpensive and it has been shown to be unbiased [53]. However, if the input feature vector is dense, hash collisions are frequent and classifier accuracy becomes worse.

5.5 Model Selection

Thus far, we have described three techniques to construct PPs and two dimension reduction techniques, all of which can be used with or without sampling the training data and several parameter choices (e.g., number of reduced dimensions d_r for FH); this leads to many possible techniques for PPs. As we describe next, we expect future systems to use a few other techniques. Hence, it is crucial to determine quickly which technique is the most appropriate for a given input dataset. We use the following model selection.

Given different PP methods $\mathcal{M}$, we select the best approach m by maximizing the reduction rate r_m for that approach:

$$m = \arg\max_{m \in \mathcal{M}} r_m[a]. \tag{8}$$

Furthermore, these methods have different applicability constraints as summarized in Table 2. We first prune $\mathcal{M}$ using these applicability constraints. To compute $r_m[a]$ quickly, we use a sample of the training data, fix $a = 0.95$, randomly choose a few different simple clauses, train the classifiers described above and use the technique that performs *better*. Our experiments show that the input dataset has the strongest influence on technique choice; that is, given a certain

type of input blobs, the same PP technique is appropriate for different predicates and accuracy thresholds etc.

5.6 Other PP details

Overfitting: To avoid overfitting on the training data, we randomly divide the input set of blobs $\mathcal{D}$ into training and validation portions. The classifiers are trained using the training portion $\mathcal{D}_{\text{train}}$ but the accuracy-data reduction curve $r[a]$ is calculated on the validation portion $\mathcal{D}_{\text{val}}$.

Classifiers built for a PP on predicate p can be reused for the PP on predicate $\neg p$: Given the classifier functions (e.g., $f_{\text{lsvm}}, f_{\text{kde}}$) built for a predicate p, note that multiplying these functions with -1 yields the corresponding classifier functions for predicate $\neg p$. Hence, the PP for predicate $\neg p$ can reuse the classifier and compute equations 3 and 4 with $-1 * f$ instead.

Input feature to PP is a simple representation of the data blob, e.g., raw pixels for images, concatenation of raw pixels over consecutive frames (of equal duration) for videos, and tokenized word vectors for documents.

6 QUERY OPTIMIZATION OVER PPS

In §5, we have seen how to construct PPs for simple clauses. Here, we describe interaction with the query optimizer which achieves the following goals.

First, for a query with a complex predicate or previously unseen predicate, which PPs may be useful? Recall that a query can use any available PP or combination of available PPs that is a necessary condition to the actual predicate. Given a complex query predicate $\mathcal{P}$, the QO generates zero, one or more logical expressions $\mathcal{E}$ that are equivalent or necessary conditions for $\mathcal{P}$ but only contain conjunctions or disjunctions over simple clauses. That is, $\mathcal{P} \Rightarrow \mathcal{E}$. The challenge, as we will show, is that there can be innumerably many choices of $\mathcal{E}$; so exploration of choices has to be quick and effective. Further details are in §6.1.

Next, how to pick the best implementation over the available expressions of PPs while meeting the query's accuracy threshold? For individual PPs, their training already yields a cost estimate and the accuracy v.s. data reduction curve. The challenge is to generate these estimates for logical expressions over PPs. Our QO extension explores different orderings of the PPs within an expression $\mathcal{E}$ and explores different assignments of accuracy to each PP which ensure that the overall expression meets the query-level accuracy threshold. Further details are in §6.2. The QO extension outputs a query plan with the chosen implementation.

Example: Consider a complex predicate of the form: $\mathcal{P} = (p \vee q) \wedge \neg r \wedge \mathcal{P}_{\texttt{rem}}$. Here p, q and r are simple clauses for which PPs have been trained and $\mathcal{P}_{\texttt{rem}}$ is the remainder of the predicate. Each PP is uniquely characterized in part by the simple clause that it mimics; we use PP_{p} to denote the PP corresponding to the simple clause p. Table 3 (right) shows the various possible expressions over PPs that may be used to support this complex predicate. We note a few points here. (1) Some parts of $\mathcal{P}$, such as $\mathcal{P}_{\texttt{rem}}$ in this example, that are attached by an '*and*' can be ignored since PPs corresponding to the rest part will be necessary conditions for $\mathcal{P}$. (2) When the predicate has a conjunction over simple clauses, PPs for one or more of these clauses can be used. This is shown in the first two rows of the table. (3) A disjunction of two PPs, e.g., $\mathsf{PP}_p \vee \mathsf{PP}_q$ is a valid PP for the disjunction $p \vee q$. The proof follows from observing Figure 7; only blobs that do not pass both the PPs will be discarded (shown using lines labeled with a '–'). As before, there will be no false positives since the actual predicate applies on the passed blobs but there can be some false negatives. A similar proof holds for a conjunction as well; an example is shown in Figure 8. Rows one and three of Table 3 show the use of the disjunction and conjunction rewrite respectively. Such rewrites substantially expand the usefulness of PPs; because otherwise PPs would need to be trained not just for individual simple clauses but for all combinations of simple clauses. (4) The predicate can also be rewritten logically, leading to more possibilities for matching with PPs; for example, $(p \vee q) \wedge \neg r \Leftrightarrow (p \wedge \neg r) \vee (q \wedge \neg r)$ leads to the PP expression shown in the fourth and fifth row of the table. (5) The number of implied expressions over PPs that correspond to a complex predicate can be substantial; the table shows eight possibilities.

Complex predicate	Implied logical expr. over PPs
$(p \vee q) \wedge \neg r \wedge \mathcal{P}_{\texttt{rem}}$	$\Rightarrow p \vee q \Rightarrow \mathsf{PP}_{p \vee q} \Rightarrow \mathsf{PP}_p \vee \mathsf{PP}_q$ $\Rightarrow \neg r \Rightarrow \mathsf{PP}_{\neg r}$ $\Rightarrow \mathsf{PP}_{(p \vee q) \wedge \neg r} \Rightarrow (\mathsf{PP}_p \vee \mathsf{PP}_q) \wedge \mathsf{PP}_{\neg r}$ $\Rightarrow \mathsf{PP}_{(p \wedge \neg r) \vee (q \wedge \neg r)} \Rightarrow \mathsf{PP}_{p \wedge \neg r} \vee \mathsf{PP}_{q \wedge \neg r} \Rightarrow$ $(\mathsf{PP}_p \wedge \mathsf{PP}_{\neg r}) \vee (\mathsf{PP}_q \wedge \mathsf{PP}_{\neg r})$

Table 3: An example rewriting of a complex predicate to expressions having conjunctions or disjunctions of probabilistic predicates.

6.1 Complex predicate to expressions over PPs

The inputs are a complex predicate $\mathcal{P}$ and a set $\mathcal{S}$ of trained PPs, each of which corresponds to some simple clause, i.e., $\mathcal{S} = \{\mathsf{PP}_p\}$. The goal is to obtain expressions $\mathcal{E}$ that are conjunctions or disjunctions of the PPs in $\mathcal{S}$ which are implied by $\mathcal{P}$, i.e., $\mathcal{P} \Rightarrow \mathcal{E}$.

If there are m PPs, i.e., $|\mathcal{S}| = m$, and n of the PPs directly match some clauses in a CNF representation of $\mathcal{P}$, then there are at least 2^n choices for $\mathcal{E}$. Since this problem has exponential-sized output, it will require exponential time.

We offer a greedy solution that is based on the intuition that expressions with many PPs will have higher execution cost; as seen in §3 early filters that have a high cost must have a relatively larger data reduction in order to perform better than the baseline plan.

The input query predicate is sent to a wrangler which greedily improves matchability with available PPs. Examples of the wrangling rules include transforming a not-equal check into disjunctions of equal checks (e.g., $t \neq 2 \Rightarrow t > 2 \vee t < 2$) or relaxing a comparison check (e.g., $t < 5 \Rightarrow t < 10$). We defer the details to Appendix A.2.

Next, we convert predicates to expressions over PPs; examples of which are shown in Table 3. For a predicate $\mathcal{P}$, let $\mathcal{P} \backslash p$ denote the remainder of the $\mathcal{P}$ after removing a simple clause p. With this notation, we use the rewrite rules below to generate expressions over PPs. All of the expressions in Table 3 can be generated by repeated application of the first three rewrite rules.

Rule R1: $p \wedge (\mathcal{P} \backslash p) \Rightarrow \mathsf{PP}_p$, **Rule R2:** $\mathsf{PP}_{p \wedge q} \Rightarrow \mathsf{PP}_p \wedge \mathsf{PP}_q$,

Rule R3: $\mathsf{PP}_{p \vee q} \Rightarrow \mathsf{PP}_p \vee \mathsf{PP}_q$, **Rule R4:** $p \wedge (\mathcal{P} \backslash p) \Rightarrow \neg \mathsf{PP}_{\neg p}$.

The fourth rule above helps for predicates with high selectivity; however, it has narrower applicability. For simplicity, we defer discussion of this rule to Appendix A.3. To construct implied logical expressions over PPs, we use the following greedy steps. (1) We limit the number

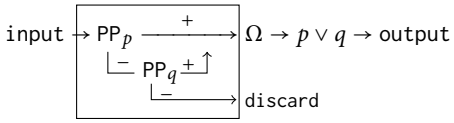


Figure 7: Injected query plan for the pattern $p \vee q \Rightarrow \mathsf{PP}_p \vee \mathsf{PP}_q$

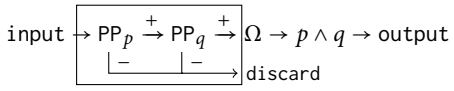


Figure 8: Injected query plan for the pattern $p \wedge q \Rightarrow \mathsf{PP}_p \wedge \mathsf{PP}_q$

of different PPs that are in any expression $\mathcal{E}$ to be at most a small configurable constant k. (2) We apply rules R2 and R3 only if the larger clause (e.g., $p \vee q$ or $p \wedge q$) does not have an available PP in $\mathcal{S}$ or if at least one of the simpler clauses has a PP that performs better (a smaller ratio of cost to data reduction $\frac{c}{r[1]}$ indicates better performance); intuitively, this prevents exploring possibilities that are unlikely to perform better.

For the example in Table 3, suppose $k = 2$ and the set of available PPs, $\mathcal{S}$, in increasing order of $\frac{c}{r[1]}$ is $\{\mathsf{PP}_{p\vee q}, \mathsf{PP}_p, \mathsf{PP}_{p\wedge\neg r}, \mathsf{PP}_{q\wedge\neg r}, \mathsf{PP}_q, \mathsf{PP}_{\neg r}\}$. It is easy to see that our algorithm only outputs three possibilities; i.e., $\{\mathcal{E}\} = \{\mathsf{PP}_{p\vee q}, \mathsf{PP}_{\neg r}, \mathsf{PP}_{p\wedge\neg r} \vee \mathsf{PP}_{q\wedge\neg r}\}$. The other possibilities are pruned by our greedy checks.

6.2 Costing query plans with PP expressions

Given a set of expressions $\{\mathcal{E}\}$ that are conjunctions or disjunctions of PPs, the goal is to compute the lowest cost query plan which meets query's accuracy threshold. If some execution plan for $\mathcal{E}$ has a per-blob cost of c and reduction-vs-accuracy of $r[a]$, then (recall from §3 that) the query plan cost is $\propto c + (1 - r[a]) * u$, where u is the cost per blob of executing the original query. u and a are inputs to the algorithm but c and $r[a]$ have to be computed.

Since the order in which the PPs in $\mathcal{E}$ execute and how the accuracy budget is allocated among the individual PPs crucially affect plan cost, we have three sub-problems. First, we have to explore different allocations of the query's accuracy budget to individual PPs. Next, we have to explore different orderings of PPs within a conjunction or disjunction; this process recurses for nested conjunctions or disjunctions. Finally, after fixing both the accuracy thresholds and the order of PPs, we have to compute the cost and reduction rate of the resulting plan. The first problem translates to a dynamic program which we omit for brevity. For the second part, recall that there are at most k PPs in any $\mathcal{E}$; if k is small, then all of the exponentially many orderings can be explored. When k is large, we use the following heuristic: consider ordering the PPs by the ratio of their intrinsic $\frac{c}{r[1]}$ and then consider all other orderings that are an edit-distance of at most 2 away from this greedy order. In practice, we found these to be the most useful orderings. The last part, computing cost and reduction rate given a fixed PP order and fixed accuracy thresholds, proceeds inductively as follows.

Base case: $\mathcal{E} = \mathsf{PP}_p$. Here the cost and accuracy vs. data reduction curve of $\mathcal{E}$ is the same as that of PP_p.

Conjunction: $\mathcal{E} = \mathcal{E}_1 \wedge \mathcal{E}_2$. Let the cost of the two logical expressions be c_1, c_2 and their accuracy vs. data reduction curves be $r_1[a], r_2[a]$ respectively. Figure 8 shows an example conjunction. Suppose each PP has been given an accuracy threshold of a_1 and a_2. We make the simplifying assumption that the PPs are independent; a fix is described in Appendix A.5. We now have:

$$\begin{array}{lcl} a & = & a_1 * a_2 \\ r[a] & = & r_1[a_1] + r_2[a_2] - r_1[a_1] * r_2[a_2] \\ c[a] & = & \min(c_1 + (1 - r_1[a_1]) * c_2,\ \ c_2 + (1 - r_2[a_2]) * c_1) \end{array} \quad (9)$$

Disjunction: $\mathcal{E} = \mathcal{E}_1 \vee \mathcal{E}_2$. Figure 7 shows an example disjunction. With the same notation as in the case of conjunction and with similar assumptions, we have:

$$\begin{array}{lcl} a & = & a_1 + a_2 - a_1 * a_2 \\ r[a] & = & r_1[a_1] * r_2[a_2] \\ c[a] & = & \min(c_1 + r_1[a_1] * c_2,\ \ c_2 + r_2[a_2] * c_1) \end{array} \quad (10)$$

Note the following intuitions for conjunction based on Eq. 9; analogous intuitions apply for disjunctions. (1) Accuracy reduces multiplicatively. (2) Data reduction ratio improves but the marginal improvement is less when many PPs are used and if the individual sub-expressions are already highly reductive. For example, if two expressions have a reduction rate of 0.1, the conjunction nearly doubles its data reduction to 0.19; however when each reduction rate is 0.8 the conjunction only increases to 0.96. (3) The cumulative cost is smaller when the sub-expression with the smaller $\frac{c}{r[a]}$ executes first. Our heuristic algorithm above is based on these intuitions.

7 CASE STUDIES

We discuss four case-studies used in our experimental evaluations: document analysis, image analysis, video activity recognition and comprehensive traffic surveillance. The input datasets have numbers of dimensions ranging from thousands (e.g., low-res images) to hundreds of thousands (e.g., bag-of-words representations of documents which can be very sparse). Some predicates are correlated (e.g., hierarchical labels of document and activity types in videos). The selectivity of predicates also varies widely; some predicates have very low selectivity (e.g., 'has truck' in traffic video). We evaluate different machine learning queries on these datasets as described below.

Case1: Document analysis. We use the LSHTC [40] dataset which contains 2.4M documents from Wikipedia. Each document is represented as a bag of words with a frequency value for each of 244K words; this vector is sparse in practice. The LSHTC dataset classifies the documents into 400K categories. The mapping between documents and categories is many-to-many; that is, a document can belong to many categories and vice versa. The dataset also offers a hierarchy over categories. We consider queries that retrieve documents having one or more categories.

Case2: Image labeling. The SUNAttribute [41] dataset contains 14K images of various scenes. The images are annotated with 802 binary attributes that describe the scene, such as 'is kitchen', 'is office', 'is clean', 'is empty' etc. We consider queries that retrieve images having one or more attributes. We also use the popular COCO [35] and ImageNet datasets [31] in a similar manner; i.e., queries retrieve images that contain one or more labels. COCO contains 120K images, each labeled with one or more of the 80 object classes. We use a subset of 110K images from ImageNet with the same 80 classes as in the COCO dataset to evaluate the cross-domain application of PPs, i.e., training PPs on COCO but testing on ImageNet.

Case3: Video activity recognition. We use the UCF101 video activity recognition dataset [46], which has 13K video clips with durations ranging from ten seconds to a few minutes. Each video clip is annotated with one of 101 action categories such as 'applying lipstick', 'rowing', etc. We consider the problem of retrieving clips that illustrate an activity.

Case4: Comprehensive Traffic Surveillance Video Analytics. The queries thus far retrieve (different) portions of the inputs. Here, we consider the problem of answering comprehensive queries on traffic surveillance videos. Our datasets include hours of surveillance videos from the DETRAC [54] vehicle detection and tracking benchmark. We design a query set, TRAF20 (§8.2), upon these videos; the queries perform machine learning actions such as vehicle detection, color and type classification, traffic flow estimation (vehicle speed and flow) etc. While DETRAC already annotates vehicles by their types (sedan, SUV, truck, and van/bus), we manually annotate the vehicles in the video with their color (red, black, white, silver and other).

8 EVALUATION

The experiments shown in this section have the following purposes:

Validating individual PPs. The first-order question we are interested in is how much speed-up can PPs offer to various machine learning inference queries over unstructured input blobs. We inject a PP into queries that have one simple predicate in §8.1. We also examine the suitability of PPs that are trained using different techniques. Our results will show that injecting PPs achieves speed-ups that are 3×–19× more than a state-of-the-art baseline [27] on different machine learning datasets.

Evaluation of our query processing system. Putting everything together, §8.2 evaluates using PPs on complex query predicates in Microsoft's Cosmos big-data cluster [11]. We demonstrate the costs to construct PPs on large datasets, how the QO chooses appropriate combinations of available PPs and the inference costs of applying PPs. These end-to-end experiments show that using probabilistic predicates can accelerate real-world machine learning inference by up to 12.5× under reasonable target accuracy and budget on training cost.

8.1 Micro-benchmarks on individual PPs

Dataset, predicates, UDFs and queries. To demonstrate that we can train PPs for a variety of datasets, we evaluate using PPs on queries that have one simple predicate. We use Cases 1-3 here; recall that the queries check for inputs that match a given category. To support these queries, we have built various feature extraction [15, 38] and classifier [3] UDFs. The classifier output, per category, is a binary column with value 1 if and only if the input blob matches that category, and the query predicates check the value of this column. For Case1, we randomly pick 140 categories, and use all categories for the other datasets. In all, this experiment has about a thousand queries and upwards of a thousand different UDFs.

Training PPs: For each query, we randomly take 60% of the entire dataset as the training set to construct the PP classifiers; the validation and testing set each takes 20% of the dataset. We also experiment with different training sizes.

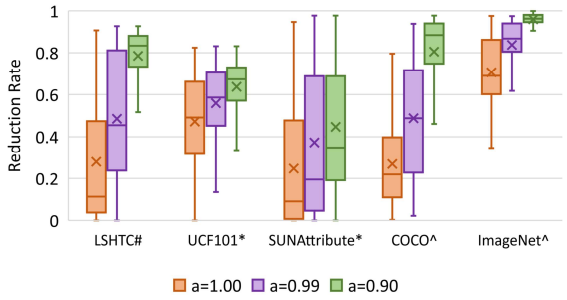


Figure 9: Whisker plots of the data reduction rates across various datasets. Each bar is a whisker plot; the lines are the min and max reduction across queries; the ends of the box are the 25th and 75th percentiles; the horizontal line in the box is the 50th percentile and x marks the average. Different PP techniques are used across datasets: # indicates PPs that use feature hashing + SVM, ∗ indicates PPs with PCA + KDE and ^ indicates PPs with a DNN.

Metrics used in our evaluations include the selectivity s_p of each predicate p, the accuracy a of the PP which is the fraction of output of the original query that is returned after using the PP, data reduction ratio $r_p[a]$ due to the PP at accuracy a. Note that accuracy is relative to the ground truth labels; the UDFs can often be imperfect. We also focus on the relative reduction ($= \frac{r_p[a]}{1-s_p}$), which is the actual number of input blobs that are dropped by the PP ($r_p[a]$) divided by the maximum possible number of input blobs that can be dropped by the PP ($1 - s_p$).

Can we train effective PPs? Building effective PPs depends on several factors; here, we consider the following key elements. (1) Do we have techniques that yield PPs with a good data reduction rate and high accuracy on a variety of datasets? (2) Are PPs trained on one dataset useful for other similar datasets?

Figure 9 shows whisker plots of the data reduction ratio ($r_p[a]$) from using PPs on different datasets. Each bar is a whisker plot; the lines are the min and max reduction across queries; the ends of the box are the 25th and 75th percentiles; the horizontal line in the box is the 50th percentile and x marks the average. A couple of points are worth noting. With a strict accuracy target $a = 1$, the PPs already achieve substantial data reduction. Half of the PPs on UCF101 filter more than 50% of the input. The data reduction varies across datasets due in part to the nature of the datasets, the queries and the predicates. Furthermore, a small trade-off in accuracy leads to much larger improvements in the reduction rates, e.g., a 1% decrease of a improves average data reduction by about 20% on COCO, ImageNet and LSHTC. Such small changes are often acceptable for aggregation queries (e.g., counting # of cars) or for queries where the desired object occurs in multiple frames (e.g., amber alert queries).

Model selection. We also note that different PP training techniques, as noted in the caption of Figure 9, achieve the best data reduction on different datasets. The LSHTC document dataset is very sparse and the query categories are linearly separable over the features, so feature hashing + SVM leads to good PPs for Case1. Video activity recognition (UCF101) is not linearly separable but the different activities in this dataset are distinctive, so PCA + KDE suffices here. Table 4 shows that PPs using SVM achieve roughly 10% less data reduction. Image category labels are not linearly separable and the blobs are highly dimensional. For the relatively simple images in SUNAttribute, PCA

Dataset	Approach	Avg. data reduction $\overline{r}$ for accuracy a		
		$\overline{r[1]}$	$\overline{r[0.99]}$	$\overline{r[0.9]}$
UCF101	**PCA+KDE**	**0.47**	**0.56**	**0.64**
	PCA + SVM	0.35	0.45	0.54
	Raw + SVM	0.35	0.47	0.59
COCO	**DNN**	**0.28**	**0.50**	**0.83**
	SVM		0.31	
ImageNet	**DNN**	**0.71**	**0.84**	**0.96**
	SVM		0.39	
	DNN trained on COCO	0.25	0.49	0.82

Table 4: Comparing the data reduction achieved by PPs that use different techniques. The best technique appears to improve data reduction by 10% to 20% in absolute terms. Finally, cross-training, i.e., using PPs trained on a different albeit similar dataset appears promising.

Dataset	Approach	PP cost to ...		Optimality for a	
		Train (per 1K rows)	Test	$a = 1$	$a = 0.9$
UCF101	PCA+KDE	14s	3ms	0.55	0.77
LSHTC	FH + SVM	1s	1ms	0.29	0.87
COCO	DNN*	110s	10ms	0.28	0.83

Table 5: The latency to train and test PPs of different types as well as the optimality gap for different accuracy targets, a, which is = $\text{avg}_p\left(\frac{r_p[a]}{1-s_p}\right)$; i.e., the average over all predicates of the fraction of blobs that are discarded by a predicate which are discarded by the corresponding PP. * indicates w/ GPU.

+ KDE leads to good PPs. However, for the more complex images in COCO and ImageNet (multiple objects in image etc., examples are in Figure 15) DNNs are needed to get useful PPs. Table 4 shows that SVM PPs on COCO and ImageNet achieve 20% to 40% lower data reduction. Compared with state-of-the-art DNNs (e.g., ResNet [21]), the DNN used for PPs here has 8 convolutional layers followed by a fully connected layer and is relatively very light-weight. Yet, the DNN PPs offer good data reduction. We believe that there is no silver bullet (i.e., best for all cases) PP approach. We use simple heuristics, e.g., do not use feature hashing for dense features, use the least complex model that returns a good data reduction etc. Nevertheless, model selection is critical. Luckily, we also see that the behavior of PP approaches for a query and dataset can be estimated well by training on a small sampled subset of the corpus which reduces the cost of model selection. Another important aspect that reduces training and model selection cost is the ability to cross-train; that is, if we can use PPs trained on a dataset for other similar datasets. Table 4 shows in red the data reduction achieved when the DNN PPs trained on COCO are used on ImageNet. We see that cross-trained PPs are not as good as PPs trained on the same dataset but they perform reasonably well especially at relaxed accuracy targets; we consider this to be a low-cost alternative to training DNN PPs on each dataset.

Costs. Table 5 reports the time to train a PP per 1000 input blobs and the time to test on each input blob. As expected, we see that the KDE and SVM PPs can process several hundreds of blobs per second per thread. Using a GPU, the DNN PPs can process only about one hundred blobs per second. The training costs are also much larger for DNN PPs. All of these timing measurements were performed on a desktop running linux with an Intel i7-5930K processor, 16 GB of RAM and an Nvidia 1080Ti GPU.

Optimality. Table 5 also estimates an optimality gap of sorts; that is, what fraction of all the input blobs that can possibly be dropped by a PP, because the blobs will not satisfy the predicate, are actually dropped by that PP (= $\frac{r_p[a]}{1-s_p}$). The table shows values averaged over all predicates. By normalizing with predicate selectivity, this number tells us the room for improvement. We see that the PPs described in this paper only achieve 28% to 55% of the optimal data reduction at $a = 1$ but at relaxed accuracy target of $a = 0.9$ they are closer to optimal. Hence, we believe that more work on novel PP techniques is warranted especially at high accuracy targets although it is apriori unclear that more data reduction can be achieved without also paying higher training and/or execution time costs.

Target a	Method	LSHTC	SUNAttribute	UCF101
0.99	PP	0.51	0.43	0.56
	PCA + Joglekar et al. [27]	0.19	0.11	0.09
	Speed-up	2.7x	3.9x	6.2x
	Joglekar et al. [27]	0.16	0.05	0.03
	Speed-up	3.2x	8.6x	19x
Target a	Method	LSHTC	SUNAttribute	UCF101
0.90	PP	0.81	0.46	0.64
	PCA + Joglekar et al. [27]	0.36	0.15	0.14
	Speed-up	2.3x	3.1x	4.6x
	Joglekar et al. [27]	0.25	0.09	0.05
	Speed-up	3.2x	5.1x	12.8x

Table 6: Empirical reduction rates on three datasets with different target filtering accuracy.

Comparing with Joglekar et al. [27] We compare the PP classifiers with Joglekar et al [27], a system optimized for processing expensive predicates. This work leverages correlation between the input columns and the UDF outputs; consequently, they drop early based on the values of the input columns. We use their code and treat each dimension of our blobs as an input column. We compare with our PPs at different target accuracy settings (a = .99/.90) on 10 randomly picked queries from each of the three cases. Table 6 shows the comparison based on the same amount of training data. The baseline system can filter some of the sparse LSHTC inputs, since each dimension of a text input depicts a word, and intuitively correlations exist between words and the document label. However, the baseline method does not work for dense machine learning blobs (e.g., images and videos). The baseline system improves marginally when it is offered the results of applying PCA over the raw data as input. The reason, we believe is that a dimension in such blobs hardly means anything, and the correlation is usually over some complex possibly non-linear combination of multiple dimensions. On the contrary, our PPs are more suited to handle machine learning blobs that have different data distributions.

8.2 Evaluating ML with PPs

TRAF-20 benchmark. The purpose of this section is to evaluate the end-to-end system speed-up from injecting probabilistic predicates. To the best of our knowledge, there is no off-the-shelf benchmark of queries with machine learning UDFs and complex predicates. Hence, we created a benchmark, TRAF-20, with 20 inference queries over datasets from Case 4 (described in §7). Five predicate columns are generated by different machine learning UDFs, including vehicle color c and type t, speed s and direction (from i/to o). These UDFs are trained over annotated inputs. The queries mimic retrieval of vehicles that meet a specified predicate (e.g., an over-speed truck or an illegal turn). TRAF-20 has complex predicates including disjunctions and conjunctions of range, equality and inequality checks. Each query is equally likely to have between one and four predicate clauses. There are no nested predicates. Table 7 shows some example predicates from TRAF-20. Suppose the speed column is discretized to 0 – 80 mph,

#clauses	**Query ID**: Predicates (Type)
1	**Q1**: t=SUV (E), **Q2**: $s > 60$ (N), **Q4**: $c \neq white$ (I)
2	**Q7**: $s > 60$ & $s < 65$ (NR) , **Q8**: $t \in \{sedan, truck\}$ (ER)
3	**Q14**: i=$pt303$ & (o=$pt335$ | o=$pt306$) (ECD)
4	**Q20**: t=SUV & c=red & i=$pt335$ & o=$pt211$ (EC)

Table 7: TRAF-20 predicate examples. We use ptX to indicate traffic intersections in the dataset. E: equality check. I: inequality check. N: real numbers. R: range check. C: conjunction. D: disjunction.

there are roughly 100 different values that different UDF-generated columns can take. Hence, the space of potential query predicates is about 100^4. Training a filter for every possible predicate may not be feasible in practice.

Method and metrics. We mimic the use of PPs in an online setting. PPs are built upon the first 1GB of input data and UDF outputs; 80% of the blobs are used for training and 20% for validation. Overall, we have built 32 PPs, all of which are trained using SVMs, each corresponding to a single predicate clause. Our system executes query plans having some appropriate combination of these PPs.

We report the training costs as well as the overall system speed-up to process the subsequent data blobs. We measure query performance using two metrics: cluster processing time and query latency; these metrics are commonly used in recent data-parallel systems [5, 11]. Cluster processing time is the overall cluster resource usage and includes the cost of executing PPs, and query latency is the end-to-end user waiting time taking PP overhead into account. We also report the empirical reduction rate and the percentage of cluster processing time saved by applying PPs. Note that query latency is affected by a small number of outlier tasks and other scheduling artifacts; hence, it is much more variable than the cluster processing time of queries.

Comparisons. We compare our query processing system, end-to-end, with two baselines. (1) Optasia [36] is a relational data-parallel platform for large-scale vision/ machine learning which is built upon Microsoft Cosmos. It does not apply any early-filtering strategy. We refer to this baseline as *NoP* in our experiments. Our system uses a similar cost-based query optimizer to translate machine learning scripts into relational operators. (2) Deshpande et al. [17], building upon [7], optimally order multiple predicates such that cheap and data-reductive predicates execute earlier in the plan. They also output conditional query plans when predicate costs or selectivity vary (e.g., $temp. > 40^{\circ}C$ has low selectivity at night). However, they still require predicate columns to be available on the inputs. We implement their scheme in our query processing system and refer to it as *SortP*.

End-to-end results. Figure 10 illustrates the speed-up in cluster processing time on 100 GBs of traffic surveillance videos relative to the baseline without PPs (*NoP*). The queries on the x-axes are ordered in increasing order of the speed-up. Table 8 reports the query execution latency for different schemes when processing different amounts of input. From Figure 10, we see that every scheme uses fewer resources than *NoP* [36]; this is as expected, since in *NoP* all blobs go through all UDFs. *SortP* [17] has a small speed-up (average is 1.2×) because based on the ordering of predicates, when predicates have multiple clauses, blobs that do not pass predicates early in the plan can avoid being processed by the UDFs which generate columns for predicates that are later in the plan. Note however that while *SortP* lowers resource usage, it substantially increases the job latency because serializing the predicates (and UDFs) leads to longer critical paths. We see

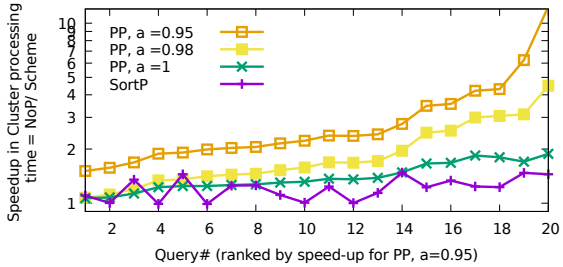


Figure 10: Evaluating TRAF20 query set on 100 GB online data. The figure shows the speed-up in cluster processing time relative to *NoP*, i.e., the total resources used to answer a query by *NoP* divided by that used by each scheme.

	System	33 GB	67 GB	100 GB
Query latency	NoP [36]	0.37	0.69	**1**
	PP (a=0.95)	0.22	0.39	0.61

Table 8: Normalized average query latency (including PP training/inference overhead) on the TRAF-20 with different input sizes.

ID	PP cons.	#PPs	PP inf.	Sub.UDF	Selectivity	Reduction
4	27s	1	2ms	23ms	0.67	11%
8	68s	2	5ms	55ms	0.41	20%
20	155s	4	12ms	85ms	0.01	60%
Avg.	79s	2.5	6ms	52ms	0.20	59%

Table 9: Training and inference overhead for deploying PPs in online machine learning query processing. PP cons. is the PP construction time (normalized to single thread) on 15K rows. PP inf. is the PP inference time per row. Sub.UDF is the subsequent UDF cost per row. Selectivity, s_p, is the fraction of rows picked by query predicate. Reduction is $\frac{NoP-PP}{NoP}$ where each term is the cluster processing time to execute the query with the corresponding scheme. Avg. is average over all TRAF-20 queries.

that our query processing system obtains large speed-ups in cluster processing time as well as query latency; especially when accuracy targets are relaxed. With an accuracy target of 1.0 (i.e., no false negatives), queries receive an average speed-up of 1.4×. For a relaxed accuracy target of 0.95, resource usage improvement ranges from 1.52× to 12.5× depending on the predicate and its selectivity, and the average query in TRAF-20 speeds up by 3.2×. Furthermore the average query latency is 60% of the latency in *NoP*. These improvements hold agnostic to data volumes; i.e., larger input sizes receive larger reductions in latency as expected.

Details. Table 9 reports additional details on the costs of training and applying PPs on some typical queries in TRAF-20 as well as the average over all queries. We report here the time to train a PP on one thread. We note in practice that multiple threads can be used, model selection is done over sampled subsets and PPs trained once are reused for other queries, all of which reduce the amortized training latency per query. We see that PP training finishes in minutes. The overhead of applying PPs is generally small, compared with the subsequent machine learning UDFs. Our QO takes 80 to 100ms to translate the query predicates into PP expressions and to parametrize these expressions. Finally, the table also shows the selectivity of each query predicate and the achieved data reduction in cluster processing time (at $a = 0.95$). On average, we achieve a 59% reduction in cluster processing time which is 74% of the theoretical maximum reduction of 80% (because the average query selectivity is 0.20). This is a sizable and promising speed-up for practical machine learning tasks. All of the algorithmic modules in our system are implemented in C/C++.

PP Corpus	Query predicate	sel.	# plans	Est. r	Picked and Alter. plans. (Est. r)
32 PPs	$t \in \{SUV, van\}$	0.41	4	0.06–0.42	$PP_{SUV} \vee PP_{van}$ (0.42), $PP_{\neg sedan} \wedge PP_{\neg truck}$ (0.40), $PP_{\neg sedan}$ (0.23)
	$s > 60 \wedge s < 65$	0.05	18	0.02–0.79	$PP_{s>60} \wedge PP_{s<65}$ (0.79), $PP_{s>60 \wedge s<70}$ (0.75), $PP_{s>60}$ (0.55)
full coverage	$s > 60 \wedge s < 65 \wedge c = white \wedge t \in \{SUV, van\}$	0.01	216	0.08–0.77	$PP_{s>60} \wedge PP_{s<65} \wedge PP_{\neg sedan} \wedge PP_{\neg truck} \wedge PP_{white}$ (0.77) $PP_{s>50} \wedge PP_{s<70}$ (0.43), $PP_{s>60} \wedge PP_{s<65} \wedge PP_{\neg sedan}$ (0.52)
16 PPs half of above dropped at random%	$t \in \{SUV, van\}$	0.41	3	0.06–0.40	$PP_{\neg sedan} \wedge PP_{\neg truck}$ (0.40), $PP_{\neg sedan}$ (0.23)
	$s > 60 \wedge s < 65$	0.05	6	0.02–0.75	$PP_{s>60 \wedge s<70}$ (0.75), $PP_{s>60}$ (0.55)
	$s > 60 \wedge s < 65 \wedge c = white \wedge t \in \{SUV, van\}$	0.01	88	0.08–0.76	$PP_{s>60} \wedge PP_{s<70} \wedge PP_{\neg sedan} \wedge PP_{\neg truck} \wedge PP_{white}$ (0.76)

Table 10: For some example queries, understanding the nature of feasible PP expressions.

Query optimizer in action. To understand how the QO chooses PP combinations, we show more detail for a few queries in TRAF-20. Recall from the method description that there are five predicate columns and our QO uses a corpus of 32 PPs while the number of possible predicates is about 100^4. By construction this PP corpus completely covers the space of the predicates, i.e., any possible predicate will have at least one PP in the corpus that is a necessary condition. For example, the vehicle type column t can take four different values $SUV, van, truck, sedan$ and the corpus contains PPs for $t = SUV$, $t = van$, $t = truck$ and $t = sedan$. For numerical columns, we train PPs for $\leq$ and $\geq$ comparisons on value boundaries, e.g., PPs for speed are of the type $s \geq v_1 \in \{40, 50, 60\}$ or $s \leq v_2 \in \{65, 70\}$. For typical queries, Table 10 shows the query predicate, the number of available PP combinations that are feasible, the range of data reduction rates achievable by the feasible PPs, the combination of available PPs picked by the QO and the reduction rates for a few alternate plans. We see that for many queries, the QO has a meaningful choice to make, i.e., there are a lot of feasible PP combinations and picking one at random is unlikely to yield close to the best possible data reduction. The table also shows that the combination picked by the QO can have multiple PPs even when the predicate has only a single clause. Furthermore, the empirical observed reduction rates are close to the estimated reduction rate and so the QO choice is nearly optimal. A key point to emphasize is that because our QO prepares appropriate PP combinations, the training overhead is reduced from per-query (there are 100^4 possible predicates) to just 32 PPs, one per simple predicate clause. Table 10 also shows results for an even smaller PP corpus, wherein for each predicate column we have randomly dropped half of the PPs that are available on that column. We see that data reduction rates of the best possible PP combination decrease but not substantially. For example, for the predicate $t \in \{SUV, van\}$, data reduction rate drops from 0.42 to 0.40. While more investigation and empirical evidence is needed, our intuition is that a small corpus of PPs suffices to provide sizable data reductions even when the space of possible predicates is large (because a complex predicate will receive data reduction as long as some combination of PPs in the corpus is a necessary condition for the complex predicate).

9 RELATED WORK

We reviewed some relevant works in §3 and in §8. Advanced indexing techniques [18] and data cubes [20] leverage the predictable nature of decision support queries and answer them directly from more compact representation. However, these approaches do not work well for machine learning inference on live streams such as audio and video, where the queries are not known a priori or are more complex.

There is a rich literature on optimizing queries with predicates: pushing predicates closer to input [49], optimal ordering of conjunctions [22], normalizing disjunctive and other complex predicates [30, 34] etc. When predicates rely on columns generated by user-defined operators, [39] shows that performance-optimal ordering of the UDFs and predicates is NP-hard. Our approach differs from these works because it uniquely adds new probabilistic predicates (PPs) rather than optimally ordering the existing predicates in the query. Approximate predicates [44] are applied to pre-filter unlikely inputs for expensive user-defined predicates; however they use the same relation as the query predicates and are not for blobs. One recent work observes that if existing column(s) in the data are correlated with user-defined predicates, then a function over those column(s) can be used to bypass the user-defined predicate [27]. While such functions over correlated columns are (simple) PPs, in our experience, such correlated columns rarely exist for ML queries. Instead, we train PPs using SVMs or kernel densities instead. For queries that apply predictive models on relational data, [13] derives implied predicates based on the details of the predictive model. Our approach differs in two ways. First, PPs are trained without any knowledge of the inference modules that are used in a query and hence PPs are more broadly usable whereas [13] applies only to decision trees and naive bayes classifiers and has a custom algorithm for each type of predictive model. Second, PPs also apply on non-relational datasets. NoScope [29] is a domain-specific model cascade for video data; it uses background subtractors, frame sampling and a simple DNN in front of the reference CNN and reports several orders of magnitude improvement on video processing rate. While we show in Appendix B comparable results on video datasets with simpler PPs, our system differs from NoScope in supporting a wider range of queries (e.g., not just selections) and datasets (e.g., not only in the video domain). IDK cascade [52] is another model cascade to accelerate heavy classification models using cheaper ones. The key difference is that PPs are not functionally equivalent to the classifiers that they bypass and so efficient PPs are available for a broader class of queries and datasets.

10 CONCLUSIONS

We focus on accelerating machine learning inference queries where classic static or post-facto optimization techniques, such as building indices or predicate push-down, are not feasible. Our key idea is to use probabilistic predicates (PPs) which execute over the raw input, without needing the predicate columns, and can successfully mirror the original query predicates. While introducing only a configurable amount of error, we show that PPs boost the performance of machine learning queries by as much as 10× on various large-scale datasets. This work is a first step towards our goal of optimizing the execution of large-scale machine learning queries on big-data engines; many open problems remain.

REFERENCES

[1] Free video trigger app. http://bit.ly/2ufJSSs.
[2] In more cities, a camera on every corner, park and sidewalk. http://n.pr/2tKQEg3.
[3] Shun-ichi Amari and Si Wu. Improving support vector machine classifiers by modifying kernel functions. *Neural Networks*, 12(6):783–789, 1999.
[4] Barak Ariel, William Farrar, and Alex Sutherland. The effect of police body-worn cameras on use of force and citizens complaints against the police: A randomized controlled trial. *J. of quantitative criminology*, 31(3):509–535, 2015.
[5] Michael Armbrust et al. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.
[6] Josh Attenberg, Kilian Weinberger, Anirban Dasgupta, Alex Smola, and Martin Zinkevich. Collaborative email-spam filtering with the hashing trick. In *6th Conf. on Email and Anti-Spam*, 2009.
[7] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *ACM SIGMOD*, 2004.
[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Comm. of the ACM*, 18(9):509–517, 1975.
[9] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.
[10] Mark W Burris. Application of variable tolls on congested toll road. *Journal of transportation engineering*, 129(4):354–361, 2003.
[11] Ronnie Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[12] Craig Chambers et al. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.
[13] Surajit Chaudhuri, Vivek R. Narasayya, and Sunita Sarawagi. Efficient evaluation of queries with mining predicates. In *ICDE*, 2002.
[14] Robert T Collins et al. A system for video surveillance and monitoring. *VSAM final report*, pages 1–68, 2000.
[15] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *CVPR*, 2005.
[16] James Davidson et al. The youtube video recommendation system. In *ACM conference on Recommender systems*, 2010.
[17] Amol Deshpande, Carlos Guestrin, Sam Madden, and Wei Hong. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, 2005.
[18] Christos Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.
[19] Gene H Golub and Charles F Van Loan. *Matrix computations*. 2012.
[20] Jim Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, 1996.
[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
[22] Joseph M Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *ACM SIGMOD*, 1993.
[23] Nacim Ihaddadene and Chabane Djeraba. Real-time crowd motion analysis. In *ICPR*, 2008.
[24] Yu-Gang Jiang, Chong-Wah Ngo, and Jun Yang. Towards optimal bag-of-features for object categorization and semantic video retrieval. In *ACM Conf. on Image and video retrieval*, 2007.
[25] Thorsten Joachims. Training linear svms in linear time. In *SIGKDD*, 2006.
[26] Manas Joglekar, Hector Garcia-Molina, Aditya Parameswaran, and Christopher Re. Exploiting correlations for expensive predicate evaluation. *arXiv preprint arXiv:1411.3374*, 2014.
[27] Manas Joglekar, Hector Garcia-Molina, Aditya Parameswaran, and Christopher Re. Exploiting correlations for expensive predicate evaluation. In *SIGMOD*, 2015.
[28] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.
[29] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing neural network queries over video at scale. *VLDB*, 2017.
[30] A Kemper, G Moerkotte, K Peithner, and M Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *SIGMOD*, 1994.
[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.
[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.
[33] Yann LeCun et al. Handwritten digit recognition with a back-propagation network. In *NIPS*, 1990.
[34] Alon Levy, Inderpal Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *VLDB*, 1994.
[35] Tsung-Yi Lin et al. Microsoft COCO: Common objects in context. In *ECCV*, 2014.
[36] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *ACM SoCC*, 2016.
[37] Yao Lu, Wei Zhang, Ke Zhang, and Xiangyang Xue. Semantic context learning with large-scale weakly-labeled image set. In *CIKM*, 2012.
[38] Bruce D Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI*, 1981.
[39] Thomas Neumann, Sven Helmer, and Guido Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE*, 2005.
[40] Ioannis Partalas et al. LSHTC: A benchmark for large-scale text classification. *arXiv preprint arXiv:1503.08581*, 2015.
[41] Genevieve Patterson, Chen Xu, Hang Su, and James Hays. The sun attribute database: Beyond categories for deeper scene understanding. *IJCV*, 2014.
[42] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*. 2012.
[43] Murray Rosenblatt et al. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, 27(3):832–837, 1956.
[44] Narayanan Shivakumar, Hector Garcia-Molina, and Chandra Chekuri. Filtering with approximate predicates. In *VLDB*, 1998.
[45] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.
[46] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. UCF101: A dataset of 101 human actions classes from videos in the wild. *Preprint arXiv:1212.0402*, 2012.
[47] Abhinav Srivastava, Amlan Kundu, Shamik Sural, and Arun K Majumdar. Credit card fraud detection using hidden markov model. *IEEE Trans. on Dependable and Secure Computing*, 2008.
[48] Ashish Thusoo et al. Hive: A Warehousing Solution Over A Map-Reduce Framework. *Proc. VLDB Endow.*, 2009.
[49] Jeffrey Ullman. Principles of database and knowledge-base systems, 1989.
[50] Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical learning theory*, volume 1. Wiley New York, 1998.
[51] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001.
[52] Xin Wang et al. IDK Cascades: Fast Deep Learning by Learning not to Overthink. *Preprint arXiv:1706.00885*, 2017.
[53] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *ICML*, 2009.
[54] Longyin Wen et al. Detrac: A new benchmark and protocol for multi-object tracking. *Preprint arXiv:1511.04136*, 2015.
[55] Xiangyang Xue, Wei Zhang, Jie Zhang, Bin Wu, Jianping Fan, and Yao Lu. Correlative multi-label multi-instance image annotation. In *ICCV*, 2011.

A ADDITIONAL QO DETAILS

A.1 Hardness of the QO problem.

Optimal choice of PPs to train: Given a query set and a constraint on the overall training budget, consider the problem of choosing which PPs to train so as to obtain the best possible speed-up over that query set. Let $TrainCost_{\mathsf{PP}_p}$ be the cost to train PP_p. Observe that the PP for predicate p will help any query q for which p is a necessary condition. Let $Queries_{\mathsf{PP}_p}$ be the set of queries that will benefit if PP_p were to be trained. For each query q in this set, let $r_p[a]^q$ denote the data reduction rate achieved from using PP_p on query q when ensuring accuracy is above a. We also know that a query can use more than one PPs. So, given a set of available PPs, $\mathcal{P}$, let $r_{\mathcal{P}}[a]^q$ be the best data reduction achieved by q through some combination of PPs in $\mathcal{P}$. Finally, let $\mathcal{Q}$ be the set of given queries, $\mathcal{S}$ be the set of all predicates in $\mathcal{Q}$ as well as all necessary conditions of those predicates and let T be the training budget. This problem becomes:

$$\max_{\mathcal{P}\subseteq\mathcal{S}} \left(\sum_{q\in\mathcal{Q}} r_{\mathcal{P}}[a]^q \right) s.t. \sum_{p\in\mathcal{P}} TrainCost_{\mathsf{PP}_p} \leq T. \tag{11}$$

We show that this problem is NP-hard by reducing set cover to a simple version of the above problem.

Proof: Recall that given a set of elements $\{1, 2, \ldots, n\}$ (called the universe) and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe. The reduction proceeds by creating

a query for each element in the universe and a predicate corresponding to each set in S with the understanding that training a PP for this predicate will help all the queries whose elements belong to that set. Hence, set the cost of training every PP to be the same and set the reduction rates to be unit; that is a query will receive the maximum benefit if it is covered by at least one PP. Note that the maximum achievable benefit to these queries will be obtained only when the union of the chosen sub-collection of sets equals the universe. To find the smallest possible sub-collection of S, we can vary the training budget from 1 to $|S| = m$ and find the smallest training budget at which the total benefit equals n. □

Optimal use of available PPs: Given a set of available PPs, $\mathcal{P}$, consider the problem of finding a combination of PPs that offer the best data reduction for a query q given accuracy target a. Let c_p be the cost and $r_p[a]$ be the data reduction rate for a PP $p \in \mathcal{P}$. We can show that this problem is NP-hard by reducing the knapsack problem to a very simple version of the above problem.

Proof: For the purposes of this reduction, suppose that only conjunctions of the available PPs are allowed. Furthermore, the above problem has two parts: how to apportion the accuracy budget among the available PPs and how to order the chosen PPs. Let us ignore the second portion (ordering PPs), and the reduction then proceeds as follows. Associate for each item a corresponding PP whose reduction rate is equal to the value of the item if the accuracy budget to this PP is at most log of the weight of the item and is zero otherwise. That is, the PP will offer reduction rate (value) only if given at least as much accuracy budget (weight). Set the log of the limit as the accuracy budget; sum of logs is product of individual accuracy budgets as per conjunction PP formula (Equation 9). □

A.2 Wrangling rules for complex predicates

Here we discuss how to wrangle predicate clauses so that they can be exactly matched onto PPs.

- Not-equals check ($f(C) \neq v$): If the range of $f(C)$ is finite and discrete, then $f(C) \neq v \Rightarrow \bigvee_{t \in Range(f(C)) \setminus v} f(C) = t$. For example, if vehicle $\texttt{type} \in \{SUV, truck, car\}$, then $\texttt{type} \neq SUV \Rightarrow \texttt{type} = truck \vee \texttt{type} = car$. This wrangling is useful if PPs exist only for the clauses on the left.
- Comparison: $f(C) > v \Rightarrow f(C) > t, \forall t \leq v$. The expression on the right relaxes the comparison and may be useful if a PP has been trained for some value t. Another rewrite is possible when $f(C)$ is finite and discrete, $f(C) > v \Rightarrow \bigvee_{t \in Range(f(C)),\, t<v} f(C) = t$. Similar rewrites exist for $>, \leq, \geq$.
- Range-check ($v_1 \leq f(C) \leq v_2$) is a special case of comparison which is bounded on both sides and can be wrangled as above.
- *No-predicate*. If some columnset C in the query output has a finite and discrete range, even a query with no predicate can benefit from PPs because $\mathbf{1} \Leftrightarrow \bigvee_{t \in Range(C)} C = t$. For the above example of vehicle type, $\mathbf{1} \Leftrightarrow \texttt{type} = car \vee \texttt{type} = truck \vee \texttt{type} = SUV$.

A.3 Negation rewrites and other details

Recall the fourth predicate rewrite rule mentioned in §6.1:

Rule 4 : $p \wedge (\mathcal{P} \setminus p) \Rightarrow \neg \texttt{PP}_{\neg p}$.

Figure 11 shows how such a PP can be used. This rule is quite powerful because predicates that have high selectivity will not yield useful PPs but their negations can achieve substantial data reductions. However, the rule has somewhat narrower applicability. As shown in Figure 11, blobs that fail the negative PP are output immediately; this requires that the schema of the query output match the schema of the query input; i.e., that the query be simply selecting a subset of blobs. Further, the rule composes in a complex way with the other rules because its application can lead to *false positives*.

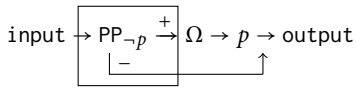


Figure 11: Injected query plan for the pattern $p \Rightarrow \neg\textbf{PP}_{\neg p}$

A.4 PP Seeding and pushdown rules

Table 11 describes our PP seeding and pushdown rules. We use a placeholder to seed a possible PP, denoted X_p, and attempt to push the placeholder down using these rules until it executes directly on the raw input; note that only predicates on a raw input can possibly be replaced with some combination of PPs. If not possible, the placeholder is simply omitted by the QO from the final plan. In the first rule, the expression on the right is less accurate, i.e., it has a given amount of false positives and false negatives. For each subsequent rule, the expressions have equivalent accuracy but the one on the right can be more performant. Some rules hold only under certain conditions. Pushdown below selection requires that the predicates p and q are independent. For the foreign-key join rule, let R and S be rowsets being equijoined on columnset $\mathcal{D}$ which is a primary key for S and a foreign key for R. This rule holds if the selection performed implicitly by the foreign-key join (recall: each row from R contributes at most one row to the join output) is independent of the predicate p. Finally, the pushdown rules for project change the columns in the predicate to invert the effect of the projection.

Seed PP for select	$\sigma_p(R) \tilde{\Leftrightarrow} \sigma_p(X_p(R))$
PP over select	$X_p(\sigma_q(R)) \overset{*}{\Leftrightarrow} \sigma_q(X_p(R))$ (additional conditions needed)
PP over foreign-key joins	$X_p(R \bowtie_{\mathcal{D}} S) \overset{*}{\Leftrightarrow} X_p(R) \bowtie_{\mathcal{D}} S$ if $p_c \subseteq R_c$ (additional conditions needed)
PP over col renaming project	$X_p(\pi_{\mathcal{C}_a \to \mathcal{C}_b}(R)) \overset{*}{\Leftrightarrow} \pi_{\mathcal{C}_a \to \mathcal{C}_b}(X_{p_{\mathcal{C}_a \to \mathcal{C}_b}}(R))$
PP over project creating new columns	$X_p(\pi_{f(\mathcal{D})=d}(R)) \overset{*}{\Leftrightarrow} \pi_{f(\mathcal{D})=d}(X_{p_{d \to f(\mathcal{D})}}(R))$

Table 11: Pushdown rules for probabilistic predicates. See §A.4.

A.5 PPs on dependent predicates

In our experiments we have observed reasonable performance for queries with multiple PPs. However, if the PPs upon multiple predicate columns are dependent, the cost and reduction rate estimation and therefore the PP planning will be suboptimal. In such case, we apply a runtime fix. If we observe that the PP cost and reduction rate at runtime differ dramatically from their estimations, we flag such

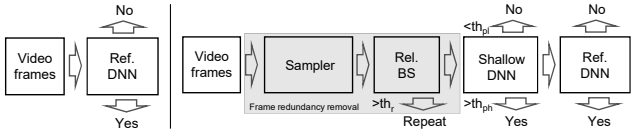


Figure 12: **Left: Original pipeline for video object detection; the reference DNN is applied on every video frame. Right: Pipeline for NoScope. Rel. BS: relative background subtraction. Ref. DNN: reference DNN.**

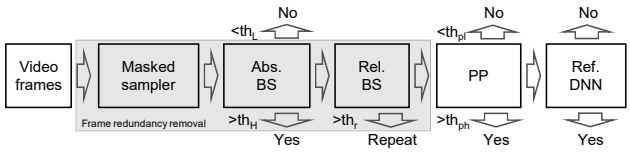


Figure 13: **Pipeline for video object detection with PPs. Abs. BS: absolute background subtraction.**

predicates as possibly dependent so that the QO will only use one PP (and not a combination of dependent PPs) in the future for that predicate. We also note that because practical accuracy targets are very close to 1, the independence assumption can be replaced with an upper bound that is fairly tight.

B SUPPLEMENTARY EXPERIMENTS

Comparison with NoScope. NoScope [29] is a system that retrieves video frames containing certain objects such as vehicles or persons. NoScope uses video-specific redundancy detection and light-weight DNNs so that the expensive DNN object detector needs to process only a few of the video frames. Figure 12 shows the original pipeline for video object detection on the left and that used by NoScope on the right. We constructed a similar pipeline inspired by PPs as shown in Figure 13. The salient differences in our pipeline are three-folds. (1) We apply a mask to eliminate unimportant video frame regions, whereas NoScope has a full scope. (2) We improve the background subtraction to a two-stage scheme. (3) To filter frames early, we use simple SVMs instead of the DNNs used by NoScope.

In more detail, our pipeline consists of: (1) *Masked sampler.* As in NoScope, we sample the video frames with different sampling rates such as 1-in-15 frames or 1-in-30 frames. We also use a mask to restrict the area-of-interest, i.e., to remove areas in the frame that have low information. An example is the area in blue in Figure 14 which will not contain any target object; such masks are available for most fixed surveillance cameras. (2) *Absolute Background Subtraction.* We use a two-stage background subtraction (BS). The first absolute BS eliminates any object against an empty footage (NoScope also uses this). (3) *Relative Background Subtraction.* A similar BS module is used to compare the current frame with the previous frame; this module detects motion. Again we compute the difference area, and if the area is below a threshold, we use the previous frame detection result. (4) *PP.* We apply an SVM PP on the raw input. Unlike the PPs used elsewhere in this paper, we configure two thresholds to not only reject blobs that are unlikely to match the predicate but also to accept blobs that are likely to match; this mimics a similar aspect of NoScope. We perform the experiments on the `coral` video clip provided by the NoScope authors. The video is 12 hours long. We

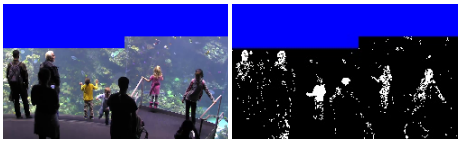


Figure 14: **Left: we apply a mask, shown in blue, on the surveillance video to restrict and accelerate the detection. Right: relative background subtraction result; white regions are with motion.**

System	Video	Data reduction		Pipeline	Accuracy
		Pre-Proc.	Early drop	Speed-up	
NoScope [29]	coral	0.998	~0.90	3500x	0.998
	coral	0.998	~0.95	5000x	0.98
PP	coral	0.993	0.93	3000x	0.997
	coral	0.9997	0.90	8200x	0.98
	square	0.967	0.76	1300x	0.912

Table 12: **Comparison with NoScope on the `coral` video clip. We show the data reduction rate during pre-processing (Pre-Proc.) as well as that due to using lightweight pre-computation. Square is another video that was provided by the NoScope authors but we did not find corresponding published results in their paper.**

Dataset	PP	ts=30%	ts=40%	ts=50%
SUNattribute	PCA+KDE	.31/.92/6s	.32/.95/7s	.35/.96/8s
UCF101	PCA+KDE	.46/.92/10s	.51/.97/12s	.54/.98/14s
UCF101	RAW+SVM	.26/.87/1s	.39/.94/1s	.43/.96/2s
LSHTC	FH+SVM	.40/.95/1s	.45/.97/1s	.48/.98/1s
COCO	DNN*	-	-	.81/.99/110s

Table 13: **For different PP methods on different datasets, with different training set sizes (ts=30%–50%) and an accuracy target of 0.99, the values shown in each table entry are the average data reduction rate/ the achieved accuracy/ and the training time per 1000 inputs. * denotes experiments using a GPU.**

train our SVM on the initial 10K frames. All of the components in our pipeline are implemented in C/C++ and OpenCV.

As shown in Table 12, our pipeline achieves comparable if not better performance. Notably, more than 99.3% of the frames are filtered in the pre-processing stage itself. SVM filters are easier to train and execute and do not require a GPU. Among all the 1.2M video frames, in our pipeline, only hundreds of frames are processed by the reference DNN object detector. NoScope, on the other hand, requires GPUs to execute its lightweight DNNs and requires considerable per-query training overhead to build these DNN filters. We also note a few more points. (1) DNN-based early filters do not appear necessary for the surveillance videos used by NoScope. Although, as we saw with the case of ImageNet, they are needed in other cases. (2) Such pipelines are only amenable for selection queries since the early filters both accept and reject frames. PPs on the other hand only reject frames that will not contribute to the actual answer. (3) Such pipelines require per-query training whereas in this paper we extend to ad-hoc queries by constructing PPs for simple predicates and using the QO to construct appropriate combinations of available PPs for a given query.

How much training data is needed to construct PPs? Table 13 demonstrates the empirical data reduction rate and accuracy on the test sets with different training sizes; PCA, if used, is based on the same 1K rows. We note that more training data usually leads to better PP classifiers in terms of reduction rate and accuracy. The training

Figure 15: Demonstration of different PP outputs on COCO. The figure shows confidences f for 4 different PPs. See text for explanation.

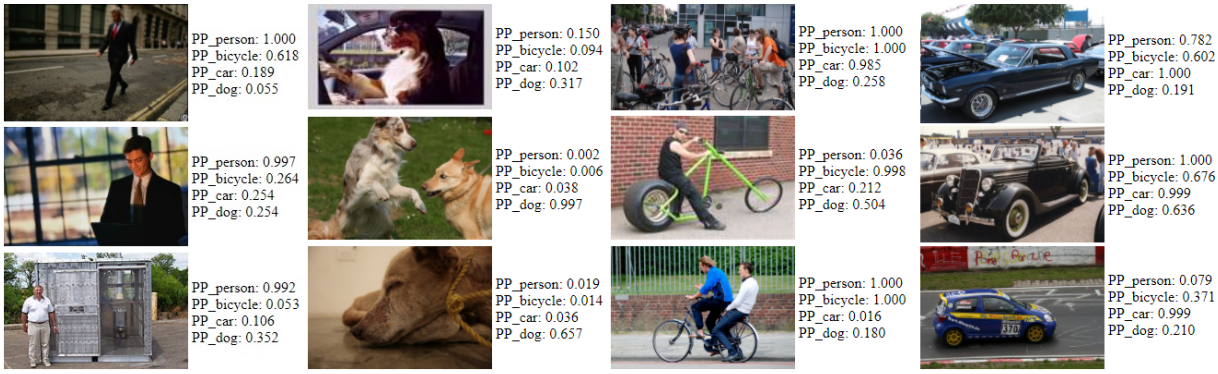


Figure 16: Demonstration of different PP outputs. The PPs are trained on COCO and applied on ImageNet. The figure shows confidences f for 4 different PPs.

cost grows sub-linearly with the training set size primarily because PCA (for dimension reduction) has a considerable fixed cost. However, since the PCA basis is specific to a dataset, it can be reused across PPs (we have not accounted for this above). We use feature hashing for the document analysis dataset (which is sparse); FH is extremely efficient, and combined with the linear SVM produces useful PPs. On the other hand, although with impressive data reduction rates, training cost for DNNs is relatively enormous.

Demonstrating PPs. Figure 15 visually demonstrates how PPs work. We show for several example images, the confidence value computed for four different PPs: 'has person', 'has bicycle', 'has car' and 'has dog'. Recall that a PP would drop blobs (images in this case) whose confidence is below a threshold that is chosen based on desired accuracy; the more blobs that can be dropped the larger the data reduction.

It is easy to see from these images that the gap between the confidence for appropriate labels and inappropriate labels is large. PP trained for 'has person' with a confidence threshold of 0.9 will achieve a data reduction of 58% and an accuracy of 100%; this is the best possible data reduction because 5-out-of-12 pictures have a person in them. PP_{has_dog} with a 0.7 confidence threshold will achieve a data reduction of 83% and accuracy of 100%. These PPs use as input the raw pixels from images. Finally, Figure 16 shows details for PPs trained on COCO being applied on ImageNet.