AQuA: Adaptive Quality Analytics

Wei Zhang
IBM Research
weiz@us.ibm.com

Martin Hirzel
IBM Research
hirzel@us.ibm.com

David Grove
IBM Research
groved@us.ibm.com

ABSTRACT

Event-processing systems can support high-quality reactions to events by providing context to the event agents. When this context consists of a large amount of data, it helps to train an analytic model for it. In a continuously running solution, this model must be kept up-to-date, otherwise quality degrades. Unfortunately, ripple-through effects make training (whether from scratch or incremental) expensive. This paper tackles the problem of keeping training cost low and model quality high. We propose AQuA, a quality-directed adaptive analytics retraining framework. AQuA incrementally tracks model quality and only retrains when necessary. AQuA can identify both gradual and abrupt model drift. We implement several retraining strategies in AQuA, and find that a sliding-window strategy consistently outperforms the rest. AQuA is simple to implement over off-the-shelf big-data platforms. We evaluate AQuA on two real-world datasets and three widely-used machine learning algorithms, and show that AQuA effectively balances model quality against training effort.

CCS Concepts

•**Applied computing** → **Event-driven architectures;**

Keywords

Events; context; machine learning

1. INTRODUCTION

Event processing systems must take context into consideration to find the best reaction to events. This context often consists of a large amount of data, and must be analyzed by machine-learning analytics to make it actionable. For example, given the input event "consumer visits store", the event processor should make a product recommendation (e.g., which movie to rent, which book to buy, or which restaurant to dine at). The context for this prediction is what similar consumers liked, and the machine-learning problem is collaborative filtering. One approach to building a context-aware event processor is to hand-integrate separate systems for event processing and analytics. Unfortunately, that approach is brittle, involves a slow data copy, and leads to stale (and hence low-quality) models.

DEBS '16, June 20-24, 2016, Irvine, CA, USA
© 2016 ACM. ISBN 978-1-4503-4021-2/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2933267.2933309

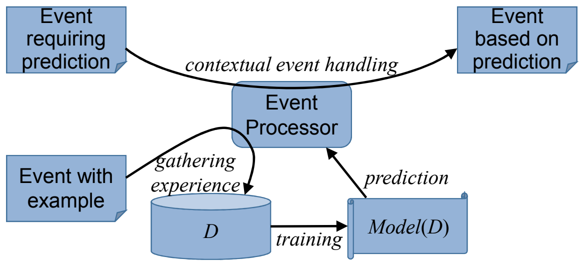


Figure 1: Event processing with analytics in META.

This paper grew out of IBM's "Middleware for Events, Transactions, and Analytics" project, or *META* for short [2], which integrates event processing with analytics. Since the middleware is pre-integrated, solution developers need not integrate by hand, leading to cheaper, more robust solutions. Furthermore, since the analytics happen in-situ, they need not copy the data and can keep models fresher. Figure 1 shows how META works. Given an event requiring prediction (such as "consumer visits store"), the event processor consults a pre-computed analytic model to make a prediction, which is reflected in the output event. Given an event with an example (such as "consumer rates purchase"), the event processor updates the stored training data D. Occasionally, the analytics train a new model $\mathit{Model}(D)$.

Given a middleware such as META, the next problem is how to keep the model quality high and the training cost low. When events with new examples Δ arrive, the old model, $\mathit{Model}(D)$, diverges from the current model, $\mathit{Model}(D \pm \Delta)$. For example, in a product-recommendation system, customers' taste may change or new products may be released; in either case, predictions made on a stale model are likely to have low quality. One simple approach is to retrain on a fixed schedule; unfortunately, that yields an arbitrary cost/quality trade-off that we found is usually suboptimal. A more sophisticated approach that received much recent attention from the systems community is incremental analytics [4, 5, 7, 20, 24, 30, 34, 35, 38]. Unfortunately, many analytics are not available in incremental form; ripple-through effects can cause incremental analytics to have high cost; and the resulting model is not guaranteed to have the best

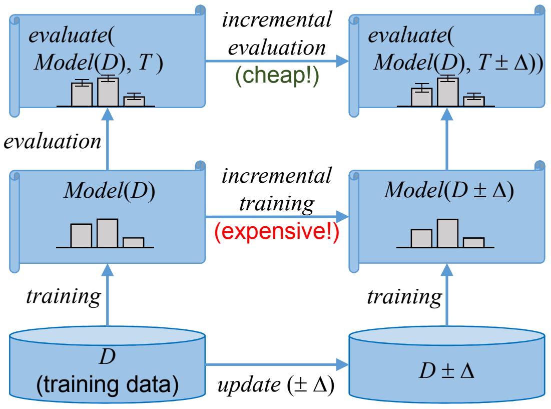


Figure 2: Incremental training vs. incremental evaluation.

quality when obsolete examples remain in the training data.

This paper proposes a complementary approach called AQuA: Adaptive-Quality Analytics. Rather than doing incremental training, it incrementally evaluates model quality. That is simple, inexpensive, and makes it possible to meet a user-specified quality threshold. Figure 2 depicts the AQuA approach and contrasts it with incremental analytics. Given the initial training data D, a machine-learning algorithm constructs an initial model *Model*(D). When an update changes the training data to $D \pm \Delta$, there are two approaches for constructing an updated model *Model*($D \pm \Delta$): either retraining from scratch (bottom right) or incremental training (middle). When D is large, retraining from scratch takes minutes or even hours.

Unfortunately, incremental training is also asymptotically expensive: even when the update Δ is small, subtle changes can ripple through the entire model, and retraining cost depends on the size of D, not Δ. Furthermore, in the presence of gradual or abrupt model drift, it is not always desirable to incrementally update the model without discarding (some) old data. We advocate continuous incremental evaluation, but retraining only on-demand, e.g., when model quality drops below a threshold.

AQuA applies to popular algorithms for common machine learning problems. We evaluated it on ALSWR [45] for collaborative filtering, KMeans [23] for clustering, and CNB [36] for classification on real-world datasets. This paper makes the following contributions:

- A novel unified approach for incremental evaluation of different machine learning algorithms in an event-processing setting.
- A retraining strategy based on sliding windows that consistently outperforms other strategies, is easy to implement, and requires no change to the underlying machine-learning infrastructure.
- An approach for handling (some forms of) abrupt model drift by detecting the change point and using only data from after that point for retraining.

Overall, this paper observes that incremental evaluation is faster than incremental training, and enables cheaply maintaining a high-quality model over changing data. And a high-quality model enables an event processing system to take better actions.

2. BACKGROUND

This section reviews machine-learning concepts and how they apply in an event-processing context. Definitions are in **bold** and referenced in *italics*. Many definitions are adopted from a standard machine-learning text book [32].

2.1 Machine Learning Basics

Example. Data instance used for *training* or *evaluation*, e.g., a triple ⟨consumer, movie, rating⟩. Typically arrives in the system as part of an incoming event.
Features. The set of attributes, often represented as a vector, associated with an example, e.g., a consumer feature vector or a movie feature vector.
Label. Value or category for an example, e.g., a movie rating. Helps event processing system to determine best action.
Supervised learning. The learner sees labeled examples and predicts labels for unseen unlabeled examples. Collaborative filtering and classification are supervised learning problems.
Unsupervised learning. The learner sees unlabeled examples and predicts labels for unseen unlabeled examples. Clustering is an unsupervised learning problem.
Training data D. Examples used to *train* a model.
Test data T. Examples used to *evaluate* the quality of a model. For example, in collaborative filtering, one can use some labeled movie ratings as training data and hold out the rest as test data. In unsupervised learning, the test data and the training data are the same.
Loss function. A function that measures the difference, or loss, between a predicted label and a true label. A typical loss function is RMSE (root mean squared error).
Cost function C. A function used in a machine learning algorithm to guide its progress. Most algorithms solve the optimization problem $\text{argmin}_\Theta\ C(\Theta, D)$, where C is the cost function, D is the given training data, and Θ is the solution to this optimization problem. For example, KMeans minimizes distances between points and cluster centroids.
Model Θ. Parameters that minimize the cost function.
Predict. Apply a model on an unlabeled example to yield a label. In an event-processing system, prediction must occur at low latency, since the label informs the output action.
Train. Find a model Θ that minimizes the cost function. In an event-processing system, training is typically a background process that proceeds concurrently with handling events in the foreground.
Training algorithm. An algorithm used to train a model, e.g., ALSWR for collaborative filtering, KMeans for clustering, or CNB for classification.[1]
Problem. A machine learning problem consists of training a model to make predictions on unseen examples.
Evaluation. Evaluation of a *supervised* learning model means applying the *loss function* to the model and the test data. Evaluation of an *unsupervised* learning model is more difficult, since generally no labeled examples are available [32]. However, since the training algorithm considers one model superior to another if it yields a smaller cost value, the *cost function* can serve as an evaluation metric.
Quality. The result of evaluating a model. This paper normalizes quality so higher values indicate better quality.

[1]This paper assumes an algorithm has been selected and configured for each problem. Algorithm selection and hyperparameter search are out of scope of this paper.

2.2 Adaptive Analytics

Adaptive analytics are only necessary when model quality degrades on new data. If the quality of an old model on new examples did not degrade, then the system could simply continue to use the old model.

New training data Δ. Examples that become available only after the initial model is trained. In an event-processing context, these examples come from events that arrive after the latest round of training.

Model drift. The phenomenon that an old model does not accurately predict new data. Model drift happens when new training examples cause models to change, either gradually or abruptly. One example for gradual model drift for collaborative filtering is when consumers rate new movies. One example for abrupt model drift for collaborative filtering is when many consumers suddenly change their taste. But the point is that there are many possible causes for model drift; rather than tailoring a solution for any one particular cause, we want to maintain a high-quality model no matter the nature of drift.

Incremental training. Update the model to reflect new training data Δ. This is a popular approach in the literature; the Δ typically consists of a small batch of new examples.

Incremental evaluation. For *supervised learning*, given a new labeled example, make a prediction and incrementally update the *loss function*. For *unsupervised learning*, make a prediction and incrementally update the *cost function*. This is the complementary approach we advocate; since evaluation is usually easy to incrementalize, doing it one example at a time incurs no performance penalty.

3. DESIGN AND IMPLEMENTATION

Based on the terminology from the previous section, this section explains AQuA's novel approach for implementing incremental evaluation and quality-directed retraining. One of the design goals was that AQuA should work with the implementations of machine-learning algorithms in off-the-shelf frameworks. For our prototype implementation, we chose Mahout [26] as the framework, as it supports a wide range of algorithms and is a stable big-data analytics engine. This section describes the interfaces for wrapping Mahout as a black-box. AQuA exploits the fact that for most machine-learning problems, the loss function (for unsupervised learning) or cost function (for supervised learning) can be incrementalized. Details for the incremental loss or cost functions are in Section 4.

3.1 AQuA Overall Workflow

Figure 3 presents the overall workflow of AQuA, assuming an initial model has been trained and evaluated. When an incoming event delivers a new training example Δ, AQuA performs several steps.

Step 1. Add Δ to the cache. Section 3.2.3 explains the cache design in detail. For unsupervised learning, Δ goes in the training cache. For supervised learning, Δ goes in an evaluation cache with a user-specified probability p and in the training cache otherwise.

Step 2. Incrementally evaluate the model. First, AQuA does a prediction for Δ. Second, AQuA incrementally updates the loss function for supervised learning or the cost function for unsupervised learning.

Step 3. Check a retraining condition to decide whether to train a new model. AQuA implements several retraining conditions, discussed below in Section 3.2.2.

Step 4. Select retraining data, if retrain is needed. AQuA implements several retraining data selection strategies, discussed below in Section 3.2.2.

Step 5. Retrain model using selected training data, and load the new model for future predictions.

None of these steps are on the critical path for event-processing itself: if the event processor needs a prediction, it can obtain that based on the existing model. However, most of these steps are low-latency, and can therefore be performed in-line with processing an event with example. The only exception is Step 5, which may be slow, and should therefore run as a background process, decoupled from event processing.

3.2 AQuA Interfaces

This section describes our interfaces to algorithms and data stores, and our suite of retraining strategies.

3.2.1 Algorithms Interface

The *AquaAlgo* class is the interface each machine learning algorithm should implement to take advantage of quality-directed adaptive analytic retraining. The methods defined in this class are *train*, *predict*, *evaluate*, and *incrementalEvaluate*. We implemented the corresponding functionality for ALSWR, KMeans, and CNB. Since such methods are central to any given machine learning algorithm, any machine learning frameworks should have them in place or be easy to modify to add such methods.

In addition, *AquaAlgo* normalizes a model's running quality against its initial quality. Higher normalized quality is better. *AquaAlgo* uses the normalized quality for retraining decisions.

3.2.2 Retraining Strategies

AQuA implements four retrain strategies, described in Figure 4. Column Strategy lists the strategy names. Columns Step 3 and Step 4 describe the retrain condition and the training data selection for the corresponding steps from Figure 3. Strategy *Fix* is quality-oblivious and serves as a baseline, emulating a system with periodic retraining but without AQuA. The other strategies are quality-directed and explore different options for which training data to use: *All* uses all data, *Gen* uses data since the last retraining, and Sw uses a sliding window w of the most recent data.

A quality threshold can be too high to maintain, so frequent retraining would be incurred. To alleviate this problem, AQuA also provides a parameter *inertia window* for users to specify. AQuA will hold at least *inertia window* data items before a retraining. That means that quality-directed strategies effectively fall back to the *fixed-size* strategy when the quality threshold is unrealistically high. The *inertia window* prevents AQuA from overreacting to early quality instability. Model quality is reset after retraining, and when evaluated on only few examples, model quality is not representative.

3.2.3 Store Interface

AQuA uses a store interface to make it possible to reuse an unmodified existing analytics framework while at the same time supporting all retraining strategies from Figure 4. We

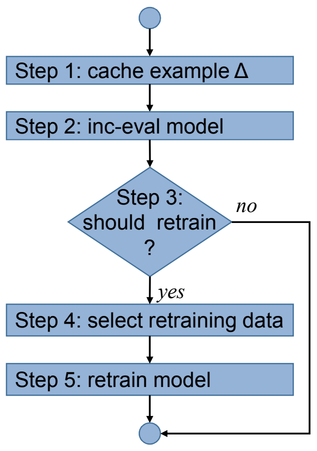


Figure 3: AQuA workflow.

Strategy	Retrain condition for Step 3	Training data for Step 4
Fix	When the number of examples since the last retraining exceeds a fixed threshold, irrespective of quality.	Training data collected since last retraining.
All	Model quality falls below a threshold.	All training data since the beginning of time.
Gen: generation.	Model quality falls below a threshold.	Training data since the last retraining.
Sw: sliding window.	Model quality falls below a threshold.	A fixed amount (w) of the most recent training data.

Figure 4: Retraining strategies.

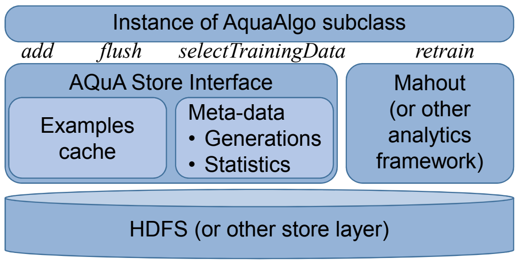


Figure 5: AQuA store interface.

have implemented this store interface to work with HDFS, because it is the underlying file system for Mahout and many other machine learning frameworks (e.g., Spark MLlib [40]). But the design is independent from the underlying store layer and can be easily implemented for a different one.

Figure 5 illustrates AQuA's store interface. When an algorithm adds a new training example, it gets cached in memory. The cache gets flushed to a new HDFS file when its size reaches a threshold or when *AquaAlgo* explicitly calls *flush* before retraining. Each *flush* goes to a unique file. In addition, the store interface manages meta-data for training and test data by maintaining two data structures. The *generations list* keeps track of flushed HDFS files between retrainings. The *statistics* record the locations of flushed HDFS files and how many examples each file contains.

When *AquaAlgo* initiates retraining, it first calls *selectTrainingData* to obtain a list of file names from the AQuA store interface according to the strategy in Figure 4. Then, it asks the off-the-shelf analytics framework to train on those files.

3.3 Abrupt Model Drift

The workflow and retraining strategies discussed so far are designed to deal well with gradual model drift, where the model slowly diverges due to the arrival of new data. However, additional work is required to deal well with abrupt model drift, where the quality of the old model suddenly drops in quality because of a fundamental change. This section defines additional quality metrics, and uses them for detecting and reacting to abrupt model drift.

3.3.1 Quality Metrics

Discussions of quality so far referred to *overall quality*, which is based on all examples used during incremental evaluation, with uniform weight.

To characterize how a model performs on more recent data, we use a widely-known technique in statistical science, the Exponential Moving Average (EMA). Consider a time series $Q = \{q_1, \ldots, q_n\}$, where q_i is the model quality evaluated by using the i-th example. Then, the following equations define the EMA α of model quality:

$$\alpha_1 = q_1 \qquad\qquad \alpha_{i+1} = \alpha_i \cdot (1-p) + q_{i+1} \cdot p$$

The older an example, the less it contributes to the EMA. In theory, p can be set between 0 and 1; in practice, p is usually below 0.1, because a larger p assigns too much weight to the recent data. However, the above equations are not practical for a long history. When $p = 0.05$, an example of age $15,000$ has weight $0.95^{15,000}$, which is indistinguishable from 0 using double floating point precision. Thus, we divide the test data into consecutive chunks of size s and record the model quality evaluated by each chunk, which we call *chunk quality*, $CQ = \{cq_1, \ldots, cq_j\}$, where $s \cdot j = n$. We use CQ instead of Q to calculate a model's EMA, which we call *EMA quality*. Incremental updates to *overall quality*, *chunk quality*, and *EMA quality* have time complexity $O(1)$.

3.3.2 Handling Abrupt Model Drift

We use a simple heuristic to detect abrupt model drift. If *EMA quality* drops at least 10% below *overall quality*, we detect a *change point*. In the hundreds of experiments we conducted on real-world dataset with different values for p (EMA probability) and s (EMA chunk size), we find this heuristic to be effective: it has neither false positives nor false negatives.

When AQuA detects a change point, a naive approach would be to immediately trigger a retraining. However, this is problematic: either the retraining uses old data from before the change point, in which case it will produce a model that is instantly stale; or the retraining uses only new data from after the change point, in which case it may over-fit since there is not enough such data available yet. To avoid this problem, AQuA waits for a user-specified number of additional incoming examples before it starts retraining, using

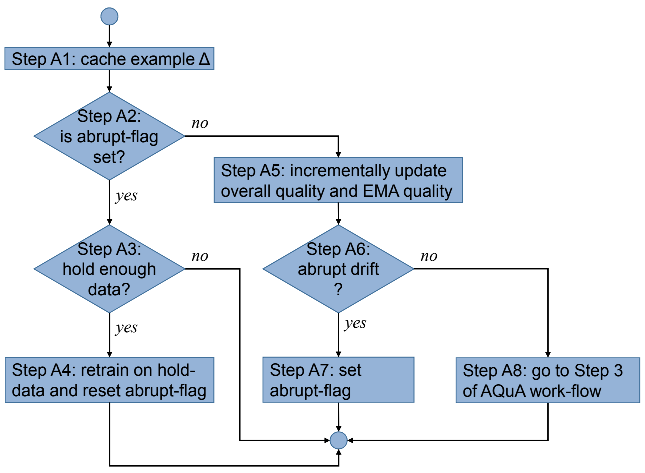

Step A1: cache example Δ
Step A2: is abrupt-flag set?
no
yes
Step A5: incrementally update overall quality and EMA quality
Step A3: hold enough data?
no
yes
Step A6: abrupt drift ?
no
yes
Step A4: retrain on hold-data and reset abrupt-flag
Step A7: set abrupt-flag
Step A8: go to Step 3 of AQuA work-flow

Figure 6: Abrupt model drift detection workflow.

only training data from after the change point. The data AQuA holds before it starts retraining is called *hold-data.*

Abrupt model drift handling is orthogonal to the retraining strategies described in Section 3.2.2. That is, only when an abrupt model drift is detected, AQuA retrains the model using the hold-data. One can freely choose different retraining strategies (e.g., S15: sliding window of width $w = 15$) to handle gradual model drift. Figure 6 presents the workflow of abrupt model drift detection:

Step A1. Add Δ to the cache, same as Step 1 of Figure 3.
Step A2. Check if the abrupt model drift flag is set true.
Step A3. If the check from Step A2 is true, check if enough training data is held since the change point was detected.
Step A4. If there is enough training data, then retrain on the hold-data and reset the abrupt model drift flag back to false.
Step A5. Incrementally update overall quality and EMA quality.
Step A6. Check if EMA quality is 10% lower than the overall quality to detect the abrupt model change point.
Step A7. If the check from Step A6 is true, set the abrupt model change flag to be true, and return.
Step A8. Jump to Step 3 of Figure 3 to detect gradual model drift, if any.

With the exception of Step A4, executing the workflow incurs only a small latency. It is designed to run in-line with the processing of each event with example. The workflow maintains state (the abrupt-flag) from one event-triggered execution to the next.

Overall, this section presented our AQuA approach for maintaining a high-quality analytic model during event processing, as well as a set of pluggable retraining strategies for handling both gradual and abrupt model drift. Next, we will look at a set of pluggable machine learning algorithms that work with AQuA.

4. MACHINE LEARNING ALGORITHMS

AQuA works for any concrete machine learning algorithm that can support prediction, non-incremental training, and both non-incremental and incremental variants of evaluation (see Section 3.2.1). This section discusses how to support those functions for three representative machine learning algorithms ALSWR, KMeans, and CNB. We chose these three algorithms to cover a diverse set of machine learning problems; AQuA also works for other algorithms not discussed here. Popular machine learning frameworks provide off-the-shelf implementations of prediction and non-incremental training. On the other hand, the description of incremental evaluation is a novel contribution of this section.

4.1 ALSWR

ALSWR (Alternating-Least-Squares with Weighted-λ-Regularization [45]) is a widely-used algorithm for *collaborative filtering.* Collaborative filtering is the following supervised learning problem: given ratings by consumers for some products, predict ratings for other products. The intuition for solving this problem is that known ratings from one consumer can predict unknown ratings from another consumer if the two consumers are similar in their known ratings.

A sample use case for collaborative filtering is for product recommendations, for instance, to recommend a movie that similar consumers have rated highly but that the consumer has not yet watched. Cast in terms of Figure 1, that means the *event requiring prediction* occurs when a consumer asks for a recommendation, and the *event with example* occurs when a consumer rates a product.

In the context of an event processing system, a batch Δ of new examples consists of new ratings by consumers for products. The word *collaborative* in collaborative filtering refers to the fact that these specific examples also predict ratings for other, similar, consumers and products. Therefore, an incremental training algorithm would have to update the model for those other consumers as well, which may affect even more consumers, and can eventually ripple through the entire population. Therefore, incremental training for collaborative filtering requires work proportional to the size $|D|$ of the training data set, not just the amount $|\Delta|$ of new data.

Evaluation for ALSWR consists of computing the root mean squared error, or RMSE. Let $\mathit{error}(t)$ be the error for one example t, defined as $|t.\mathit{rating} - \mathit{Model}(D).\mathit{predict}(t)|$. Then evaluation for the entire test data T uses the formula:

$$\mathit{evaluate}(\mathit{Model}(D), T) = \sqrt{\frac{\sum_{t \in T} \mathit{error}(t)^2}{|T|}} \qquad (1)$$

For *incremental evaluation*, we separately maintain the running total for $\mathit{sum} = \sum_{t \in T} \mathit{error}(t)^2$ and for $\mathit{count} = |T|$ while examples trickle in one event at a time. Given an incoming event with a new example t', incremental evaluation simply updates $\mathit{sum} \mathrel{+}= \mathit{error}(t')^2$ and $\mathit{count} \mathrel{+}= 1$. That means that after several new examples Δ, the current evaluation is simply $\mathit{evaluate}(\mathit{Model}(D), T \pm \Delta) = \sqrt{\mathit{sum}/\mathit{count}}$. Incremental evaluation requires only $|\Delta|$ steps. That means that for ALSWR, if $|\Delta| \ll |D|$, incremental evaluation is much faster than incremental training.

4.2 KMeans

KMeans [23] is a widely-used algorithm for *clustering.* Clustering is the following unsupervised learning problem: given a set D of examples with feature vectors, find subsets (clusters) whose feature vectors are similar. The KMeans algorithm starts by randomly picking k examples as cluster centroids. Then it repeats the following steps for i iterations:

assign each example in D to the cluster around the nearest centroid, and update each centroid based on the examples assigned to its cluster. KMeans relies upon a distance function that quantifies the similarity between feature vectors.

A sample use case for clustering is anomaly detection, for instance, detecting that an insurance claim is unlike others that have been seen before, and should therefore be audited by a case worker. Cast in terms of Figure 1, that means that an event with an unlabeled insurance claim can play both the role of the *event requiring prediction* and the *event with example.* More generally, the primary advantage of unsupervised learning is that it does not require labeled examples.

Given a batch Δ of new examples (i.e., feature vectors), each new example gets added to the cluster with the closest centroid. However, adding an example to a cluster may cause its centroid to shift, which may cause other examples to move to a new cluster, which may cause other centroids to shift as well. In other words, the effect of one new example can eventually ripple through the entire data set D. Therefore, incremental training for KMeans requires work proportional to $|D|$.

Evaluation for KMeans consists of computing the average distance of all test examples from the centroids of their predicted clusters. Let $\mathit{error}(t)$ be the error for one example t, defined as $\mathit{distance}(t, \mathit{Model}(D).\mathit{predict}(t).\mathit{centroid})$. The formula for evaluating clusters over test data T is:

$$\mathit{evaluate}(\mathit{Model}(D), T) = \frac{\sum_{t \in T} \mathit{error}(t)^2}{|T|} \qquad (2)$$

As before, *incremental evaluation* maintains the running total $\mathit{sum} = \sum_{t \in T} \mathit{error}(t)^2$ and $\mathit{count} = |T|$. Given an incoming event with a new example t', it adds $\mathit{sum} \mathrel{+}= \mathit{error}(t')^2$ and $\mathit{count} \mathrel{+}= 1$. After several new examples Δ, the up-to-date evaluation is $\mathit{evaluate}(\mathit{Model}(D), T \pm \Delta) = \mathit{sum}/\mathit{count}$. Incremental evaluation requires only $|\Delta|$ predictions. That means that for KMeans, if $|\Delta| \ll |D|$, incremental evaluation is much faster than incremental training.

4.3 CNB

CNB (Complementary Naive Bayes [36]) is a widely-used algorithm for *classification.* Classification is the following supervised learning problem: given a fixed set of categorical classes (which can be represented by integers $\{0, \ldots, k-1\}$), given a training set D of examples labeled with their class, predict the class of new unlabeled examples.

A sample use case for classification is when a patient is being admitted to a hospital, the receptionist can use the displayed symptoms and other features to route the patient to the right department. Cast in terms of Figure 1, that means the *event requiring prediction* occurs when a receptionist admits a patient, and the *event with example* occurs when a doctor diagnoses a patient.

A standard machine learning text book observes that Naive Bayes is the only learning method that does not perform an explicit search through the space of possible hypotheses [31]. Therefore, CNB is unusual in that it has a simple incremental training algorithm. Given a batch Δ of new examples, it can update the model in $|\Delta|$ steps without having to individually reconsider $|D|$ old examples.

Evaluation for CNB can simply track how often the prediction is correct on average, i.e., how often the true class equals the predicted class. The error for one example t is $\mathit{error}(t) = \textbf{if } t.\mathit{class} = \mathit{Model}(D).\mathit{predict}(t) \textbf{ then } 0 \textbf{ else } 1$. The following formula evaluates CNB on test data T:

Algorithm	Training: non-incremental	Training: incremental	Evaluation: incremental
ALSWR	$O(i\|D\|)$	$O(i\|D\|)$	$O(\|\Delta\|)$
KMeans	$O(i\|D\|)$	$O(i\|D\|)$	$O(\|\Delta\|)$
CNB	$O(\|D\|)$	$O(\|\Delta\|)$	$O(\|\Delta\|)$
NB	$O(\|D\|)$	$O(\|\Delta\|)$	$O(\|\Delta\|)$
Lin.Reg.	$O(i\|D\|)$	$O(i\|D\|)$	$O(\|\Delta\|)$
Log.Reg.	$O(i\|D\|)$	$O(i\|D\|)$	$O(\|\Delta\|)$
N.Network	$O(i\|D\|)$	$O(i\|D\|)$	$O(\|\Delta\|)$

Table 1: Computational complexity analysis of widely-used machine learning algorithms, where i is the number of iterations, $|D|$ is the initial training data set size, and $|\Delta|$ is the number of new examples.

$$\mathit{evaluate}(\mathit{Model}(D), T) = \frac{\sum_{t \in T} \mathit{error}(t)}{|T|} \qquad (3)$$

Again, *incremental evaluation* maintains the current running total $\mathit{sum} = \sum_{t \in T} \mathit{error}(t)$ and $\mathit{count} = |T|$. Given an incoming event with a new example t', add $\mathit{sum} \mathrel{+}= \mathit{error}(t')$ and $\mathit{count} \mathrel{+}= 1$. The current evaluation after several new examples Δ is $\mathit{evaluate}(\mathit{Model}(D), T \pm \Delta) = \mathit{sum}/\mathit{count}$. That means that for CNB, both incremental training and incremental evaluation require $|\Delta|$ steps.

4.4 Summary of Complexity Analysis

Table 1 summarizes the time complexity of seven widely-used machine learning algorithms. In addition to the three discussed in detail above, it adds Naive Bayes (NB), Linear Regression (Lin.Reg.), Logistic Regression (Log.Reg.) and Neural Network (N.Network). NB is a similar algorithm to CNB, but usually performs worse than CNB. Lin.Reg., Log.Reg., and N.Network are more advanced classification techniques that are based on a Stochastic Gradient Descent (SGD) solver. Incremental evaluation usually has a better time complexity than incremental training. The only exceptions are CNB and NB, whose incremental training has the same complexity as incremental evaluation. However, off-the-shelf implementations of non-incremental machine learning algorithms are more widely available than incremental ones. Therefore, a contextual event processing system is likely to have only non-incremental training at its disposal.

5. EXPERIMENTAL METHODOLOGY

AQuA is largely independent from the specific machine learning algorithm and underlying implementation framework. To evaluate its utility, we seek to realistically simulate scenarios in which predictive analytics are being applied to an incoming event stream. This implies a need for large datasets with timestamped elements.

The Netflix and Wikipedia datasets are the largest publicly available datasets that meet our criteria. The Netflix dataset is 1.4GB and contains 100 million ⟨user,movie,rating⟩ examples from 1998 to 2005. Ratings range from 1 (least favorite) to 5 (most favorite). The Wikipedia dataset is 44GB and contains edit history for all Wikipedia documents from 2002 to August, 2014 (over 14 million documents). Both datasets have been used extensively in big data research [12,

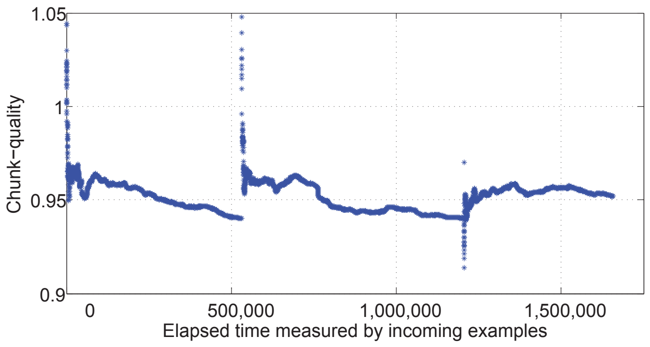


Figure 7: Quality over time for CNB, using strategy S15 with quality threshold 94%.

Strategy	Training data size		Training time	
	Overall	In-strategy	Overall	In-strategy
Fix	49	13	73	16
All	74	4	25	5
Gen	33	4	39	3
S10	3	0	27	2
S15	6	0	15	2
S20	14	0	22	1

Table 2: Sub-optimal scores for ALSWR+KMeans+CNB. Lower is better, since a lower number indicates there are fewer cases where the strategy gets beaten by other strategies (Overall) or by the same strategy with a different quality threshold (In-strategy).

15, 18, 25, 37, 44].

For the purpose of evaluating model quality for CNB, we need to assign a ground truth label to each Wikipedia document. We do this by using the categories assigned by human editors to the document. By sampling 25 popular categories (e.g., science) we define a subset of the dataset that contains over 2 million documents. For KMeans, we use Cosine distance between the vectors. We use standard tf-idf vectorization to obtain a numerical representation of each Wikipedia document. Using a sparse vector representation, the size of the dataset is 8GB (multiple orders of magnitude smaller than using dense vectors).

We use the term **whole-world** dataset to refer to the entire data set, i.e., 100 million ratings in Netflix and over 2 million documents in Wikipedia. We sort the whole-world dataset by timestamp, and use the oldest 20% of the whole-world data as training data to generate an initial model. The remaining 80% of the data is presented in timestamp order to AQuA as an incoming training data stream.

We use Mahout 0.9 [26] as the base machine learning framework. Since exploring distributed scale-out is not the primary goal of our work, we run all KMeans and CNB experiments on one server, which has 16 2-way 2.0 GHz cores and 130GB of memory; and we run all ALSWR experiments on another server, which has 20 2-way SMT 2.4 GHz Intel Xeon cores and 260GB of memory.

6. EXPERIMENTAL RESULTS

This section presents the results of experiments designed to answer fundamental questions about AQuA:

- Section 6.1: How does each retraining strategy trade off training cost against model quality? Is a quality-directed retraining system better than a quality-oblivious one?
- Section 6.2: What is the right retraining strategy when there is abrupt model drift?
- Section 6.3: How does AQuA compare with incremental training in terms of processing speed and model quality?
- Section 6.4: What AQuA parameters should a user tune?

6.1 Evaluating Retraining Strategies

To illustrate how we compare retraining strategies, we first discuss results for the CNB classification algorithm in detail. Figure 7 illustrates how the model quality changes across time during one particular benchmark run of CNB with strategy S15 (sliding window of width $w = 15$, see Figure 4) and quality threshold 94%. Recall that AQuA normalizes quality so the initial quality is 1. As new examples arrive, quality slowly degrades until it reaches the threshold. At that point, AQuA trains a new model, causing quality to jump up. This process repeats.

We ran hundreds of experiments and summarized each benchmark run by two numbers: accumulative chunk-quality (i.e., the area under the curve in Figure 7) and retraining effort (measured by total training time or by total number of examples used to for all retrainings in the run). Figure 8 plots the reciprocal of quality (y-axis) against the retraining effort (x-axis). On both axes, lower is better. The Pareto frontier nearest the origin holds optimal trade-offs (i.e., better quality with less retraining effort).

Most strategies from Figure 4 are represented by multiple data points in each of the plots in Figure 8. For the Fix strategy, each data point characterizes the quality-effort trade-off for a chosen training dataset size, e.g., 10% of the whole-world dataset. For the quality-directed retraining strategies (All, Gen, and Sw), each data point characterizes the quality-effort trade-off for a chosen quality threshold, e.g., 90%. We use the same set of quality thresholds (90%, 92.5%, 94%, and 95%) for each quality-directed strategy. A higher quality threshold yields a smaller inverse quality (lower y-axis value).

Continuing with the example of CNB, Figure 8(e) shows that the sliding-window strategies (S10, S15, and S20) do best. The All strategy does worst, because it trains with a larger data set but arrives at the same quality. Furthermore, the sliding-window strategies are the most stable in the sense that changing the quality threshold moves them along the Pareto frontier, but does not yield a totally better result. Most of the other plots in Figure 8 look similar to Figure 8(e), and we summarize them in Table 2.

Table 2 reports the *sub-optimal score* and *in-strategy sub-optimal score* for all strategies over all three algorithms. We say a data point p_1 **dominates** data point p_2 in the plot if p_1 takes less training effort *and* yields better model quality than p_2. For each data point p, its **sub-optimal score** is the number of points that dominate p, and its **in-strategy sub-optimal score** is the number of points with the same strategy as p that dominate p. We generalize the scores from individual points to strategies by summation.

The *sub-optimal score* characterizes how well a strategy

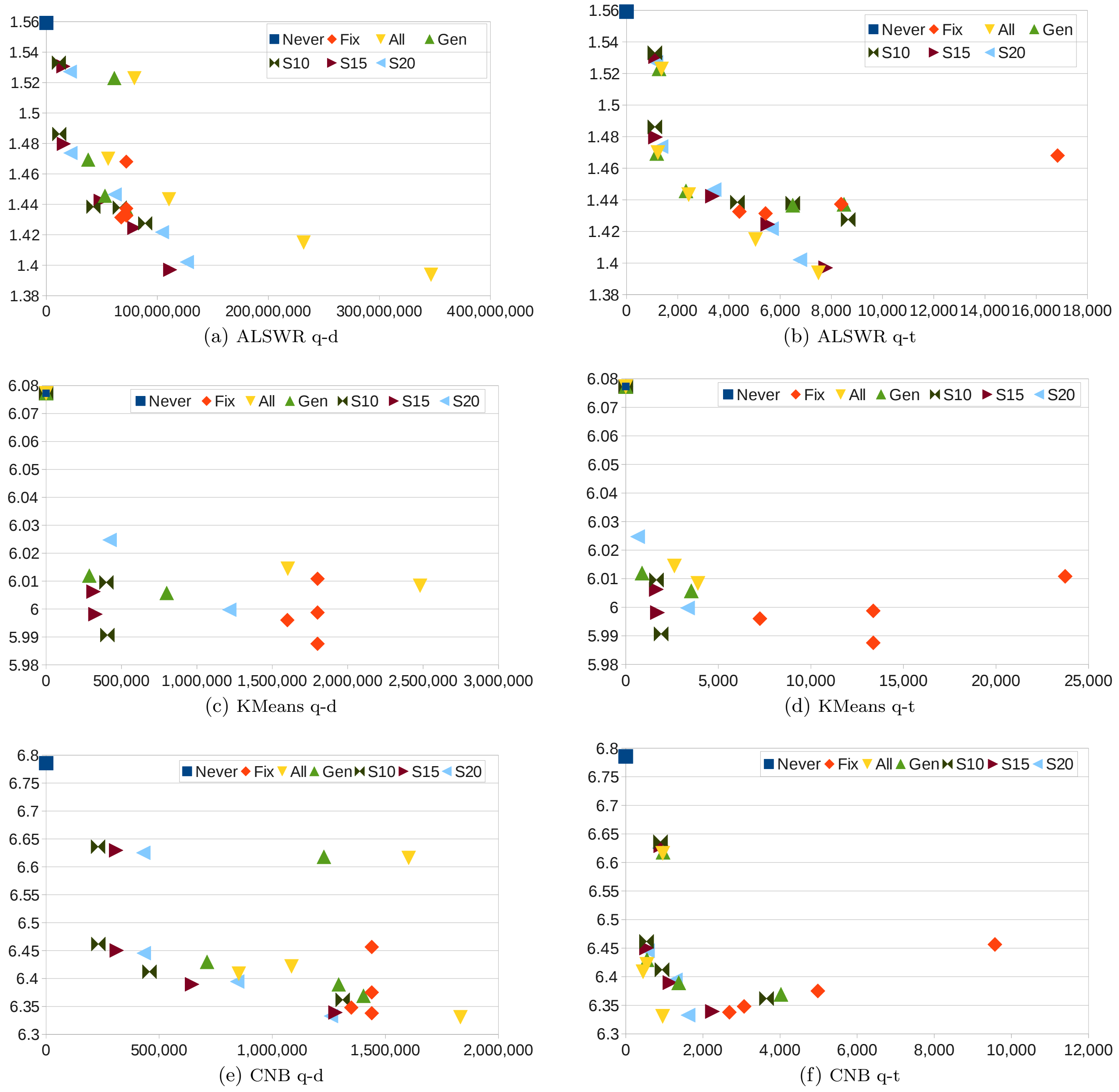
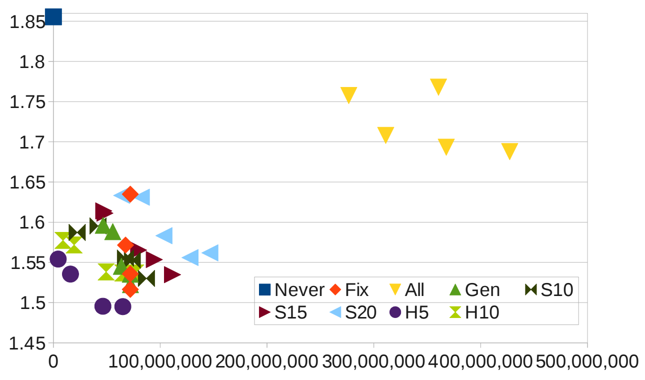
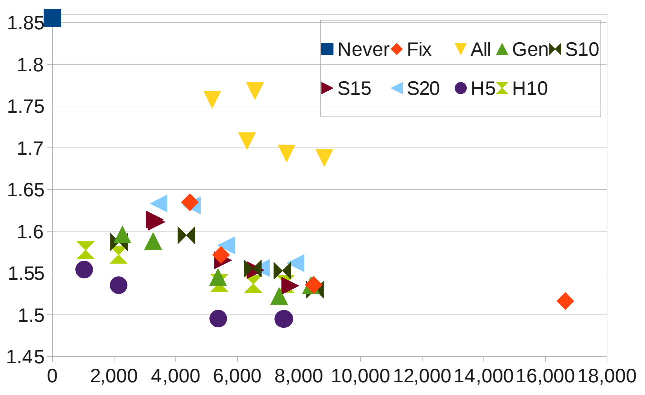


Figure 8: We plot the reciprocal of quality (y axis) against retraining effort (x axis). Quality is measured by the model's accumulative *chunk*-quality; lower y values indicate better quality. The left column plots quality vs. training data size, measured in number of training data examples (i.e., *q-d* plot), the right column plots quality vs. training time, measured in seconds (i.e., *q-t* plot); lower x values indicate less retraining effort. The Pareto frontier nearest the origin holds optimal trade-offs (i.e., better quality with less retraining effort). In these experiments, we used quality thresholds of 80%, 82.5%, 85%, 87.5%, and 90% for ALSWR, 90%, 95%, 96%, 97%, and 98% for KMeans, and 90%, 92.5%, 94%, and 95% for CNB. A higher quality threshold yields a smaller value for reciprocal of quality (lower y-axis value in these plots).



(a) ALSWR abrupt model drift q-d



(b) ALSWR abrupt model drift q-t

Figure 9: Similar plots to Figure 8 (i.e., *q-d* plot and *q-t* plot) for scenario with abrupt model drift. H5 and H10 are strategies that detect abrupt model drift, and use hold-data of 5% and 10% of the whole-world data size, respectively.

Strategy	Training data size		Training time	
	Overall	In-strategy	Overall	In-strategy
Fix	38	4	26	0
All	172	2	116	2
Gen	25	0	17	1
S10	33	1	25	1
S15	51	0	28	0
S20	96	1	51	1
H5	1	1	1	1
H10	16	0	11	0

Table 3: Sub-optimal scores for ALSWR in the presence of abrupt model drift. Lower is better.

compares against other strategies, the lower the better. The *in-strategy sub-optimal score* characterizes a strategy's stability: a lower in-strategy sub-optimal score indicates a more stable strategy, since different parameter settings make different but not strictly better trade-offs. Sliding-window strategies consistently offer the best trade-off between retraining effort and model quality, and they are stable. In addition, the Fix strategy, which is quality-oblivious, consistently performs worse than quality-directed retraining strategies.

6.2 Abrupt Model Drift

To test if AQuA can handle abrupt model drift, we inverted all the Netflix movie ratings (e.g., change 5 to 1, 4 to 2, etc.) starting at the middle of the incoming training data stream. For the change-point strategies, we experimented with two different hold-data sizes: 5% and 10% of the whole-world data size (denoted H5 and H10). When no abrupt model drift was detected, the change-point strategies fall back to S15 as the default retraining strategy.

Figure 9 plots accumulative chunk-quality (y axis) against retraining effort (x axis) in the presence of abrupt model drift. The change-point strategies H5 and H10 are close to the Pareto frontier. Table 3 tallies up the overall and in-strategy sub-optimal scores from Figure 9. When there is an abrupt model drift, the sliding-window strategy with abrupt model drift detection works best. Strategy All is the worst, because it always contains conflicting training examples. Furthermore, the Fix and All strategies are less stable than others.

6.3 Comparison with Incremental Training

The experimental results so far focused on the cost of retraining. Here, we report numbers on the cost of incremental evaluation, i.e., the time to update the quality estimate when a new example arrives. Note that for ALSWR, each example is a movie rating, and for KMeans and CNB, each example is a Wikipedia document. We measured 3.9μs for ALSWR, 4.5ms for KMeans, and 4.3ms for CNB. For most real-world event processing applications, these costs are unlikely to become the bottleneck. For comparison, a widely-cited incremental training paper [5] reports that incremental training takes at least 1% of the time of from-scratch training, which takes over 1,000 seconds. That works out to >10s, which is 4 orders of magnitude worse than our incremental evaluation.

Next, we want to compare the quality of incremental evaluation and incremental training. To obtain an upper bound for an incremental training system's model quality, we evaluate the new model using the test data collected since the previous retraining at the end of each AQuA retraining phase. We then compare this to the quality when AQuA started retraining, which is the model quality AQuA produces. On average, the upper-bound quality of incremental training is at most 13.5% higher for ALSWR, 1.2% higher for KMeans, and 3.7% higher for CNB.

Our results indicate that AQuA can significantly reduce training effort with modest impact on model quality. A user can easily decide how to balance training effort and model quality by setting the quality threshold for AQuA.

6.4 AQuA Tunable Parameters

Being an adaptive system, AQuA requires users to provide only four parameters: *quality threshold*, *inertia window size*, *sliding window size*, and *hold-data size*. We expect that users with some experience with the data set and analytics should find it easy to supply usable values for these parameters.

For example, consider when a user migrates a system that was based on a fixed-size retraining schedule (corresponding to the Fix strategy) to AQuA. Let N be the size of the original schedule. If the user demands a relatively low *quality-*

threshold, AQuA would rarely trigger retraining, thus saving a significant amount of training effort while satisfying the user's demand. If the *quality-threshold* was set unreasonably high, as a rule of thumb, one can set *inertia window size* and *sliding window size* to the original size N; AQuA would then automatically degenerate to the old fixed-size schedule system. If the quality-threshold is set at a reasonable value, then AQuA can guarantee a model of that quality threshold while minimizing the retraining effort. Throughout our experiments, we start with a reasonable *inertia window size* (i.e., the amount of data that can sufficiently train a model to be useful), then experiment with *sliding window size* of 1×, 1.5×, and 2× the *inertia window size*. One can set the *inertia window size* to the original fixed size N.

For *hold-data size*, our experience indicates that setting it to the *inertia window size* (or a fraction of that, e.g., 0.5×) can achieve the best tradeoff, because AQuA suffers less from stale data.

7. RELATED WORK

This paper was part of a broader research project called META: Middleware for Events, Transactions, and Analytics [2]. But whereas this paper deals with machine learning, other papers to come out of the META project do not. Instead, they investigate data-store issues including snapshot consistency [9], event processing over a JSON store [13], and translating rules to queries [39].

The goal of this paper is to maintain a model for a changing data set with high quality and low performance cost. This is addressed by classic incremental frameworks like Rete [16] or (stream-)relational view maintenance [19, 41]. Whereas these frameworks assume centralized data with immediate and exact descriptive analytics, this paper investigates distributed data with delayed and approximate predictive analytics.

Big data analytics is an extensively studied area. Many large-scale machine learning platforms exist [10, 17, 25, 27]. Mahout [26] and MLlib [40] implement machine-learning algorithms in Bulk Synchronous Parallel (BSP) style [43]. Several recent systems [4, 5, 7, 20, 24, 30, 34, 35, 38] incrementalize distributed big-data frameworks such as MapReduce [11] and Dryad [22]. These incremental systems help only when the input change is small enough for the incremental algorithm to be faster than from-scratch recomputation. Furthermore, they all focus on performance, without measuring or considering quality. This paper is complementary: we show how to make quality-directed retraining decisions for Pareto-optimal results, irrespective of whether the underlying framework is incremental or not.

Fluxy is a framework that avoids retraining when quality-of-data (QoD) exceeds a threshold [14]. In contrast, we incrementally evaluate the quality of machine-learning models.

The systems community is not alone in investigating how to maintain a model for a changing data set: this has also been studied in machine learning. For instance, Masud et al. show how to maintain high-quality classifiers when concepts drift [29] or evolve [1]. Hoi et al. use multiple-kernel learning to adapt to changing data by adjusting both the kernels and their relative weights [21]. And machine unlearning removes the effect of old data from a model [8]. Like AQuA, these papers focus on quality of the model. However, all of these papers exploit characteristics of specific machine learning algorithms (i.e., classifiers). In contrast, we take a black-box, systems approach and enable the user to trade quality for performance.

Transfer learning [42] is a method that reuses the features or models learned from one training run to the next. While promising for certain tasks (e.g., auto-encoder), it is difficult to generalize it to all learning tasks and it is difficult to decide if the learning transfers the negative knowledge without evaluating the model.

Irregular sampling [28] is a sampling method that reconstructs statistical properties within a non-uniformed time series. AQuA treats each incoming example as a testing datum to continuously evaluate the model, thus the statistical properties are well maintained and evaluated.

Online learning [6] is a highly active research area in machine learning. Stochastic gradient descent (SGD) is the most adopted online learning method [3], and HogWild! parallelizes SDG [33]. AQuA is complementary to HogWild!; HogWild! can be viewed as yet another machine learning framework to be retrained when model quality falls below a target threshold or abrupt model drift is detected.

8. CONCLUSION

This paper tackles the problem of how to maintain a high-quality analytics model for contextual event processing when training data keeps changing. We observe that incremental evaluation of machine learning algorithms is usually faster than incremental training. Based on this observation, we propose AQuA, a quality-directed adaptive analytic retraining framework. We evaluate AQuA on two large real-world datasets and three widely-used machine learning algorithms. The results demonstrate that the quality-directed approach strikes a better balance between model quality and training effort than the traditional quality-oblivious approaches, and that a near-optimal tradeoff between model quality and training effort is possible in the presence of either gradual or abrupt model drift.

Acknowledgment

We thank Jian Tan and Li Zhang for suggesting the EMA algorithm to detect abrupt model change. We thank Andrew Palumbo for helping answer Wikipedia dataset vectorization questions in Mahout. We thank Ben Herta for setting up the servers to run the experiments. We thank Jinfeng Yi for helping clarify clustering algorithms. We thank Haibo Chen, Bo Li, Josh Milthorpe, Feng Qin, Alex Tarvo, Junming Xu, Yiqing Yang, Ce Zhang, and the anonymous reviewers for providing constructive feedback for this paper.

9. REFERENCES

[1] T. Al-Khateeb, M. M. Masud, L. Khan, C. C. Aggarwal, J. Han, and B. M. Thuraisingham. Stream classification with recurring and novel class detection using class-based ensemble. In *International Conference on Data Mining (ICDM)*, pages 31–40, 2012.

[2] M. Arnold, D. Grove, B. Herta, M. Hind, M. Hirzel, A. Iyengar, L. Mandel, V. Saraswat, A. Shinnar, J. Siméon, M. Takeuchi, O. Tardieu, and W. Zhang. META: Middleware for events, transactions, and analytics. *IBM Journal of Research and Development*, 60(2–3):15:1–15:10, 2016.

[3] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1995.

[4] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental sliding window analytics. In *International Middleware Conference*, pages 61–72, 2014.

[5] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *Symposium on Cloud Computing (SoCC)*, 2011.

[6] L. Bottou. Online learning and stochastic approximations. *On-Line Learning in Neural Networks*, pages 9–42, 1998.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *Conference on Very Large Data Bases (VLDB)*, pages 285–296, 2010.

[8] Y. Cao and J. Yang. Towards making systems forget with machine unlearning. In *Symposium on Security and Privacy*, pages 463–480, 2015.

[9] F. Chirigati, J. Siméon, M. Hirzel, and J. Freire. Virtual lightweight snapshots for consistent analytics in NoSQL stores. In *International Conference on Data Engineering (ICDE), Industrial Track*, 2016.

[10] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Conference on Neural Information Processing Systems (NIPS)*, 2012.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[12] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144, 2014.

[13] M. Enoki, J. Siméon, H. Horii, and M. Hirzel. Event processing over a distributed JSON store: Design and performance. In *Conference on Web Information System Engineering (WISE)*, pages 395–404, 2014.

[14] S. Esteves, J. a. N. Silva, J. a. P. Carvalho, and L. Veiga. Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets. *Journal of Parallel and Distributed Compututing (JPDC)*, 2015.

[15] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2012.

[16] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[17] Giraph. http://giraph.apache.org/. Retrieved February, 2016.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[19] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *International Conference on Management of Data (SIGMOD)*, pages 328–339, 1995.

[20] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Operating Systems Design and Implementation (OSDI)*, 2010.

[21] S. C. H. Hoi, R. Jin, P. Zhao, and T. Yang. Online multiple kernel classification. *Machine Learning*, 90(2):289–316, 2013.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[23] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, Sept. 2006.

[24] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Symposium on Cloud Computing (SoCC)*, pages 51–62, 2010.

[25] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. In *International Conference on Very Large Data Bases (VLDB)*, volume 5, pages 716–727, Apr. 2012.

[26] Mahout. http://mahout.apache.org/. Retrieved February, 2016.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[28] F. Marvasti. *Nonuniform Sampling: Theory and Practice*. Information Technology. Kluwer, New York, 2001.

[29] M. M. Masud, J. Gao, L. Khan, J. Han, and B. M. Thuraisingham. A multi-partition multi-chunk ensemble technique to classify concept-drifting data streams. In *Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD)*, pages 363–375, 2009.

[30] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[31] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[32] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[33] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Conference on Neural Information Processing Systems (NIPS)*, 2011.

[34] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and

notifications. In *Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2010.

[35] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[36] J. Rennie, L. Shih, J. Teevan, and D. Karger. Tackling the poor assumptions of Naive Bayes text classifiers. In *International Conference on Machine Learning (ICML)*, 2003.

[37] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *International Conference on Management of Data (SIGMOD)*, pages 979–990, 2014.

[38] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs. In *Conference on Very Large Data Bases, Industrial Track*, pages 1736–1747, 2012.

[39] A. Shinnar, J. Siméon, and M. Hirzel. A pattern calculus for rule languages: Expressiveness, compilation, and mechanization. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 542–567, 2015.

[40] Spark MLib. http://spark.apache.org/docs/1.1.1/mllib-guide.html. Retrieved February, 2016.

[41] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. In *Conference on Very Large Data Bases (VLDB)*, pages 702–713, 2015.

[42] L. Torrey and J. Shavlik. Transfer learning. In E. S. Olivas, editor, *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques: Algorithms, Methods, and Techniques*. IGI Global, 2009.

[43] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[44] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. S. Dhillon. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. In *International Conference on Very Large Data Bases (VLDB)*, pages 975–986, Sept. 2014.

[45] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Conference on Algorithmic Aspects in Information and Management (AAIM)*, pages 337–348, 2008.