Edge-based Discovery of Training Data for Machine Learning

Ziqiang Feng, Shilpa George, Jan Harkes, Padmanabhan Pillai†, Roberta Klatzky, Mahadev Satyanarayanan
Carnegie Mellon University and †Intel Labs
{zf, shilpag, jaharkes, satya}@cs.cmu.edu, padmanabhan.s.pillai@intel.com, klatzky@cmu.edu

***Abstract*—We show how edge-based *early discard* of data can greatly improve the productivity of a human expert in assembling a large training set for machine learning. This task may span multiple data sources that are live (e.g., video cameras) or archival (data sets dispersed over the Internet). The critical resource here is the attention of the expert. We describe *Eureka,* an interactive system that leverages edge computing to greatly improve the productivity of experts in this task. Our experimental results show that Eureka reduces the labeling effort needed to construct a training set by two orders of magnitude relative to a brute-force approach.**

I. Introduction

Deep neural networks (DNNs) have transformed computer vision. By supervising the training of DNNs, a domain expert with no programming skills or computer vision background can create highly accurate classifiers in domains such as medical research, ecology, and military intelligence. The discriminative power of these classifiers can be impressive, embodying questions that are hard for a human non-expert to answer. Examples include: "Is this skin lesion melanoma or basal cell carcinoma?"; "Does this pathology image show pagetoid spread or nuclear atypia?"; "Is this a caterpillar of the pest moth *Cactoblastis cactorum* or of a benign moth?"; "Is this seismic image indicative of an underground nuclear explosion or a minor earthquake?"

Through the use of Web-based training tools, a domain expert can produce highly accurate image classifiers. Virtually all the effort is in assembling a large training data set, labeling the data, running a long-running batch job to perform training, and then evaluating the resulting classifier. The use of DNNs avoids the need to perform explicit *feature extraction,* which typically requires creation of custom code and hence requires programming skills.

The empowerment of domain experts by DNNs introduces a new problem: namely, *how to collect a large enough training data set and label it.* This is the problem addressed by this paper. DNNs typically need to be trained on thousands of labeled examples before their accuracy reaches an acceptable level. How many hours will it take for a domain expert to assemble such a large training data set for a rare phenomenon? How much junk will she need to wade through? An expert cannot delegate the task of generating training data to a less skilled person. Crowd-sourcing approaches such as those using Amazon Mechanical Turk (AMT) that have been successful in other contexts [1], [2] are not applicable here for two reasons. First, crowds are not experts. By definition, only the expert possesses the depth of domain-specific knowledge needed to reliably distinguish true positives from false positives, and avoid false negatives. Second, access to data sources may be restricted for reasons such as patient privacy, national security, or business policy.

In this paper, we show how edge computing can be used to create a human-in-the-loop system called *Eureka* for human-efficient discovery of training data from distributed data sources on the Internet. "Human-efficient" in this context refers to (a) leveraging human judgement and expertise; (b) avoiding long stalls awaiting results; and (c) avoiding overloading the expert with a flood of irrelevant results. Eureka views user attention as the most precious resource in the system. This resource is used well if most of it is consumed in examining results that prove to be true positives. It is used poorly if most of it is spent waiting for results, or dismissing frivolous false positives. To this end, Eureka uses an iterative workflow that starts from a handful of examples and evolves through increasingly sophisticated classifiers that lead to discovery of more examples.

This paper focuses on the setting of multiple data sources dispersed across the Internet. The data source may be live (e.g., recently-captured video streams from a swarm of drones, recent sensor data in automated manufacturing, etc.) or archival (e.g., video from previous drone flights, multi-year patient records from a large hospital system, etc.). In this setting, the terms "edge" and "cloudlet" correspond to their familiar interpretations. However, the concepts underlying Eureka are more broadly applicable. In all settings, the edge is a location that can read from a data source at very high bandwidth and low latency. A cloudlet is the compute infrastructure at this location. Archival data may often be concentrated in a single cloud data center. In that setting, the data set can be statically or dynamically partitioned into subsets that are each processed by a different machine in the data center. Those machines play the role of cloudlets, and their locations can be viewed as the edge. Eureka concepts can also be extended to hardware implementations of storage that embed processing capability. Various forms of "active disks" and "intelligent storage" have been proposed over the years, as discussed in Section VII. In that setting, the embedded processing capability within storage is effectively the "cloudlet", and its location is the "edge".

In all settings, high bandwidth and low latency access to

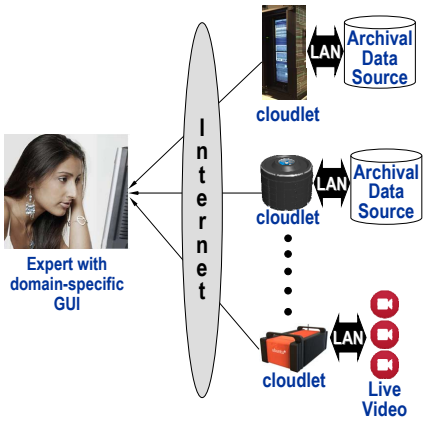


Figure 1. Eureka System Architecture

An infectious disease expert has just learned that a shy rodent, long considered benign, may be the transmitter of a new disease. There is a need to create an accurate image classifier for this rodent so that it can be used in public health efforts to detect and eradicate the pest. The expert has only a few images of the rodent, but needs thousands to build an accurate DNN. There are likely to be some untagged occurrences of this rodent in the background of images that were captured for some other purpose in the epidemiological image collections of many countries. A Eureka search can reveal those occurrences. In the worst case, it may be necessary to deploy cloudlets with large arrays of associated cameras in the field to serve as live data sources for Eureka.

Figure 2. Example: Infectious Disease Control

data is used by domain-specific code executing on a cloudlet to perform *early discard*. In other words, the rejection of clearly irrelevant data happens as early as possible in the processing pipeline that stretches from the data source to the user. With this architecture, the bandwidth demand between cloudlet and user is typically many orders of magnitude smaller than the bandwidth demand between data source and cloudlet. In addition to early discard, Eureka also harnesses high levels of parallelism across multiple cloudlets. This speeds up discovery of training data for rare phenomena.

The focus on early discard makes it natural to associate Eureka with edge computing rather than cloud-based settings. This is because of two reasons. First, ingress bandwidth savings from edge-located sensors is a major part of the rationale for edge computing. That is precisely what early discard achieves. In cloud-based settings, the need for bandwidth savings within a data center is less compelling because bandwidth is plentiful. The user could simply run her GUI on a virtual desktop that is located within the data center, thus completely avoiding large-volume data transfers out of the cloud. Second, Eureka is able to use both live and archival data in an edge computing setting. The aligns well with another aspect of edge computing, which is the ability to create real-time services on edge-sourced data — early discard is the "service" in this case.

In summary, Eureka can be viewed as an architecture that trades off computing resources (e.g., processing cycles, network bandwidth, and storage bandwidth) for user attention. Edge computing is the key to making this tradeoff effective. We describe the iterative workflow of Eureka in Section II, present its design and implementation in Section III, derive an analytical model of its workflow in Section IV, and report on experiments in Sections V and VI. We describe related work in Section VII, and close in Section VIII.

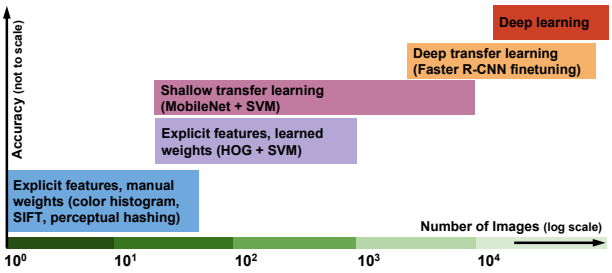
II. Discovery Workflow

Eureka views data sources as unstructured collections of *items*. Early in our implementation, an item referred to a single image in JPEG, PNG, or other well-known format. We have since extended Eureka to treat a short segment of a video stream as an item (Section III). We plan to further extend Eureka to other data types such as whole-slide images in digital pathology [3], map data from OpenStreetMap, and other forms of domain-specific multi-dimensional data. In each case, what constitutes an item will be specific to its data type. For simplicity, we focus on images in this section.

Structured information, such as a SQL database, may sometimes be available for a data source. This can be applied in a pre-processing step to shrink the size of the data source. For example, when searching medical images, patient record details such as age, gender, and prior medical history may be used to reduce the number of images that need to examined with Eureka. In this paper, we will assume that any possible pre-processing to shrink data sources has already been done prior to the start of the Eureka workflow.

Figure 1 illustrates the system architecture for the Eureka workflow. A domain-specific front-end GUI runs on a client machine close to the expert. An early-discard back-end runs at each cloudlet, LAN-connected to its data source. The back-ends execute in parallel, and transmit thumbnails of undiscarded images to the front-end. Each thumbnail includes back pointers to its origin cloudlet and full-fidelity image on that cloudlet. The expert sees a merged stream of thumbnails from all back-ends. If a thumbnail merits closer examination, a mouse click on it will open a separate window to display the full-fidelity image and its associated meta-data. Thumbnails are queued by the front-end, awaiting the expert's attention. If demand greatly exceeds available attention, queue back pressure throttles cloudlet processing.

As a working example, consider the scenario described in Figure 2. Starting from just a few example images, how can the expert bootstrap her way to the thousands to tens of thousands of images needed for deep learning? We start with the premise that the *base rate* [4] is low — i.e., that the rodent is rarely seen, and hence there are very few images in which it appears. If a good classifier already existed, discard-based search could be used in parallel on a very large number of cloudlets. The number of false positives would be low,



Each type of model requires a different number of examples before its accuracy starts to improve steadily. Their accuracies saturate at different levels when sufficient data is given.

Figure 3. Training Data Set Size vs. Accuracy

and the rate of true positives would be reasonably high. The expert would neither waste her time rejecting obvious false positives, nor would she waste time waiting for the next image to appear. Most of her time would be spent examining results that prove to be true positives. Alas, this state of affairs will only exist at the end of the Eureka workflow. No classifier exists at the beginning. What can we use for early discard at the start of the workflow?

Based on our experience using Eureka, Figure 3 illustrates the tradeoff that exists between classifier accuracy (higher is better, but not to scale) and the amount of labeled training data that is available. Using an iterative workflow, Eureka helps an expert to efficiently work her way from the extreme left, where just a handful of labeled images are available, to the extreme right, where thousands of training images are used to train a DNN. At the extreme left, the Eureka GUI allows simple features such as color and texture to be defined by example patches outlined on the few training examples that are available. This defines a very weak classifier that can be used as the basis of early discard. Because of the weakness of the classifier, there are likely to be many false positives. Unless the expert restricts her search to just a few data sources, she will be overwhelmed by the flood of false positives. Buried amidst the false positives are likely to be a few true positives. As the expert sees these in the result stream, she labels and adds them to the training set. Over a modest amount of time (tens of minutes to a few hours, depending on the base rate and number of cloudlets), the training set is likely to grow to a few tens of images. At this point, there is a sufficient amount of training data to create a classifier based on more sophisticated features such as HOG, and learned weights from a simple machine learning algorithm such as SVM. The resulting classifier is still far from the desired accuracy, but it is significantly improved. Since the improved accuracy reduces the number of false positives, the number of data sources explored in parallel can be increased by recruiting more cloudlets. Once the training set size reaches a few hundreds, shallow transfer learning can be used. This yields an improved classifier that further reduces false positives, and allows further expansion in the number of data sources, thus speeding up the search.

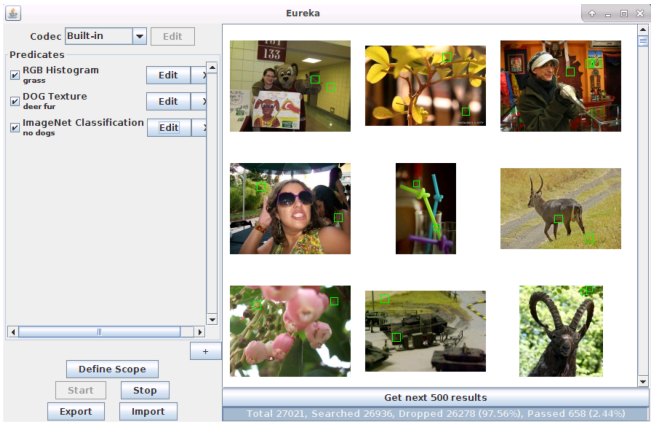


Figure 4. Front-end GUI at the User

Once the training set size reaches a few thousands, deep transfer learning can be used and beyond that, deep learning. This iterative workflow can be terminated at any point if a classifier of sufficient accuracy has been obtained.

Throughout this iterative workflow, the most precious resource is the attention of the human expert. Eureka helps to optimize the use of this scarce resource in two ways. First, it enables immediate use of more accurate classifiers as they are created. Second, as improved classifiers become available, Eureka allows the search to be easily expanded to more data sources, thus harnessing more parallelism to increase the rate at which results are delivered to the expert.

The classifier generated at the end of Eureka's workflow may have *bias*, which is an essential part of expertise. In real life, we overcome bias through mechanisms such as obtaining a second opinion on a medical diagnosis. It is not Eureka's goal to generate a classifier that beats the expert — after all, "your model is only as good as your training data." Rather, our goal is to help in capturing expertise in the form of a training set, which is then used to train a DNN. This DNN will inevitably reflect the bias of the expert who trained it. In future, we envision multiple experts each training a different DNN to allow for "second opinions." Higher-order machine learning approaches can integrate DNNs from several experts into a single classifier.

III. System Design and Implementation

Eureka is designed around two major considerations. The first is *software generality,* allowing use of computer vision code written in a wide range of programming languages and libraries. The second is *runtime efficiency*, allowing rapid early discard of large volumes of data. The Eureka implementation substantially extends the OpenDiamond® platform for discard-based search [5], [6], [7], which was developed in 2003–2010 (prior to the emergence of DNNs). This platform has been used to implement many applications in the medical domain, and they provide a rich collection of domain-specific GUIs that helped us to conceptualize the Eureka workflow. It also gave us a robust framework for

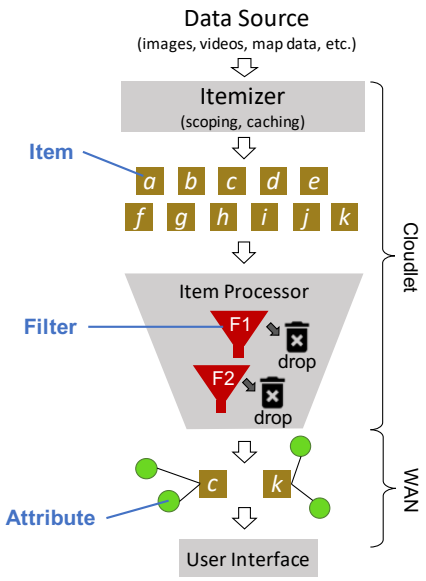


Figure 5. Execution Model

early discard that we were able to use as a starting point in implementing Eureka. Finally, it comes with a rich collection of simple image processing filters that are valuable in the early iterations of a workflow (i.e., at the left of Figure 3).

Eureka is controlled and operated from a domain-specific front-end GUI running on the human operator's computer. Figure 4 shows an example of such a GUI. It allows the user to construct a search query that takes the form of an early-discard pipeline of cascaded filters. The system deploys this set of filters across many cloudlets that host the data collections, and begins searching the associated data in parallel. Only the results that pass these early-discard filters are transmitted and displayed to the user. Figure 5 depicts the logical execution model of Eureka.

A. Data Model and Query Formulation

1) Item: Eureka views data sources as unstructured collections of *items*. Our current implementation supports images, individual frames of a video, or overlapping segments from a video. We focus on images in this paper. The appropriate granularity of items depends on the task. For example, an object detection task may use individual frames as items, while an activity recognition task may use 10-second segments as items. Items are considered independently by Eureka. Item attributes (Section III-A3) facilitate post-analysis of Eureka results using traditional big data technologies such as MapReduce or Spark.

2) Filter: A *filter* is an abstraction for executing any computer vision code in Eureka. A filter's main function is to inspect items, declare which ones are clearly "irrelevant", and then discard them. Eureka defines a simple API between filters and the Eureka runtime. A filter uses these APIs to get user-supplied parameters (e.g., example texture patches) for this query. A filter is required to implement a scoring function, `score(item)`, where it examines a given item and outputs a numeric score. The runtime applies the filter's score function to each item, and if the returned score exceeds a user-provided threshold, the item is deemed to *pass*; otherwise the item is discarded. An early-discard query pipeline consists of multiple filters, with corresponding parameters and passing thresholds. The system requires an item to pass all of the filters (logical conjunction) before transmitting and presenting it to the user. This effectively implements the Boolean operator AND across filters. Eureka could easily be extended to support the full range of Boolean operators and expressions. Eureka performs short-circuit evaluation: once an item fails a filter, it is discarded without further evaluation by later filters in a cascade.

3) Attribute: A filter can also attach *attributes* to an item as a by-product of scoring. Attributes are key-value pairs that can represent arbitrary data, and are accessed using the `get-attribute(item, key)` function, and can be written using the `set-attribute(item, key, val)` interface. The primary purpose of the attribute abstraction is to facilitate communication between filters, where a filter gets attributes set by another. Attributes are analogous to columns in relational databases but with significant differences. In Eureka, attributes are rarely complete for all items (rows) in the data, both due to early-discard of items in the pipeline and due to fast-aborted searches. Additionally, unlike most databases, where the schema tends to be stable, new attributes may be created rapidly in each new query as the user applies new filters (e.g., a retrained DNN). Finally, the user can designate a set of interesting attributes to be retrieved along with the items. Unwanted attributes are stripped off before Eureka transmits results back to the user to reduce bandwidth demand over the WAN. Returned attributes can be used for analysis using other tools.

4) Examples: Figure 6 shows several example filters. While some filters output a pass/fail boolean result (e.g., JPEG decoder), others output a numeric score (e.g., SVM confidence) that can be compared with a threshold. As explained in Section III-B4, the use of numeric scores allows reuse of cached filter execution results even when filter thresholds are changed. The SVM filter in Figure 6 illustrates how attributes enable communication between filters. It uses the `mobilenet_pool1a` attribute created by the MobileNet [8] filter, which in turn uses the `rgb` attribute created by the JPEG decoder, forming a chain of *dependency*. This attribute mechanism allows decomposition of complex tasks into independent, manageable, and reusable components, while still adhering to the filter chain abstraction.

B. Eureka Edge Implementation

Most of the Eureka system runs on cloudlets, colocated with distributed data sources. The cloudlets both store the

Filter	Synopsis
JPEG decoder	jpeg_decode() → bool Decodes a JPEG image. Set-attributes: rgb Returns true if successful, false otherwise.
SIFT matching	sift_match(distance_ratio: float, example: Image) → int Finds matched SIFT keypoints between example and test image. Get-attributes: rgb Returns number of matched keypoints.
MobileNet classification	mobilenet_classify(target_class: string, top_k: int) → bool Classifies image into ImageNet classes and test if target_class is in top_k predictions. Get-attributes: rgb Set-attributes: mobilenet_pool1a Returns true if target_class is in top_k predictions of the test image, false otherwise.
SVM	svm(training_data: List⟨Image⟩) → float Train an SVM with the given training set, using MobileNet's 1024-dimensional feature as SVM input. Then use the SVM to classify the test image. Get-attributes: mobilenet_pool1a Returns probability of the test image being positive.

Each filter has algorithm-specific parameters, get-/set-attributes and return scores. The user can specify a threshold on each filter's return score to drop objects below the threshold.

Figure 6. Examples of Eureka Filters

collected data, and execute queries. As mentioned earlier, Eureka has been designed with software generality and runtime efficiency in mind. This section describes how key components of the Eureka backend address these dual goals.

1) Filter Container: Eureka encapsulates each filter in its own Docker container. This facilitates use of many different frameworks with varying software dependencies concurrently within a single query. For example in Figure 6, the JPEG decoder may be a proprietary library, while SIFT is written in OpenCV, MobileNet in TensorFlow, and SVM in Scikit-learn. Some filters may depend on specific versions of software (e.g., a specific release of TensorFlow).

The containers representing filters can access multi-core CPU resources as well as specialized hardware (e.g., GPUs). For typical uses of GPUs such as DNN inference, we batch incoming items to exploit the efficient batch processing capability of popular deep learning frameworks. Eureka reuses running containers whenever possible, e.g., when the same filter is used in multiple queries. To simplify filter development, we have implemented Docker base images for different Linux distributions. These include all of the logic needed to interface with Eureka as well as a skeleton filter. A developer only needs to add code for the computer vision algorithm that corresponds to the filter being implemented.

2) Itemizer: The itemizer obtains raw data from its data source, transforms it into an item stream, and injects this stream into the execution pipeline. The simplest case just involves loading individual files from disk. More generally, the itemizer can preprocess the data from its native format and transform it into the items needed for the query. For example, it can take continuous data (e.g., streaming or stored video), and emit multiple separate items at a granularity appropriate for the query (e.g., individual frames for object detection, or overlapping short video clips for activity recognition). In addition to selecting granularity, users can also set the *scope* of the itemizer. This can limit the search to only a subset of underlying data based on metadata attributes such as geographical location and recorded date or time. To take advantage of temporal locality in the iterative workflow, the itemizer aggressively caches items that it emits. This *item caching* is in addition to the *result caching* that is described in Section III-B4. The existence of temporal locality in Eureka workloads is in contrast to stream analytics, where each item is processed only once.

3) Item Processor: The item processor is responsible for much of the query execution in Eureka. It evaluates the set of early-discard filters on each item independently, exploiting available data parallelism and multiple cores when available. It uses filter scores and supplied thresholds to decide whether an item should be discarded. As mentioned in Section III-A2, short-circuit evaluation of the filter chain is used to implement early discard. The item processor communicates to the filter containers using a narrow API, of which the three main functions (`score`, `get-attribute` and `set-attribute`) were described earlier. These attributes are retained in memory for access by downstream elements of the filter chain. The narrow-waist API and the use of Docker containers simplify the conversion of off-the-shelf computer vision code into a Eureka filter. Only the items passing all filters (usually only a tiny subset of the entire item stream) are sent to the user. Each transmitted item includes selected attributes that can be used as the basis

of front-end operations such as aggregations and joins.

4) Result and Attribute Cache: Eureka workloads typically exhibit two important properties. First, queries are aborted long before running to completion. As soon as a user decides to refine filters or their thresholds, she is effectively starting a new iteration. Second, as mentioned in Section III-B2, Eureka workloads exhibit high temporal locality because of iterative refinement. Since a new filter chain may overlap with earlier filter chains, the same items may be re-evaluated by the same filters with the same parameters. For example, after viewing some results, a user may retrain an SVM filter with a larger data set. In this case, only the SVM filter is changed — all the other filters repeat computations from the previous query. Alternatively, the user may lower the SVM's threshold. This may enable her to discover some previously missed true positives (i.e., false negatives in the current iteration).

Eureka's reuse of cached results preserves strict execution fidelity as long as a filter is deterministic — i.e., its execution on the same item with the same parameters always produces the same result. To preserve execution fidelity, Eureka immediately detects if the code or parameters of a filter are changed, thus rendering its cache entries stale. This is implemented as follows. Filter execution accesses a subset of attributes (`in_attrs`). It may update or create a set of attributes (`out_attrs`), and typically outputs a `score`. Eureka stores two types of cache entries in a Redis database. In the *result cache*, it writes:

```
(item_id, filter_id, filter_params[]) →
    (score, {h(a) for a in in_attrs},
       {h(b) for b in out_attrs} )
```

where `h(·)` is the hash digest of its argument. In the *attribute cache*, it writes: `h(b)` → `b` for all output attributes in `out_attrs`. When evaluating a query on an item, Eureka first identifies all filters in the query and retrieves all cache entries with the matching `item_id`, `filter_id` and `filter_params[]`. It then validates the cache entries by examining their chains of dependency. A cache entry is deemed valid if and only if all hash digests of its input attributes match the hash digests of the output attributes of another validated entry, or a newly-executed filter. If a valid entry is found, cached results are used; otherwise, the filter is re-executed. This approach avoids redundant execution, ensures correctness of cached results, and minimizes re-computations of hash digests. Compared to whole-query caching (i.e., hashing all filters together as a single cache key), Eureka's finer-grain approach provides more opportunities to reuse prior results.

Note that result caching is very different from blindly precomputing and indexing features or metadata ahead of time. Especially for DNN features, such *a priori* indexing can be quite inefficient or even impossible. There is a vast array of different DNNs that could be used in queries. This set is constantly growing as new deep learning techniques emerge. Furthermore, any given network may be trained on different data, resulting in a very different set of features extracted. Finally, because of early abort of queries and early discard, many filters may never be executed on all data. Hence, aggressive precomputation may be wasteful. In contrast, the attribute and result caching approach used in Eureka can be seen as a lazy or just-in-time partial indexing system [9]. Of course, Eureka can make use of preindexed attributes if they have already been created.

IV. Matching System to User

The previous sections describe how Eureka is able to execute queries efficiently at the edge. A key metric is how well the system can utilize the human expert's time and attention, which is the most precious resource in the system. Here, we present an analysis of the workflow between Eureka and the expert, with the goal of optimizing delivery of items for evaluation. The optimal workflow delivers candidate images *at a rate matching the expert's ability to evaluate them.* Too fast a delivery will waste system resources, and generate a backlog of wasted work that may never be seen by the user; too slow a delivery will frustrate the expert with idle waiting time. Note that although computers usually process much faster than humans, the expert can still be forced to wait when the target phenomena are sufficiently rare (i.e., very low base rate) and the filters are highly selective. Another constraint is that the delivered images should be reasonable candidates; that is, images deemed as obvious negatives should be avoided. This depends, of course, on the set of early-discard filters currently deployed.

In this section we introduce an analytic model to explore parameters that will govern the expert's waiting time. The model is idealized, as its purpose is to show how the various parameters interact, rather than to simulate an actual use case. Figure 7 shows the notation used in the model. Using this notation, Figure 8 defines several well-known metrics that pertain to classifier accuracy.

The base rate represents the "rarity" of the search target. In many valuable use cases of Eureka, the base rate is very low. This makes it difficult for the classifier, even if highly accurate, to yield candidates at a fast rate for the expert to evaluate. Given a single data source, such as a cloudlet that stores and processes data collected from one or a few cameras mounted in some physical environment, we can also compute the *pass rate* — the average fraction of the W images in the world that it passes on for expert inspection.

$$pass\_rate = BR \cdot TPR + (1 - BR) \cdot FPR$$

A. Result Delivery Rate

To evaluate the temporal performance of the system, it is necessary to determine two further parameters: The first is the average time for the filter set to evaluate a single image on a cloudlet and decide whether it is a candidate to pass to the expert. We denote this as t_c. The second parameter is the

TP	True positive: items from the population that are instances of the class, and that are so designated by the classifier.
FP	False positive: items from the population that are not instances of the class, but are accepted as instances by the classifier.
TN	True negative: items from the population that are not instances of the class, and that are so designated by the classifier.
FN	False negative: items from the population that are instances of the class, but that are rejected as instances by the classifier.
W	The size of the entire image population.

Figure 7. Notation

$$TPR = \frac{TP}{TP + FN} \quad \text{True positive rate (aka "recall" or "hit rate")}$$

$$FPR = \frac{FP}{FP + TN} \quad \text{False positive rate (aka "false alarm rate")}$$

$$BR = \frac{TP + FN}{W} \quad \text{Base rate (aka "prevalence")}$$

Figure 8. Metrics Pertaining to Classifier Accuracy

time for the expert to evaluate a candidate image. Although this is inevitably going to vary with the images and the target class, human decision times are likely to be on the order of many seconds or tens of seconds for non-trivial decisions. We denote this as t_h.

Given a single data source (cloudlet with stored camera data), we define the average time to deliver a new image to the expert as *time to next result*, or $TTNR$. This depends on the above parameters in the following way:

$$TTNR = \frac{t_c}{pass\_rate}$$

As we add more data sources to the system, edge computing makes high degrees of coarse-grained parallel processing possible. For N sources,

$$TTNR(N) = \frac{t_c}{N \cdot pass\_rate}$$

B. Optimization Metric

In general, $TTNR(N)/t_h$ is a measure of how well the system's delivery matches the user's time-scale. We will call this ratio the *User:System match*. When the system is performing optimally, the User:System match will approximate 1.0. That is, each new item is delivered when the expert finishes evaluating its predecessor. A match > 1.0 means that the user is waiting; a match < 1.0 means that the system is running ahead, wasting bandwidth and compute resources on candidates the expert may never look at.

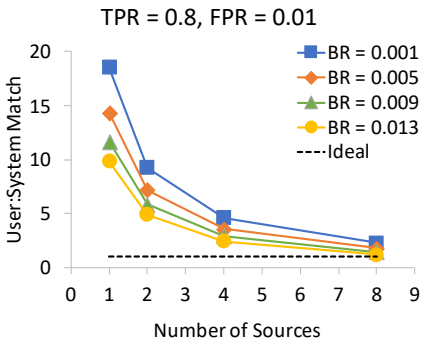
Time is not the only constraint, however. There is also the need to keep the expert occupied with meaningful decisions that will ultimately improve the query. It is always easy for the system to send in more "junk" to avoid letting the user wait. If the User:System match is high, that is, the filters are stringent, the user may wait a long time to see the next candidate. It may be tempting to increase the rate of system delivery by lowering the thresholds on the filters of the query. Given a low base-rate environment, this unfortunately tends to waste the user's time by presenting obviously bad candidates (false positives from the system). In essence, a high false positive rate produces "false positive rage" on the part of the user! – i.e, extreme annoyance. Ultimately, the solution is to iteratively improve the query by combining multiple filters, and incorporating newly found examples to improve accuracy. Ideally, this will reduce false positives without increasing false negatives.

C. Analytical Results

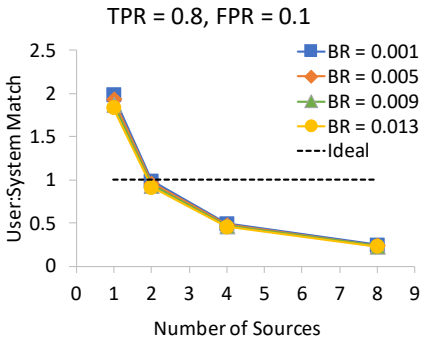
The User:System match depends on the true positive rate TPR, false positive rate FPR, base rate BR of the target phenomenon, and the number of sources N. It also depends on the time for the classifier to process an item at one source t_c relative to the time for the human user to evaluate a delivered item t_h. In example calculations, we assume that the time for the system to process a single item from one source is a factor of 5X faster than the human time to process a delivered image; other assumptions would scale the results differently, but the qualitative pattern would be the same.

Because we are dealing with problems where the base rate is low, the false positive rate governs the system more than the true positive rate. This effect is shown in Figure 9. Figure 9(a) illustrates the User:System match ratio when the classifier is highly accurate: $TPR = 0.8, FPR = 0.01$. Under these conditions, the User:System Match is highly dependent on the base rate and number of sources. Because target items are scarce, and most of the items that the system delivers are true positives, it is highly beneficial to deliver from more sources – more so when the base rate is lower. An optimal ratio is approached here with 8 sources, regardless of base rate. Even at the lowest base rate, the user accepts 1 in 13 items presented, and at the highest base rate, half of the presented items are true positives.

Figure 9(b), in contrast, shows the disastrous effect of a high false positive rate. As the number of sources increases, the user is flooded with irrelevant data and the system backs up, wasting resources. Base rate is irrelevant here because it is not the sparse targets that are the problem. Rather it is the overwhelming flood of false positives. At the lowest base rate, the user rejects 125 items for each one accepted.



(a) Low False Positive Rate



(b) High False Positive Rate

This graph shows User:System match as the number of data sources, N, processed in parallel increases. The ideal match of 1.0 is shown by the dotted line. Base rate (BR) is a parameter. In (a), the classifier has $TPR = 0.8$ and $FPR = 0.01$; in (b) TPR is unchanged, but $FPR = 0.1$.

Figure 9. Negative Effect of High False Positive Rate

D. Summary

In this section, we presented an idealized model to show how the various parameters interact in Eureka, from which implications can be drawn about critical parameters governing system performance. We acknowledge limitations of this calculation. In particular, it is not intended to predict numerical outcomes in real-world use cases of Eureka, where the base rate is generally not precisely known; the delivery is not regular, particularly at the beginning of the process where few examples are available; and the ratio of machine to human processing time may be greatly skewed in favor of the machine. Nonetheless, by characterizing the User:System match, the model provides an observable metric that can be used in Eureka to recruit or omit cloudlets and thus move toward an optimal match.

The analysis further suggests some practical strategies for iterative Eureka searches. First, in the initial iterations, where the query's filter set will have poor accuracy, it does not make sense to scale out the number of sources, as it will just overwhelm the user. Second, for phenomena with such low base rates, it makes sense to focus first on reducing FPR. Once the FPR is under control, additional data sources can be added to optimally match the expert.

Dataset	YFCC100M	ImageNet	COCO
# images	99.2 million	1.2 million	330 thousand
# classes	unlabeled	1000	80

Figure 10. Comparison of Well-Known Image Datasets.

V. Experimental Methodology

As described in Section II, the iterative workflow of Eureka enables users to efficiently discover training data from distributed data sources by incrementally improving the classifier accuracy. We apply this iterative workflow to collect training data for several selected targets and try to answer the question: *Is Eureka effective in reducing the time taken to create a training set?*

We consider two main evaluation criteria:

- the total elapsed time taken to discover a training set of a certain size;
- productivity: number of training examples discovered per unit time, over all of the iterations of Eureka.

We compare Eureka against two alternatives: BRUTE FORCE and SINGLE-STAGE EARLY-DISCARD. With BRUTE FORCE, the user manually inspects and labels every image. For targets with a low base rate this requires the user to plow through many images before collecting even a small training set. SINGLE-STAGE EARLY-DISCARD improves the situation for the user by allowing them to use a simple filter chain to perform early-discard [5]. These filters are based on the initial set of sample images of the target, and are essentially identical to the queries used in the first iteration of the Eureka experiments. However, there is no use of machine learning or of iterative refinement.

Our experiments use 99.2 million images from the Yahoo Flickr Creative Commons 100 Million (YFCC100M) dataset [10]. This is by far the largest publicly available multimedia dataset. The images in it are representative of real-world complex scenes as opposed to datasets such as ImageNet [11] and COCO [12], both of which are curated and have a relatively few, evenly-distributed classes. Figure 10 compares these datasets. As Section VI shows, the number of images that need to be processed in order to discover just 100 true positives easily exceeds the sizes of the ImageNet and COCO datasets.

We run experiments on 8 cloudlets. Each cloudlet has a 6-core/12-thread 3.6GHz Intel® Xeon® E5-1650 v4 processor, 32GB DRAM, a 4 TB SSD for image storage, and an NVIDIA GTX 1060 GPU (6GB RAM). The 99.2 million images in YFCC100M are evenly divided among the cloudlets. Each cloudlet accesses its own subset of the data from its SSD, thus ensuring high bandwidth and low latency. The user's GUI connects to the cloudlets over the Internet.

VI. Experimental Results

We present results for three experimental case studies. In each, we attempt to build a labeled training dataset for a novel target. The three targets chosen are (in descending

(a) Deer

(b) Taj Mahal

(c) Fire hydrant

Left column: examples used in the initial Eureka iteration. Right column: examples discovered using Eureka, and used as training data in following iterations.

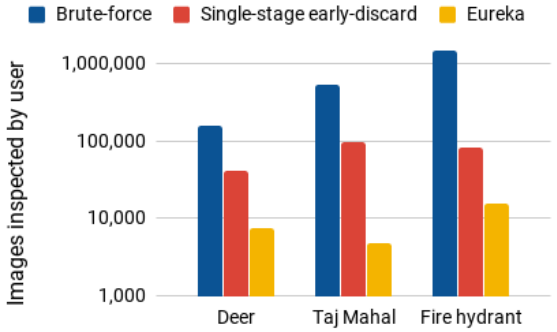
Figure 11. Examples of Targets Used in Our Case Studies

order of base rate): (1) deer, (2) Taj Mahal, and (3) fire hydrant. Figure 11 gives examples of each target. There are no publicly available labeled datasets or off-the-shelf detectors for these targets. Although no specialized expertise is needed to identify these targets, they are fairly rare in Flickr photos and still serve as an initial proof of the Eureka concept. We defer formal user studies in domains such as healthcare or national security to future work.

For each target we started with an initial set of 5 example images. These were used to bootstrap an initial set of filters for the first Eureka iteration. Over five iterations with increasingly sophisticated and better trained filters, we collected a total of approximately 100 new images of the target. The GUI assists users with no programming skills or computer vision background to easily create filters by using example patches from the training images collected so far. For the first two iterations we used explicit features such as color, texture, or shape to perform early-discard of irrelevant data. Once we collected more than 10 positive examples, we used a MobileNet + SVM filter, in which an SVM is trained over the 1024-dimensional feature vectors obtained using MobileNet. The GUI allows the user to easily add and remove examples to just-in-time train the SVM. We incrementally improve classifier accuracy by retraining the SVM when the collected example set is approximately doubled. In general, the optimal point to switch filter types or retrain

Target	Estimated base rate	Images inspected by user	Positive examples discovered
Deer	0.07 %	7,447	111
Taj Mahal	0.02 %	4,791	105
Fire hydrant	0.005%	15,379	74

Figure 12. Summary Results for Case Studies



This graph shows the estimated amount of human attention needed (Y-axis, log scale) using three different approaches to acquire a training set of fixed size (111 images of deer, 105 images of Taj Mahal, and 74 images of Fire hydrant).

Figure 13. Number of Images Presented to User

a machine learning model depends on the characteristics of data, target, and filters, and can be highly empirical. Our rule-of-thumb is to escalate to advanced filters when adding more training examples to the current machine learning algorithm fails to improve the quality of the results.

To allow unbiased evaluation of filters, we used a disjoint subset of the source data in every iteration so that the data being processed was never seen before by the filter. Hence, these experiments did not take advantage of result caching.

Figure 12 summarizes the overall results of using Eureka to build a training set of approximately 100 examples for the three targets. Although the specific numbers are primarily determined by dataset characteristics and filter quality, they give an intuition about the targets' rarity and the human effort spent. We estimate the base rates in YFCC100M based on the metadata of Flickr tags, titles, and descriptions. This metadata is only used in analysis of the results and *not* used in the search process. Although this measure is subject to inclusion error (tag without actual target) and exclusion error (target without tag), it provides at least a crude estimate of the prevalence of the targets in the dataset.

Figure 13 (Y-axis in log scale) compares Eureka to alternatives in the number of images the user has to inspect when building a training set of given size. For brute force, this number is extrapolated using the estimated base rate. For single-stage early-discard, this is based on the precision of the filters used in the first Eureka iteration. We see that single-stage early-discard reduces demand for human attention by an order of magnitude over brute force. Eureka reduces demand by a further order of magnitude.

	Filters	Examples		Items processed	Items shown	New hits	Pass rate	Precision	Elapsed time	Product-ivity
		Pos	Neg							
	Initial set of images	5								
1	RGBhist x2 + DoG texture	5	0	991,814	1,836	5	0.19%	0.27%	12.53	0.40
2	RGBhist x2 + DoG texture	10	0	652,357	2,002	5	0.31%	0.25%	13.98	0.36
3	MobileNet + SVM	15	15	90,047	1,704	17	1.89%	1.00%	11.40	1.49
4	MobileNet + SVM	32	32	130,266	1,204	35	0.92%	2.91%	8.25	4.24
5	MobileNet + SVM	67	67	247,039	701	49	0.28%	6.99%	10.27	4.77

Number of initial examples = 5
Items processed = Images processed by machine on edge nodes
Items shown = Images passing all filters, transmitted and shown to user
New hits = Images labeled as true positives by user in that iteration
Pass rate = Items shown / Items processed
Precision = New hits / Items shown
Elapsed time = Wall clock time of that iteration (minutes)
Productivity = New hits / Elapsed time (# per minute)

Figure 14. Case Study: Building a Training Set for a Deer DNN

	Filters	Examples		Items processed	Items shown	New hits	Pass rate	Precision	Elapsed Time	Product-ivity
		Pos	Neg							
	Initial set of images	5								
1	RGBhist + SIFT + Person	5	0	850,352	3,741	4	0.44%	0.11%	30.17	0.13
2	HOG x2	9	0	245,315	352	5	0.14%	1.42%	9.88	0.51
3	MobileNet + SVM	14	14	228,266	343	13	0.15%	3.79%	8.07	1.61
4	MobileNet + SVM	27	27	590,187	172	37	0.03%	21.51%	20.50	1.80
5	MobileNet + SVM	64	64	633,560	183	46	0.03%	25.14%	15.63	2.94

Columns have the same meaning detailed in Figure 14.

Figure 15. Case Study: Building a Training Set for a Taj Mahal DNN

A. Case Study: Deer

Figure 14 shows the results for the deer training set. For the first iteration we used an RGB histogram filter and a DoG (Difference of Gaussian) texture filter, which use color and texture features respectively to pass the images. Patches of deer fur from the bootstrapping images were given to the DoG texture filter. Patches defining the color of the deer fur and verdure of the scene constituted the two RGB histogram filters. Note that although we use filter names such as RGB, DoG, and SIFT in our current implementation, we expect that a production version of Eureka would use more accessible descriptions such color, texture, and shape. Although the user does not need to know the underlying computer vision algorithms of these filters, she would need to know that a specific filter is indicative of the target class. This is, of course, an essential part of domain expertise.

Iterations lasted 8–14 minutes, with the variability reflecting image processing time on cloudlets, filter accuracy, and human inspection time. Since deer constitute a deformable and varying class of objects, the RGB histogram and DoG texture filters showed no improvement over two successive iterations. The MobileNet + SVM filter introduced in iteration 3, in contrast, showed substantial improvements across iterations in terms of precision. Most importantly, it also resulted in greater productivity (last column). In five iterations, the productivity increased from 0.40 new positives per minute to 4.77, an improvement of more than 10X.

B. Case Study: Taj Mahal

Figure 15 details the steps in obtaining a dataset of 105 positive examples of the Taj Mahal. Since the Taj Mahal is a rigid structure with distinctive features, we used a SIFT (Scale Invariant Feature Transform) filter. The knowledge of the target helped us to include other filters such as an RGB histogram filter (for the white marble) and a human body filter. The choice of the body filter is based on the intuition that the Taj Mahal is a popular tourist destination and is likely to have people in most target images.

On the second iteration, two HOG filters (Histogram of Oriented Gradients) were created based on the 9 available examples, to capture the shape of (1) minarets, and (2) small domes. From the third iteration onwards we used a MobileNet + SVM filter which was improved in each iteration by adding the new positive examples obtained by the prior iteration. The returned false positives were mostly buildings such as Humayun's tomb and Sikandara which closely resemble Taj Mahal's dome and entrance. As can be seen, there is an improvement of user productivity in each iteration. In the final iteration, over a quarter of the items presented to the user are true positives.

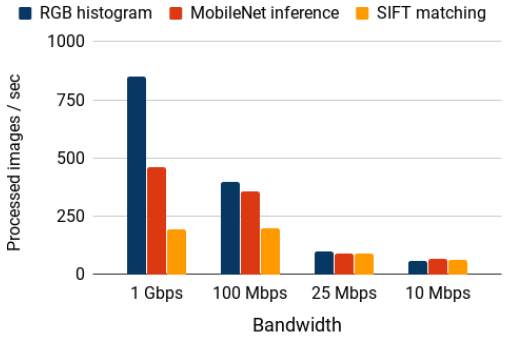
C. Case Study: Fire Hydrant

The base rate of fire hydrant is much lower than the first two targets chosen. In Figure 16, we present measurements from building a training set of 74 fire hydrant images. A HOG filter was used initially to capture the shape of

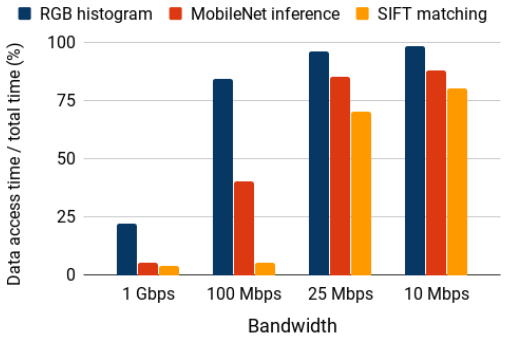
	Filters	Examples		Items processed	Items shown	New hits	Pass rate	Precision	Elapsed Time	Product-ivity
		Pos	Neg							
	Initial set of images	5								
1	HOG x2	5	0	524,136	6,643	6	1.27%	0.09%	13.00	0.46
2	HOG x3	11	0	523,008	3,133	5	0.60%	0.16%	15.15	0.33
3	MobileNet + SVM	16	16	210,688	1,775	9	0.84%	0.51%	7.68	1.17
4	MobileNet + SVM	25	25	517,789	2,856	24	0.55%	0.84%	17.52	1.37
5	MobileNet + SVM	49	49	973,828	972	30	0.10%	3.09%	23.18	1.29

Columns have the same meaning detailed in Figure 14.

Figure 16. Case Study: Building a Training Set for a Fire Hydrant DNN



(a) Image Processing Rate



(b) Fraction of Time Spent on Data Access

Figure 17. Effect of Bandwidth between Cloudlet and Data Source

the hydrants. From the third iteration onwards, using a MobileNet + SVM filter helped to improve precision. Many of the remaining false positives returned by Eureka include British royal mail boxes and traffic cones that resemble fire hydrants. In the later iterations, the classifier accuracy improves significantly but user productivity stalls. This is due to the low inherent base rate and the fact that we have only 8 data sources, resulting in wait time for the user. In this situation, as discussed in Section IV, Eureka should add additional data sources to speed up discovery.

D. The Necessity of Edge Computing

Edge computing is a key enabler of Eureka as it allows the user to scale out to many data sources without stressing the WAN. The proximity of cloudlets to data sources is crucial for efficiency — typically providing LAN connectivity (1 Gbps or higher) to an archival data source.

To study the importance of proximity, we throttled the network bandwidth between cloudlets and their data sources using the Linux command line tool `tc qdisc`. We ran experiments at 1 Gbps, 100 Mbps, 25 Mbps and 10 Mbps. The US national average broadband value of 18.7 Mbps in 2017 [13] lies towards the lower end of this range.

We selected three filters (ordered by increasing cost in terms of computation time): RGB histogram, MobileNet inference, and SIFT matching. Figure 17(a) reports the processing throughput on the cloudlets (processed images per second, higher is better) as we decrease the bandwidth from 1 Gbps to 10 Mbps. At 1 Gbps, the RGB histogram filter achieves significantly higher throughput than MobileNet and SIFT because it is the least computationally expensive. As the bandwidth decreases, its throughput decreases drastically. SIFT, the most expensive filter, is still bound by computation at 100 Mbps, but also suffers from low bandwidth starting at 25 Mbps. Under 25 Mbps, there is only marginal difference in throughput among the three filters, implying data access has become the bottleneck for all of the filters. This can be confirmed in Figure 17(b), where we measure the percentage of total run time spent on retrieving data as opposed to computation (lower is better). At the lowest bandwidth of 10 Mbps, even the most computationally expensive SIFT filter spends 80% of its run time retrieving data, and the RGB filter spends 98%! This confirms the fundamental premise that edge computing is essential for the success of Eureka.

VII. Related Work

While machine learning has long been used in computer vision, the use of DNNs only dates back to the 2012 victory of Krizhevsky, Hinton, and Sutskever in that year's ImageNet competition [14]. Since then, DNNs have become the gold standard for accuracy in image classification. As discussed earlier, DNNs enable direct creation of classifiers by domain experts once the training framework has been set up. No coding is required of them. Their primary task becomes creation of a large, accurately-labeled training set. Since accurate labeling follows implicitly from expertise, finding an adequate number of training examples becomes the dominant challenge. For DNNs to deliver good results, the training set sizes have to be very large: typically on

the order of 10^3 to 10^4 items. It is very time consuming and laborious to manually discover so many true positive items for a phenomenon whose intrinsic base rate is low. This effort cannot be outsourced to non-experts or crowd-sourced. It requires the painstaking effort of domain experts.

To the best of our knowledge, no previous work has studied the above problem specifically in the context of experts. Previous efforts to construct large training sets have typically used crowd-sourcing to achieve high degrees of human parallelism in the discovery process. The work of Vijayanarasimhan et al. [1] and Sigurdsson et al. [2] are two recent examples of this genre. Unfortunately, by definition, expertise in any domain is rare. That is the whole point of using the term "expert" to refer to such a person. Large investments of time in education, training, and experience, possibly amplified by rare genetically-inherited talents, are needed to produce expertise. Experts are not cheap, and crowd-sourcing them is not viable. Further, as discussed in Section I, even if crowd-sourcing experts were viable, there may be stringent controls on the accessibility of data sources. In the extreme case, only one expert may have access.

In spite of sustained efforts in unsupervised and semi-supervised learning [15], [16], [17], [18], supervised learning still provides the best accuracy for domain-specific computer vision tasks. As long as an expert needs to manually create a large and accurately-labeled training set from huge volumes of unlabeled data, Eureka will be valuable.

Eureka substitutes machine parallelism (growing number of cloudlets in the later stages of the workflow) for human parallelism. Rather than trying to harness many experts to work on the same task in parallel, Eureka tries to improve the efficiency of a single expert in discovering training data. We are not aware of any other work that has studied this specific problem. This is an inherently human-in-the-loop problem. Hence, the vast body of "big data" solutions to date (including well-known tools such as Hadoop) have little to offer in this context. Virtually all of those solutions use a batch approach to processing, and depend on optimizations such as deep pipelining that have little value in a context where a user may abort processing at any moment [7].

The approach of using early discard for human-in-the-loop searches of unindexed image data was introduced in 2004 by Huston et al. [5] in a system called *Diamond* [6], [7]. This was, of course, long before the emergence of DNNs and the resulting problem of creating large training sets. Eureka applies the Diamond concepts of early discard, filter cascades, and iterative query refinement to machine learning. As mentioned in Section III-B, the Eureka implementation leverages and extends the OpenDiamond® platform.

The term "early" in "early discard" refers to the processing pipeline from data storage (disk or SSD) through various stages of server hardware, operating system, and application layers, to transmission across the Internet, further processing at the client operating system and application layers, and eventual processing by the human expert. The ability to discard irrelevant data as early as possible in this pipeline improves the efficiency of the system. As discussed earlier in this paper (Section VI-D), edge computing helps in the Internet context by avoiding WAN transfer of items that can be deemed irrelevant by cloudlet processing.

In the current Eureka implementation, filters that perform early discard execute as Docker containers on cloudlets. Further efficiency in early discard could be achieved by executing filters even closer to data storage. This could leverage the large body of work that has been done in the context of *active disks* and *active flash,* also referred to as *intelligent storage* [19], [20], [21], [22], [23], [24], [25], [26], [27]. Closely related are approaches such as Abacus [28], Coign [29], River [30] and Eddies [31] that dynamically relocate execution relative to data based on current processing attributes. Also related is the large body of work on operator ordering for query optimization in relational database systems. This has been a staple of database research over almost its entire history, starting from System R [32] down to today [33], [34], [35], [36], [37].

There has been recent work on the use of *teacher models* for early discard in computer vision [38], [39]. This approach assumes that an accurate, domain-specific teacher model exists to label data. In the absence of such a pre-trained teacher model, there is no alternative to using a real human expert. This is Eureka's sweet spot.

VIII. Conclusion and Future Work

DNNs have greatly increased the accuracy of computer vision on image classification tasks, as shown by the near-human accuracy of recent face recognition software such as DeepFace [40], FaceNet [41] and OpenFace [42]. This success comes at a high price: namely, the need to assemble a very large labeled dataset for DNN training. How to efficiently use an expert's precious time and attention in creating such a training set is an unsolved problem today.

Eureka leverages edge computing to solve this problem. The essence of the Eureka approach is an iterative workflow in which the training examples that have been discovered so far can be immediately used to improve the accuracy of early discard filters, and thus improve the speed and efficiency of further discovery. Our experiments show that Eureka can reduce by up to two orders of magnitude the amount of data that a human must inspect to produce a training set of given size. Our analysis reveals the complex relationship between filter accuracy and parallel search of multiple data sources.

Many future directions are suggested by this work. A natural next step would be to extend the Eureka implementation beyond simple images and video to a wider range of data types, such as whole-slide images in pathology [3], map data, and multi-spectral images. Although this paper focused on independent Internet-based data sources, the Eureka concept has relevance to many other settings, as

discussed in Section I. Today, many large archival datasets are stored in a single cloud data center. Extending Eureka to support this setting would be valuable. The concepts of "edge" and "cloudlet" will need to be re-interpreted, and the implementation will need to be extended to reflect this new setting. From the viewpont of validation, it would be valuable to explore how domain experts use Eureka and whether they benefit from it. This work would include creation of GUIs that bridge the semantic gap between domain-specific concepts and raw data. Another area of future work would be to enhance Eureka's early discard efficiency by using processing embedded in storage when it is available. As discussed in Section VII, modern implementations of concepts such as active disk and active flash have emerged recently and these would be good performance accelerators for early discard. Eureka could also be extended to support commercial products such as IBM's Neteeza appliances [43], which provide specialized edge computing for storage. Finally, our discussion of Eureka so far has focused on its use by a single expert at a time. For reasons discussed in Sections I and II, it is rare that multiple experts are simultaneously available to work on the task of creating a training set for a DNN. However, if one has the good fortune to benefit from multiple experts, it would be valuable to extend the architecture shown in Figure 1 to a multi-user setting. The changes to the Eureka front-end are likely to be deep and extensive if the goal is to allow these experts to actively collaborate in a joint task. The ability to share newly-discovered training examples and newly-created filters, and thereby accelerate the process of discovery would effectively create a new form of computer-supported cooperative work (CSCW). It also offers the possibility of developing new multi-expert techniques to overcome the problem of single-expert bias that was discussed in Section II.

In closing, Eureka is based on the premise that at least some aspects of domain-specific human expertise can be captured in DNNs. Its goal is to help experts efficiently capture their own expertise in the form of a large training set. This difficult and human-intensive task is likely to remain important well into the future, as long as supervised learning remains central to the creation of DNNs.

ACKNOWLEDGEMENTS

We wish to thank our shepherd, Ganesh Ananthanarayanan, and the anonymous reviewers for helping us improve the technical content and presentation of this paper. We greatly appreciate the help of Luis Remis in helping us acquire the YFCC100M dataset. We thank Vishakha Gupta-Cledat, Christina Strong, Luis Remis, and Ragaad Altarawneh for their insightful discussions on Eureka in cloud-based settings. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0051 and by the National Science Foundation (NSF) under grant number CNS-1518865. Additional support was provided by Intel, Vodafone, Deutsche Telekom, Verizon, Crown Castle, NTT, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view(s) of their employers or the above-mentioned funding sources.

REFERENCES

[1] S. Vijayanarasimhan and K. Grauman, "Large-scale live active learning: Training object detectors with crawled data and crowds," *International Journal of Computer Vision*, 2014.

[2] G. A. Sigurdsson, G. Varol, X. Wang, A. Farhadi, I. Laptev, and A. Gupta, "Hollywood in homes: Crowdsourcing data collection for activity understanding," in *European Conference on Computer Vision*, 2016.

[3] A. Goode, B. Gilbert, J. Harkes, D. Jukic, and M. Satyanarayanan, "Openslide: A vendor-neutral software foundation for digital pathology," *Journal of Pathology Informatics*, September 2013.

[4] S. K. Lynn and L. F. Barrett, "'Utilizing' signal detection theory," *Psychological science*, 2014.

[5] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki, "Diamond: A storage architecture for early discard in interactive search." in *Proceedins of USENIX Conference on File and Storage Technologies*, 2004.

[6] M. Satyanarayanan, R. Sukthankar, A. Goode, N. Bila, L. Mummert, J. Harkes, A. Wolbach, L. Huston, and E. de Lara, "Searching Complex Data Without an Index," *International Journal of Next-Generation Computing*, 2010.

[7] M. Satyanarayanan, R. Sukthankar, L. Mummert, A. Goode, J. Harkes, and S. Schlosser, "The Unique Strengths and Storage Access Characteristics of Discard-Based Search," *Journal of Internet Services and Applications*, 2010.

[8] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[9] M. Satyanarayanan, P. Gibbons, L. Mummert, P. Pillai, P. Simoens, and R. Sukthankar, "Cloudlet-based Just-in-Time Indexing of IoT Video," in *Proceedings of the IEEE 2017 Global IoT Summit*, Geneva, Switzerland, 2017.

[10] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li, "YFCC100M: the new data in multimedia research," *Communications of the ACM*, 2016.

[11] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, 2015.

[12] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *European Conference on Computer Vision*. Springer, 2014.

[13] Akamai, "Q1 2017 state of the Internet / connectivity report," 2017.

[14] D. Parthasarathy, "A Brief History of CNNs in Image Segmentation: From R-CNN to Mask R-CNN," https://blog.athelas.com/a-brief-history-of-cnns-in-image-segmentation-from-r-cnn-to-mask-r-cnn-34ea83205de4, April 2017, last accessed on May 17, 2018.

[15] X. Wang and A. Gupta, "Unsupervised learning of visual representations using videos," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015.

[16] C. Doersch, A. Gupta, and A. A. Efros, "Unsupervised visual representation learning by context prediction," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015.

[17] I. Misra, C. L. Zitnick, and M. Hebert, "Shuffle and learn: unsupervised learning using temporal order verification," in *European Conference on Computer Vision*, 2016.

[18] M. Noroozi and P. Favaro, "Unsupervised learning of visual representations by solving jigsaw puzzles," in *European Conference on Computer Vision*, 2016.

[19] A. Acharya, M. Uysal, and J. Saltz, "Active Disks: Programming Model, Algorithms and Evaluation," in *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 1998.

[20] K. Keeton, D. Patterson, and J. Hellerstein, "A Case for Intelligent Disks (IDISKs)," *ACM SIG on Management of Data Record*, 1998.

[21] E. Riedel, G. Gibson, and C. Faloutsos, "Active Storage for Large-Scale Data Mining and Multimedia," in *Proceedings of Very Large Data Bases*, 1998.

[22] X. Ma and A. Reddy, "MVSS: An Active Storage Architecture," *IEEE Transactions On Parallel and Distributed Systems*, 2003.

[23] G. Memik, M. Kandemir, and A. Choudhary, "Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads," in *Proceedings of the International Conference on Parallel Processing*, 2000.

[24] J. Rubio, M. Valluri, and L. John, "Improving Transaction Processing using a Hierarchical Computing Server," Laboratory for Computer Architecture, The University of Texas at Austin, Tech. Rep. TR-020719-01, July 2002.

[25] R. Wickremisinghe, J. Vitter, and J. Chase, "Distributed Computing with Load-Managed Active Storage," in *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, 2002.

[26] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman, "Active Flash: Out-of-core Data Analytics on Flash Storage," in *Proceedings of the 28th IEEE Mass Storage Symposium*, 2012.

[27] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines," in *Proceedings of the File and Storage Technologies Conference*, 2013.

[28] K. Amiri, D. Petrou, G. Ganger, and G. Gibson, "Dynamic Function Placement for Data-Intensive Cluster Computing," in *Proceedings of USENIX Annual Technical Conference*, 2000.

[29] G. Hunt and M. Scott, "The Coign Automatic Distributed Partitioning System," in *Proceedings of USENIX Operating Systems Design and Implementation*, 1999.

[30] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick, "Cluster I/O with River: Making the Fast Case Common," in *Proceedings of Input/Output for Parallel and Distributed Systems*, 1999.

[31] R. Avnur and J. Hellerstein, "Eddies: Continuously Adaptive Query Processing," in *Proceedings of ACM SIG on Management of Data*, 2000.

[32] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access path selection in a relational database management system," in *Proceedings of ACM SIG on Management of Data*, 1979.

[33] P. Menon, T. C. Mowry, and A. Pavlo, "Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last," *Proceedings of Very Large Data Bases*, 2017.

[34] I. Trummer and C. Koch, "Solving the join ordering problem via mixed integer linear programming," in *Proceedings of ACM SIG on Management of Data*, 2017.

[35] K. Dursun, C. Binnig, U. Cetintemel, and T. Kraska, "Revisiting reuse in main memory database systems," in *Proceedings of ACM SIG on Management of Data*, 2017.

[36] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources," *Proceedings of the Very Large Data Bases*, 2017.

[37] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: self-regulating stream processing in heron," *Proceedings of the Very Large Data Bases*, 2017.

[38] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: optimizing neural network queries over video at scale," *Proceedings of the Very Large Data Bases*, 2017.

[39] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri, "Accelerating machine learning inference with probabilistic predicates," in *Proceedings of ACM SIG on Management of Data*, 2018.

[40] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of IEEE Computer Vision and Pattern Recognition*, 2014.

[41] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of IEEE Computer Vision and Pattern Recognition*, 2015.

[42] B. Amos, B. Ludwicznik, and M. Satyanarayanan, "OpenFace: A general-purpose face recognition library with mobile applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-16-118, June 2016.

[43] IBM, "Introducing the next step of Netezza's evolution: IBM Integrated Analytics System," https://www.ibm.com/analytics/netezza, 2018.