# Stop Chasing Trends: Discovering High Order Models in Evolving Data

Shixi Chen[†], Haixun Wang[‡], Shuigeng Zhou[†], Philip S. Yu[‡]

[†]*Fudan University, China,* {chensx,sgzhou}@fudan.edu.cn

[‡]*IBM T. J. Watson Research, USA,* {haixun,psyu}@us.ibm.com

***Abstract*— Many applications are driven by evolving data — patterns in web traffic, program execution traces, network event logs, etc., are often non-stationary. Building prediction models for evolving data becomes an important and challenging task. Currently, most approaches work by "chasing trends", that is, they keep learning or updating models from the evolving data, and use these impromptu models for online prediction. In many cases, this proves to be both costly and ineffective — much time is wasted on re-learning recurring concepts, yet the classifier may remain one step behind the current trend all the time. In this paper, we propose to mine high-order models in evolving data. More often than not, there are a limited number of concepts, or stable distributions, in the data stream, and concepts switch between each other constantly. We mine all such concepts offline from a historical stream, and build high quality models for each of them. At run time, combining historical concept change patterns and cues provided by an online training stream, we find the most likely current concept and use its corresponding models to classify data in an unlabeled stream. The primary advantage of the high-order model approach is its high accuracy. Experiments show that in benchmark datasets, classification error of the high-order model is only a small fraction of that of the current best approaches. Another important benefit is that, unlike state-of-the-art approaches, our approach does not require users to tune any parameters to achieve a satisfying result on streams of different characteristics.**

## I. Introduction

The primary task of data mining is to develop models based on existing data. In classification, usually the training data is fixed, for example, it is stored in a data warehouse, and the models, once trained from the stored data, can be applied to future data without much change. Thus, the knowledge discovery process can be regarded as consisting of two sequential phases: a *training* phase, where models are learned from past data, and a *testing* phase, where models are applied on the future data.

Recently, much work focuses on mining evolving data [1], [2], [3], [4], [5], [6], [7], [8], [9]. Evolving data, or data with changing class distribution, blurs the boundary between the *training* and the *testing* phases. Because data's class distribution is no longer stationary, models trained from past data cease to be valid for future data. In order to maintain reasonable prediction accuracy, *training* must be carried out continuously so that models are up-to-date as data evolves.

This introduces negative impacts on the accuracy of a stream classifier. Model training is often a time consuming, offline process. To keep up with the high data throughput in testing, we create impromptu models of low quality. In particular, it is hard to find out what data an up-to-date model should rely on. A large set of data may include changing concepts, and a small set will cause model over-fitting.

### *Our Motivation*

As data streams through the learning system, we train individual models from small windows on the stream as if taking fast snapshots of the evolving data. After a certain amount of time, we have accumulated many snapshots. The question is, can we mine these historical snapshots to derive a big picture about the underlying data generating mechanism, and stop wasting time taking endless snapshots?

This is desirable because big pictures are more revealing, and likely to have more predictive power, than individual snapshots. When data evolves, base models trained directly from small data chunks will become unstable. Instead of chasing ephemeral patterns in the data stream, we should learn a high-level, stable model from historical base models.

In this paper, we show that this approach is not only desirable, but also feasible. In fact, many systems work in a limited set of states, and within each state, data's class distributions are stable. For example, in network and system monitoring, most of the time the system is in a stable state. When certain events occur (e.g., heap exceeds physical memory), the system goes into another state (e.g., one characterized by paging operations). The state may switch back again (e.g., when memory usage recedes). As another example, we predict traffic patterns in a metropolitan road network. Under normal conditions, traffic behaves in one way, and under other conditions, e.g., after an accident, traffic behaves in another way. Note in both cases above, transitions among stable concepts may occur at any time, instead of exhibiting simple patterns such as periodicity.

### *State of the Art*

The idea of developing a meta-model from a set of base models is not new. For instance, meta-learning [10], query by committee [11], and other ensemble methods [12] have been studied extensively. The purpose of these approaches is to integrate results from multiple learning systems in order to reduce the inductive bias of individual learners, so that the meta-model is more accurate. However, they do not deal with the challenges of evolving data.

Much effort has been made to adapt traditional classifiers for evolving data [1], [2], [3], [4], [5], [9]. Most work, however, falls into the category of *chasing the snapshots*, that is, it learns and re-learns models in the evolving data, and then it uses a single snapshot or several snapshots stitched together to make predictions, instead of trying to reveal the big picture, that is, the underlying data generating mechanism.

The possibility that history always repeats itself, that is, only a limited number of concepts exists in an endless data stream, has prompted researchers to devise more efficient learning schemes. The RePro approach [13] seeks to group two base models if they tent to agree with each other in classifying a training dataset. Thus, large amount of historical snapshots are iteratively reduced to a much small number of concepts. A flaw in this pioneering approach is that we cannot afford to perform pair-wise comparison between every two models to find "equivalent classes" of concepts, and relying on an arbitrary training dataset as a similarity measure is just as problematic.

*Our Approach*

In this paper, we propose a novel approach for mining evolving data. We train a high-order model from base models to eliminate the need of re-learning classifiers over and over again.

The first step toward building a high-order model is to capture all stable concepts in the evolving data. However, as in the examples we mentioned above, concept changes may occur at any time, instead of exhibiting simple patterns such as periodicity [14]. The second component of the high-order model is the concept change patterns, which are also learned from the historical data, that is, we analyze how individual concepts interact with each other by collecting the statistics of concept changes. At runtime, with cues from an online training stream, the high-order model identifies the current concept in the stream and uses offline trained classifiers corresponding to the concept for prediction.

The primary advantage of our approach is its very high accuracy. Experiments show that in benchmark datasets, classification error of the high-order model is only about one tenth of the current best approaches. Furthermore, unlike state-of-the-art approaches, the high-order model has no user parameters. It does not require users to tune any parameters on the basis of the characteristics of different data streams in order to attain satisfying classification accuracy.

## II. Building a High-order Model

In this section, we discuss the process of building a high-order model. Assuming we have a sufficiently large historical dataset (labeled and ordered by time), our primary task is to group its data into a number of clusters, each corresponding to a stable concept.

### A. Concept Clustering

The historical dataset $D$ is ordered by time. Although class distribution in the dataset is changing overtime, there is still a high probability that adjacent records belong the same concept, as in most applications it is highly unlikely that state transitions would occur for every record.

We call a segment of data that belongs to the same concept a *concept occurrence*. In the time-ordered historical dataset, one concept usually has multiple occurrences. Thus, the concept clustering process can be divided into two steps: first find all independent concept occurrences, then group occurrences into unique concepts. This approach also avoids the problem of having to predetermine the number of concepts, or number of concept changes. In this spirit, we take the following steps in our concept clustering approach.

1) First we partition the historical stream into *data blocks* of equal size. The block size should be small enough (e.g., 2–20) such that data within a block represents a same concept with high probability. Then we group adjacent blocks into *data chunks*, each representing an concept occurrence, and the boundary between two chunks represents a concept change.
2) We group the chunks into a number of unique concepts, each of which consisting of a set of chunks representing occurrences of the concept.

The above steps are illustrated by an example in Figure 1, where we start with 16 data blocks, $1, \cdots, 16$. We then perform clustering twice. The first time is in step 1, where consecutive blocks in the stream are clustered into chunks $a, \cdots, f$, and then in step 2, where non-consecutive chunks are clustered into concepts $A$, $B$ and $C$.

As shown in Figure 1, our method is similar to agglomerative hierarchical clustering in unsupervised learning. Thus, the entire process of concept clustering can be visualized by a "dendrogram", a tree representing the order of merging. After merging is complete, we find the "best" cut in the dendrogram, which produces the final concepts. Before we describe the algorithm, we give an objective function that defines the strategy of merging and finding the best cut.

### B. The Objective Function

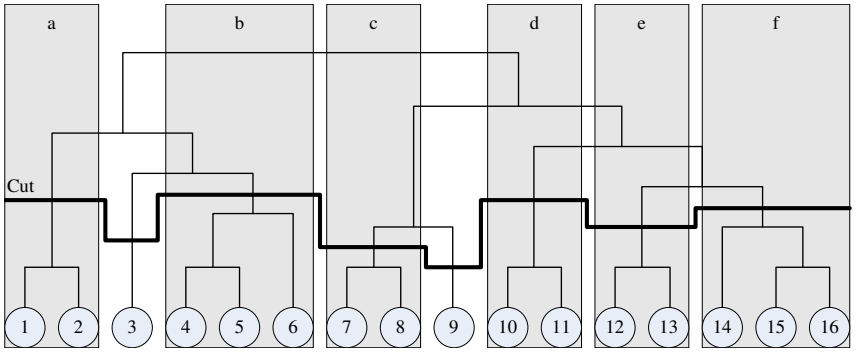
Given a set of clusters each of which represents a concept occurrence (step 1) or a concept (step 2), how do we measure its "goodness"?

Let $P$ be a set of clusters that defines a partition of $D$. From each cluster $D_i$, we learn a base model $M_i$ by any method (such as decision tree, Naïve Bayes, or SVM) designed for mining stationary data. In both of the two steps, let $Q(P)$ denote the quality of a partition $P$. Clearly, $Q(P)$ must have the following properties: i) grouping records belonging to the same concept increases the quality, and ii) grouping records belonging to different concepts decreases the quality.

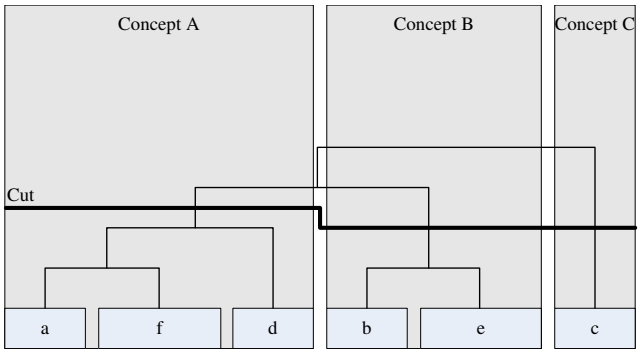
To this purpose, we propose the following objective function.

$$Q(P) = \sum_{D_i \in P} |D_i| \cdot Err_i \tag{1}$$

where $Err_i$ denotes $M_i$'s validation error rate. The validation error is derived using the technique of holdout validation: In




(a) Joining Adjacent Data Blocks into Chunks




(b) Joining Chunks into Stable Concepts

Fig. 1: Mining Underlying Concepts from Historical Dataset via Concept Clustering

each $D_i$, we randomly choose half of the data for testing, and the remaining half for training[1]. Intuitively, the lower the $Q$ value, the better the partition.

However, a low $Q$ may not always mean a good cluster. For instance, if each $D_i$ has only one tuple, or each $D_i$ consists of tuples of the same class only, then we have $Q = 0$ since $Err_i = 0, \forall i$. Nevertheless, in our case, $P$ is not an arbitrary partition of the data. In step 1, data in each cluster is contiguous. In step 2, a cluster is made up of clusters found in step 1. Given that each $D_i$ must contain at least two tuples so that the training and the testing dataset are mutually exclusive and non-empty, we can preclude the case where $|D_i| = 1, \forall i$. Our assumption that each tuple is regarded as an independent random sample[2] from the distribution precludes the 2nd case in which each cluster consists of contiguous data of the same class.

It should be clear that $Q(P)$ in Eq. 1 satisfies the criteria we set forth: i) it favors merging clusters that represent the same concept, as a larger training dataset is more resilient to the overfitting problem, which lowers classification error; ii) it favors separating data of different concepts, as a training dataset with conflicting concepts always leads to a less accurate classifier.

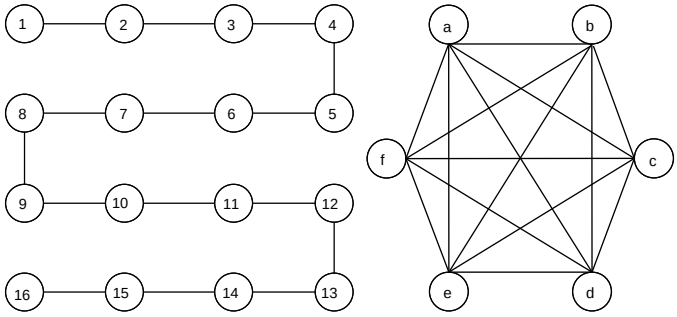
[1]A k-fold cross-validation is preferable, but we adopt a simple method in order to speed up the learning process.

[2]Here we do not consider the case where (continously drawn) samples are likely to have correlated class labels, i.e., one sample is likely to have the same label as its previus sample.

### C. The Concept Clustering Algorithm

A basic problem of hierarchical clustering is to decide which pair of clusters should be merged first, and how to find the final set of clusters when everything is merged. We address this problem in the setting of clustering concepts.

We propose a single algorithm to perform concept clustering in both steps. Our algorithm is based on the agglomerative hierarchical clustering method. The input to the algorithm is a set of nodes, each node representing a data block (step 1), or a data chunk (step 2). Initially, each node is a cluster by itself. The algorithm then iteratively merge small clusters into big clusters.




(a) Clustering Blocks (b) Clustering Chunks

Fig. 2: Input to the Clustering Algorithm

Figure 2 shows two set of inputs to the clustering algorithm

(corresponding to two steps respectively). We use an edge between two nodes to indicate that the two nodes can be merged. Web In step 1, edges are added between only neighboring nodes (data blocks), which means each cluster is made up of contiguous data, while in step 2, the input is a complete graph, meaning any two nodes (data chunks) can be merged to form a concept. This unifies the two clustering problems into one framework.

*1) Order of Merging:* Various strategies are available for deciding which pair of clusters should be merged first. One of the best method is to choose the merger that results in the least increase (or greatest decrease) in $Q$. Assuming we merge two clusters $D_u$ and $D_v$ into $D_w$, we can compute the increase of $Q$ as follows:

$$\begin{aligned}\Delta_Q(u,v) &= |D_w| \cdot Err_w - |D_u| \cdot Err_u - |D_v| \cdot Err_v \\ &= |D_u| \cdot (Err_w - Err_u) + |D_v| \cdot (Err_w - Err_v)\end{aligned} \tag{2}$$

This strategy finds the local optimal partition for each merger. However, in order to compute $\Delta_Q(u,v)$, we need to know $Err_w$, which is not available unless we train a classifier on $D_u \cup D_v$. Consequently, for a complete graph, in order to find the best merger among $n$ clusters, we need to train and test $n(n-1)/2$ classifiers, one for each possible merger, which is time consuming. Hence, this strategy is employed only for step 1, in which case each cluster has only two involved edges.

We propose another strategy for step 2 based on model similarity. Assume there is a function $\mathrm{sim}(M_u, M_v)$, which measures the similarity between the models $M_u$ and $M_v$ learned from cluster $D_u$ and $D_v$, then we can express the "distance" between the two clusters by:

$$\mathrm{dist}(u,v) = |D_u| \cdot (1 - \mathrm{sim}(M_u, M_v)) + |D_v| \cdot (1 - \mathrm{sim}(M_u, M_v)) \tag{3}$$

We assess the similarity between two models by checking to what extent they agree with each other when classifying a sample dataset $S$.

$$\mathrm{sim}_S(M_i, M_j) = \frac{|\{x \in S | M_i(x) = M_j(x)\}|}{|S|} \tag{4}$$

where $M_i(x)$ is $x$'s class predicted by base model $M_i$.

A problem may arise in choosing the sample test data $S$. To make results consistent, all comparisons must be based on sample datasets of the same distribution. If the sample set is too large, it consumes too much time to measure the similarity. If the sample set is too small, the result may be inaccurate. We solve this problem by using a dynamically formed sample set.

Let $D_i^{test}$ be the data left out for testing at node $D_i$. Before performing any mergers, we gather the test data $\{D_i^{test}\}$ from all nodes, and randomly shuffle them into a list $L$. Let $L_k$ denote the top $k$ records of $L$. Clearly, $L_k$ is a random sample of the original dataset.

Each time two clusters are merged to a new cluster $D_u$, we train a new model $M_u$ from the merged data, and we use $M_u$ to predict records in $L_k$, where $k = |D_u^{test}|$, and store its predictions in an array $A_u[1, \cdots, k]$. To measure the similarity between clusters $D_u$ and $D_v$, we compare the predictions stored in $A_u[1, \cdots, i]$ and $A_v[1, \cdots, i]$, where $i = \min(|D_u^{test}|, |D_v^{test}|)$. Thus, there are no more than $i$ comparisons for one evaluation of the similarity function, and the number of predictions is minimized by sharing the same sample dataset.

Based on above discussion, we use Esq. 2 (step 1) and Eq. 3 (step 2) to find the closest pair of clusters and merge them. Our experiments show that these strategies work well in practice.

In our implementation, we store candidate mergers in a linked list, and a min-heap is maintained to manage all candidate mergers along with their distances as keys, so that we can find the best candidate merger by picking the top element from the heap in logarithmic time at each step. After each merger, the candidate mergers and the heap are updated by linking the neighbors of old clusters to new ones and recalculating their distances. The mergers are repeated until the list is empty, or only one cluster remains.

*2) The Final Cut:* A dendrogram is a tree that shows the order of merging. Figure 1 shows two dendrograms, one of merging data blocks, the other of merging data chunks. After merging is complete, we cut off some of the last mergers (root and nodes close to the root of the dendrogram) to find the best partition that gives the smallest $Q$.

Each node in the dendrogram represents a dataset, with the root node representing the entire dataset, and each of its descendant a subset of the entire dataset. For each node $w$, let $D_w$ denote the dataset represented by $w$. We define $Err_w^*$ as the error of the local optimal partition, that is, $Err_w^* = \frac{1}{|D_w|} \min\{Q(P) \mid \text{P is a partition of } D_w\ \}$.

Let $P_w$ be the optimal partition of $D_w$, i.e., $Err_w^* = \frac{1}{|D_w|} Q(P_w)$. By the property of our clustering algorithm, if $D_w$ is merged from $D_u$ and $D_v$, then $P_w$ is either a partition consisting of the sole member $D_w$, or an union of $P_u$ and $P_v$, the best partition for $D_u$ and $D_v$.

This enables us to compute $Err_w^*$ in the process of merging as follows. If $w$ is a leaf node, we have:

$$Err_w^* = Err_w$$

where $Err_w$ is the error of the classifier built on $D_w$, otherwise assuming $w$ has two child nodes $u$ and $v$, then

$$Err_w^* = \min\left\{Err_w, \frac{|D_u| \cdot Err_u^* + |D_v| \cdot Err_v^*}{|D_w|}\right\}$$

After we build the complete dendrogram, we perform the final cut from top to bottom. Since $Err_w^* = \frac{1}{|D_w|} Q(P_w)$ for every cluster, the best partition $P_w$ can be established by checking values of $Err_w^*$ and $Err_w$. In words, we split the nodes in the dendrogram from top to bottom. Initially, the root node is the sole member of the current partition. For each node $w$ in the partition, if $Err_w^* < Err_w$, it is split into the two corresponding child nodes in the dendrogram, and the current partition is updated by replacing $w$ with its child nodes. The splits are repeated until $Err_w^* = Err_w$ for every remaining node $w$ in the partition.

Note that we cannot cut during the process of merging by simply stop merging a node $w$ with other nodes if $Err_w^* < Err_w$ because the value of $Q(P)$ is not monotonically decreasing in the process of merging.

### D. Complexity Analysis and Optimizations

The concept clustering algorithm is executed offline. The most time consuming part is building classifiers and measuring classifiers' similarity. More specifically, the algorithm preforms $n-1$ mergers and trains $O(n)$ classifiers, where $n$ is the number of nodes. Ideally, if every merger is between two clusters similar in size, then the total time is logarithmically linear to the size of the dataset.

There are several opportunities for optimization. First, mergers in the final stage of clustering are usually wasted: the clusters to be merged are of different concepts, and their merger will be discarded in the final cut. However, these mergers are the most time consuming ones, because clusters in the final stage are on the same scale in size as the entire dataset. We can optimize by terminating the merging in advance when we are confident that no additional mergers can help finding better partitions. For example, we can remove all candidate mergers with a cluster $u$ if $u$ contains at least 2000 records and $Err_u$ is at least 20% greater than $Err_u^*$.

Second, if clusters to be merged are always unbalanced in size, time complexity will become higher (unless the base classifier supports incremental learning). More specifically, if the dendrogram is a balanced tree, the aggregated size of data in all of the mergers is $O(n)$. However, in the extreme case when small clusters successively merge with a large cluster, the dendrogram will be extremely unbalanced, and this number becomes $O(n^2)$. Fortunately, it is highly unlikely that concept clustering will result in an extremely unbalanced dendrogram. If occasionally we do need to merge a large cluster with a very small one, a possible optimization is to simply reuse the existing classifier from the large cluster.

## III. Applying the High-order Model

In this section, we discuss how to use the high-order model trained offline to perform online prediction.

### A. Overview

The input to the classifier is a stream of unlabeled data $X = \{x_1, \cdots, x_t\}$, where $x_t$ is a record (or a set of records) of timestamp $t$. Because the concepts in the stream are changing, a stream of labeled data $Y = \{y_1, \cdots, y_t\}$ is also made available. We assume records with the same timestamp (e.g. $x_i$ and $y_i$) are generated by the same mechanism. In practice, $Y$ is usually created by labeling a subset of $X$ online. For instance, in financial fraud detection, a small subset of transactions are investigated and labeled. Thus, the labeled data usually lags behind the unlabeled data due to the labeling overhead, but the lag can usually be ignored with regard to the rate of concept change. In our analysis, we assume the prediction of $x_t$ makes use of labeled data $\{y_1, \cdots, y_{t-1}\}$.

Given that we already have the high-order model, which contains all historical concepts, our primary task in online prediction is to use $Y$ to identify the current concept in $X$, and use the classifier that corresponds to the concept to predict $X$, instead of learning new models from $Y$.

A naïve way to identify the current concept is to study the distribution of the labeled data in the most recent window of $Y$, and choose one of the known concepts that best fits the distribution. This approach has several problems. First, it will take at least one window of data to catch up with the concept change. Second, it is difficult to pick the window size. If the window size is too small, the system will be very sensitive to noisy or biased data, and if the window is too large, it will be unable to detect concept change in a timely manner.

We use a probabilistic method to identify the current concept — we compute the probability of each known concept being the *current* concept. We then ensemble the classifiers of these concepts weighted by these probabilities to classify the streaming data. To certain extent, we are training a Hidden Markov Model (HMM) from concept changing data streams. Each state of the HMM corresponds to a stable class distribution, or a concept, and the current observed data are generated based on the class distribution of the current state. Thus, given a sequence of observations, we can use a Viterbi-like algorithm to find the most likely sequence of underlying concepts. Since our goal is to accurately classify the current data, we only need to find out the current concept. In the following we present our approach of finding the current concept. We leave the study of the analogy between classifying concept shifting data stream and learning HMMs to future work.

### B. Predicting the Current Concept

Let $c$ be a concept. We define $c$'s *active probability* as the probability that $c$ is the current concept. We denote $c$'s priori active probability at time $t$ given data $\{y_1, \cdots, y_{t-1}\}$ as $P_t^-(c)$, and $c$'s posteriori active probability given data $\{y_1, \cdots, y_t\}$ as $P_t(c)$. More specifically,

$$P_t^-(c) = p(C_t = c | y_1, \cdots, y_{t-1})$$

$$P_t(c) = p(C_t = c | y_1, \cdots, y_t)$$

where $C_t$ is a random variable that denotes the concept at time $t$. Below, we omit $C_t$ for simplicity.

Our goal is to compute the active probability of each underlying concept, so that we know which concept is the most likely current concept.

- At time $t = 1$, we assign an equal active probability to every underlying concept:

  $$P_1(c) = \frac{1}{N}$$

  where $N$ is the total number of concepts.
- We estimate $P_t^-(c)$, the priori probability at time $t$ using the posteriori probabilities at time $t-1$:

  $$P_t^-(c) = \sum_i P_{t-1}(i)\chi(i, c) \tag{5}$$

where $\chi(i,c)$ denotes the probability of changing to concept $c$ when concept $i$ is previously active. Recall that $Len_i$ is the average lasting time (in records) of concept $i$, and $Freq_i$ is the frequency of concept $i$ in history, $\chi(i,c)$ can be computed as

$$\chi(i,j) = \begin{cases} 1 - \frac{1}{Len_i} & \text{if } i = j \\ \frac{1}{Len_i}\frac{Freq_j}{1-Freq_i} & \text{if } i \neq j \end{cases} \quad (6)$$

where $\frac{1}{Len_i}$ is the probability of concept change when $i$ is active, and $\frac{Freq_j}{1-Freq_i}$ is the probability of $j$'s being the next concept when concept changes from $i$.

- We compute $P_t(c)$, the posteriori probability of concept $c$, by combining $P_t^-(c)$ and labeled data $y_t$:

$$\begin{aligned} P_t(c) &= p(c|y_1, \cdots, y_t) \\ &= \frac{p(c|y_1, \cdots, y_{t-1})p(y_t|c, y_1, \cdots, y_{t-1})}{p(y_t|y_1, \cdots, y_{t-1})} \\ &= \frac{p(c|y_1, \cdots, y_{t-1})p(y_t|c)}{p(y_t|y_1, \cdots, y_{t-1})} \\ &\propto P_t^-(c)p(y_t|c) \end{aligned} \quad (7)$$

where $p(y_t|c)$ is the probability of generating record $y_t$ when $c$ is active. Since the historical dataset is limited, it is usually impossible to count the frequency for every possible $y_t$. Here, we estimate $p(y_t|c)$ using $M_c$, the base classifier trained from data of concept $c$. Let

$$\psi(c, y_t) = \begin{cases} 1 - Err_c & \text{if } M_c \text{ correctly classifies } y_t \\ Err_c & \text{if } M_c \text{ misclassified } y_t \end{cases} \quad (8)$$

Intuitively, if $y_t$ is generated by concept $c$, then $M_c$, the classifier for concept $c$, should classify $y_t$ correctly if $M_c$ is error free. We assume that $p(y_t|c)$ is approximatively proportional to $\psi(c, y_t)$ on each concept $c$. It follows that

$$\begin{aligned} P_t(c) &\propto P_t^-(c)p(y_t|c) \\ &\propto P_t^-(c)\psi(c, y_t) \end{aligned} \quad (9)$$

Now we obtain the value of $P_t(c)$ after we normalize the values of the right part to ensure that the sum of active probabilities equals to 1.

In summary, we compute the active probabilities in time linear to the number of underlying concepts. In the above reasoning, we assume that the rate of generating records is constant. If records are generated in variable rate, the equations can be easily revised and adapted to the situation. Also, Eq. 5 describes the concept-shifting scenario, but in practice, the result is also well applicable to concept-drifting data streams.

### C. Predicting Streaming Data

To classify an unlabeled record $x_t$, the simplest way is to choose the underlying concept that has the largest active probability, and use its classifier to predict the record. This method is feasible when there is always a clear current concept. However, during a concept change, the current concept may be a mixture of multiple stable concepts, thus the simple method will give sub-optimal results.

In this paper, we propose to predict the record by weighing underlying concepts by their active probabilities. Given an unlabeled record $x_t$, the high-order classifier outputs $\texttt{Highorder}(l|x_t)$, the probability that $x_t$ belongs to class $l$:

$$\texttt{Highorder}(l|x_t) = \sum_c P_t^-(c)M_c(l|x_t) \quad (10)$$

If a unique answer is desired, we select the class that has the largest probability:

$$\texttt{Highorder}(x_t) = \arg\max_l \texttt{Highorder}(l|x_t) \quad (11)$$

If there are a lot of records to be predicted (much more than the labeled records used to predict the current concept), and only the unique answer is required, we can speed up the prediction by pruning some underlying concepts whose active probabilities are too low. Specifically, for each unlabeled record, we enumerate the underlying concepts in decremental order of their active probabilities. Each concept's classifier is used to predict the record, and the overall classification result is updated. Once we find that the unique answer is impossible to be changed again by the remaining concepts, we terminate the enumerating and get the final answer. In usual situation, only one classifier is used to classify the record because the current concept is very clear.

## IV. Experiments

We conducted experiments on a variety types of concept-changing data. We compare the effectiveness and efficiency of our high-order approach against state-of-the-art methods, and demonstrate that our approach outperforms the others to a great extent.

### A. Data Streams

We employ three benchmark datasets, covering concept-shifting, concept-drifting, and sampling-changing data streams, in our experiments. Each dataset is partitioned into two parts, the first part is used as the historical dataset for training and the second part for testing.

*Stagger:* The Stagger dataset [15], [16], [17], [18], [13] simulates concept shift. Each record has three attributes: color $\in \{green, blue, red\}$, shape $\in \{triangle, circle, rectangle\}$, and size $\in \{small, medium, large\}$. There are altogether three concepts, A: class = $positive$ iff color = $red$ and size = $small$; B: class = $positive$ iff color = $green$ or shape = $circle$; C: class = $positive$ iff size = $medium$ or $large$. For any other case, class = $negative$.

The default frequency of concept change is $\lambda = 0.001$, i.e. there is a probability $\lambda$ to change the current concept before generating each record. The transition among concepts is controlled by the $z$ parameter of Zipf distribution, and in our experiments, $z$ is set to 1.

We use the Stagger data generator to randomly generate a total of 600 thousand records. The first 200 thousand is used for training, and the following 400 thousand for testing.

TABLE I: Benchmark Data Streams

| Data Stream | Stagger | Hyperplane | Intrusion |
|---|---|---|---|
| Continuous | 0 | 3 | 34 |
| Discrete | 3 | 0 | 7 |
| # of Concepts | 3 | 4 | Unknown |
| Historical Data | 200,000 | 200,000 | 1,000,000 |
| Test Data | 400,000 | 400,000 | 3,898,431 |

*Hyperplane:* The hyperplane dataset [1], [15], [2], [17], [6], [19], [13], [5] simulates concept drift by continuously moving a hyperplane in a $d$-dimensional space. Records are randomly generated and uniformly distributed in $[0, 1]^d$. Those records satisfying $\sum_{i=1}^{d} a_i x_i \geq a_0$ are labeled as positive, otherwise negative. We choose the value of $a_0$ so that the hyperplane cuts the multi-dimensional space into two parts of the equal volume, that is $a_0 = \frac{1}{2}\sum_{i=1}^{d} a_i$. Thus, roughly half of the space is positive, and the other half negative.

We create four concepts in the data. Each concept is defined by a hyperplane selected randomly. The default frequency of concept change is $\lambda = 0.001$. When a concept changes, the hyperplane corresponding to the current concept slowly drifts to the hyperplane that embodies the next concept. This is done by slowly changing the values of $a_i$'s. The drifting finishes within an average of 100 steps, and the next concept becomes the current concept. This mechanism simulates a concept-drifting data stream.

We use the hyperplane generator to randomly generate a total of 600 thousand records. The first 200 thousand is used for training, and the following 400 for testing.

*Network Intrusion:* The network intrusion dataset [20] was used in the KDDCUP'99 competition. It can be used to simulate sampling change. The competition task was to build a network intrusion detector, a predictive model capable of distinguishing between "bad" connections, called intrusions or attacks, and "good", or normal connections. The dataset contains roughly 4.9 million records, which includes a wide variety of intrusions simulated in a military network environment. Different periods witness bursts of different intrusion classes. The first 1 million records are used as historical dataset, and the remaining about 4 million is the test dataset.

### B. Competitors

We compare our high-order model with two existing algorithms. In the experiments, all algorithms will first process the historical dataset to train a data stream classifier, and then evaluate their effectiveness and efficiency on the test dataset. The decision tree classifier C4.5 release 8 implemented by Quinlan [21] is used as the common base classifier for consistency. Algorithm parameters are tuned to achieve best possible effectiveness. All of the three algorithms, including the high-order model approach, are implemented in C++ and compiled in optimization configuration. The tests are conducted on a machine with two Pentium 4 2.80GHz CPUs and 1GB main memory.

*RePro:* RePro [13] is presumably the best existing algorithm on classifying concept-changing data streams. It represents the family of algorithms that remember historical concepts and reuse pre-learned patterns for reappearing concepts. RePro employs a trigger detection algorithm to detect concept change, and then it identifies whether a new concept is a reappearing one or a new one that requires learning. RePro also learns transition patterns among concepts to help prediction.

In our experiments, the trigger window size in RePro is set to 20, the stable learning data size is set to 200, the trigger error threshold is set to 0.2, and other three threshold parameters are set to 0.8.

*Weighted Classifier Ensemble (WCE):* WCE [6] is an alternative competitive algorithm, which represents a family of algorithms that use ensemble classifiers for prediction. It divides the recent window of data into sequential chunks of fixed size, and builds a classifier from each chunk. The classifiers in the ensemble are judiciously weighted based on their expected classification errors on the test data under the concept changing environment.

In our experiments, the size of the chunks in WCE is set to 100, and the number of chunks is set to 20.

TABLE II: Comparison in Error Rates

| Data Stream | Stagger | Hyperplane | Intrusion |
|---|---|---|---|
| High-order | 0.0020035 | 0.0254519 | 0.0000665 |
| RePro | 0.0275480 | 0.1882158 | 0.0011210 |
| WCE | 0.0584363 | 0.1141258 | 0.0014962 |

TABLE III: Comparison in Test Times (sec)

| Data Stream | Stagger | Hyperplane | Intrusion |
|---|---|---|---|
| High-order | 2.1467 | 3.2917 | 54.2448 |
| RePro | 3.1345 | 24.2116 | 182.8444 |
| WCE | 6.3360 | 9.9606 | 16.0634 |

TABLE IV: Building Phase in High-order Model

| Data Stream | Stagger | Hyperplane | Intrusion |
|---|---|---|---|
| Build Time | 12.9898 | 52.7291 | 714.1117 |
| # of Concepts | 3 | 4 | $11 \pm 2$ |

### C. Experimental Results

All the experiments are conducted for 20 runs to obtain the average performance. If possible, datasets of different content are randomly generated for each run.

*1) Comparison in Effectiveness and Efficiency:* Table II presents the average error rates for each algorithm on different data streams. One can find that the classification error of the high-order model is about one or two tenth of the error of the best of other algorithms. This result demonstrates that the high-order model approach is the clear winner of the algorithms on all three types of data stream.

Table III presents the test time for each algorithm. The test time is the sum of the time spent on classification and the time spent for additional online training (on the test datasets). This result shows that the efficiency of the high-order model is comparable with other algorithms, and it is even the best for some types of data streams. This is because the high-order model builds and uses only a small number of base classifiers to predict the test data, and the number of such classifiers

equals to the number of distinct stable concepts. Although the high-order model approach needs to use the ensemble of all classifiers in prediction, it avoids online training of new base classifiers or learning new patterns.

The accuracy of RePro is better than WCE for the Stagger dataset, however, its online prediction is very time consuming for the Hyperplane and the Network Intrusion data streams. This is due to the intrinsic problems of the RePro algorithm. RePro has a lot of user parameters, which depends on the characteristics of data streams. These parameter values affect how RePro detects concept changes, identifies reappearing concepts, learns and stores concepts, and chooses prediction model. A misassignment of the parameters will cause RePro to fail to manage historical concepts correctly. Actually, when the data streams become somewhat complicated, finding a perfect set of parameters becomes an impossible task. In that case, RePro may learn illusive concepts due to the noises or the biases in the data, which introduces more and more of historical concepts as time passes. This problem is concealed if the dataset is very small. Yet if the dataset becomes larger, a large set of historical concepts will certainly slow down online classification, which is why RePro becomes so inefficient in our experiments.

WCE is a relatively simple algorithm. Only recent patterns are stored and the number of base classifiers WCE holds is always constant. Since the size of a data chunk is usually very small, the complexity of each base classifier is very low and hence the classification is still fast. However, because WCE cannot remember the historical concepts and too few records are utilized to train a base classifier, the accuracy of classification becomes a major problem.

Table IV shows the time required to build the concept model from historical dataset. It is obvious that the building phase is much slower than the running phase (while processes the dataset of the same scale in size). However, because the building phase is only required to run once, the time required by the building phase is still acceptable.

*2) Impact of Changing Rate:* We also studied the impact of the rate of concept changing on the effectiveness and efficiency of the stream classifiers. We generated data streams with varied concept changing frequency and examined their performance. Figure 3 shows the results. Note that the X-axis indicates the reciprocal of changing rate, which equals to the average length of a concept appearance. It can be observed that for RePro and WCE, when the changing rate increases, the error rate rises significantly. This is because when the concept changes frequently, RePro and WCE must update the prediction model frequently, which means they may miss the latest concept all the time. However, the high-order model approach maintains low prediction error in all situations.

The test time of RePro increases rapidly when the changing rate increases. This is due to the fact that RePro learns new patterns and enumerates every historical concept to find reappearing concept at each time concept changes. Larger changing rate also gives RePro more chances to learn illusive concepts from noisy or biased data and expanding the set of historical concepts, which results in more and more time being required to process a concept change. The test time of WCE is reduced when the changing rate increases. This is resulted from the instance-based pruning method used in the WCE algorithm. Larger changing rate reduces the number of useful base classifiers in predicting the next record and hence the classification time is reduced. The test time of the high-order model approach only depends on the number of concepts and the classification time of the base classifiers, consequently the changing rate has no impact on the test time.

*3) Effects of the Scale of the Historical Datasets:* We also examined how historical dataset size would affect the time used to build a high-order model, as well as the accuracy of the model. The experimental results are showed in Figure 4. The results justify the use of larger historical datasets, as they help to improve the quality of base classifiers, which is one of the reasons why the high-order model approach is so outstanding. More specifically, for the Stagger data stream, since the concepts are very simple, once the historical dataset is sufficiently large to contain all the concepts, the classification accuracy quickly becomes optimal. For Hyperplane, since the class distribution is modeled by a hyperplane, which is difficult to be modeled by a decision tree (we used C4.5 for base classifiers), a large training dataset could help reduce the classification error. In summary, when the size of the historical datasets increases, more data can be utilized to train base classifiers and the error rate gradually drops.

Figure 4 also shows the time to build a high-order model with varying historical dataset size. It testifies that the build time of high-order models is near linear to the size of the historical dataset. Finally, it shows that the impact on test time quickly decays with the increasing size of the historical dataset.

*4) Catching up with Concept Change:* Figure 5 illustrates how classification error varies during the concept change for each algorithm. The error rates are tested for $1000$ runs and the average results are evaluated. In general, the high-order model approach is able to catch the current concept promptly in all cases, while other algorithms always have a large lag on catching up.

For the Stagger data streams, once the concept changes, the high-order model quickly reduces the error rate to zero after seeing a few records. RePro has to wait for data to fill a "trigger window" before it detects the concept change. Occasionally, if the error rate in the first trigger window following the concept change does not exceed the threshold, or the trigger window contains data of mixed concepts or bias data, RePro has to wait more time to revise the prediction model. WCE always recovers from the concept change about one chunk of data after concept change, because WCE never revises the prediction model before the current chunk of data becomes full.

For the Hyperplane data streams, the concept does not change instantly. Instead, the current concept will gradually drift from the previous concept to the next concept, which is achieved by continuously moving the hyperplane which mod-

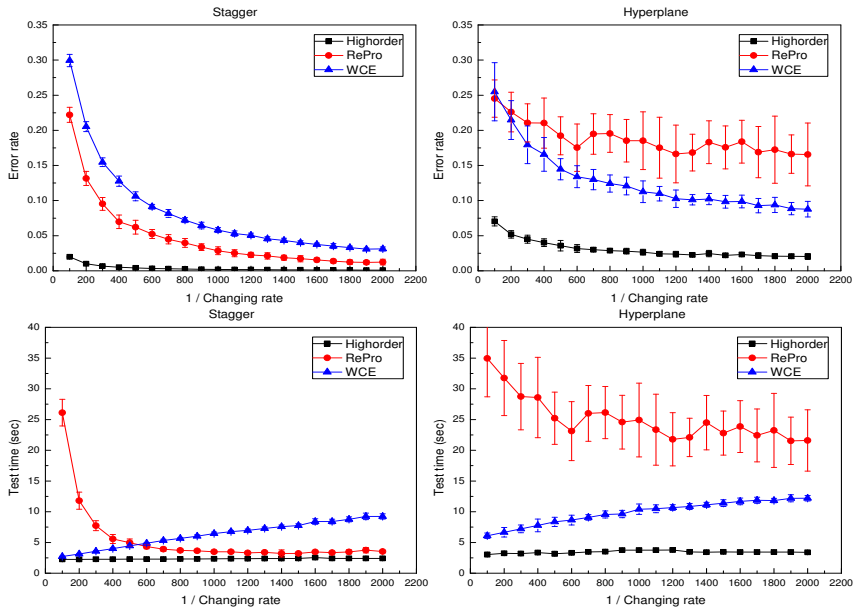



Fig. 3: Impact of Changing Rate

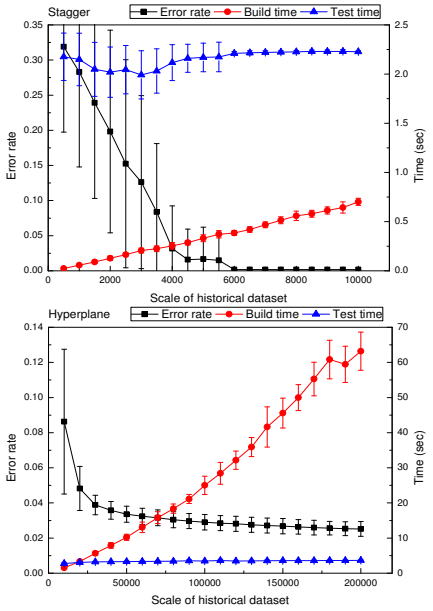



Fig. 4: Impact of the Scale of the Historical Datasets

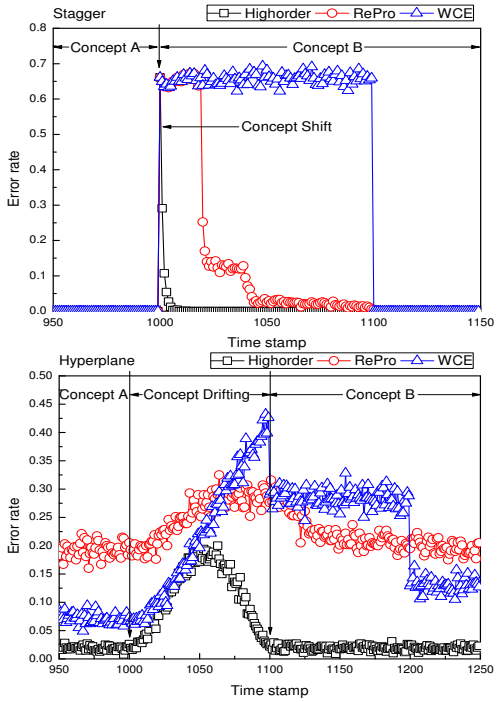



Fig. 5: Error Rates during Concept Change

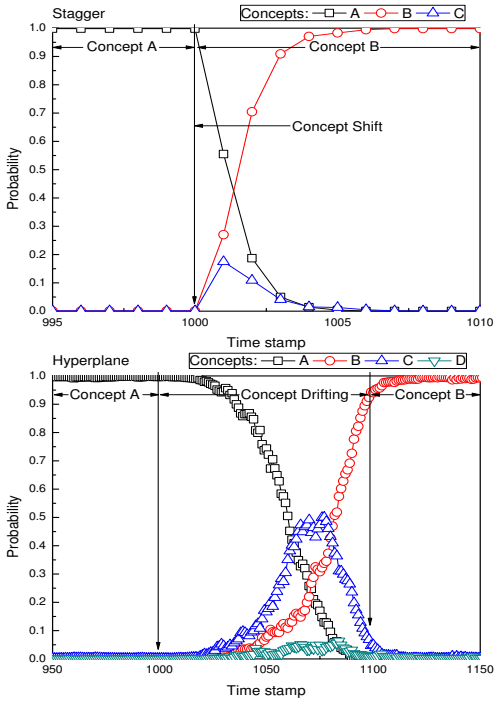



Fig. 6: Probabilities of Stable Concepts during Concept Change in High-order Model

els the class distribution. In our experiments, the drifting will be completed in 100 steps for each concept change. The error rate of the high-order model reaches the peak in the middle of the drifting interval, and it swiftly returns to the optimal range when drifting completes and the concept becomes stable again. Because the decision tree classifier requires a large amount of training data to get a good classification accuracy for the hyperplane dataset, RePro is not able to gather sufficient data to build base classifiers, and therefore the error rate of RePro is quite large all the time. Besides, RePro cannot update the prediction model in time after the drifting completes. WCE is able to ensemble a number of previous learned classifiers for prediction, thus the error rate of WCE is somewhat better than

RePro. However, WCE has a large lag to revise the prediction model after concept change.

Figure 6 demonstrates how the probabilities of stable concepts vary during concept change for the high-order model approach. The probabilities are tested for 100 runs and the the average results are evaluated. For the Stagger data streams, the high-order model approach can determine the current concept just a few records after concept change. For the Hyperplane data streams, during the concept drifting, the historical concept which is mostly similar to the current drifting concept has the largest probability. These enable the high-order model to preserve the optimal classification accuracy all the time.

## V. Conclusions

This paper advocates exploiting historically trained models to improve the qualities of stream classifiers for evolving data. Using a novel concept clustering algorithm, we collect all existing concepts in the evolving data. We then predict the current concept by evaluating the possibility for each concept being the current one. We report significant improvement of classification accuracy, as we are the only approach that manages to use all data scattered in the stream but pertaining to a unique concept to train a model for that concept. This solves the fundamental problem of model over-fitting in classifying data streams.

## References


[1] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *SIGKDD*. San Francisco, CA: ACM Press, 2001, pp. 97–106. [Online]. Available: citeseer.nj.nec.com/hulten01mining.html

[2] W. N. Street and Y. Kim, "A streaming ensemble algorithm (SEA) for large-scale classification," in *SIGKDD*, 2001. [Online]. Available: citeseer.nj.nec.com/489183.html

[3] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang, "Multi-dimensional regression analysis of time-series data streams," in *VLDB*, Hongkong, China, 2002.

[4] S. Guha, N. Milshra, R. Motwani, and L. O'Callaghan, "Clustering data streams," in *FOCS*, 2000, pp. 359–366.

[5] H. Wang, J. Yin, J. Pei, P. S. Yu, and J. X. Yu, "Suppressing model overfitting in mining concept-drifting data streams," in *SIGKDD*, 2006.

[6] H. Wang, W. Fan, P. S. Yu, and J. Han, "Mining concept-drifting data streams using ensemble classifiers," in *SIGKDD*, 2003.

[7] Y. Chi, P. S. Yu, H. Wang, and R. Muntz, "Loadstar: A load shedding scheme for classifying data streams," in *SIAM Data Mining*, 2005.

[8] Y. Chi, H. Wang, and P. S. Yu, "Loadstar: Load shedding in data stream mining," in *VLDB*, 2005, pp. 1303–1305.

[9] P. Wang, H. Wang, X. Wu, W. Wang, and B. Shi, "On reducing classifier granularity in mining concept-drifting data streams," in *ICDM*, 2005.

[10] P. Chan, "An extensible meta-learning approach for scalable and accurate inductive learning," Ph.D. dissertation, Columbia University, 1996. [Online]. Available: citeseer.ist.psu.edu/article/chan96extensible.html

[11] H. S. Seung, M. Opper, and H. Sompolinsky, "Query by committee," in *Computational Learning Theory*, 1992, pp. 287–294. [Online]. Available: citeseer.ist.psu.edu/seung92query.html

[12] T. G. Dietterich, "Ensemble methods in machine learning," *Lecture Notes in Computer Science*, vol. 1857, pp. 1–15, 2000. [Online]. Available: citeseer.nj.nec.com/dietterich00ensemble.html

[13] Y. Yang, X. Wu, and X. Zhu, "Combining proactive and reactive predictions for data streams," in *SIGKDD*, 2005, pp. 710–715.

[14] J. Yang, W. Wang, and P. Yu, "Discovering high order periodic patterns," *Knowledge and Information Systems Journal (KAIS)*, vol. 6, no. 3, pp. 243–268, 2004.

[15] J. Z. Kolter and M. A. Maloof, "Dynamic weighted majority: A new ensemble method for tracking concept drift," in *ICDM*, 2003.

[16] K. O. Stanley, "Learning concept drift with a committee of decision trees," in *Technical Report AI-03-302, Dept. of Computer Sci., Uni. of Texas at Austin, USA*, 2003.

[17] A. Tsymbal, "The problem of concept drift: definitions and related work," in *Technical Report TCD-CS-2004-15, Dept. of Computer Sci., Trinity College Dublin, Ireland*, 2004.

[18] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," in *Machine learning*, 1996.

[19] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *VLDB*, 2003.

[20] KDDCUP-1999, "The third international knowledge discovery and data mining tools competition," http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 1999.

[21] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.


## Appendix

Algorithm 1 outlines the concept clustering algorithm used by high-order model without code for optimization. Most details are omitted to make the pseudo code more readable.

**Algorithm 1** Concept Clustering

**Input**

$V$: A set of nodes

$E$: A set of possible joins

$D_u$: Labeled records contained by every node $u \in V$

**Output**

$P$: A set of clusters denoting the final partition

**Algorithm**

// Initialize clusters
**for all** $u \in V$ **do**
  $D_u^{train} \leftarrow$ random half records of $D_u$
  $D_u^{test} \leftarrow D_u - D_u^{train}$
  $M_u \leftarrow$ train a classifier from $D_u^{train}$
  $Err_u \leftarrow$ the error rate of $M_u$ classifying $D_u^{test}$
  $Err_u^* \leftarrow Err_u$
// Repeat joining pairs of clusters
**while** $E \neq \emptyset$ **do**
  $(u, v) \leftarrow \arg\min_{(u,v)\in E} \mathrm{dist}(u, v)$
  $w \leftarrow$ a cluster joined from $u$ and $v$
  $V \leftarrow V \cup \{w\} - \{u, v\}$
  $E \leftarrow E - \{(u, v)\}$
  $D_w \leftarrow D_u \cup D_v$
  $D_w^{train} \leftarrow D_u^{train} \cup D_v^{train}$
  $D_w^{test} \leftarrow D_u^{test} \cup D_v^{test}$
  $M_w \leftarrow$ train a classifier from $D_w^{train}$
  $Err_w \leftarrow$ the error rate of $M_w$ classifying $D_w^{test}$
  $Err_w^* \leftarrow \min(Err_w, \frac{|D_u|\cdot Err_u^* + |D_v|\cdot Err_v^*}{|D_w|})$
  **for all** $x : (x, u) \in E \vee (x, v) \in E$ **do**
    $E \leftarrow E \cup \{(x, w)\} - \{(x, u), (x, v)\}$
// Select the best partition through dendrogram
$P \leftarrow V$
**while** $\exists w \in P : Err_w^* < Err_w$ **do**
  $(u, v) \leftarrow$ the pair of clusters joined into $w$
  $P \leftarrow P \cup \{u, v\} - \{w\}$
return $P$