AnySC: Anytime Set-wise Classification of Variable Speed Data Streams

Jagat Sesh Challa, Poonam Goyal, Vijay M. Giri, Dhananjay Mantri, Navneet Goyal
ADAPT Lab, Dept. of Computer Science & Information Systems, Pilani Campus
Birla Institute of Technology & Science, Pilani, India
Email: {jagatsesh,poonam,f2014151,f2015139,goel}@pilani.bits-pilani.ac.in

***Abstract*—Classification of data streams has gained a lot of popularity in recent years owing to its multiple applications. In certain applications like community detection from text feeds, website fingerprinting attack, etc., it is more meaningful to associate class labels with groups of objects rather than the individual objects. This kind of classification problem is known as *the set-wise classification problem*. The few algorithms available in literature for this problem are budget algorithms, i.e. they are designed to process fixed maximum stream speed, and are not capable of handling variable and high speed streams. We present ANYSC which is the first *anytime set-wise classification* algorithm for data streams. ANYSC handles variable inter-arrival rate of objects in the stream and performs classification of test entities within any available time allowance, using a proposed data structure referred to as *CProf-forest*. The experimental results show that ANYSC brings in the features of an anytime algorithm and outperforms the existing approaches.**

***Keywords*-Data Mining; Data Streams; Data Classification**

I. INTRODUCTION

A data stream is characterized by continuously arriving data objects at a fast rate, ordered by time. Mining data streams is typically constrained by limited available time to process the incoming objects and limited memory to store them. Due to increasing utilities of data generating systems such as web, sensor networks, retail chains, etc., classification in data streams has become very popular [1]–[6]. These algorithms build a classification model on the initial training corpus, which is used to classify the test objects arriving in the stream. These algorithms allow only a single pass for classifying each test object.

In many applications such as community detection from text feeds, website fingerprinting attack, retail chain analysis, etc., the classification labels are not associated with individual data objects, but with groups of objects. Each group is treated as an indivisible *entity* with an associated class label. And, a class label can be meaningfully assigned to an entity only by studying the overall distribution pattern of objects in it, rather than studying a single object. This kind of problem is known as the *Set-wise Classification problem* (defined in [7]) and can be found in the following stream applications:

Community Detection using text feeds: Community detection allows us to find groups of users who have common interests [8]. These groups can be used for targeted advertising on social networks or viral advertising campaigns. There can be multiple users belonging to various communities creating text feeds on a social networking website such as twitter. Using these feeds (or tweets), we can predict the community to which each user belongs to. This problem can be viewed as a set-wise classification problem. Each tweet can be considered as a data object, each user can be considered as an entity (associated with tweets tweeted by him), and a community class label can be associated with each user, with each class having multiple users associated. We can clearly observe that based on a single tweet, we may not be able to meaningfully assign a class label to a user. However, by studying the pattern formed by a set of tweets tweeted by a user, we can study his behavior and appropriately assign a class label. In this problem, we can first construct an initial classification model over a given sample of tweets from various users (entities) with known community class labels. Then in the stream we will receive and process real-time tweets that are being posted by various users. The stream can receive tweets from both labeled and unlabeled users. The tweets from labelled users (training entities) can be used to incrementally update the classification model. The tweets from unlabelled users can be used to construct the test entities (unlabelled users), whose class labels can be predicted using the above model. As more tweets belonging to a user arrive, the prediction becomes more accurate.

Website Fingerprinting attack: It is a Trac Analysis Attack, where network attackers try to breach web navigation security and privacy [9], [10]. Certain web users accessing web pages use anonymous communication mechanisms to hide the content and meta data exchanged between the browser and the server hosting the web page using methods like Tor network [11]. An Attacker can use ML techniques to identify the web page accessed by a user by capturing the network packets secretly (even encrypted ones). Such attackers can target individuals, businesses and governments. In order to prevent them, researchers study different attack schemes and provide counter measures. This attack can be modeled as set-wise classification problem over a stream of network packets captured when various users are accessing the web. Given a set of web pages that are being accessed by the users, each web page shall have different network traces associated where each trace consists of uplink and downlink packets generated when a user loads the web page. Each packet contains information like time, direction and

978-1-5386-5035-6/18/$31.00 ©2018 IEEE

length in bytes. We can consider a group of consecutive packets (known as a *burst*) going in a specific direction as a data object. Each burst is characterized by *burst length* and *direction*, which are the features used by the attackers [9]. The set of packets (bursts) exchanged (uplink and downlink) between the user and the server for loading a complete web page forms a *trace*, which can be considered as an entity. Each trace can be associated with a class label (web page label). So, in this problem, an individual packet (or a burst) may not be associated with a class label since similar packets can occur over multiple web pages. Instead, a class label is associated with a set of packets (or set of bursts), which forms a trace (entity). We can build an initial classification model over a given sample of packets from various traces with known class labels. In the stream, the packets received from labeled traces can be used to incrementally update the training model, and packets from unlabeled traces can be used to construct test entities whose class label can be predicted using the classification model.

To the best of our knowledge, only a couple of approaches have been proposed for set-wise classification for data streams. The approach in [7] formally defines this problem and presents a classification model & method for classifying test entities. Its classification model consists of sets of class profiles (one set for each class). A class profile is an object that characterizes the average distribution pattern of a set of entities (see Section II). A test entity is matched with all class profiles (from all classes) to find the closest, whose class label is assigned to the test entity. The second approach [10] extends the previous by using an ensemble of classifiers such as nearest neighbors, bayesian classifier, etc.

Limitations of Budget Approaches: The above algorithms take a fixed duration of time (*budget*) for assigning class label to each test entity. However, real-time streams do not have constant speed and thus cannot guarantee availability of such budget for every object received. For example, in community detection using text feeds, the rate of arrival of tweets is not fixed. It can vary depending upon many factors. Similarly, in website fingerprinting attack, the number of packets exchanged in the network is high during peak hours and low during other times. The classification model of the above algorithms pose a flat linear structure (list of class profiles) that can not handle *variable inter-arrival rate* of objects, which is a key requirement in the above applications. When the stream speed is lesser than their budget, they successfully process the incoming objects, and then sit idle for the remaining time allowance without any effort to improve the accuracy. And when the stream speed is higher than the budget, they fail to execute! An ideal algorithm, however, should be able to process any stream speed, giving the best possible result within any given time allowance, even if it is approximate, and refine its accuracy with increase in time allowance. Such an algorithm is known as an *anytime algorithm*. The above algorithms are budget algorithms and lack such capabilities.

A few algorithms were proposed in literature for anytime classification in data streams based on nearest-neighbors [12] and Bayesian Classifiers [13], [14]. However, none of them were designed for set-wise classification of data streams.

A. Our Contributions

- We propose ANYSC, which is an *Anytime Set-wise Classification algorithm* for data streams. To the best of our knowledge, this is the first such approach.
- To make it anytime, we propose a data structure, *CProf-forest*, which is built over the initial training data, and serves as the classification model for processing the incoming test objects. The *CProf-trees* in the CProf-forest store class profiles at hierarchical granularities. ANYSC incrementally updates the CProf-forest using the objects arriving in the stream that belong to the labeled entities, within any given time allowance dictated by the stream speed. The incremental updates are deferred in case the time allowance expires before the update finishes, and are completed alongside subsequent updates occurring on the same traversal path in the underlying CProf-tree.
- ANYSC leverages the hierarchical structure of CProf-trees to classify the test entities within any given processing time allowance (see Section III).

The experimental results presented in Section IV show that ANYSC can: (i) handle variable stream speeds and produce accurate classification results; (ii) handle very high speed streams with reasonable performance, unlike the baseline approach (SC) [7] that fails to execute when speed exceeds its budget; (iii) give very high classification accuracy (compared to SC) when stream speed is low, since it makes use of greater time available to refine the result to the greatest possible degree. The experiments also demonstrate the applicability of set-wise classification problem over both the applications described above.

II. BACKGROUND

Let there be N training entities (labeled) in the entire corpus denoted by $\mathcal{E}_1...\mathcal{E}_N$, where each entity $\mathcal{E}_i$ has e_i data objects in it. Let there be a total of c different classes, with associated class labels $[1...c]$. Let d be the dimensionality of the dataset. Each object in an entity is a d-dimensional vector. Similarly, let there be a set of n test (unlabeled) entities ($\mathcal{T}_1...\mathcal{T}_n$). Let the data objects in the stream be received in the form of tuples - $< Y_1, entityid_1, label_1 >$ ... $< Y_r, entityid_r, label_r >$ ... and so on. Y_r is a d-dimensional object which could either belong to a training entity or to a test entity; $entityid_r$ is the id of the entity to which the object belongs to; and $label_r$ is its class label. If Y_r belongs to a training entity, its class label will be in the range $[1...c]$, and if it belongs to a test entity, its class label will be -1. So, the stream receives a mixture of objects belonging to training and test entities. Note that training

and test entities are disjoint. *So, given a set of training entities $\mathcal{E}_1...\mathcal{E}_N$, the problem of set-wise classification in data streams is: constructing a classification model using a sample of objects from training entities, using which we can classify test entities whose objects arrive in the stream.*

A. Set-wise Classification model [7]

The classification model for set-wise classification [7] (SC), is constructed over an initial sample of data objects from training entities and is then incrementally updated using the training objects received in the stream. The model consists of: c *classes*, p *class profiles* and N *training entities* ($c << p << N$). Each class contains a subset of class profiles and each class profile is an aggregate of a subset of training entities.

An entity (either training or test) is represented by a q-dimensional vector known as *fingerprint*, which contains the distribution pattern of objects belonging to it. Initially, a set of q data objects known as *anchor points* (denoted by $W_1...W_q$) are selected from the initial training sample, k-means clustering (with $k = q$) is performed over it and the final centroids are chosen as the anchors. Note that q is an input parameter. The distribution pattern of objects for any given entity is captured as a frequency vector around these q anchors forming a histogram like structure, which is the fingerprint. The value of q dictates the granularity of the fingerprint. It is defined as follows:

Definition 1: (FINGERPRINT). Let there be r objects denoted as $Y_1...Y_r$, in a given entity $\mathcal{E}_j$. Let these objects be assigned to their respective closest anchor points (one in $W_1...W_q$), resulting in a partitioning consisting of q clusters $C_1...C_q$. Let the corresponding (relative) cluster frequencies be denoted as $f_1...f_q$, where $\sum_{i=1}^{q} f_i = 1$. Then, the *fingerprint* of $\mathcal{E}_j$, defined with respect to the above q anchors, is denoted by the q-dimensional vector $[f_1...f_q]$.

Given a set of data objects $Y_1...Y_r$ belonging to an entity $\mathcal{E}_j$, the fingerprint of $\mathcal{E}_j$ by constructed by incrementally adding each object to it. For each Y_i, we first identify the closest anchor point using *euclidean distance* as the distance measure. Say the index of selected anchor point is ind in the fingerprint vector. Let the number of objects updated in the fingerprint of $\mathcal{E}_j$ before adding Y_i be $|U|$. Then the new fingerprint vector after adding Y_i to $\mathcal{E}_j$ can be computed as:

$$f'_i = \begin{cases} \frac{f_i.|U|}{|U+1|} & i \neq ind \\ \frac{f_i.|U|+1}{|U+1|} & i = ind \end{cases} \quad (1)$$

After constructing the fingerprints for the training entities in the initial data sample, the class profiles are created over them. A *class profile* contains the average fingerprint characterization for a given set of entities. It is defined as:

Definition 2: (CLASS PROFILE). Given a set of entities $S = \mathcal{E}_1...\mathcal{E}_s$ belonging to one class with label l and associated q-dimensional fingerprints $L_1...L_s$. A class profile is a tuple of: (i) A q-dimensional vector AG_S containing the sum of the fingerprints of all the entities $\in S$ across all dimensions, i.e. $AG_S[p] = \sum_{i=1}^{s} L_i$ for each dimension $p \in [0...q-1]$. (ii) An integer value s storing the number of entities aggregated in this class profile. (iii) A class label l.

Class profiles are used for predicting class labels of test entities. There can be multiple class profiles associated with each class and to construct them, k-means is applied over the entities belonging to a given class, and one class profile is constructed for each cluster by aggregating the entities belonging to it. Let p_i be the number of class profiles for the class with label i. The total number of class profiles in the model will be $p = \sum_{i=1}^{c} p_i$. Thus, the SC model comprises a set p of class profiles with associated class labels.

B. Processing Incoming Stream Objects

The stream receives a mixture of objects belonging to labeled and unlabeled entities. The objects from labeled (training) entities are used to incrementally update the classification model. The objects from unlabeled (test) entities are used to construct the test entities, which are assigned appropriate class labels using the classification model.

For incoming object $Y_i \in$ training entity $\mathcal{E}_j$, the following steps are executed to incrementally update the classification model: (i) Identify the closest anchor point. (ii) Update the fingerprint of $\mathcal{E}_j$ using Eq. 1. (iii) Determine the closest class profile within the same class and re-assign $\mathcal{E}_j$ to it if necessary, and update the class profile statistics.

Since, class profiles are simple aggregate vectors, they posses additivity property by which we can add (or subtract) an entity fingerprint vector to (or from) it. This property is used to update the class profile statistics, wherein we subtract the old fingerprint vector from it and add the newly updated one. If the incoming object $Y_i \in$ a test entity $\mathcal{T}_j$, the first two steps are the same as the above. In the third step, a class label is assigned to $\mathcal{T}_j$ if it was not assigned previously, or $\mathcal{T}_j$ is re-assigned to the most appropriate class (either same or a different class). Before a class label is assigned to it, the algorithm lets the test entity accumulate at least min_stat objects in it. This allows the test entity to accumulate sufficient number of objects into it before it can be meaningfully assigned to a class. As the number of objects being accumulated to an entity increases, its class label prediction becomes more accurate, since its objects' distribution pattern shall be known more accurately. So, to assign a class label to $\mathcal{T}_j$, all p class profiles (belonging to all c classes) are scanned to identify the closest, whose class label is assigned to $\mathcal{T}_j$. Cosine similarity is used to measure distance between a fingerprint and a class profile, both of which are q-dimensional vectors.

The model described above has certain limitations and cannot be used over variable stream speeds (as explained in Section I). Its limitations and how they have been addressed by ANYSC, is described in Section V.

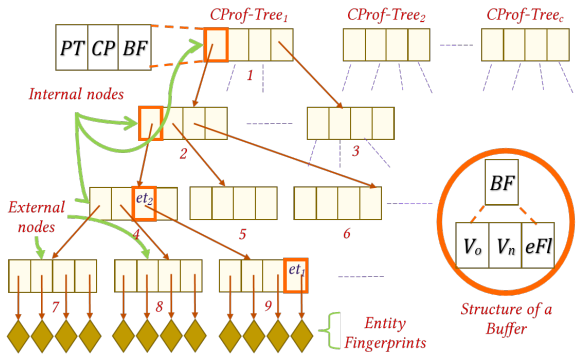


Figure 1: CProf-forest

III. ANYSC

We first present the data structure - *CProf-forest* (Fig. 1), which serves as the classification model for ANYSC. It consists of *CProf-trees* whose total number is equal to the number of classes (c). CProf-trees store hierarchy of class profile vectors in their internal nodes. The design of CProf-tree is inspired from ClusTree [15], which stores a hierarchy of micro-cluster vectors used for clustering in streams. Although both of them have same skeletal structure, they are conceptually different. The problem addressed by both of these structures are fundamentally different. Also, the workflow of *insert-and-update* algorithm in both of them is totally different: top-down in case of ClusTree and bottom-up in case of CProf-tree. We define CProf-tree as follows:

Definition 3: CPROF-TREE. It is a height balanced multi-dimensional indexing structure (see Fig. 1) having the following properties:

- All nodes (both internal and external) contain between m and M entries. The root has at least one entry. Fig. 1 shows a CProf-tree with $m = 2$ and $M = 4$.
- All nodes store a pointer $Parent$ which points to the parent entry.
- An entry e in an external node stores the q-dimensional fingerprint vector representing an entity.
- An entry e in an internal node stores: (i) a pointer PT_e pointing to the root of sub-tree beneath e; (ii) a class profile CP_e storing the aggregate of all the entity fingerprints indexed at the leaf level of the subtree rooted at e; (iii) a *buffer* BF_e, which consists of 2 q-dimensional vectors - V_o & V_n, and a flag eFl which is set to TRUE if buffer is empty. BF_e is used for deferred updation of CProf-tree that occurs while processing high speed streams (see Section III-A).

Definition 4: CPROF-FOREST. It is collection of c CProf-trees, one for each class.

The hierarchical nature of CProf-tree enables us to represent the distribution patterns of the objects belonging to a class in a fashion of hierarchical granularities of various levels. The root of each tree summarizes the distribution pattern of all the entities belonging to a class at coarsest granularity. And, as we keep going down the levels in the tree, the granularity of the class profiles (stored at internal nodes) becomes finer and finer, until we reach the leaf nodes containing entity fingerprints, which are of finest granularity. Consider the CProf-tree in Fig. 1. We can arrange the nodes in increasing level of granularity as follows: $1 < 2, 3 < 4, 5, 6 < 7, 8, 9$. This model of hierarchical granularities helps in classification of test entities within any given time allowance dictated by varying arrival rate (see Section III-B).

A CProf-tree for a given class is constructed by dynamic insertion (of logarithmic cost) of training entities belonging to it, in a top-down recursive fashion similar to that of an R-tree. The only difference is that the distance metric is computed between the entity fingerprint vector and class profile vectors stored at the internal nodes (using cosine similarity metric), rather than with the Minimum Bounding Rectangles as in R-trees (refer [16] for more details). The node overflows are handled in a similar manner as that of an R-tree and the node splits propagate upwards leading to CProf-tree growing upwards. The vectors in the buffers of CProf-trees are left empty during the creation of the initial CProf-forest in the training phase. *So, given a set of training entities $\mathcal{E}_1 ... \mathcal{E}_N$, with associated labels l_1 l_2 ... l_N, we construct a CProf-forest over an initial sample of objects $\in$ training entities, which serves as the classification model for classifying test entities $\mathcal{T}_1$ $\mathcal{T}_2$ ... whose objects are arriving in the stream with variable inter-arrival rate.*

A. Anytime Incremental Model Update

The classification model is incrementally updated using training objects received in the stream, within the variable time allowance dictated by variable inter-arrival rate of objects. It is illustrated with Fig. 1 & Algo 1 and explained as follows:

Let the arriving training object be $< Y_r, entityid_r, label_r >$. Firstly, we need to update the fingerprint of the entity to which Y_r belongs to (Algo 1:line 3), which is the entity with id = $entityid_r$. This entity will be indexed in a leaf node of the CProf-tree with class label - $label_r$. To do this update, we first identify the nearest anchor point among q anchors and update the fingerprint using Eq. 1. This operation can be done in $\mathcal{O}(1)$ time by additionally indexing all the entities in a separate array with their respective ids as the rank. After this, we need to perform a bottom-up update of the tree in which the above entity is present, in order to accommodate the change that has just happened. We need to update the class profile aggregates along the path from the current entity to the root node. For this, we make use of the additivity property of the vector AG (stored in class profiles), where at each class profile in this bottom-up path, we subtract the old aggregate of the entities present in the sub-tree rooted at it and add the new aggregate (lines 13-25). For example, consider Fig. 1. Let the entity which just got updated be stored at entry et_1 of leaf node 9. The entry et_2 at node 4 contains a class profile that

stores the aggregate of all the fingerprints indexed at node 9. To accommodate this change, we update the class profile indexed at et_2, by subtracting the old fingerprint vector of entity at et_1 from it ($vect\_old$), and adding the newly updated fingerprint to it ($vect\_new$). We then proceed for subsequent iteration and in this way the update propagates up to the root along the path consisting of entries highlighted in orange in the figure. However, if during the bottom-up update, the time allowance for processing this object expires (triggered typically due to arrival of a new object in the stream), we *defer* its completion. For this, we use V_o and V_n vectors stored in the buffers for temporarily storing the old and new aggregate vectors, respectively (lines 7-12). Say, the time allowance has expired before the class profile indexed at et_2 got updated, we simply leave the old and new fingerprint vectors of entity indexed at et_1 in the buffer of its parent entry et_2, and exit to process the newly arrived object. Next time when a new update triggered by the insertion of another data object passes through et_2, the vectors in its buffer are taken out and processed along with (lines 13-25). The class profile at et_2 will be updated with respect to the old and new vectors stored in the buffer in a similar manner explained above (lines 17-21), along with the recent old and new vectors (lines 16 & 22). And then the combined update propagates upwards in subsequent iterations. This way the deferred update triggered by an old object can reach the root and get eventually completed. Note: In this algorithm, the update of a class profile is a *unit operation*. Unit operation is an atomic non-interruptible operation, i.e., we will process the next object only after this operation completes, even if the next object arrives before it finishes.

In the above algorithm, it is possible that multiple vectors, resultant of update of different entities, get accumulated in the same buffer. However, owing to the additive nature of these vectors, the algorithm's correctness is not affected and the class profiles in the tree remain consistent. The above bottom-up update is of *logarithmic order* of the number of entities indexed in the tree.

Algorithm 1: INSERT AND UPDATE IN CPROF-TREE

```
1  procedure INSERT-AND-UPDATE-IN-CPROF-TREE()
       Input : A CProf-tree CPT_i, a training object: Y_r, entityid_r, label_r
       Output: Y_r inserted into CPT_i and CPT_i updated bottom-up until
               time allows
2      vect_old = E_{entityid_r}.fingerprint;
3      Update E_{entityid_r}.fingerprint with respect to Y_r;
4      vect_new = E_{entityid_r}.fingerprint;
5      curr_entry = ENTRY_OF(E_{entityid_r});
6      while NODE_OF(curr_entry) != root do
7          if NEW_OBJECT_ARRIVED then
8              foreach dimension p ∈ [0...q] do
9                  NODE_OF(curr_entry).parent.BF.V_n[p] +=
                     vect_new[p];
10                 NODE_OF(curr_entry).parent.BF.V_o[p] +=
                     vect_old[p];
11             NODE_OF(curr_entry).parent.BF.eFl = FALSE;
12             return
13         curr_entry = NODE_OF(curr_entry).parent;
14         foreach dimension p ∈ [0...q] do
15             temp = curr_entry.CP[p];
16             curr_entry.CP.AG[p] -= vect_old[p];
17             if curr_entry.BF.eFl == FALSE then
18                 curr_entry.CP.AG[p] -= curr_entry.BF.V_o[p];
19                 curr_entry.CP.AG[p] += curr_entry.BF.V_n[p];
20                 curr_entry.BF.V_o[p] = 0;
21                 curr_entry.BF.V_n[p] = 0;
22             curr_entry.CP.AG[p] += vect_new[p];
23             vect_old[p] = temp;
24             vect_new[p] = curr_entry.CP.AG[p];
25         curr_entry.BF.eFl = TRUE;
```

Algorithm 2: CLASSIFYING A TEST ENTITY

```
1  procedure CLASSIFY-TEST-ENTITY()
       Input : A CProf-forest Forest, a test entity T_r
       Output: Class label assigned to T_r
2      Initialize c Priority Queues PQ_1 ... PQ_c;
3      foreach CProf-tree CPT_i of Forest do
4          foreach entry e in CPT_i.root do
5              PQ_i.ADD(e);
6      while TRUE do
7          foreach CProf-tree CPT_i of Forest do
8              temp = REMOVEMIN(CPT_i);
9              foreach entry e in temp.PT do
10                 PQ_i.ADD(e);
11         if (NEW_OBJECT_ARRIVED or all leaves are scanned) then
12             T_r.label = CLOSEST(REMOVEMIN(PQ_1),
                 REMOVEMIN(PQ_2), ... , REMOVEMIN(PQ_c)).label;
13             return
```

B. Anytime Classification of test entities

Let the arrived test object be $< Y_r, entityid_r, -1 >$. First we update the fingerprint of the entity $\mathcal{T}_{entityid_r}$ with respect to Y_r as was done in case of training objects. If $\mathcal{T}_{entityid_r}$ doesn't exist in this test set, we first create it and then update. Once $\mathcal{T}_{entityid_r}$ contains at least min_stat objects updated in its fingerprint, we can meaningfully assign it to a class. The process of classifying a test entity is illustrated in Algo 2. Essentially, we do a *best-first* traversal of the CProf-trees using c number of priority queues, one for each class. These traversals can be interrupted anytime, i.e. whenever a new object arrives in the stream we can assign a class label to the current test entity on the basis of class profiles of different trees visited until now, and then exit to process the newly arrived object. The class label assigned should be the most accurate for the given processing time allowance. For this, we shall have to compare the current test entity with respect to class profiles/entities belonging to all the classes and then assign the class label of the closest class profile/entity (using cosine similarity as distance measure). However, we may not have sufficient time to scan all the class profiles/entities of all classes. So, we start our comparisons with the class profiles stored in the root node of all the trees. This is because the entries in the root node summarize all the entities of a given class. So, we add all the entries of the roots to their respective priority queues (lines 2-5). And we iteratively refine our search space by removing the closest entry to the test object from each priority queue, and add its child entries into the same priority queue (lines 7-10). This step increases the granularity of the search space, because the class profiles stored in the entries which got just added to the priority queues are of finer granularity than those that were removed. In this way we iteratively refine our search space in every tree in best-first manner, until a point of time

when the priority queues are only filled with leaf entries that index entities. However, our processing time allowance may not allow us to reach this point. So, whenever time allowance expires, we assign the current test entity to the class of the most closest entry (could be an entity or a class profile) among all the entries in all the priority queues, and then exit to process the newly arrived object (lines 11-13). So, the label assigned to test entity will be the most accurate one for the given time allowance. If we have sufficient time allowance to reach the leaf levels of the CProf-trees, the classification accuracy will be highest. So, when processing time allowance is high at lower stream speeds, or when we execute without any constraint on processing time allowance for each object (*non-anytime mode*), we reach finer granularities of the CProf-trees and get highly accurate results (see Fig. 4 and Table I). Also, as the test entities evolve with more number of objects being added into them, the classification accuracy also improves. The unit operation in the above algorithm is removal of an entry from a PQ and adding its children to it.

We conducted an empirical study, where we tested our approach by refining only a subset of top k nearest classes rather than refining all the classes (see Fig. 2). The results clearly show that it is sufficient to refine top 4 classes to get accurate results for all datasets used in experimentation.

IV. Experimental Analysis

All experiments were conducted over a workstation with 32 GB RAM, Intel i7 processor and Ubuntu 14.04 installed in it. All programs were implemented in C. The description datasets used for experimentation is presented herein.

1) Synth: It a synthetic dataset generated using guidelines given in [7]. It has 1M 10-dimensional objects generated using *gaussian mixture models*. It consists of 50 classes, with each class having 20 entities (total 1000 entities).

2) ECover: This is generated by converting the Forest Cover dataset (FC) [17] into an applicable form using guidelines given in [7]. We use first 10 dimensions of FC, which are numeric. It has 0.58M objects distributed in 10 different classes. Objects in each class have been clustered into 150 different entities, giving a total of 1500 entities.

3) CDetect: We created this dataset by collecting tweets of various Indian celebrities that include - Politicians, Sportsmen, Journalists, Film stars, Philanthropists and Businessmen (6 classes). A maximum of 3200 historical tweets were collected from each user using *Tweepy API* on 15 Aug 2018. The data set consists of 1,688,365 tweets (data objects), collected from 751 users (entities), belonging to 6 communities (class labels). We extracted keywords from the tweets using Stanford Postagger [18] and represented each tweet as vector around these keywords. We use Jaccard Coefficient as the distance measure to compute distance between two objects (tweets), unlike euclidean distance for other dataset. Also, the distribution of entities per class is skewed in this dataset (14-455 entities/class), unlike the previous two which were uniform.

4) WebFing: The Website Fingerprinting attack dataset has been borrowed from [9]. It consists of approx. 111M bursts (data objects) categorized into 41K traces (entities) from 2000 web pages (classes). This dataset also has uniform distribution of entities/class (190-210).

We set 80% of the entities as training entities and 20% as test entities for all datasets. We build the initial training model using 20% of the objects from training entities. The remaining objects arrive in the stream in a random mixture of training and test objects. The values of fanout parameters for CProf-tree have been set to: $m = 2$ and $M = 4$, & min_stat is set to 50. The term SC indicates the baseline result generated by using the method proposed in [7].

To simulate a stream with varying inter-arrival rates, we use Poisson streams, which is a stochastic model used to model random arrivals [19]. It takes in a parameter λ which controls the speed of the stream. For, $\lambda = \frac{1}{x}$, the model generates an expected number of λ objects per second (*ops*), with expected inter-arrival rate of x sec. between any two consecutive objects.

We use *accuracy* as the measure of evaluation for all datasets except CDetect, for which we use *F1-score*. This is because CDetect has skewed distribution of entities per class, and hence use of accuracy can give misleading results.

A. Experimental Results

In the first experiment, we measure the classification accuracy of test entities for AnySC with increase in number of classes refined, at speed (λ) = 40,000 ops & q = 40 for *Synth* and *ECover* datasets. The results presented in Fig. 2 clearly show that accuracy increases until we reach 4 classes and then remains steady for sometime and later shows a decline. The initial increase is because, when we refine lower number of classes, we may not reach the closest class profile. This problem reduces with increase in number of classes refined. The decline in later part of the curve is because, as the number of classes refined increases, after a point in the curve, the additional classes being added for refinement does not involve any of the closest probable classes that could help in improving classification accuracy. Rather, they only add up to the computation overhead, and does not allow us to reach finer granularity levels of CProf-trees, leading to reduction in classification accuracy. It can be observed from Fig. 2 that refining only the top 4 classes yields better

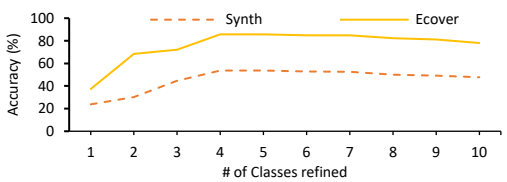
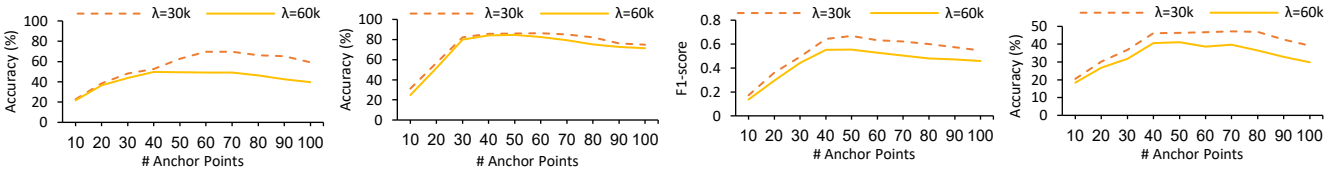
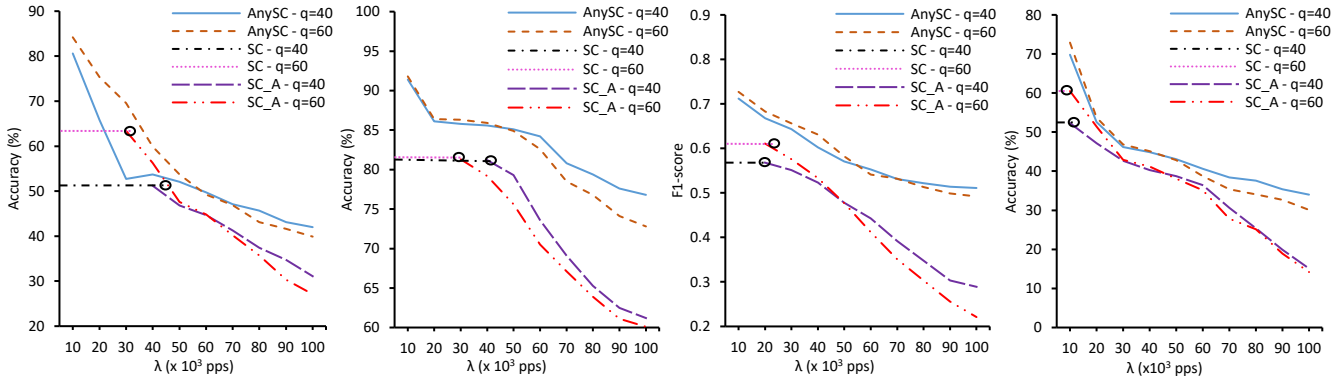


Figure 2: Effect on Accuracy with variation in number of classes refined at λ=60,000 ops



(a) (b) (c) (d)

Figure 3: Effect of varying q on performance for (a) *Synth* (b) *ECover* (c) *CDetect* (d) *WebFing* datasets



(a) (b) (c) (d)

Figure 4: Effect of varying λ on performance of ANYSC for (a) *Synth* (b) *ECover* (c) *CDetect* (d) *WebFing* datasets

classification results for both the datasets. Similar pattern was observed for other datasets as well. This is also evident from results presented in the last column of Table I, which gives the %age of test entities whose true class value actually lies within top 4 closest classes refined by ANYSC when run in non-anytime mode. Therefore, in subsequent experiments, we restrict our search to *top 4* closest classes and utilize the spare time to reach finer granularity levels within the closest classes to obtain more accurate results.

In the next experiment, we study the classification accuracy/ F1-score of ANYSC with variation in number of anchor points (q) at two different stream speeds - λ=30,000 (moderate speed) and λ=60,000 (high speed). The results presented in Fig. 3 show that the accuracy increases with increase in anchor points up to a certain point and then declines for all the datasets at both the stream speeds. The decline is because, as the number of anchor points increase, the distance computation cost increases due to which the speed handling capacity reduces and thus with lower processing time allowances at higher speeds, we will not be able to reach finer granularities in CProf-trees.

Next, we tested the performance of ANYSC with variation in stream speed and compared it with that of SC. The number of class profiles for SC has been set to 100 for Synth & ECover, 150 for CDetect & 20,000 for WebFing. We studied the accuracy/ F1-score at q = 40 & 60. The results presented in Fig. 4 clearly show that ANYSC is able to give a classification result at all the speeds, unlike SC which is limited to giving results up to its budget respective to the dataset (budget speeds are marked with circles and their values are given in Table I). We can also observe that the accuracy at lower stream speeds is much higher for ANYSC than SC whose accuracy is constant for all speeds up to its budget. This is because ANYSC refines its search space to finer granular levels of CProf-trees (upto the entities at the leaf level), which is not possible in SC. Also note that the rate of reduction in accuracy with increase in stream speed is greater for q=60, which is due to increase in cost of distance computations.

To get a fair baseline comparison, we tried to convert the SC model into an anytime model, where we do a linear scan of the class profiles list only to the extent time allows. For e.g., if we have a list of 100 class profiles, and time allowance is sufficient to scan only the first 15 class profiles, we assign class label to the test entity only on the basis of these 15 class profiles scanned. We refer to this strategy as SC_A. Fig. 4 also presents results comparing SC_A with ANYSC. Note that SC and SC_A shall give the same result until SC's budget speed. The results clearly show that accuracy of ANYSC is higher than SC_A. This is because, giving a class label to a test entity on the basis of only a few scanned class profiles doesn't guarantee the scanning of its actual closest class profile, which is the case in SC_A.

Next, we compared the accuracy/ F1-score of ANYSC with that of SC at SC's budget. The results presented in Table I clearly show that ANYSC outperforms in every case. ANYSC is giving better accuracy than SC even at the budget speed of SC. We also present the accuracy of ANYSC without applying the anytime feature i.e. each time the refinement will take place up to the finest level in the CProf-trees without any constraint on time allowance. ANYSC gives remarkable performance in this case. However, this

Table I: Comparison of AnySC with SC at its budget and non-anytime performance of AnySC at q = 40

Dataset	Max. Speed of SC (budget) (aprox.)	Accuracy /F1 of SC at its budget	Accuracy/ F1 of AnySC at SC's budget	Non-anytime accuracy /F1 of AnySC	Non-anytime speed of AnySC (approx.)	% test entities with actual class within top 4 classes refined
Synth	46,000 ops	51.2	53.4	80.7	14,000 ops	85.1
ECover	41,000 ops	81.2	85.6	91.5	11,000 ops	95.6
CDetect	25,000 ops	0.568	0.651	0.715	9,000 ops	90.3
WebFing	12,000 ops	52.3	65.1	69.8	12,000 ops	80.1

hampers its speed handling capacity, as it takes more time to reach finest granularity levels (entity fingerprints at leaf nodes). Note that the difference observed in non-anytime accuracy (column 5) and percentage of test entities with true class value within top 4 refined classes in non-anytime mode (column 7), is because of anomalies present in the dataset.

We also studied the evolution of test classification accuracy with stream progression for both ANYSC and SC. Due to limited space we don't present the results here. However, we observed that the accuracy increases with more objects received for both SC and $AnySC$. This is because: (i) model gets incrementally updated and drift in patterns is captured as more training objects are received; (ii) the distribution patterns of the test entities are predicted more accurately when more test objects are received.

V. DISCUSSION

ANYSC addresses the following limitations of SC:

Flat Structure Limitation: SC's model is a linear flat structure consisting of a list of p class profiles. Test entities are assigned labels by identifying the closest in this list using a linear scan (of $\mathcal{O}(p)$ cost) that takes a fixed amount of time (budget) to complete, irrespective of the stream speed. ANYSC addresses this limitation by using *CProf-forest*, which stores class profiles at hierarchical granularities and thus enables us to classify test entities within any given time allowance.

Classification Accuracy at low speeds: At low speed streams, when the time available is more than the budget, SC doesn't utilize the excess time to improve the classification accuracy. However, $AnySC$ makes best use of such time by reaching finest levels in the trees, thus giving very high classification accuracy.

Efficient incremental update: The cost of incremental update of the classification model in $AnySC$ is little higher than the SC because of the bottom-up update in CProf-trees. However, due to the support given by buffers, we can always defer & merge multiple such updates (as explained in Section III-A) and traverse together leading to reduction in the update cost. This reduction is visible in $AnySC$'s capability of handling high speed streams.

Efficient Class Assignment: Class label assignment cost is logarithmic for ANYSC, unlike linear cost for SC.

VI. CONCLUSIONS & FUTURE WORK

We presented ANYSC, which is the first anytime set-wise classification algorithm for data streams. It uses a data structure, CProf-forest, which stores class profiles at hierarchical granularity in its trees, and serves as the classification model for classifying test entities whose objects arrive with variable inter-arrival rate. The experiments shows that ANYSC is able to bring in anytime features and outperform SC in all aspects. In future, we plan to adopt ANYSC to image data.

REFERENCES

[1] R. Jin and G. Agrawal, "Efficient decision tree construction on streaming data," in *KDD '03*. ACM, New York, USA, 2003, pp. 571–576.

[2] D. Jankowski, K. Jackowski, and B. Cyganek, "Learning decision trees from data streams with concept drift," *Procedia Computer Science*, vol. 80, pp. 1682 – 1691, 2016.

[3] Q. Xue, B. W. Cao, W. Chang-wei, Y. Ping-gang, and L. Yong-hong, "Study on application of bayesian classifier model in data stream," in *ICCIS '10*, 2010, pp. 1312–1315.

[4] T. T. T. Nguyen et al., "Variational inference based bayes online classifiers with concept drift adaptation," *Pattern Recognition*, vol. 81, pp. 280–293, 2018.

[5] Y.-N. Law and C. Zaniolo, "An adaptive nearest neighbor classification algorithm for data streams," in *PKDD '05*. Springer-Verlag, Berlin, Heidelberg, 2005, pp. 108–120.

[6] C. C. Aggarwal, *Data Classification: Algorithms and Applications*, 1st ed. Chapman & Hall/CRC, 2014.

[7] C. C. Aggarwal, "The setwise stream classification problem," in *KDD '14*. ACM, New York, USA, 2014, pp. 432–441.

[8] W. Silva et al., "A methodology for community detection in twitter," in *WI '17*. ACM, NY, USA, 2017, pp. 1006–1009.

[9] M. Liberatore and B. N. Levine, "Inferring the source of encrypted http connections," in *CCS '06*. ACM, New York, USA, 2006, pp. 255–263.

[10] S. Jhaver, "Large scale data mining with applications in social computing," Master's thesis, Univ. of Texas, Dallas, 2014.

[11] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, "Anonymous connections and onion routing," *IEEE Journal on Selected Areas in Comm.*, vol. 16, no. 4, pp. 482–494, 1998.

[12] K. Ueno et al., "Anytime classification using the nearest neighbor algorithm with applications to stream mining," in *ICDM '06*. IEEE Computer Society, Washington DC, USA, 2006, pp. 623–632.

[13] T. Seidl et al., "Indexing density models for incremental learning and anytime classification on data streams," in *EDBT '09*. ACM, New York, USA, 2009, pp. 311–322.

[14] P. Kranen, M. Hassani, and T. Seidl, "Bt*: An advanced algorithm for anytime classification," in *SSDBM'12*. Springer-Verlag, Berlin, Heidelberg, 2012, pp. 298–315.

[15] P. Kranen et al, "The clustree: Indexing micro-clusters for anytime stream mining," *Knowl. Inf. Syst.*, vol. 29, no. 2, pp. 249–272, 2011.

[16] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, 1984.

[17] J. A. Blackard, "Comparison of neural networks and discriminant analysis in predicting forest cover types," Ph.D. dissertation, Colorado State Univ, Fort Collins, USA, 1998.

[18] (2018) Stanford postagger. Last accessed: 01 Aug, 2018. [Online]. Available: https://nlp.stanford.edu/software/tagger.shtml

[19] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, 2nd ed. Wiley-Interscience, 2000.