Interactive Demonstration of Probabilistic Predicates

Yao Lu\textsuperscript{1,2}, Srikanth Kandula\textsuperscript{2}, Surajit Chaudhuri\textsuperscript{2}
\textsuperscript{1}UW, \textsuperscript{2}Microsoft

ABSTRACT

We will demonstrate a prototype query processing engine that uses probabilistic predicates (PPs) to speed up machine learning inference jobs. In current analytic engines, machine learning functions are modeled as user-defined functions (UDFs) which are both time and resource intensive. These UDFs prevent predicate pushdown; predicates that use the outputs of these UDFs cannot be pushed to before the UDFs. Hence, considerable time and resources are wasted in applying the UDFs on inputs that will be rejected by the subsequent predicate. We use PPs that are lightweight classifiers applied directly on the raw input and filter data blobs that disagree with the query predicate. By reducing the input to be processed by the UDFs, PPs substantially improve query processing.

We will show that PPs are broadly applicable by constructing PPs for many inference tasks including image recognition, document classification and video analyses. We will also demonstrate query optimization methods that extend PPs to complex query predicates and support different accuracy requirements.

ACM Reference Format:

1 INTRODUCTION

Relational data engines are increasingly being used to analyze data blobs such as unstructured text, images and videos [4, 5, 12, 17]. Such queries begin by applying user-defined functions to extract relational columns from blobs. Consider the following example which finds red or blue SUVs from city-wide surveillance video:

\begin{verbatim}
SELECT cameraID, frameID, 
C1(F1(vehicleBox)) AS vehType, 
C2(F2(vehicleBox)) AS vehColor
FROM (PROCESS inputVideo

PRODUCE cameraID, frameNum, vehicleBox
USING VehDetector)

WHERE vehType = 'SUV' \land (vehColor = "red" \lor "blue"");
\end{verbatim}

Here, VehDetector extracts one or more bounding boxes that contain a vehicle from each video frame, \(F_1\) and \(F_2\) extract relevant features from each bounding box and finally \(C_1\) and \(C_2\) are classifiers that identify the vehicle type and color using these features. The query plan is shown in Figure 1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD‘18 Demo, June 10–15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06... $15.00
https://doi.org/10.1145/3183713.3183751

How to execute such machine learning queries efficiently? Clearly, traditional query optimization techniques such as predicate pushdown are not useful here because they will not push predicates below the user-defined functions (UDFs), which generate the columns used by the predicate. In the above example, vehType and vehColor are available only after VehDetector, the feature extractors and classifiers have been executed. Although the predicate has limited selectivity (perhaps 1-in-100 images have red or blue SUVs), every video frame has to be processed by all of the UDFs.

\begin{verbatim}
Input \rightarrow VehDetector \rightarrow F_1, F_2 \rightarrow C_1, C_2 \rightarrow \sigma_{SUV} \land (\sigma_{red} \lor \sigma_{blue}) \rightarrow Result
\end{verbatim}

Figure 1: Query plans to retrieve red or blue SUVs from traffic surveillance videos. Materializing the vehType and vehColor columns (underlined) accounts for almost all of the query cost, leaving little room for traditional optimization.

\begin{verbatim}
Input \rightarrow PP_{SUV}, PP_{red}, PP_{blue} \rightarrow VehDetector \rightarrow F_1, F_2 \rightarrow C_1, C_2 \rightarrow \sigma_{SUV} \land (\sigma_{red} \lor \sigma_{blue}) \rightarrow Result
\end{verbatim}

Figure 2: We construct and apply probabilistic predicates (PPs, dash-underlined) to filter data blobs that do not verify the predicates.

In our prior work [13], we proposed the notion of probabilistic predicates (PPs), which are binary classifiers that apply on the unstructured input and reduce the query cost by avoiding processing data blobs that will not pass the query predicate. Figure 2 shows a query plan with PPs; if the query predicate has a small selectivity and the PP is able to discard half of the frames that do not have the target objects, the query may speed up by 2x. This demonstration will showcase our query processing engine to accelerate machine learning inference jobs by using probabilistic predicates. Our demo will use a variety of queries on images, documents and videos to show that PPs are applicable for a broad set of machine learning inference tasks. Users will be able to interact with our system in various ways including submitting new queries and comparing performance with and without PPs. Performance boosts of as much as 10x can be observed in this interaction.

Since query predicates can be diverse, trivially constructing a PP for each query predicate is unlikely to scale. Hence, we only constructs PPs for simple predicates and assembles, at query compilation time, an appropriate combination of available PPs that has the lowest cost, is within the accuracy target and is a necessary condition for (i.e., semantically implies) the original query predicate. We will demonstrate this functionality which can be embedded into a standard cost-based query optimizer; users will be able to see the various available plan choices, pick accuracy targets and examine job results.

2 THE DEMO SYSTEM

Demo system. The prototype used in our prior work [13] ran on a production cluster and extended a proprietary query processing
for experiments that construct PPs online. Unlike the actual UDF classifiers which take complex features as inputs (some example features are shown in Table 1), we use PPs that apply over raw input blobs, i.e., they take as input pixels from images and videos and bag-of-words representations for documents. The modified platform, as shown in Figure 3(c), takes two additional inputs compared to baseline systems: (1) the list of available probabilistic predicates and (2) a desired accuracy threshold for the query. The modified query optimizer injects an appropriate combination of PPs for the query based on the accuracy threshold; the PPs, shown in the figure as green dotted circles, execute directly on raw inputs and the remaining query plan is semantically equivalent to the baseline query plan.

**Key technical challenges:** The prototype system has been built to answer the following technical questions.

- **Filtering rate and efficiency:** PPs have to apply on the raw input which can be highly dimensional and arbitrarily distributed. If PPs are not efficient and/or do not lead to sizable data reduction, then query performance can worsen instead of improving. Hence, we use a variety of classifiers to construct PPs including SVMs, kernel density functions and deep NNs; different techniques are appropriate for different queries and input types.
- **Query precision and recall:** Whereas conventional predicate pushdown produces deterministic results, how the classifiers used as probabilistic predicates will function on previously unseen inputs is unknown. Hence, filtering with PPs is parametrized over a precision-recall curve; different filtering rates (and hence speed-ups) are achievable based on the desired accuracy.
- **Handling complex predicates:** Since query predicates can be diverse, trivially constructing a PP for each query predicate is unlikely to scale. To generalize, we construct PPs for only simple clauses and extend the query optimizer to assemble, at query compilation time, an appropriate combination of PPs that has the lowest cost, is within the accuracy target at query compilation time, an appropriate combination of PPs for the query based on the accuracy threshold; the PPs, shown in the figure as green dotted circles, execute directly on raw inputs and the remaining query plan is semantically equivalent to the baseline query plan.

**Scope and limitations.** We build probabilistic predicates for simple clauses of the form \( f(g_i(b), \ldots) \phi \), where \( f \) and \( g_i \) are functions, \( b \) is an input blob, \( \phi \) is an operator that can be \( =, \neq, <, \leq, >, \geq \) and \( \epsilon \) is a constant. We builds PPs using diverse techniques and uses only PPs that are useful, i.e., high data-reduction, accuracy and throughput. With these PPs, the query optimizer in our system

---

**Table 1:** A partial list of ML modules provided in our system.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Extraction - RGB Histogram</td>
<td>Extract RGB histogram feature.</td>
</tr>
<tr>
<td>Feature Extraction - HOG</td>
<td>Extract Histogram of Gradient feature.</td>
</tr>
<tr>
<td>Feature Extraction - Raw Pixels</td>
<td>Extract raw pixel feature.</td>
</tr>
<tr>
<td>Feature Extraction - PyramidHSVHist</td>
<td>Extract Pyramid HSV histogram feature.</td>
</tr>
<tr>
<td>Classifier/regressor - Linear SVM</td>
<td>Apply linear SVM on feature vector.</td>
</tr>
<tr>
<td>Classifier/regressor - Random Forest</td>
<td>Apply Random forest on feature vector.</td>
</tr>
<tr>
<td>Keypoint Extraction - SIFT</td>
<td>Extract SIFT keypoints in given image region.</td>
</tr>
<tr>
<td>Tracker - KLT</td>
<td>Tracking keypoints using KLT tracker.</td>
</tr>
<tr>
<td>Tracker - CamShift</td>
<td>Tracking objects using CamShift tracker.</td>
</tr>
<tr>
<td>Segmentation - MOG</td>
<td>Mixture of Gaussian background subtraction.</td>
</tr>
</tbody>
</table>

---

**Figure 3:** Comparing the unmodified query processing system on the top with the proposed system on the bottom. Key changes are in the training and use of probabilistic predicates (PPs). See [13] for details.

**Figure 4:** An example query in our system.

```python
def query(input, output)
    #define the ML pipeline; second parameter is schema
    let veh = input.apply_UDF(VehDetector, "frameID, boxID, int, vehBoxCoord");
    let text = veh.apply_UDF(1, "frameID, boxID, int, feature array");
    let callout = veh.apply_UDF(2, "frameID, boxID, int, feature array");
    let vehType = callout.apply_UDF(1, "frameID, boxID, int, vehTypeID string");
    let vehColor = callout.apply_UDF(2, "frameID, boxID, int, vehColor rating");
    output = vehType.join(vehColor, "frameID", boxID) / join two streams
    filter("vehTypeID:SV", "vehColor:blue") / apply a three-clause predicate
    .distinct("frameID") / pick frames satisfying the filter

SystemP.execute(videos, stdout, //define system input/output
                 query1, //ML query handle
                 use_PPs = True, //switch on/off PPs
                 target_accuracy = 0.99 //specify target filtering accuracy
```
The goals of our demonstration are listed below:

- **Broad applicability of PPs.** By using queries over documents, images and videos and by training PPs using a variety of techniques, we make a case that probabilistic predicates are broadly useful.
- **Query optimization with PPs.** Given complex query predicates, the query optimizer in our system picks which PPs to apply and determines their parameters for different precision/recall settings. We will show modified query plans for different target accuracy thresholds.
- **End-to-end system demonstration of query processing with PPs.** Our system offers the users an interface to submit different machine learning queries (such as the example in Figure 4) and will show the behavior with and without the use of PPs.

To demonstrate broad applicability of PPs, we consider different inference queries on different inputs. Some of the input blobs are highly dimensional (e.g., high-res video clips) whereas others are sparsely distributed (e.g., Wikipedia documents in bag-of-words format). Furthermore, several of the considered inference queries use non-linear classifiers (e.g., object recognition) as well as neural networks (e.g., image labeling). In more detail:

- **Case1: Image labeling.** The COCO dataset [11] has 120K images, each labeled with one or more of 80 object classes. Queries in this scenario retrieve images that contain objects satisfying the predicate which is a conjunction, disjunction, negation of one or more clauses over class labels such as ‘has person’ ∨ ‘has dog’ etc. We also generate similar queries over images and classes from the ImageNet [1] dataset. Figure 6 shows some example images from COCO.

- **Case2: Video activity recognition.** We use a popular video activity recognition dataset, UCF101 [16], which has 13K video clips ranging from ten seconds to several tens of seconds. 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; some examples are in Figure 7.
are correlated with a user-defined predicate, then some function over those column(s) can be constructed and used to bypass the user-defined predicate [7]. While such functions over correlated columns are (simple) PPs, in our experience, correlation between input columns and predicates is harder to capture when inputs are images, videos or documents. Hence, we train PPs using SVMs, kernel densities, or shallow CNNs. Another prior work, NoScope [8], is a domain-specific model cascade customized for selection queries on videos. NoScope inserts a query-specific shallow DNN before a complex DNN and accepts/rejects frames early using the shallow DNN. We demonstrate gains across a wider range of datasets including images and documents, across a wider range of inference queries (accepting frames early only works for selection queries) and does not require per-query training (i.e., uses a small corpus of PPs for a large set of query predicates).

5 CONCLUSION

We focus on speeding-up 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 $10\times$ on various large-scale datasets. This work is only a first step towards the larger goal of optimizing the execution of large-scale machine learning queries on relational big-data engines; open problems remain especially in dealing with correlated predicate clauses and handling queries with complex relational operations.

REFERENCES