Journal of Big Data

RESEARCH Open Access

Mining frequent itemsets from streaming transaction data using genetic algorithms

Sikha Bagui* and Patrick Stanley

*Correspondence:
bagui@uwf.edu
Department of Computer Science, University of West Florida, Pensacola, FL, USA

Abstract
This paper presents a study of mining frequent itemsets from streaming data in the presence of concept drift. Streaming data, being volatile in nature, is particularly challenging to mine. An approach using genetic algorithms is presented, and various relationships between concept drift, sliding window size, and genetic algorithm constraints are explored. Concept drift is identified by changes in frequent itemsets. The novelty of this work lies in determining concept drift using frequent itemsets for mining streaming data, using the genetic algorithm framework. Formulas have been presented for calculating minimum support counts in streaming data using sliding windows. Testing highlighted that the ratio of the window size to transactions per drift was a key to good performance. Getting good results when the sliding window size was too small was a challenge since normal fluctuations in the data could appear to be a concept drift. Window size must be managed in conjunction with support and confidence values in order to achieve reasonable results. This method of detecting concept drift performed well when larger window sizes were used.

Keywords: Frequent itemsets, Streaming data, Concept drift, Sliding window, Association rules mining, Genetic algorithms, Data mining

Introduction

Today's digital world is constantly generating data from traffic sensors, health sensors, customer transactions, and various other Internet of Things (IoT) devices. Continuous never-ending streams of Big Data are creating new sets of challenges from the perspective of data mining. Mining only static data in snapshots of time is no longer useful. Streaming data, being dynamic or volatile in nature, has changing patterns over time, and this is more technically known as concept drift. Algorithms developed for mining streaming data have to be able to detect and work with concept drifts, hence the need for new streaming data mining approaches. This work looks at an important data mining technique, frequent itemset mining, applied to streaming transaction data, in the presence of concept drift.

Frequent itemset mining, a precursor to association rule mining, typically requires significant processing power since this process involves multiple passes through a database, and this can be a challenge in large streaming datasets. Though there is great deal of progress in finding frequent itemsets and association rules in static or permanent databases [1, 8], since the landscape of data is changing, algorithms have to be effectively extended to streaming data with concept drift. There are a number of questions that must be answered in order to explore this: (i) Can frequent item sets be mined while the streaming data is appearing without reference to a permanent master database; (ii) Are there techniques to reduce the time complexity of mining frequent itemsets in order to manage dynamic data; (iii) Are there challenges that appear as a result of a streaming environment? To address the issue of time complexity, the use of genetic algorithms has been considered; this technique has shown to reduce time complexity of mining frequent itemsets and association rules in static databases [8].

Using genetic algorithms to mine frequent itemsets in streaming data with concept drift is relatively unstudied, and there are a number of critical issues that must be considered. One critical issue specifically explored in this work, is the relationship between the rate of concept drift in streaming data and the convergence of a genetic algorithm used to mine frequent itemsets from that data.

In this study, genetic algorithms are used to mine frequent itemsets from streaming data in the presence of concept drift, with the aid of sliding windows. The novelty of this work lies in determining concept drift using frequent itemsets for mining streaming data, using the genetic algorithm framework.

Testing is done using a test harness, allowing for streaming data as needed. The test harness also allows the control of some of the fundamental variables, including the data being streamed, the number of transactions before a concept drift is introduced, and the size of the sliding window. The other variables, which are related to the genetic algorithm, include population size, crossover probability, mutation probability, and the convergence rate of the genetic algorithm.

The rest of this paper is presented as follows. To provide the background and context, in the introduction, a high-level overview of streaming data in a sliding window, concept drift, sliding window frequent itemsets, and finally genetic algorithms, is presented. The following section presents the related works. "Methods" section presents a detailed discussion of the proposed approach. Next, an example that illustrates how the proposed approach is different from the standard Apriori algorithm is presented. This is followed by the results and discussions, which includes a discussion of the points of interest that were uncovered during testing. Finally, the conclusion and future works sections are presented.

Streaming data in a sliding window

In most modern data environments, some static data store like a database is assumed, but this might not always be the case. Data can arrive at a computer port continuously and frequently. Further, the arrival of this data is a necessary part of a business model, much like the situation of ships bringing cargo to a port at a city. If the port tried to store all of the cargo, space could quickly become an issue. Also, while some cargo is typically stored, it must be processed and removed on average at least as quickly as the cargo arrives. Computer systems handling the persistent arrival have a similar problem regarding arrival, space and processing.

The processing that needs to be done on streaming data most often involves extracting some useful information from the data. There are some unique constraints to consider when dealing with streaming data, for example, data could be limited to one-time access, data could be unbounded in volume, and real time response may be necessary for the data to be useful [7, 12, 13, 22]. These works discuss models for processing streaming data, one of which is the sliding window model. The sliding window keeps track of current transactions, letting the oldest transaction to be deleted when the new transaction arrives. Hence, the newest *n* transactions always appear in the sliding window, where *n* is the sliding window size.

Figure 1 illustrates a sliding window of size 5. {Item B, Item D, Item E} are no longer in the window. This data has been deleted. The boxed area presents the current window. The new transactions are the two transactions above the window. They have not yet been included in the window. Once {Item B, Item C} has been included, {Item C, Item D} will be removed.

Concept drift

Mining information from static data results in static information. In streaming data, the information learned is not static. It is dynamic. When the nature of information in data changes over time, this is called a concept drift. It may also be important to capture the drifting of a concept. In this paper, concept drifts are measured in terms of frequent itemsets, by change in frequent itemsets. Other works have also looked at using frequent itemsets for mining streaming data [7]. Gama [7] mentions that the most difficult problem in mining frequent itemsets from streaming data is that itemsets that were infrequent in the past might now become frequent, and itemsets that were frequent in the past might become infrequent. Hence, in addition to concept drift detection, management of concept drifts are also an important aspect of mining streaming data.

Sliding window frequent itemsets

An association rule is a conditional statement that says, if item A exists in a transaction then it is likely that item B is in that transaction. In association rule mining, support is defined as items found with some minimum frequency in the whole dataset. Confidence is defined as an item's frequency in relation to the set of items that contain the supported items. A frequent itemset is an itemset that meets the minimum support threshold. The

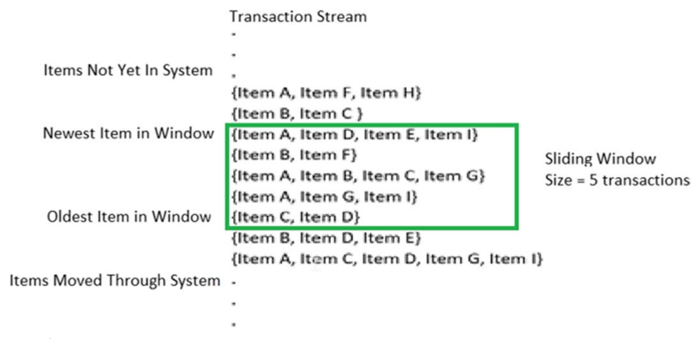


Fig. 1 Sliding window

challenge is to find frequent itemsets in sliding windows of streaming data. Before presenting the formulas that were used for calculating support counts in sliding windows, the background on the general Apriori algorithm is presented.

Given:

- sliding window length = 20
- minimum support = 0.3
- minimum confidence = 0.6.

And,

- 6 transactions contain (A), hence support is 6/20.
- 8 transactions contain (B), hence support is 8/20.

And, A and B appear together in a transaction 4 times, then:

- support (A ⇒ B) = 4/20 or 0.2.
- confidence (A ⇒ B) = P(B|A) = (support_count(A ∪ B))/(support_count(A)) = 4/6 or 0.66.

Confidence (A ⇒ B) > minimum confidence (of 0.6), so this is a strong association rule.

Hence, for static data, a frequent itemset will be any itemset that has support > 0.3.

For streaming data, a sliding window is used, hence the modified support_count to determine frequent itemsets is:

$$support\_count = \left(size\ of\ sliding\ window * minimum\ support\right)$$

Hence the sliding window frequent itemset is:

$$Support(A \Rightarrow B) > \left(size\ of\ sliding\ window * minimum\ support\right)$$

The genetic algorithm support_count will be:

$$support\_count = \left(size\ of\ sliding\ window * support * confidence\right)$$

Hence, the itemset (A ∪ B) will be a considered genetic algorithm frequent itemset when:

$$Support\ (A \Rightarrow B) > \left(size\ of\ sliding\ window * support * confidence\right)$$

The metric used in the fitness function of this genetic algorithm is the genetic algorithm frequent itemset inclusion threshold:

$$\left(size\ of\ sliding\ window * support * confidence\right)$$

Genetic algorithms

The genetic algorithm mimics the action of natural selection. First, the structure of the genes for an individual are defined. These are portrayed as a list of 0 s and 1 s. The individuals are then evaluated for fitness using a fitness function. Then the best individuals are crossed, that is, some of their genes are swapped. Next the individuals are mutated, that is, one or more of their genes might be randomly flipped from 1 to 0 or vice versa. Once the genetic operators have been used to define a new population, the fitness function is applied again. This process repeats until a successful individual is found or the maximum number of generations is found.

Related works

Mining association rules and frequent itemsets have a well established history and in fact [15, 20, 22] discuss algorithms to accomplish these tasks in the context of streaming data. The accumulative model, the sliding window model, and the weighted accumulative model are presented by Yu and Chi [22] as ways of handling streaming data. The accumulative model and weighted accumulative models keep all the data in a data store. However, in the weighted accumulative model, data becomes less important as it gets older. In Krempl et al. [13] 's discussion of the challenges facing this kind of research is a lack of commonly accepted set of environments, protocols and benchmarking methods necessary for testing and comparison. In the context of mining frequent itemsets, Gama [7] states that concept drift is the most difficult problem. Hoens et al. [10] presents a method of detecting and adapting to concept drift while focusing on classification learning. Kim and Park [11] and Wang and Abraham [21] also consider the detection of concept drift.

Bull [4] considers genetic algorithms as one of the most useful optimization algorithms, but only if the function being optimized is cheap to calculate. Rabinovich and Wigderson [17], on the other hand, believes they can be used on hard practical problems. There has been a good deal of research concerning the convergence rate of genetic algorithms. Forrest and Mitchell [5], Ruholla and Smith [19], Aldallal [2], Lin et al. [14], Angelova and Pencheva [3] and Pellerin et al. [16], all consider ways to improve the convergence rate in genetic algorithms. These studies affect the convergence rate by manipulating the genetic operators. According to Aldallal [2], the most common metric of convergence in a genetic algorithm is the number of generations required for an individual to meet the fitness requirements. However, He and Lin [9] considers the "normalized geometric mean of the reduction ratio of the fitness difference per generation" as a better metric that is still easily calculable. Ghosh et al. [8] and Rangaswamy and Shobha [18] discuss mining frequent itemsets and association rules with genetic algorithms.

While all of the individual pieces related to this work have been addressed independently, there is no research directly addressing all these issues together. That is, there is no work addressing the problem of mining frequent itemsets from streaming data with genetic algorithms in the presence of concept drift, the problem being addressed in this work.

Methods

This section offers a detailed description of the operations of the proposed system.

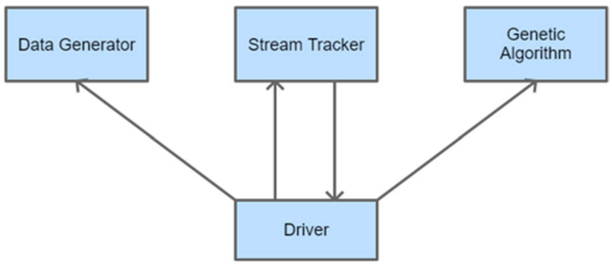


Fig. 2 System overview

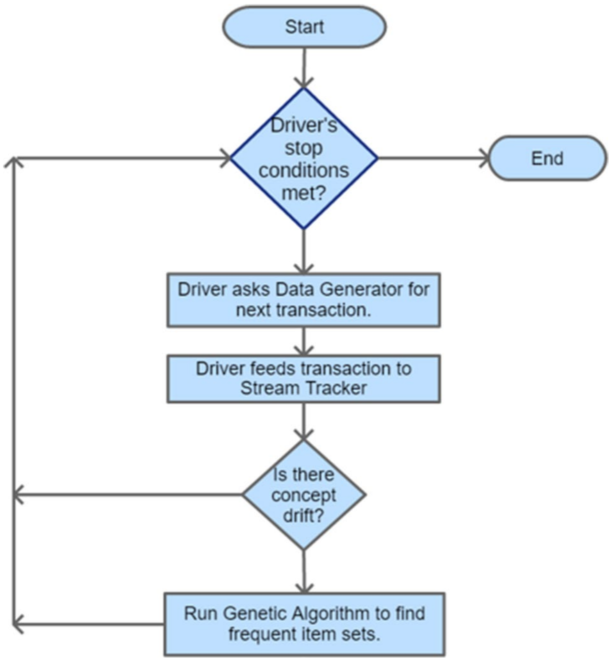

Start
Driver's stop conditions met?
End
Driver asks Data Generator for next transaction.
Driver feeds transaction to Stream Tracker
Is there concept drift?
Run Genetic Algorithm to find frequent item sets.

Fig. 3 System operations

Overview

In order to study the dynamics of this problem, a system was developed that includes a data generator, a genetic algorithm [6], a stream tracker and a driver, as shown in Fig. 2. The driver is the main instrument that sets all the necessary variables. The driver asks the data generator for the next item, passes the item to the stream tracker, detects the concept drift and starts the genetic algorithm when needed.

The workflow, presented in Fig. 3, can be presented as follows.

- The driver gets a transaction from the data generator.
- The driver enqueues that transaction in the stream tracker.
- The driver gets a reference to the stream tracker and initializes the genetic algorithm with that and the required support and confidence.
- On initialization, the genetic algorithm makes a copy of the queue to be used for further processing during the fitness evaluations.

- When the driver detects a concept drift, it resets the queue in the genetic algorithm before starting the genetic algorithm's find frequent items function.

Concept drift detection

In streaming data, frequent itemsets for a stable concept would be identified by the set of frequent itemsets remaining constant in both number and content, despite data flowing through the window. In other words, if the set of itemsets changes in any way, then it will be considered as a concept drift occurring.

The driver detects concept drift by keeping a copy of the stream tracker's set of potential frequent items. Each time an item is added to the queue, the new potential frequent itemset is compared with the old one, and if it has changed, the concept has potentially changed. This, along with a check that the frequent itemset has at least two members and that the sliding window is full, triggers the driver to start the genetic algorithm.

Data generator

The data generator continuously emits single transactions when called. It takes one argument, the number of items that it is required to emit, before switching to a different concept. It does this by cycling through and returning elements of one text file of transactions until the required number of emissions is met. It then gets a different file and does the same.

Sliding window model: stream tracker

In a specialized queue, the stream tracker keeps track of the number of transactions specified as the size of the sliding window. When a transaction is enqueued, if the required size of the window is met, the oldest item is automatically dequeued. When a transaction is enqueued, the individual items in the transaction are added to a dictionary with the counts in the sliding window. On dequeuing a transaction, the item counts are decremented. This allows the stream tracker to return a set of items that could potentially be considered part of a set of frequent items based on their frequency in the sliding window. This set is the template for a potential individual in the genetic algorithm. Any itemset in the dictionary with a count greater than the genetic algorithm frequent itemset inclusion threshold (*size of sliding window*support*confidence)* is the set of potential frequent itemsets. If a frequent itemset's count is lower than that calculation, then the item cannot possibly be part of a genetic algorithm frequent itemset.

Genetic algorithms

A genetic algorithm is an algorithm that mimics a natural selection. As presented in Fig. 4, it:

1. Generates a random population of individuals.
2. Evaluates each individual's fitness using a fitness function.
3. Applies a mating operation based on a crossover probability and with more fit individuals having a higher chance of being chosen for mating.
4. This creates a new population.

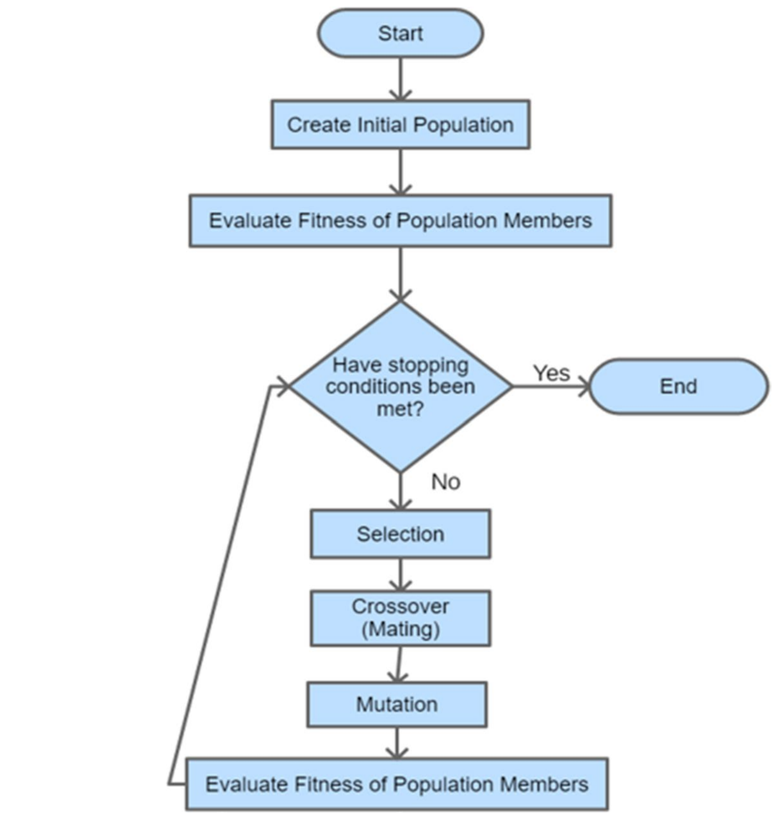

Start
Create Initial Population
Evaluate Fitness of Population Members
Have stopping conditions been met?
Yes
End
No
Selection
Crossover (Mating)
Mutation
Evaluate Fitness of Population Members

Fig. 4 Operations of the genetic algorithm

5. The new population is subjected to mutation with each gene's chance of being mutated being determined by the mutation probability, usually a very low chance.
6. The new population's fitness is evaluated. If the fitness of a best individual has not reached some desired level, and the maximum number of generations has not been reached, then the process starts over at step 3.

To implement the genetic algorithm, the open source library, Distributed Evolutionary Algorithms in Python [6] was used. This framework allows the user to easily define the parts of a genetic algorithm that are fairly common, like the definition of an individual as a list of 0 s and 1 s with each 0 or 1 representing a single gene. At the same time, it provides the flexibility to defining the methods and registering them with the framework as needed to define the required genetic algorithm.

Individuals and genes

In this implementation, the individuals were made up of a list of 0 s and 1 s that map to the sorted list of possible frequent items from the stream tracker, for example:

Frequent items: [apple, banana, candy, diapers, soap]
Random Individual: [0,1,1,0,1]
Gives a potential frequent itemset consisting of {banana, candy, soap}

Population size

To find multiple sufficient itemsets rather than just one best itemset, it is important to maintain a population of unique individuals. Hence the population size was based on the potential itemset's length or some maximum size. The rule used for the size of the population was:

$$minimum(2^{(length\_of\_genes-1)}, 100).$$

This kept population sizes down while ensuring unique individuals throughout the population.

Fitness function

To determine the fitness of an individual, the fitness function would get the items that were mapped to the 1 s in the individual, then loop through a snapshot of the queue checking to see if the set of the individual's items were a subset of each transaction and keep count of this. The final fitness function was:

$$count\ of\ transactions/size\ of\ sliding\ window.$$

Tournament selection for mating

Tournament selection for mating involves randomly choosing a small number of individuals (in this case, 3) and taking the one with the highest fitness and putting it in the new population. This was done without removing the original individuals from the old population.

Two point crossover

Once individuals were chosen for mating, they were subjected to a two-point crossover operation. This involved finding a random middle section from two individuals and swapping the sections. Two random numbers were chosen, the first greater than the 0 index and then from that index to the second to last index in the list. Once the middle section was found, it could be swapped with the same section from the other individual (see Fig. 5).

Mutation

The last genetic operation that was applied before re-evaluating each individual's fitness was the mutation operator (see Fig. 6). Usually, a random number is generated for each bit in the gene and if the random number is lower than the mutation

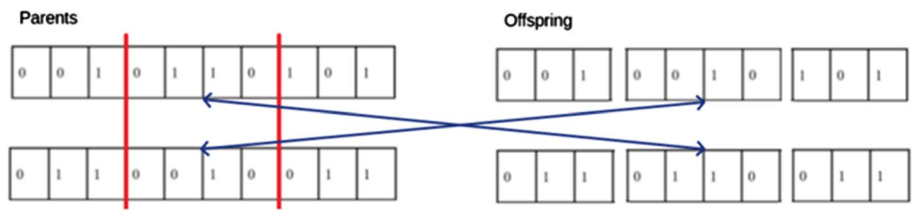


Fig. 5 Crossover operation

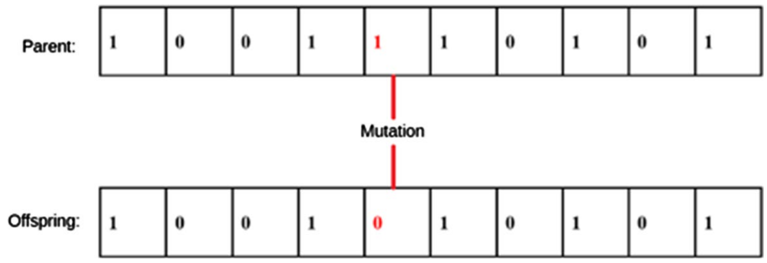
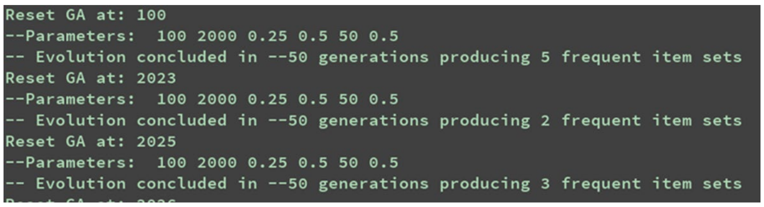


Fig. 6 Mutation operation



```
Reset GA at: 100
--Parameters:  100 2000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 5 frequent item sets
Reset GA at: 2023
--Parameters:  100 2000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 2 frequent item sets
Reset GA at: 2025
--Parameters:  100 2000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 3 frequent item sets
```

Fig. 7 Screen example

probability then the bit is flipped. The mutation probability was often very low, that is, under 5%. However, due to the uniqueness requirements, the mutation operator was used to remove duplicates from the population. Each new individual that was being added to the population was converted to a string and if that string was already in a temporary set of strings, the individual was mutated again. Though data was not kept on the mutation rate, spot-checking indicated that this resulted in an unusually high rate of mutation.

Evaluation procedure

Testing this application involved setting various variables in the driver and capturing the output stream and evaluating the resulting output. For example, Fig. 7 shows the parameters at this point:

- Sliding window size: 100 transactions
- Concept drift every: 2000 transactions
- Support: 0.25
- Confidence: 0.5
- Number of generations: 50
- Crossover probability: 0.5.

When the sliding window reached capacity at 100 transactions, the genetic algorithm evolved for 50 generations and was able to find 5 frequent itemsets. The concept was detected as stable until transaction 2023, where the algorithm was triggered again but only 2 itemsets were found. At transaction 2025, the algorithm was once again triggered and 3 itemsets were found.

A comparative example

This section presents an example of the Apriori algorithm working in a sliding window, and then presents the genetic algorithm's generation of frequent itemsets using a sliding window. When sliding windows are used in Apriori, support_count is defined as the size of the sliding window multiplied by the minimum support; when genetic algorithms are used in frequent itemset calculations, Apriori's support_count is modified by multiplying the confidence to Apriori's support_count. This makes the genetic algorithm's frequent itemsets more inclusive. That is, even if an itemset was not included as a frequent itemset using the Apriori approach, it would be included as a genetic algorithm frequent itemset. This is illustrated next—first the frequent itemset generation for the Apriori algorithm with the sliding window is presented and then the genetic algorithm frequent itemset generation with the sliding window is presented.

The Apriori frequent itemsets with the sliding window

The Apriori algorithm is the standard algorithm for mining frequent itemsets. Frequent itemsets can be further processed to generate association rules. First, a definition of some terms in relation to the Apriori algorithm: A transaction is one line or row in the dataset. An itemset is any subset of the set of all individual items. *K-itemset* is an itemset that contains *k* items. Support is the count of the number of transactions in which a particular itemset can be found. Support is also represented as the percent of transactions in which a particular itemset can be found. Confidence is used when determining association rules. Confidence is the probability of a subset of transactions occurring given some particular itemset. Support and confidence are defined by conditions of the problem, and can be set as needed. A frequent itemset is any itemset where the support count is above some value.

Given a database of transactions:

{Item 2, Item 4, Item 6}
{Item 2, Item 4, Item 1, Item 5}
{Item 6, Item 4, Item 1, Item 3}
{Item 2, Item 6, Item 4, Item 1}
{Item 2, Item 6, Item 4, Item 3}

Apriori follows the following pattern:

- Generate all candidate 1-itemsets and calculate their support.
- Eliminate any 1-itemsets that do not meet the minimum support criteria.
- Combine the remaining 1-itemsets into all possible candidate 2-itemsets and calculate their support.
- Eliminate any 2-itemsets that do not meet the minimum support criteria.
- Combine the remaining 2-itemsets into all possible candidate 3-itemsets and calculate their support.
- Eliminate any 3-itemsets that do not meet the minimum support criteria.

- This is repeated until no new "candidate" itemsets can be created.

Given:

- Minimum support > 0.5
- Support_count = 2.5.

The support_count for the Apriori algorithm using a sliding window of 5 is calculated as:

$$Support\_count = size\ of\ sliding\ window * minimum\ support$$

which is

$$5 * 0.5 = 2.5.$$

Hence the support_count is 2.5.

Table 1 presents the general pattern of Apriori.

From Table 1: First the 1-itemsets are counted. Items 3 and 5 do not have the minimum support_count, hence are eliminated. 2-itemsets are created out of the remaining 1-itemsets. Sets {Item 1, Item 2}and {Item 1, Item 6} did not reach the required minimum support_count and hence are eliminated. Next the 3-itemsets are generated. There are 4 possible 3-itemsets and all are eliminated except {Item 2, Item 4, Item 6}. There are no possible 4-itemsets. This leaves a total of 9 frequent itemsets that meet the support criteria.

The genetic algorithm frequent itemsets with the sliding window

When genetic algorithms are used to create frequent itemsets, the first step is to search for 2-itemsets or more. The support_count criteria is also adjusted, as discussed previously. This is illustrated using the itemset {Item 1, Item 6} from the Apriori example above.

Given:

- Minimum support = 0.5
- Minimum confidence = 0.6.

For a rule: Item 1 ⇒ Item 6

Table 1 Apriori example

1-Itemsets	Count	2-Itemsets	Count	3-Itemsets	Count
Item 1	3	~~Item 1, Item 2~~	~~2~~	~~Item 1, Item 2, Item 4~~	~~2~~
Item 2	4	Item 1, Item 4	3	~~Item 1, Item 2, Item 6~~	~~1~~
~~Item 3~~	~~2~~	~~Item 1, Item 6~~	~~2~~	~~Item 1, Item 4, Item 6~~	~~2~~
Item 4	5	Item 2, Item 4	4	Item 2, Item 4, Item 6	3
~~Item 5~~	~~1~~	Item 2, Item 6	3		
Item 6	4	Item 4, Item 6	4		

This itemset {Item 1, Item 6} did not meet the criteria for a frequent itemset in the regular Apriori algorithm. But, Item 1 is in 3 transactions which satisfies the required support. And, of the 3 transactions that Item 1 appears in, Item 6 appears in 2 of them. That is, this rule has a confidence of 2/3 or 0.66, which exceeds the stated minimum of 0.6, thus this is a valid association rule.

The proposed genetic algorithm implementation uses the genetic algorithm frequent itemset inclusion threshold (*size of sliding window*support*confidence*) as the support_count required for inclusion as a valid frequent itemset. In this case, the support_count is:

$$5 * 0.5 * 0.6 = 1.5.$$

Hence, itemset {Item 1, Item 6} would be a genetic algorithm frequent itemset. The genetic algorithm frequent itemset inclusion threshold is set up in this manner to be able to include itemsets that contain both the consequent and antecedent of a potential association rule as part of the frequent itemset.

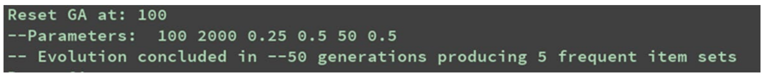
As transactions come in, a count is kept of how many transactions each individual appears in, and those that do not meet the minimum support count threshold are eliminated. In this example, Item 5 is eliminated from a possible mapping in the genome. Given that there are five eligible items in this dataset, the first step is to start by creating a random population of $2^{(5-1)}$ or 16 individuals, each with 5 genes and each gene mapping to one of the possible items.

Note: Not all individuals are included in the following example presented in Table 2.

[Item 1, Item 2, Item 3, Item 4, ~~Item 5~~, Item 6]

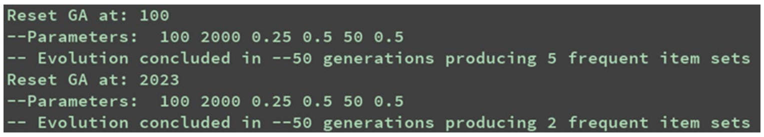
Table 2 Genetic algorithm example

Individual	Maps to	Fitness is checked	Select for mating	New Individuals	Mutate New Individuals	Maps to	Fitness
[01101]	[Item 2, Item 3, Item 6]	1	Individuals 2, 3	[0**100**1] Child 1	[010*0*1]	[Item 2, Item 6]	3
[00011]	[Item 4, Item 6]	4	Individuals 2, 3	[1**001**0] Child 2	[10010] No genes mutated	[Item 1, Item 4]	3
[11000]	[Item 1, Item 2]	2	Individuals 2, 4	[00011] No crossover. Crossover probability not exceeded Individual 2 carried to the next generation	[0*0*011]	[Item 4, Item 6]	4
[01010]	[Item 2, Item 4]	4	Individuals 2, 4	[01010] No crossover. Crossover probability not exceeded Individual 4 carried to the next generation	[01010] No genes mutated	[Item 2, Item 4]	4



```
Reset GA at: 100
--Parameters:  100 2000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 5 frequent item sets
```

Fig. 8 Maximum generations



```
Reset GA at: 100
--Parameters:  100 2000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 5 frequent item sets
Reset GA at: 2023
--Parameters:  100 2000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 2 frequent item sets
```

Fig. 9 Concept drift detection

The above process is repeated until some stopping condition is met. This stopping condition includes a maximum number of generations or some fitness condition. Only the generation of frequent itemsets is shown, and not the association rules that would follow.

Results and discussions

Test results have been presented for: (i) maximum generations; (ii) concept drift detection; (iii) varying sliding window sizes; and finally, (iv) some examples of ideal operations.

Maximum generations

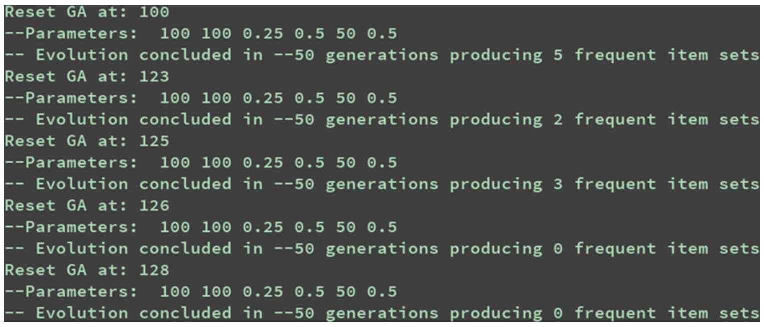
Figure 8 presents the results of maximum generations. The parameters listed in Fig. 8 are (in order):

- Sliding window size: 100 transactions
- Concept drift every: 2000 transactions
- Support: 0.25
- Confidence: 0.5
- Maximum number of generations: 50
- Crossover probability: 0.5.

Concept drift detection

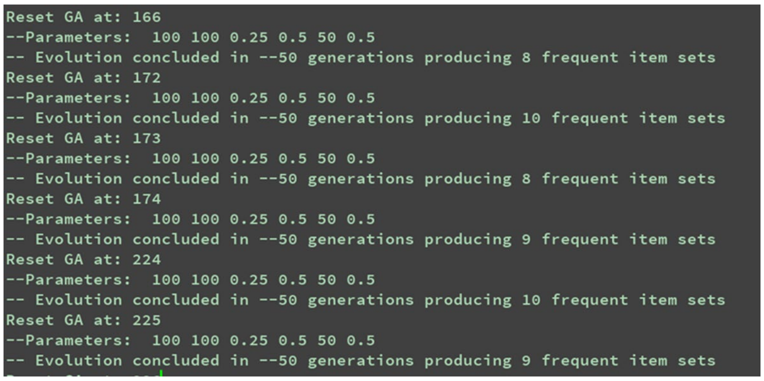
Detecting concept drift involved keeping track of a list of items that had occurred more times than the minimum support count. These counts were kept in a dictionary and updated for each new transaction, and then the dictionary was looped over to create a new 'possible itemset list'. The new list was compared to the old list, and if they were different, then the concept was considered to be drifting. This approach required a dictionary update for each item in a new transaction plus a loop through all of the items in the dictionary to assemble the new list. For these tests, this method of detecting concept drift triggered as expected and performed well, especially when a larger window size was used. Figure 9 presents concept drift detection.

From Fig. 9, a stable concept at transaction number 100 can be detected. No work is required until transaction 2023, when a new concept is first detected. At this point the



```
Reset GA at: 100
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 5 frequent item sets
Reset GA at: 123
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 2 frequent item sets
Reset GA at: 125
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 3 frequent item sets
Reset GA at: 126
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 0 frequent item sets
Reset GA at: 128
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 0 frequent item sets
```

Fig. 10 Small concept drift illustration, illustration 1



```
Reset GA at: 166
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 8 frequent item sets
Reset GA at: 172
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 10 frequent item sets
Reset GA at: 173
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 8 frequent item sets
Reset GA at: 174
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 9 frequent item sets
Reset GA at: 224
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 10 frequent item sets
Reset GA at: 225
--Parameters:  100 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 9 frequent item sets
```

Fig. 11 Small concept drift illustration, illustration 2

whole second data file (new concept) is loaded into the sliding window, while 77% of the sliding window still contains the old concept.

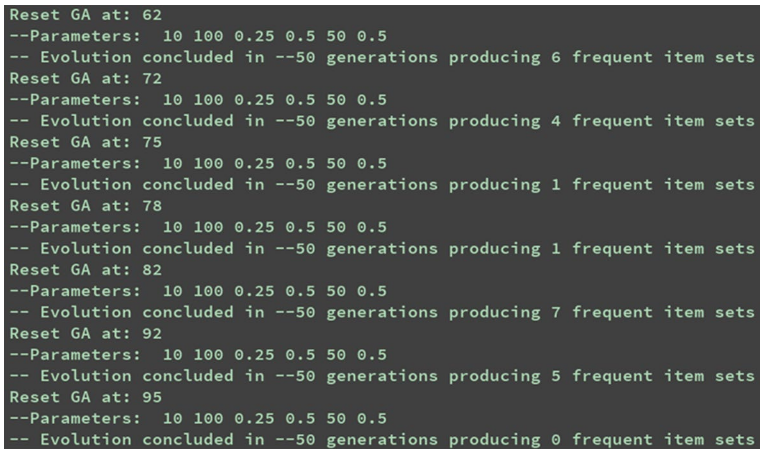
This testing highlighted that the ratio of the window size to transactions per drift was a key to good performance. This is because, if a fixed random window size is used, as transactions per drift get larger, the data from the perspective of the sliding window begins to look static. An example of this can be seen in Fig. 9 which shows a concept drift of 2000 transactions.

In Fig. 10, both the sliding window and the transactions per drift are set to 100. As expected, with the new data filling the window immediately after filling up with the first dataset, there is no stability. Figure 11 shows the next data change and the instability in the number of itemsets found during these repeated runs. Also, in this run, the genetic algorithm was called to run about 60 times during 400 transactions with only 27 runs producing any itemsets at all.

Also, there were problems getting results when the sliding window size was too small. When the support count was very small, that is, 0, 1, 2, there were problems because of normal fluctuations in the data appearing to be a concept drift.

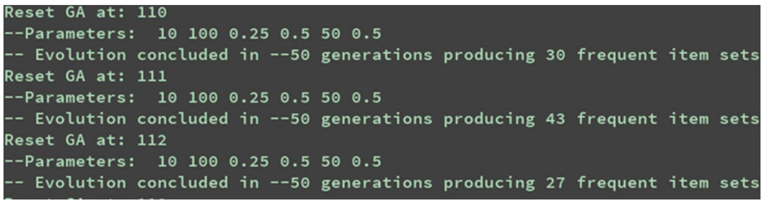
Varying sliding window sizes

In Fig. 12 the window size is 10 and the concept drift is 100. Looking at a region of concept drift triggers between transactions 72 and 95, the concept should become stable.



```
Reset GA at: 62
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 6 frequent item sets
Reset GA at: 72
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 4 frequent item sets
Reset GA at: 75
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 1 frequent item sets
Reset GA at: 78
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 1 frequent item sets
Reset GA at: 82
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 7 frequent item sets
Reset GA at: 92
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 5 frequent item sets
Reset GA at: 95
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 0 frequent item sets
```

Fig. 12 Small sliding window size, illustration 1



```
Reset GA at: 110
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 30 frequent item sets
Reset GA at: 111
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 43 frequent item sets
Reset GA at: 112
--Parameters:  10 100 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 27 frequent item sets
```

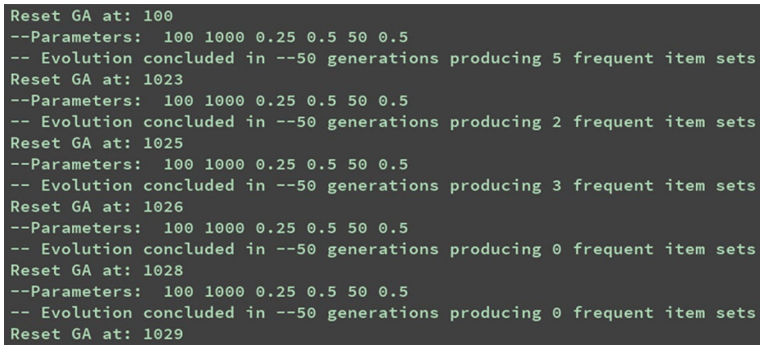
Fig. 13 Small sliding window size, illustration 2

The frequent itemset support count for this would be $10 * 0.25 * 0.5 = 1.25$. Hence, almost every item could be part of a frequent itemset, and a large number of frequent itemsets can be generated, as can be seen in Fig. 13. These sets have no informative value. This shows that, using this technique, the window size must be managed in conjunction with support and confidence values in order to achieve reasonable results.

Examples of ideal operations

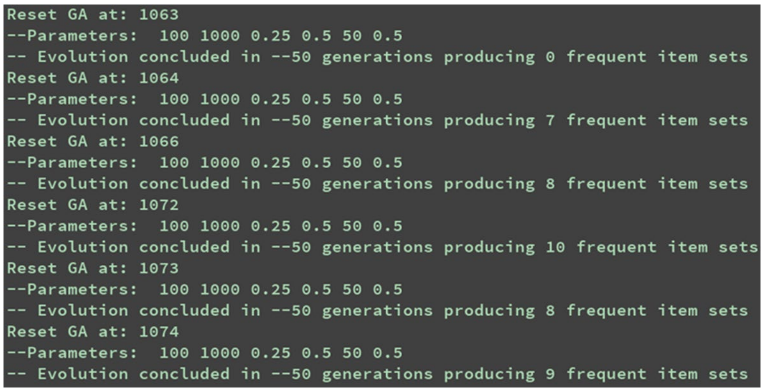
Under ideal operations, while the genetic algorithm was being repeatedly triggered during a high rate of concept drift, the number of valid frequent itemsets generated would drop to zero, and then as the new data began to fill the window, the itemset count would rise to the number of frequent itemsets expected for the new dataset. The genetic algorithm would not be triggered again until the concept detector began to trigger. Also, while this algorithm generated a very good set of rules, it would sometimes miss one, as would be expected by the random process of a genetic algorithm.

Figure 14 presents a stable concept at transaction 100 with 5 sets detected. This situation is stable until transaction 1023, where the concept is starting to drift and the 5 rules diminish to 0 rules by transaction 1026. From transaction 1026 to 1063 the genetic algorithm is triggered 11 times with no frequent sets detected. Figure 15 presents the algorithm being triggered and finding 7 then 8, 10, 8 and finally 9 frequent itemsets. The concept is stable until transaction 2024. Note that the stable concept of 9 itemsets at transaction 1074 is less than the 10 sets found just 2 transactions earlier. Since the



```
Reset GA at: 100
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 5 frequent item sets
Reset GA at: 1023
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 2 frequent item sets
Reset GA at: 1025
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 3 frequent item sets
Reset GA at: 1026
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 0 frequent item sets
Reset GA at: 1028
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 0 frequent item sets
Reset GA at: 1029
```

Fig. 14 Ideal operation illustration 1



```
Reset GA at: 1063
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 0 frequent item sets
Reset GA at: 1064
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 7 frequent item sets
Reset GA at: 1066
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 8 frequent item sets
Reset GA at: 1072
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 10 frequent item sets
Reset GA at: 1073
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 8 frequent item sets
Reset GA at: 1074
--Parameters:  100 1000 0.25 0.5 50 0.5
-- Evolution concluded in --50 generations producing 9 frequent item sets
```

Fig. 15 Ideal operation illustration 2

window is filling up with data for this concept, generally, an increasing number of itemsets would be expected. However, genetic algorithms should not be expected to necessarily get all of the itemsets all of the time.

Changing the fitness function to allow early convergence would be a great improvement. One simple way to do this would be to allow some small number of rules relative to the total expected possible rules to constitute convergence.

Discussion

A few issues led to the mitigation of some of the expected control of the genetic algorithm. Initially the algorithm was converging quickly on 1 or 2 itemsets. This is the normal expected behavior for a genetic algorithm that is looking for the 'best' individual. However, the goal was to find as many of the frequent itemsets as possible. This convergence on the local maxima was due to the population becoming dominated by individuals with exactly the same gene itemset. This led to limiting the population size and altering the mutation function to ensure a population of unique individuals. This resulted in the loss of ability to control those two factors of the genetic algorithm. Since the stopping requirement was based on the weakest individual meeting the fitness requirement or the maximum number of generations being met, the result was that the maximum number of generations was always met. This implied that the variables of the genetic algorithm were not as free to be manipulated. The first conclusion is related to this diversity issue in the genetic population for this type of problem. The nature of this problem required finding all or nearly all of a set of solutions, however since the genetic algorithm tended to optimize to a particular solution, the makeup of the population was altered. To address this issue, all individuals in the population were required to be unique. The effect of this decision was twofold. First, this limited the population size to a maximum of $2^{(\text{number of possible items} - 1)}$, ensuring the possibility of a population of unique individuals, but taking away control of population size. Second, in order to ensure uniqueness in the population, the mutation operation was tracked, and if a new individual was a duplicate, then that individual was mutated. This implied that the mutation probability was no longer reflective of the actual mutation rate, nor was it a valid control mechanism. This left only the crossover probability, which seemed to have little effect on the operation of the genetic algorithm. The most concerning result of this development was that the genetic algorithm never converged before the maximum number of generations were concluded. Even in the best circumstance, when the window was initially filled with five copies of the same data file and no concept drift, the genetic algorithm was terminating at the maximum number of generations.

While the algorithm did offer good rules within certain parameters, testing was done in a very controlled environment. Under production conditions the performance could be very different. There are a number of things that could be done to help, for instance the fitness function could be parallelized, or, each time the genetic algorithm is started, the whole process could be started in its own thread. If the concept values have not changed, it might be possible to determine valid itemsets in live stream. This would obviate the need for the snapshot method being used now, but add the complexity of terminating a thread as 'failed' if the concept drifted while it was running. Also researching ways to help the genetic algorithm find multiple rules while still achieving convergence without the artificial 'maximum generations' stopping, would lead to better data on this subject.

Conclusion

This paper presents a study of mining frequent itemsets from streaming data. An approach using genetic algorithms is presented and various relationships between data stream drift rate (concept drift), sliding window size and genetic algorithm constraints have been explored. Useful formulas are presented for calculating minimum support counts for determining frequent itemsets in streaming data using sliding windows. And, detecting concept drift involved keeping track of a list of items that had occurred more times than the minimum support count.

Testing highlighted that the ratio of the window size to transactions per drift was a key to good performance. But, getting good results is difficult when the sliding window size was too small, because then normal fluctuations in the data appeared to be a concept drift. Hence, the window size must be managed in conjunction with support and confidence values in order to achieve reasonable results. This method of detecting concept drift performed well when larger window sizes were used.

Using the setup described, the important variables of the genetic algorithm were ultimately decided by the data at run time. Mutation rate, individual size, and population size were all beyond external control. Crossover was specified but had no effect on speed of convergence for the genetic algorithm, as the genetic algorithm always took the maximum number of generations while still achieving good results.

Future work

There are promising areas for future work. First, the genetic algorithm could be threaded allowing the stream to continue while the frequent itemsets are being generated. Second, the genetic algorithm could be altered to allow convergence on some small number of valid rules. This would allow convergence before the maximum number of generations. Third, improvements in and possible parallelization of the fitness function should be considered. Fourth, testing on live data could lead to insights that are not considered due to the nature of some particular test data. Further, buffering incoming data beyond the sliding window, within limits, might allow more time for the genetic algorithm to converge.

Acknowledgements
This work has been partially supported by the Askew Institute of the University of West Florida.

Authors' contributions
This work was mainly conceptualized by SB and PS. The programming was mainly done by PS. Both authors participated in the write-up of the paper. Both authors read and approved the final manuscript.

Authors' informations
Dr. Sikha Bagui is Professor and Askew Fellow in the Department of Computer Science, at The University West Florida, Pensacola, Florida. Dr. Bagui is active in publishing peer reviewed journal articles in the areas of database design, data mining, BigData, pattern recognition, and statistical computing. Dr. Bagui has worked on funded as well unfunded research projects and has numerous peer reviewed publications. She has also co-authored several books on database and SQL. Bagui also serves as Associate Editor and is on the editorial board of several journals.

Patrick Stanley holds a Master of Science from The University West Florida and currently works as a Full Stack Software Engineer for the University of North Florida.

Funding
This work was not funded.

Availability of data and materials
Not applicable.

Competing interests
None of the authors have any competing interests.

Received: 12 March 2020 Accepted: 17 July 2020
Published online: 25 July 2020

References

1. Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: Proceedings of the ACM SIGMOD international conference on management of data, Washington, D.C., USA. 1993. pp. 207–216.
2. Aldallal AS. Avoiding premature convergence of GA in informational retrieval systems. Int J Intell Syst Appl Eng. 2015;2(4):80. https://doi.org/10.18201/ijisae.78975.
3. Angelova M, Pencheva T. Tuning GA parameters to improve convergence time. Int J Chem Eng. 2011. https://doi.org/10.1155/2011/646917.
4. Bull AD. Convergence rates of efficient global optimization algorithms. J Mach Learn Res. 2011;12(88):2879–904.
5. Forrest S, Mitchell M. What makes a problem hard for a GA? Some anomalous results and their explanation. GAs Mach Learn. 1993. https://doi.org/10.1007/978-1-4615-2740-4_6.
6. Fortin F-A, De Rainville F-M, Gardner M-A, Parizeau M, Gagné C. DEAP: evolutionary algorithms made easy. J Mach Learn Res. 2012;13:2171–5.
7. Gama J. A survey on learning from data streams: current and future trends. Prog Arti Intell. 2012;1(1):45–55. https://doi.org/10.1007/s13748-011-0002-6.
8. Ghosh S, Biswas S, Sarkar D, Sarkar PP. Mining frequent itemsets using GA. Int J Artif Intell Appl. 2010;1(4):133–43. https://doi.org/10.5121/ijaia.2010.1411.
9. He J, Lin G. Average convergence rate of evolutionary algorithms. IEEE Trans Evol Comput. 2016;20(2):316–21. https://doi.org/10.1109/tevc.2015.2444793.

10. Hoens TR, Polikar R, Chawla NV. Learning from streaming data with concept drift and imbalance: an overview. Prog Artif Intell. 2012;1(1):89–101. https://doi.org/10.1007/s13748-011-0008-0.
11. Kim Y, Park CH. An efficient concept drift detection method for streaming data under limited labeling. IEICE Trans Inf Syst. 2017;100(10):2537–46. https://doi.org/10.1587/transinf.2017edp7091.
12. Kolajo T, Daramola O, Adebiyi A. Big data stream analysis: a systematic literature review. J Big Data. 2019;6:47. https://doi.org/10.1186/s40537-019-0210-7.
13. Krempl G, Zliobaite I, Brzezinski D, Hullermeier E, Last M, Lemaire V, Noack T, Shaker A, Sievi S, Spiliopoulou M, Stefanowski J. Open challenges for data stream mining research. ACM SIGKDD Explor Newsl. 2014;16(1):1–10. https://doi.org/10.1145/2674026.2674028.
14. Lin W-Y, Lee W-Y, Hong T-P. Adapting crossover and mutation rates in GAs. J Inf Sci Eng. 2003;19(5):889–903.
15. Nedunchezhian R, Geethanindhini K. Association rule mining on Big Data—a survey. Int J Eng Res Technol. 2016;5(5):42–6.
16. Pellerin E, Pigeon L, Delisle S. Self-adaptive parameters in Genetic Algorithms. In: Proceeings SPIE 5433. Data mining and knowledge discovery: theory, tools, and technology VI. 2004. https://doi.org/10.1117/12.542156.
17. Rabinovich Y, Wigderson A. Techniques for bounding the convergence rate of GAs. Random Struct Algorithms. 1999;14(2):111–38. https://doi.org/10.1002/(sici)1098-2418(199903)14:2%3c111:aid-rsa1%3e3.0.co;2-6.
18. Rangaswamy S, Shobha G. Optimized association rule mining using GA. J Comput Sci Eng Inf Technol Res. 2012;2(1):1–9.
19. Ruholla J-M, Smith BK. Fluid GA (FGA). J Comput Des Eng. 2017;4(2):158–67. https://doi.org/10.1016/j.jcde.2017.03.001.
20. Vijayarani S, Sathya P. An efficient algorithm for mining frequent items in data streams. Int J Innov Res Comput Commun Eng. 2013;1(3):742–7.
21. Wang H, Abraham Z. Concept drift detection for streaming data. In: 2015 international joint conference on neural networks (IJCNN). 2015. https://doi.org/10.1109/ijcnn.2015.7280398.
22. Yu PS, Chi Y. Association rule mining on streams. Encyclopedia of Database Systems. 2009. https://doi.org/10.1007/springerreference_63188.

Publisher's Note