Polishing the Right Apple: Anytime Classification Also Benefits Data Streams with Constant Arrival Times

Jin Shieh Eamonn Keogh
Dept. of Computer Science & Engineering
University of California, Riverside
{shiehj, eamonn}@cs.ucr.edu

Abstract— **Classification of items taken from data streams requires algorithms that operate in time sensitive and computationally constrained environments. Often, the available time for classification is not known a priori and may change as a consequence of external circumstances. Many traditional algorithms are unable to provide satisfactory performance while supporting the highly variable response times that exemplify such applications. In such contexts, *anytime algorithms*, which are amenable to trading time for accuracy, have been found to be exceptionally useful and constitute an area of increasing research activity. Previous techniques for improving anytime classification have generally been concerned with optimizing the probability of correctly classifying *individual objects*. However, as we shall see, serially optimizing the probability of correctly classifying *individual objects K* times, generally gives inferior results to batch optimizing the probability of correctly classifying *K objects*. In this work, we show that this simple observation can be exploited to improve *overall* classification performance by using an anytime framework to allocate resources among a set of objects buffered from a fast arriving stream. Our ideas are independent of object arrival behavior; and, perhaps unintuitively, even in data streams with constant arrival rates our technique exhibits a marked improvement in performance. The utility of our approach is demonstrated with extensive experimental evaluations conducted on a wide range of diverse datasets.**

Keywords-anytime algorithms; classification; nearest neighbor; streaming data

I. INTRODUCTION

Classification of data arriving from a data stream is often more difficult than the batch situation because the algorithm must operate in a time sensitive and computationally constrained environment. Traditional algorithms are often unable to provide satisfactory performance while supporting the highly variable arrival rates that typify such applications. For example, a single data stream may produce items to be classified at a rate that can range from milliseconds to minutes [1][22]. Traditional classification algorithms typically lack the mechanism for providing an intermediate result prior to completion, and contract-based algorithms require the time duration prior to execution [28]. In such contexts, *anytime algorithms* have been found to be exceptionally useful, and have recently been the subject of extensive research efforts [8][12][16][17][22][27].

Anytime algorithms are algorithms which are amenable to variable response times, by exchanging the quality of response as a function of time [9][28]. In the case of classification, *quality* is measured by the probability of correct classification. More concretely, an anytime algorithm, after a short period of initialization, can always be interrupted to return some intermediate result. This flexibility in response time allows anytime algorithms to be used with great success in real-world environments with variable constraints [8][13][23].

For anytime classification, one well established technique is the anytime nearest neighbor classification algorithm [22]. This algorithm retains the strong points of the nearest neighbor algorithm, its simplicity and generality[1], while greatly mitigating the problem associated with the linear time complexity at classification time, a function of its "lazy" behavior.

Previous techniques for improving anytime classification have generally been concerned with optimizing the probability of correctly classifying *individual objects*. In this work, we show that substantial improvement in overall classification accuracy performance can be achieved if the optimization is performed relative not to each *individual object*, but rather to a (possibly quite small) *set* of objects.

Our technique is a generalized framework which utilizes a scoring function that estimates the intermediate result quality of an object being processed. Here, the *quality* is an estimate of the (relative) likelihood that we have the correct class label for the object. Objects with a high initial quality are unlikely to significantly improve their quality, even with additional computation time. In contrast, objects with poor initial quality have much greater room for improvement, and are thus deserving of more resources. Using this intuition, our framework intelligently and dynamically schedules computational resources for each object. We show that the lack of such inter-object consideration would otherwise result in poor allocation of computation time and lead to reduced performance. As expressed by the well-known idiom, we would be *polishing the wrong apple* if we allocated resources to an instance whose class label is unlikely to change, even with more resources.

Our methodology is invariant to object arrival behavior, and perhaps unintuitively, it is notable in that even with an *uniform* object arrival rate we are capable of attaining a

[1] I.e. The ability to use any distance measure, the ability weight features, etc.

marked improvement in classification performance. This is in contrast to the usual motivation for anytime algorithms, which are typically presented to mitigate the effects of *variable* object arrival behavior [12][22][27].

The remaining sections of the paper are organized as follows: In Section II, we provide background material on anytime algorithms and review related work on classification techniques. We then present an overview of the anytime nearest neighbor classifier in Section III. Section IV motivates and introduces techniques for improving classification accuracy by using a scoring function to measure intermediate result quality and performing computational resource allocation. Section V provides additional details regarding the selection and formulation of a scoring function. In Section VI we verify the utility of our framework with experimental evaluation conducted on a wide range of diverse datasets. Lastly, Section VII offers some discussion and suggests directions for future work.

II. Background and Related Work

Algorithms are considered *anytime* if they exhibit specific characteristics [9][28], notably:

- After a short period of initialization, the algorithm becomes *interruptible*. That is, an intermediate result can be returned at any time up to completion.
- The quality of this result is measurable and improves with additional computation time.
- The change in quality is typically characterized by diminishing returns, with the largest gains found in the initial stages of computation.
- An interrupted execution of the algorithm can later be resumed for additional refinement without significant overhead.

Figure 1 illustrates the prototypical tradeoff between result quality and computation time in an anytime algorithm. Such flexibility is advantageous when available computation time is not known a priori (e.g. in data streams).

Due to their utility in real-world settings, anytime algorithms have been extensively studied [11] and have found application in a number of diverse domains. Applications range from path planning in real time strategy games [3] to the clustering of time series [17]. Significant work has also been done to adapt or view [21] well established machine learning algorithms under the anytime framework. Some examples include Bayesian networks [13], decision trees [10], nearest neighbors [22], and inductive logic programming [18].

For anytime nearest neighbor classification of objects, performance has been shown to be improved by reordering the training set. One technique for generating such an ordering is by repeatedly moving the worst exemplar to the end of the list so that the most characteristic exemplars are examined first [22] [25].

When classifying objects from a data stream it may not be necessary or advantageous to compute classifications serially, where the available computation time for each object is the interarrival time between itself and the next successive object. A more general methodology is to

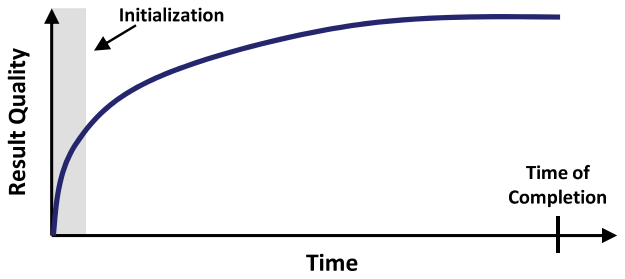

Figure 1. Anytime algorithms are interruptible after initialization. This plot shows the increase in result quality with additional computation time

separate the direct relationship between arrival rate and object computation time. Instead, the available computation time is dictated by some external variable or dependency. This approach is less constrained and allows for the concurrent processing of more than one object. For example, in real-world monitoring scenarios, a typical query may be: "*Monitor object-stream X and event-stream Y* (let λ_X, λ_Y be the arrival rates for *X* and *Y*, respectively, where $\lambda_X >> \lambda_Y$), *classify objects arriving in X, and return their classification upon the next event in Y.*"

Note that if the object processing strategy is serial, then simply optimizing the classification of each object *individually* is clearly the optimal policy. However, if objects are processed concurrently, then it is possible that we can do much better than simply optimizing each object classification in isolation [12][16]. This follows from the observation that in virtually any set of objects, the change in result quality with additional computation time will likely vary greatly between each object. To obtain the greatest increase in performance, we simply need a way to estimate and then process the object(s) that can benefit the most from additional computation. While this appears to be a "chicken-and-egg" paradox (since during classification, we obviously don't know if we have the true class label), as we shall see, at least in the case of the nearest neighbor algorithm, that we can cheaply obtain an approximation of how likely we have the correct label, a *confidence* measure.

In this work, we apply this intuition to present a general framework which can be used to increase the overall performance in data stream classification. Our method uses a "confidence" scoring function to estimate the quality of intermediate results and assigns computation time to objects with the lowest classification confidence. We demonstrate in our experimental evaluation that once we have a reasonably accurate scoring function, we can improve overall classification performance. Note that classification remains under anytime conditions and that the end times for objects are never known a priori.

We have chosen to explore our observations with the nearest neighbor algorithm. The nearest neighbor algorithm has proven to be one of the most frequently deployed classifiers, with accuracy which is competitive with other techniques [22] [26]. The nearest neighbor algorithm has the advantage of being non-parametric, capable of handling a large number of classes, and easily adaptable to datasets

which are dynamically changed or updated without retraining or overfitting. Its primary disadvantage is the linear time dependence on the number of training exemplars, a property which can be mitigated by indexing the data (when applicable), or with anytime algorithms such as the one presented in [22]. While the work in [22] allows the algorithm to be used in streams where *one* object may arrive before the current object has completed processing, it does not leverage the idea of varying computational resources according to result quality when classifying a *set* of concurrent objects.

III. Anytime Nearest Neighbor Classification

The nearest neighbor algorithm, well known for its utility and range of applicability, is easily extended to fit under the anytime framework. This section presents a review of the anytime nearest neighbor classification (ANNC) algorithm [22]. In Section IV, we will discuss how to generalize ANNC for concurrent classification.

For clarity, the notation used in the following sections is first presented in TABLE I.

TABLE I List of Notation

Name	Description
$\boldsymbol{q}$	An object to classify. Contains the following fields:
$\boldsymbol{q.pos}$	Current training set position
$\boldsymbol{q.class}$	Current classification label
$\boldsymbol{q.dist}$	Current nearest neighbor distance
$\boldsymbol{q.stopped}$	Flag denoting user stoppage
$\boldsymbol{Q}$	A set of objects to classify
$\boldsymbol{Q_i}$	$\boldsymbol{Q_i} \in \boldsymbol{Q}$. Equivalent to $\boldsymbol{q}$ above
$\boldsymbol{Q'}$	A set of objects to classify, which are concurrent and in memory
$\boldsymbol{D}$	The set of training objects
$\boldsymbol{D_i}$	$\boldsymbol{D_i} \in \boldsymbol{D}$
$\boldsymbol{D_i.class}$	The class label of this object
$\boldsymbol{M}$	Number of objects which can be buffered in memory
NumClasses($\boldsymbol{D}$)	Returns the number of unique class labels represented within training set $\boldsymbol{D}$
ScoreFcn($\boldsymbol{q}$)	A function that returns a score which estimates the intermediate result quality of object $\boldsymbol{q}$
Distance(,)	A context appropriate distance measure

Given an object to classify, $\boldsymbol{q}$, and a set of training instances, $\boldsymbol{D}$, the ANNC algorithm finds the entry $\boldsymbol{D_i}$ in $\boldsymbol{D}$ which minimizes *Distance*($\boldsymbol{q}$, $\boldsymbol{D_j}$). That is:

$$\forall D_j \in D, Distance(q, D_i) \leq Distance(q, D_j)$$

The returned classification is the corresponding class label for $\boldsymbol{D_i}$, $\boldsymbol{D_i.class}$.

Note that we have not explicitly defined the *Distance* measure used in ANNC. The ANNC can use any distance measure (L^p-Norm, Hamming distance, graph edit distance, Dynamic Time Warping, etc.) that is appropriate to a specific context (time series, strings, graphs, categorical data, etc.). If only a *similarity* measure is available, i.e. the cosine similarity measure, we can simply define the 'distance' as the reciprocal of the similarity measure.

A sketch of the ANNC algorithm is shown in TABLE II. Lines 1-8 initialize $\boldsymbol{q}$ to an initial result. First, let ***NumClasses(D)*** be the number of unique class labels in $\boldsymbol{D}$, where the first ***NumClasses(D)*** instances of $\boldsymbol{D}$ contain exactly one exemplar from each class label. Then, the initialization period does the following: $\boldsymbol{q}$ is iterated over the first exemplar from each class and an initial classification is obtained and saved as the intermediate result. While the algorithm cannot be interrupted during this period, it *only* has to iterate ***NumClasses(D)*** times, where ***NumClasses(D)*** << $|\boldsymbol{D}|$ and thus the time duration spent in initialization is marginal.

Following initialization, the object being classified can be stopped, then resumed at any time leading up to the completion of training set evaluation. Lines 9-14 iterate through the remaining training instances in $\boldsymbol{D}$ or until stopped and update the nearest neighbor for $\boldsymbol{q}$ accordingly. $\boldsymbol{q.class}$ is then returned as the classification result for $\boldsymbol{q}$.

The generic algorithm just presented falls under the anytime framework and achieves increased quality of results (classification confidence) as a function of additional time.

TABLE II Anytime Nearest Neighbor Classifier

	Function *AnytimeClassifier*($\boldsymbol{q}$, $\boldsymbol{D}$)
1	$\boldsymbol{q.dist} \leftarrow \infty$
2	$\boldsymbol{q.class} \leftarrow$ null
3	**for** $i \leftarrow 1$ to ***NumClasses(D)***
4	Dist ← *Distance*($\boldsymbol{q}$, $\boldsymbol{D_i}$)
5	**if** Dist < $\boldsymbol{q.dist}$
6	$\boldsymbol{q.dist} \leftarrow$ Dist
7	$\boldsymbol{q.class} \leftarrow \boldsymbol{D_i.class}$
8	$\boldsymbol{q.pos} \leftarrow$ ***NumClasses(D)***
9	**while** !$\boldsymbol{q.stopped}$ and $\boldsymbol{q.pos} < |\boldsymbol{D}|$
10	$\boldsymbol{q.pos} \leftarrow \boldsymbol{q.pos} + 1$
11	Dist ← *Distance*($\boldsymbol{q}$, $\boldsymbol{D_{q.pos}}$)
12	**if** Dist < $\boldsymbol{q.dist}$
13	$\boldsymbol{q.dist} \leftarrow$ Dist
14	$\boldsymbol{q.class} \leftarrow \boldsymbol{D_{q.pos}.class}$

This, in essence, is the current methodology for ANNC, introduced in [22], except that work also suggests optimizing individual object classification by identifying heuristics for ordering the training set entries so that the most characteristic exemplars are examined early on. This simple idea is useful enough to have found real-world applications; for example, it is used in a surveillance system created by Toshiba [24].

In this work, we adopt the complementary methodology of optimizing performance across a set of objects. In the following sections, the motivation and advantages behind such a method are presented and techniques for improving overall classification performance are introduced.

IV. Concurrent Object Evaluation

For the purpose of explanation, it was convenient to illustrate anytime classification with regards to a single object. However, streaming data classification is more

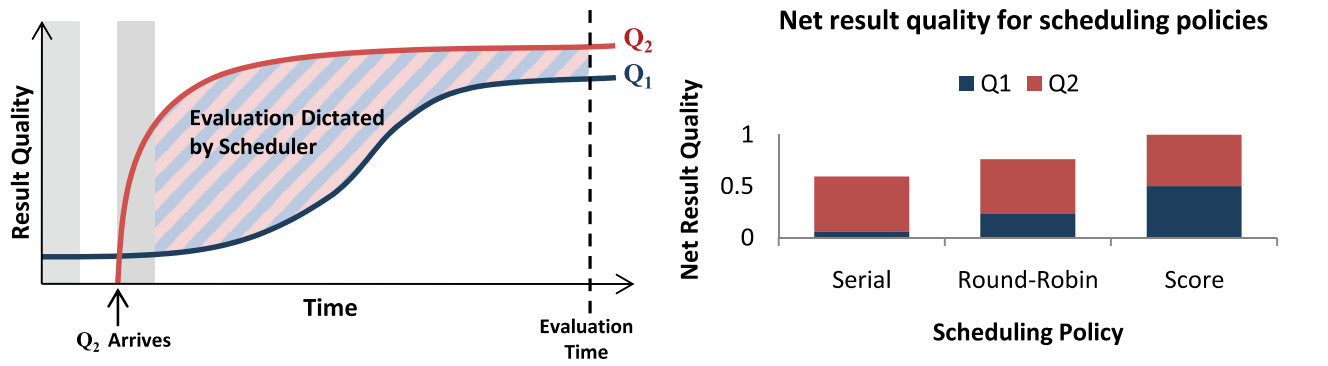

Figure 2. Net result quality across concurrent queries for various scheduling policies. *Left*) Result quality over time for two queries Q_1 and Q_2 (note that Q_2 arrives shortly after Q_1). *Right*) Net result quality (Q_1 and Q_2) for various scheduling policies at evaluation time (hatched line)

accurately exemplified by a *sequence* of objects. Given this consideration, our work examines the set of concurrently processed objects and optimizes the scheduling of computational resources towards maximizing overall classification performance. Such evaluation is significant in that simply optimizing individual object classification can result in poor *overall* computational resource allocation and performance. For expository purposes, consider a simple example:

Suppose we have a set of objects to concurrently classify. After some period of initial processing, in all but the most pathological cases, we would expect that this set of objects will contain intermediate results which span some range in quality. As a simple example, suppose we have a database of ten million objects, consisting of two classes of automobiles, Japanese and American, and we have a pool of instances to classify: {1995 Toyota Corolla, 2000 Ford Escort, 1998 Honda Civic,..., 1957 Hudson Hornet[2]}. Because the Toyota Corolla is the bestselling car in the world, once we have examined just the first 100 objects in the training database we are practically guaranteed to have seen several examples of Corollas. There is, therefore, little utility in comparing this instance to the rest of the database. In contrast, the Hornet is so rare, and so unlike other American cars, that it is very unlikely to have been encountered in the first 100 items visited. Its current nearest neighbor is as likely to be Japanese as American.

Scheduling policies which do not take into account the diversity in the set of currently processing objects will almost certainly result in suboptimal resource allocation by scheduling computation time to objects which already have excellent intermediate results (i.e. Toyota Corollas). This is time that can be better spent on objects which may benefit most from additional computation. However, we do not know the true class labels of the objects, and therefore how can we know the likelihood that the *tentative* label is correct? Our task resembles a well observed problem found in nature, and our solution is also very similar to the solution that has evolved in nature.

Many birds and small mammals have large broods, and the parents must allocate food resources among them [19]. From the point of view of the parent who is trying to maximize her (less often, "his") reproductive success, the optimal thing to do is to feed each offspring equally. However, birds cannot differentiate between their offspring, so they cannot use any algorithm that requires labeling their young. If their algorithm is to feed the most aggressive young (a literally *greedy* algorithm), then the next time they return with food, that aggressive offspring, fortified by recent feeding, will be able to force itself to the front again and beg for more food [20]. This will continue indefinitely until the weaker siblings starve[3].

The solution to this problem is that the young signal their hunger level to the parent by the frequency of chirping. The parent's optimal algorithm is reduced to feeding the young that signals the greatest need. This solution is nearly universal among birds, and may have evolved independently in different species [20].

We can see that our problem is nearly identical. We have resources (CPU time) that we must distribute among objects that have possibly varying levels of need (i.e. varying levels of confidence in their classification). As we shall show in the following section, our solution is nearly identical; we simply need to have each object "*signal*" its need, based on an estimate of how confident it feels about its current label. Of course, we can never know exactly the probability of an object having the correct classification; however, as we shall see in Section VI, a simple heuristic is sufficient to produce improvements in overall accuracy.

A. *Scheduling Policies*

If a data stream is sparse, then objects are processed in isolation and without interference from other objects (in effect, complete serial processing). In such scenarios, the best performance can be achieved by utilizing any of the previous approaches which optimize for single objects. However, in most data streams we can expect to encounter at

[2] A rare American-made automobile.

[3] We know that this would happen, because it *does* happen in cases of brood-parasites such as the cuckoos

some point a set of objects which have overlapping lifetimes. Thus, given a data stream which contains a set of $\boldsymbol{Q}$ objects distributed over the duration of the stream, at any given time within the lifetime of the data stream, there exists some subset, $\boldsymbol{Q'}$ of $\boldsymbol{Q}$, consisting of $|\boldsymbol{Q'}|$ number of objects being classified, $|\boldsymbol{Q}| \geq |\boldsymbol{Q'}| \geq 0$. If the number of objects is neither zero nor singular ($|\boldsymbol{Q'}| > 1$), and because only one object may be processed at a time, what is the scheduling approach that attains the best overall performance over $\boldsymbol{Q'}$?

Recall that the end time of each object is never known a priori. Consequently, it is not possible to identify a global optimum, over $\boldsymbol{Q}$, beforehand. However, we *can* optimize locally, over $\boldsymbol{Q'}$, to improve performance. Let us first introduce an illustrative example, consisting of two objects $\boldsymbol{Q_1}$ and $\boldsymbol{Q_2}$, $\boldsymbol{Q'} = \{\boldsymbol{Q_1}, \boldsymbol{Q_2}\}$. Assume $\boldsymbol{Q_2}$ arrives shortly after $\boldsymbol{Q_1}$ and that the quality of result as a function of computation time for each of the objects is known (as shown in Figure 2 *Left*).

One scheduling technique is the sequential or serial scheduling of objects. That is, we classify each object, $\boldsymbol{Q_i}$, using the ANNC algorithm described in Section III and return a response when complete or upon the next arriving object. Under serial scheduling, $\boldsymbol{Q_1}$ is stopped prematurely as a result of $\boldsymbol{Q_2}$'s arrival. From the result quality over time plot in Figure 2 *Left* we can see that $\boldsymbol{Q_2}$ obtains a high quality result immediately following initialization, whereas $\boldsymbol{Q_1}$ requires additional computation time to reach a higher level in quality. Due to the sequential nature of this scheduling algorithm, a clearly suboptimal result is obtained (see Serial Scheduling Policy in Figure 2 *Right)*. The bulk of computation time is devoted to $\boldsymbol{Q_2}$, resulting in a marginal improvement in net quality. Notice that a much larger increase in overall quality could have been achieved by continuing to schedule $\boldsymbol{Q_1}$ even after the arrival of $\boldsymbol{Q_2}$.

Another scheduling policy is to divide computation time equally among concurrently processing objects in a round robin fashion. The result shown in Figure 2 *Right* is an improvement over serial scheduling; however, the intuition established in the previous solution is as follows: since $\boldsymbol{Q_2}$ is able to obtain a high quality result quickly and $\boldsymbol{Q_1}$'s quality remains poor, computation time is still best spent on classifying $\boldsymbol{Q_1}$.

Let us now see how we can achieve even better results than round robin scheduling. Our intuition dictates that we should refrain from wasting computational resources on an object when another object exists whose quality is lower. Observe that such a scheme can be achieved if we have a scoring function which returns an indicator of intermediate result quality per object. Then the scheduling policy is simply to schedule the object with the lowest score. For example, if we use the plots in Figure 2 *Left* as our scoring mechanism and schedule the computation accordingly, the net result quality is shown in Figure 2 *Right* and is clearly the best of the three presented techniques.

While it is impossible to have prior knowledge which gives the exact change in quality over time, the score scheduled technique remains highly effective if scoring functions which are a close estimator of result quality exists. We leave the discussion of scoring functions to Section V; first, the score scheduling algorithm is presented.

B. Batch Evaluation

To introduce the use of a scoring function, we first consider the slightly simplified task of classifying a set of objects in batch fashion. In the next section, extensions necessary for a streaming environment will be presented. The batch score scheduled algorithm is shown in TABLE V. Note that the segments of code devoted to the initialization and updating of objects have been summarized in TABLE III and TABLE IV, respectively.

The batch algorithm begins by initializing each object in the input set (Lines 1-2). A priority queue containing the entire set of objects is then created (Line 3). The priority queue ordering is dictated by the scoring function, *ScoreFcn* which provides an estimate of the quality of an object's intermediate result. At each iteration, the object with the lowest score (lowest classification confidence) is scheduled for computation and updated (Lines 4-8).

TABLE III INITIALIZING OBJECT CLASSIFICATION

	Function *Initialize*($\boldsymbol{q}$, $\boldsymbol{D}$)
1	$\boldsymbol{q.dist} \leftarrow \infty$
2	$\boldsymbol{q.class} \leftarrow$ null
3	**for** $i \leftarrow 1$ to *NumClasses*($\boldsymbol{D}$)
4	Dist $\leftarrow$ *Distance*($\boldsymbol{q}$, $\boldsymbol{D_i}$)
5	**if** Dist < $\boldsymbol{q.dist}$
6	$\boldsymbol{q.dist} \leftarrow$ Dist
7	$\boldsymbol{q.class} \leftarrow \boldsymbol{D_i.class}$
8	$\boldsymbol{q.pos} \leftarrow$ *NumClasses*($\boldsymbol{D}$)

TABLE IV UPDATING OBJECT CLASSIFICATION

	Function *Update*($\boldsymbol{q}$, $\boldsymbol{D}$)
1	**if** !$\boldsymbol{q.stopped}$
2	$\boldsymbol{q.pos} \leftarrow \boldsymbol{q.pos} + 1$
3	Dist $\leftarrow$ *Distance*($\boldsymbol{q}$, $\boldsymbol{D_{q.pos}}$)
4	**if** Dist < $\boldsymbol{q.dist}$
5	$\boldsymbol{q.dist} \leftarrow$ Dist
6	$\boldsymbol{q.class} \leftarrow \boldsymbol{D_{q.pos}.class}$

TABLE V BATCH SCORE SCHEDULED CLASSIFIER

	Function *BatchScoreScheduledAnytimeClassifier* ($\boldsymbol{Q}$, $\boldsymbol{D}$, *ScoreFcn*)
1	**for** $i \leftarrow 1$ to $\|\boldsymbol{Q}\|$
2	*Initalize*($\boldsymbol{Q_i}$, $\boldsymbol{D}$)
3	*PriorityQueue* Queue($\boldsymbol{Q}$, *ScoreFcn*)
4	**while** Queue.*Size* > 0
5	$\boldsymbol{q} \leftarrow$ Queue.*RemoveMin*()
6	*Update*($\boldsymbol{q}$, $\boldsymbol{D}$)
7	**if** !$\boldsymbol{q.stopped}$ and $\boldsymbol{q.pos} < |\boldsymbol{D}|$
8	Queue.*Add*($\boldsymbol{q}$)

Computational time for initialization is $O(C\ Numclasses(\boldsymbol{D}))$ per query, with C as the cost per *Distance*(,) invocation. The priority queue is initialized in $O(S\ |\boldsymbol{Q}|)$ time and the cost for each iteration of the score scheduled computation is at most $O(log|\boldsymbol{Q'}|)$ to *RemoveMin*( ) and $O(C+S+log|\boldsymbol{Q'}|)$ for the re-

insertion/update, given that ***Q'*** is the number of currently processing queries, *S* is the cost per *ScoreFcn*( ) invocation, when utilizing a standard heap-based priority queue. We assume that objects can be asynchronously stopped and can be lazily removed from the priority queue or in the case with limited memory, purged at a cost of $O(log|\boldsymbol{Q}'|)$.

Given our intuition that the distribution of intermediate result quality across a set of concurrently processing objects is skewed, the number of scheduling iterations (and its requisite overhead) can be significantly reduced by simply tracking the confidence score of the next minimum item in the queue, *T*, and performing computations on the current object until stopped or its confidence score exceeds *T*. Another technique for reducing framework overhead would be to schedule computational resources at a coarser granularity, that is, to increase the resources allocated per iteration.

C. Streaming Evaluation

Extensions to the batch algorithm are necessary to make the score scheduled algorithm suitable for a streaming context. More concretely, we must incorporate the consideration of the online arrival of objects and a finite memory for buffering of concurrent objects. Our complete framework is presented in TABLE VI.

TABLE VI STREAMING SCORE SCHEDULED CLASSIFIER

	Function *StreamingScoreScheduledAnytimeClassifier* (***D***, *ScoreFcn*, ***M***)
1	*PriorityQueue* Queue(*ScoreFcn*)
2	**while** *ContinueClassification*
3	**if** *NewObject*
4	***q*** ← *GetNewObject*()
5	*Initalize*(***q***)
6	**if** Queue.*Size* > ***M***-1
7	Queue.*RemoveMax*()
8	Queue.*Add*(***q***)
9	**else**
10	***q*** ← Queue.*RemoveMin*()
11	*Update*(***q***, ***D***)
12	**if** !***q.stopped*** and ***q.pos*** < |***D***|
13	Queue.*Add*(***q***)

In Line 1, we initialize an empty priority queue and set the ordering of objects to be dictated by the scoring function, *ScoreFcn*. Line 2 examines a user-updateable flag to determine if this online algorithm should continue. The previous batch algorithm was able to simply check queue size for termination; however, an empty queue is not similarly indicative in an online, streaming environment (consider the case where bursts of queries are separated by long interarrival times, resulting in empty queues). The algorithm then checks to see if a new object has arrived and fetches it accordingly (Lines 3-4). Incoming objects are initialized immediately (Line 5) and inserted into the queue (Line 8). In the case that the current queue size plus the new object will exceed the maximum buffer size, ***M***, the object with the largest score, is stopped and evicted (Lines 6-7). This is the in-memory object with the highest classification confidence. Lines 10-13 schedule the minimum scored object for processing and are identical to the batch algorithm. The computational time is the same as the batch algorithm, with the addition that the priority queue is empty at initialization and must also support the *RemoveMax*( ) operation, which has $O(log|\boldsymbol{Q}'|)$ time complexity.

V. SCORING FUNCTION

Recall that the scoring function estimates intermediate result quality. In a classification context, a high score would imply that the current classification label is unlikely to change, even if given sufficient time to run until completion. Conversely, a low score is characteristic of an object whose classification is likely to change.

Given that the nearest neighbor classifier seeks to find the entry in the training set with the minimum distance, one simple scoring method is to use the current best-so-far nearest neighbor distance as the estimate for classification confidence. The distance value is inverted so that higher distances correspond to lower scores:

$$Score_{BsfDistance}(q) = -min(Distance(q, D_i)) \quad 1 \le i \le q.pos$$
$$= -q.dist$$

Note that we are not claiming that this scoring method is the optimal one; our claim is merely that it is empirically successful for many datasets. The framework is agnostic to the scoring method, and a user with context specific knowledge can formulate and tune a custom scoring function accordingly. Furthermore, this generality allows for applicability even in settings where the raw data may not be directly accessible. For an example, one can imagine a black box scenario where we only need to observe distances between objects and not require direct access to the objects themselves.

For comparative purposes, we will use the round robin scheduling policy as a competitive baseline. As shown by the example illustrated in Figure 2, round robin can offer a significant improvement over serial scheduling. Furthermore, and in contrast to the $Score_{BsfDistance}$ method, round robin has the advantage of being starvation free. This property can mitigate the adverse effects of outliers and prevent otherwise a potential monopolization of computational resources.

VI. EXPERIMENTAL RESULTS

In this section, we examine the utility of our score scheduled anytime nearest neighbor classifier by conducting experimental evaluation of a wide range of diverse classification datasets [2][15]. A list of the datasets used in our experiments and their attributes are shown in TABLE VII. Note that the data and code used in this work is archived at [29], with annotations to allow reproduction of results.

For all datasets, we obtained testing and training splits using 10-fold cross validation. The training exemplar order for each fold is randomly permuted and all features are used. Note that the training set invariant, where one exemplar from each class is encountered first (during initialization), is preserved.

Our dataset evaluation uses the Euclidean distance as the distance function. An exception is the commercial entomology case study presented in Section VI.D. There, we use a compression based distance measure which has been shown to have utility for differentiating textures [5].

Recall that given objects q and d, each with n-dimensions the Euclidean distance is:

$$EuclideanDistance(q,d) \equiv \sqrt{\sum_{i=1}^{n}(q_i - d_i)^2}$$

While Euclidean distance may not be the optimal distance measure for every dataset, it has been shown to be very competitive across many domains [7][14]. As our primary objective is simply to show the *improvement* as a result of using score scheduling to allocate computational resources, the selection of Euclidean distance as the distance measure is appropriate.

TABLE VII DATASETS USED FOR EXPERIMENTAL EVALUATION

Name	Classes	Attributes	Instances
Two-Pattern	4	128	5,000
AIBO Robot	2	100	12,100
Gun	2	150	200
Face	16	131	2,231
Leaf	6	150	442
CBF	3	128	1,000
Moth	35	Image (~500x800)	772
JF	2	2	20,000
Letter	26	16	20,000
Pen Digits	10	16	10,992
Ionosphere	2	32	351

A. *Classification of Streaming Data*

To evaluate the utility of score scheduled classification, we simulate the classification of data streams with varying rates of arrival. Our experimental data stream exhibits the characteristic of constant or uniform arrival between successive objects, and the exact interarrival time between each object is modeled as the number of training set exemplars ($|\boldsymbol{D}|$) which can be evaluated:

$$InterArrivalTime(r) = \lfloor |D| r \rfloor \quad 0.1 \le r \le 1$$

The arrival rate is modeled as a function of $|\boldsymbol{D}|$ on an account of its generality across all datasets. This is in contrast to concrete numerical values (e.g. the data stream operates at 100Hz) which may not always be applicable or meaningful (due to the wide variability in dataset characteristics: number of classes, feature space, exemplars available).

Objects from the testing set, $\boldsymbol{Q}$, enter the data stream in accordance with the interarrival time until exhausted. For r = 1, the interarrival time between successive objects is exactly the time needed to evaluate the entire training set and thus is equivalent to complete serial classification. Our experiment concludes when mean interarrival time has elapsed following the arrival of the last object from the test set.

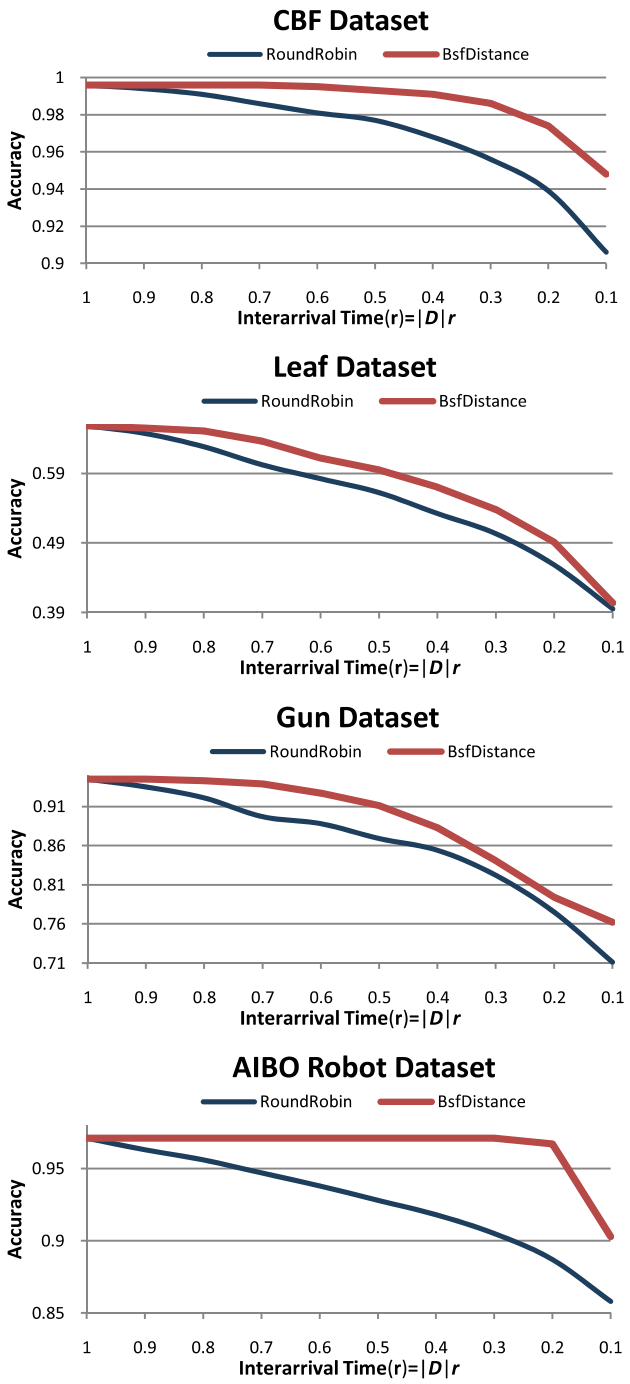

Figure 3. Classification accuracy of score scheduled anytime classifier on constant data streams with varying rates of arrival

Classification accuracy is computed from the predicted test labels and the true test labels. Note that this experimental setup obtains classification accuracies which are dependent on the order of arrival from the testing set. To remove such bias, we average the classification accuracy for each

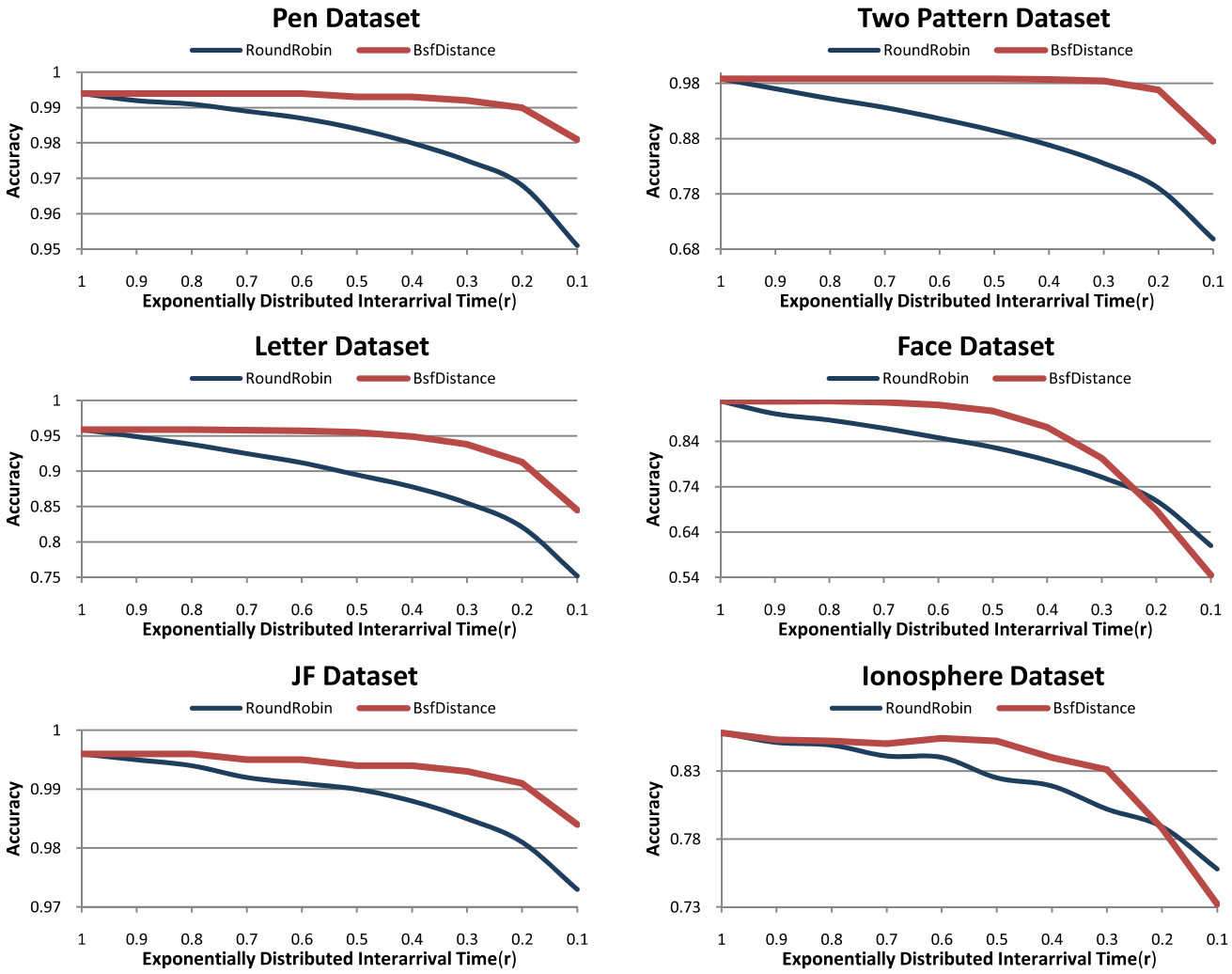

Figure 4. Classification accuracy on data streams with exponentially distributed interarrival times

testing/training split over 10 random permutations of the testing set. For this experiment, we assume that the cardinality of objects being concurrently classified, $|\boldsymbol{Q}'|$, can be accommodated by the memory buffer ($\boldsymbol{M} > |\boldsymbol{Q}'|$).

As a general rule, we expect accuracy to decrease as the arrival rate increases. This behavior can be attributed to the reduced available computation time on average per object for faster streams. However, we have occasionally observed higher accuracy with increased arrival rate, beyond an expected variability. This phenomenon is often caused by non-separable classes or the presence outliers/noisy data and resulting in a scenario where intermediate results have the correct classification but the final nearest neighbor is of a different class.

The classification accuracies from our score scheduled approach on a variety of datasets are shown in Figure 3. From the results, we see that we are typically able to obtain an increase in accuracy over the round robin baseline. This confirms our intuition that $Score_{BsfDistance}$ is a good indicator of result quality by allocating additional computational resources to objects which need it more.

Overall, round robin is a fairly competitive baseline. This can be expected, as prior work [22] has shown that many datasets have query objects which follow the prototypical result quality over time behavior depicted in Figure 1. That is, even evaluating just a small portion of the training set can obtain a high quality result quickly, with additional evaluation characterized by diminishing returns.

B. *Effects of Constrained Memory on Classification Accuracy*

Performance degradation can occur when objects are stopped prematurely and evicted from the buffer as a result of memory constraints. As the score for each object is an estimate of how confident we are about its class label, it is a principled way of determining the eviction policy when encountering memory constraints. That is, we simply evict the object with the highest confidence score. For the round robin baseline, a randomly chosen object is evicted. We conducted experiments which varied the available memory buffer size, $\boldsymbol{M}$, from $|\boldsymbol{Q}|$ to $0.05*|\boldsymbol{Q}|$. From the four datasets we evaluated for constant streams, the change in accuracy was negligible. For each dataset the net change in accuracy

per arrival rate was 1 percent or less. This indicates that our methodology succeeds in evicting objects which are most likely to have their true class label.

C. *Streams with Non-Uniform Arrival*

Data streams are often modeled with non-uniform interarrival times. In this experiment, we show the accuracy of score scheduled classification on such streams. We simulate a data stream with an arrival process which is modeled to be Poisson distributed with mean interarrival times matching the constant streams presented in Section VI.A. The arrival rates are:

$$\lambda = \left\{ \frac{1}{\lfloor |D| r \rfloor} \ , \ 0.1 \le r \le 1 \right\}$$

Figure 4 shows the classification accuracy for exponentially distributed interarrival times, computed as a function of *r*. As shown on the Two-Pattern dataset, we are able to obtain a definite increase in accuracy over the round robin baseline. For the Face and Ionosphere dataset, we see that our scheduling technique is able to improve performance until r < [0.2-0.3], upon which there is insufficient computation time to accurately discern a meaningful and differentiating confidence value. Round robin outperforms in this scenario because it is fair for all entries in memory. Similar results are seen across the remaining datasets.

Overall, while round robin again performs competitively, our score based scheduling is able to obtain a definitive increase in overall accuracy for almost all values in the parameter set.

D. *A Case Study in Commercial Entomology*

Several species of moths are harmful to agriculture. For example, *Epiphyas postvittana*, the Light Brown Apple Moth (LBAM) have larvae that feed on leaves and buds of plants, reducing photosynthetic rate, which in turn leads to general weakness and disfigurement.

In grapes and citrus, LBAM larvae can feed directly on the fruit, and the resulting damage renders fruit unmarketable. The LBAM is native to Australia, but appeared in California in 2007. Since that time, the California Department of Food and Agriculture has spent $70 million on attempts to eradicate it from California. If not eradicated, it is estimated it could cause $140 million in damage each year [6].

Of course, attempts at eradication must be very careful; many moths are important pollinators of plants. For example, the Yucca moth (*Tegeticula maculata*) is the only animal that is the right size and shape to pollinate yucca flowers. If it is accidentally eradicated along with the LBAM, yucca flowers would be threatened, which could further affect additional fauna. Note that the LBAM is just one of the hundreds of insects which are known to be harmful to agriculture, livestock, or humans.

With this in mind, several companies, including ISCA Technologies of Riverside CA, are developing AVIDs, Automated Visual Identification Devices, which can recognize individual species or genera. Most of these systems currently just count the target insect, however systems are being developed that selectively trap or kill only the target insect, and release all others. In order to be effective AVIDs must be cheaply mass produced, and therefore have limited computational resources.

Figure 5. *Left*) The adult Light Brown Apple Moth is harmless to agriculture, however it's larval form. *Right*) causes extensive damage to several commercially important crops

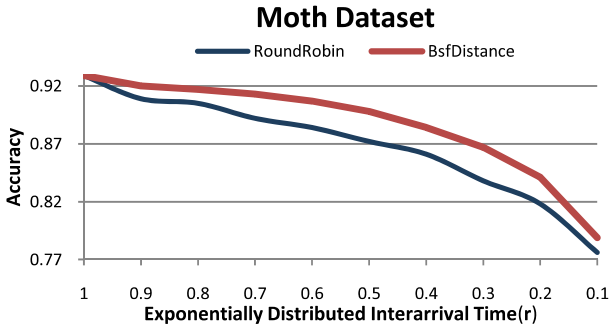

Figure 6. Insect classification with memory buffer constrained to four objects

Recent work has shown that it is possible to accurately classify moths using a compression-based distance measure [5]. The distance measure is effective, but not being a metric it does not allow an efficient indexing mechanism to make classification more tractable. Below we describe our initial experiments to port the compression-based distance measure to resource limited hardware using our anytime framework.

As a preprocessing step, we first cluster the original moth data into three dominant clusters to obtain more exemplars per grouping. We then simulate non-uniform insect arrival [4] using the methodology described in Section VI.C. Due to resource constraints inherent in our target environment, we set the available memory size to 5 percent of the testing set, resulting in a memory buffer of only four objects. Our classification accuracy compared to round robin for different arrival rates is shown in Figure 5. We see that the score scheduled approach consistently out performs round robin.

VII. Conclusion and Future Work

In this work, we presented a generalized framework which utilizes a scoring function that estimates the intermediate result quality of an object being classified. Our framework extends existing anytime algorithms to a *set* of

concurrently processing objects by dynamically scheduling computational resources for each object (in accordance with its score). We showed over a wide range of diverse datasets that the lack of such inter-object consideration would otherwise result in poor allocation of computation time and lead to reduced performance.

As future work, we look forward to examining more in-depth, the utility of this framework and considering the interplay between variability in object duration, amount of concurrency, and different scoring methods. Additional real world case studies would also reinforce the wide range applicability of our framework.

ACKNOWLEDGMENT

This work was funded by NSF awards 0803410 and 0808770. We would like to thank Anna Watson and Michael Mayo for the moth dataset.

REFERENCES

[1] Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. 2004. On demand classification of data streams. In *Proceedings of the Tenth ACM SIGKDD international Conference on Knowledge Discovery and Data Mining* (Seattle, WA, USA, August 22 - 25, 2004). KDD '04. ACM, New York, NY, 503-508.

[2] Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository [http://www.ics.uci.edu/~mlearn/MLRepository.html].

[3] Butt, R. and Johansson, S. J. 2009. Where do we go now?: Anytime algorithms for path planning. In *Proc. of the 4th international Conference on Foundations of Digital Games* (Orlando, Florida, April 26 - 30, 2009). FDG '09. ACM, New York, NY, 248-255.

[4] Byers, J.A. 1996. Temporal clumping of bark beetle arrival at pheromone traps: Modeling anemotaxis in chaotic plumes. J. Chem. Ecol. 22:2143-2165.

[5] Campana, B. and Keogh, E. 2010. A Compression Based Distance Measure for Texture. In *Proceedings of the SIAM International Conference on Data Mining*, SDM 2010.

[6] CISR: Light Brown Apple Moth. Retrieved Jan 15, 2010, from http://cisr.ucr.edu/light_brown_apple_moth.html.

[7] Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., and Keogh, E. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1542-1552.

[8] Esmeir, S. and Markovitch, S. 2005. Interruptible anytime algorithms for iterative improvement of decision trees. In *Proc. of the 1st international Workshop on Utility-Based Data Mining* (Chicago, Illinois, August 21 - 21, 2005). UBDM '05. ACM, New York, NY, 78-85.

[9] Grass, J. and Zilberstein, S. 1995 *Anytime Algorithm Development Tools*. Technical Report. UMI Order Number: UM-CS-1995-094., University of Massachusetts.

[10] Grumberg, O., Livne, S., and Markovitch, S. 2003. Learning to order BDD variables in verification. *Journal of Artificial Intelligence Research.*

[11] Hansen, E. A. and Zilberstein, S. 1996. Monitoring anytime algorithms. *SIGART* Bull. 7, 2 (Apr. 1996), 28-33.

[12] Hui, B., Yang, Y., and Webb, G. I. 2009. Anytime classification for a pool of instances. *Mach. Learn*. 77, 1 (Oct. 2009), 61-102.

[13] Hulten, G. and Domingos, P. 2002. Mining complex models from arbitrarily large databases in constant time. In *Proc. of the Eighth ACM SIGKDD international Conference on Knowledge Discovery and Data Mining* (Edmonton, Alberta, Canada, July 23 - 26, 2002). KDD '02. ACM, New York, NY, 525-531.

[14] Keogh, E. and Kasetty, S. 2002. On the need for time series data mining benchmarks: a survey and empirical demonstration. In *Proceedings of the Eighth ACM SIGKDD international Conference on Knowledge Discovery and Data Mining* (Edmonton, Alberta, Canada, July 23 - 26, 2002). KDD '02. ACM, New York, NY, 102-111.

[15] Keogh, E., Xi, X., Wei, L. & Ratanamahatana, C. A. (2006). The UCR Time Series Classification/Clustering Homepage: www.cs.ucr.edu/~eamonn/time_series_data/

[16] Kranen, P. and Seidl, T. 2009. Harnessing the Strengths of Anytime Algorithms for Constant Data Streams. In *Proceedings of Conference on Machine Learning and Knowledge Discovery in Databases:* 31-31.

[17] Lin, J., Vlachos, M., Keogh, E., and Gunopulos, D. 2004. Iterative incremental clustering of time series. EDBT '04.

[18] Lindgren., T. 2000. Anytime inductive logic programming. In *Proc. of the 15th International Conference on Computers and Their* Applications. 439-442.

[19] MacNair, M.R. and Parker, G.A. 1979. Models of parent–offspring conflict. III. Intra-brood conflict. Anim. Behav., 27: 1202–1209.

[20] Manser, M.B., and G. Avey. 2000. The effect of pup vocalisations on food allocation in a cooperative mammal, the meerkat (Suricata suricatta). Behavioral Ecology and Sociobiology 48(November):429.

[21] Roy, N. and McCallum, A. 2001. Toward Optimal Active Learning through Sampling Estimation of Error Reduction. In *Proc. of the Eighteenth international Conference on Machine Learning* (June 28 - July 01, 2001). C. E. Brodley and A. P. Danyluk, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 441-448.

[22] Ueno, K., Xi, X., Keogh, E., and Lee, D. 2006. Anytime Classification Using the Nearest Neighbor Algorithm with Applications to Stream Mining. In *Proc. of the Sixth international Conference on Data Mining* (December 18 - 22, 2006). ICDM.

[23] Seidl, T., Assent, I., Kranen, P., Krieger, R., and Herrmann, J. 2009. Indexing density models for incremental learning and anytime classification on data streams. EDBT '09, vol. 360. ACM, New York, NY, 311-322.

[24] Toyoshima, I. 2008. Surveillance system, surveillance method and computer readable medium. U.S. Patent Application 20,090,322,875.

[25] Wilson, D. R. and Martinez, T. R. 2000. Reduction Techniques for Instance-Based Learning Algorithms. *Mach. Learn.* 38, 3 (Mar. 2000), 257-286.

[26] Xi, X., Keogh, E., Shelton, C., Wei, L., and Ratanamahatana, C. A. 2006. Fast time series classification using numerosity reduction. In *Proceedings of the 23rd international Conference on Machine Learning* (Pittsburgh, Pennsylvania, June 25 - 29, 2006). ICML '06, vol. 148. ACM, New York, NY, 1033-1040.

[27] Yang, Y., Webb, G., Korb, K., and Ting, K. M. 2007. Classifying under computational resource constraints: anytime classification using probabilistic estimators. *Mach. Learn.* 69, 1 (Oct. 2007), 35-53.

[28] Zilberstein, S., and Russell, S. 1995. Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation*. Kluwer Academic Publishers.

[29] http://sites.google.com/site/icdm10annc.