National College of Ireland

Investigation Into Parameterisation of the FIMOTS Algorithm for Computational Efficiencies on FPM on Streams of Data

MSc Research Project
Master's in Data Analysis

Rory O'Loughlin
Student ID: 17132835

School of Computing
National College of Ireland

Supervisor: Jorge Basilio

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Rory O'Loughlin

Student ID: 17132835

Programme: Masters in Data Analytics **Year:** 2

Module: Masters Project

Supervisor: Jorge Basilio

Submission Due Date: 16/08/2021

Project Title: Investigation Into Parameterisation of the FIMOTS Algorithm for Computational Efficiencies on FPM on Streams of Data

Word Count: 8792 **Page Count** 25

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Rory O'Loughlin

Date: 11/08/2021

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□

Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Investigation Into Parameterisation of the FIMOTS Algorithm for Computational Efficiencies on FPM on Streams of Data

Rory O'Loughlin
Student Number: 17132835

Abstract

Frequent pattern mining (FPM) has long been a data science technique with applications in tendency analysis and association rules. Technological evolutions have meant that data is no available within streams and expectations exist for near real-time processing as opposed to the batch processing of static datasets from the past. This introduces complexity in how existing algorithms may maintain accurate results of frequent patterns as new transactions arrive to the stream and older ones become less important. Usually there is a trade-off between the accuracy of the results and the amount of processing and memory required to achieve those results. This paper investigates the FIMoTS algorithm for FPM on data streams and determines how the accuracy of the frequent patterns identified by it is impacted by the introduction of a parameter which reduces the amount of computation required. Depending on the datasets used, it has been found that a large reduction in the amount and time spent in computation can be achieved with no loss of precision and a minor (<1%) reduction in recall.

1 Introduction

With recent advances in technology such as social media, IoT and sensor technology the amount of valuable data which is now available for data mining is ever increasing. The volume of data being generated is so large that in many circumstances the storage requirements are so large that batch analysis may be impractical [1]. Also, in certain areas – such as economic and social media analysis – the expectation is for immediate or near real-time updates [2]. These requirements are shifting the trends for analysis to the incoming data streams in real-time instead of the classic approach of writing updates to a database and performing a batch process during and end of day window.

Frequent pattern mining is a technique commonly used to determine items occurring together most frequently within a database of transactions first introduced within [3]. The problem is defined thus: let $\boldsymbol{D}$ be the set of transactions ($\boldsymbol{T}$) and $\boldsymbol{I} = \{ i_1, i_2, i_3, \dots, i_m \}$ the set of all uniquely occurring items within $\boldsymbol{D}$. Each transaction $\boldsymbol{T}$ consists of a set of items such that $\boldsymbol{T} \subseteq \boldsymbol{I}$. Frequent pattern mining is concerned with finding collections of items (itemsets) which occur with a frequency greater than the *support s*, where *s* is usually a fixed percentage of $|\boldsymbol{D}|$. An itemset with *k* items is called a *k-itemset*. Clearly for $k > 1$ items are said to occur frequently together and from this associations may be identified (for example, the probability of purchasing item A if item B is purchased).

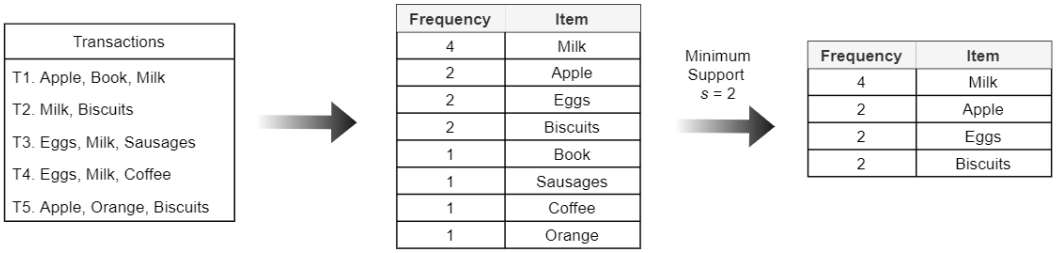

Figure 1: Example Support Calculation

The example from Figure 1 shows a support calculation for a list of grocery transactions. For a minimum support 40% ($5 * 40\% = 2$), milk, apple, eggs and biscuits are frequent items.
Within retail situations it is very effective at determining links between products often sold together and so advising on advertising or product placements to available of customers pre-dispositions to purchase certain items. The first step in this process is the identification of the commonly occurring itemsets within a transaction set. There has been much investigation in this area for static datasets – where all transactions are available within a database and algorithms seek to determine all itemsets above a defined *support* value with maximal efficiency. Some of the common algorithms include: Apriori [4] and FP-Tree [5].
Apriori was the initial algorithm proposed which is based on an iterative process of determining frequent k-itemsets, from these construct candidate N+1 itemsets and scanning the transaction database to identify which are frequent. FP-Tree is generally considered to out-perform the others in terms of computational efficiency as the algorithm avoids the expensive candidate generation step of Apriori by building a tree structure of all transactions in the database which may be mined to determine the frequency of itemsets.
The general understanding when these algorithms were introduced was that the datasets were static and so optimizations in terms of numbers of database scans and candidate generation were favoured. With frequent pattern mining on streams of data, the focus must extend to ensuring the state of the frequency can be incrementally updated efficiently, ideally without requiring the full sets of data to be retained within memory [6].
Figure 2 illustrates the problem when attempting to determine frequent itemsets within a stream of data. As the dataset moves from W_1 to W_2 and W_3, new transactions enter the dataset (e.g., in Period 6) and others leave (Period 1). The difficulty is updating the running list of frequent itemsets which is very different in nature to efficiently calculating this information from a static dataset. The Lossy Count[7] algorithm is an early example of an approximate method for calculating frequent itemsets within data streams where the amount of memory is limited. Thus, the pruning which does occur introduces the possibilities of errors which are managed within an error tolerance. Mining may result in false positives existing between the minimum frequency and an accepted error threshold. There are many similar algorithms which utilize approximates when managing resources whilst calculating frequent patterns within streams such as FP-Streams, estDec, SWP-Tree, CP-Tree.

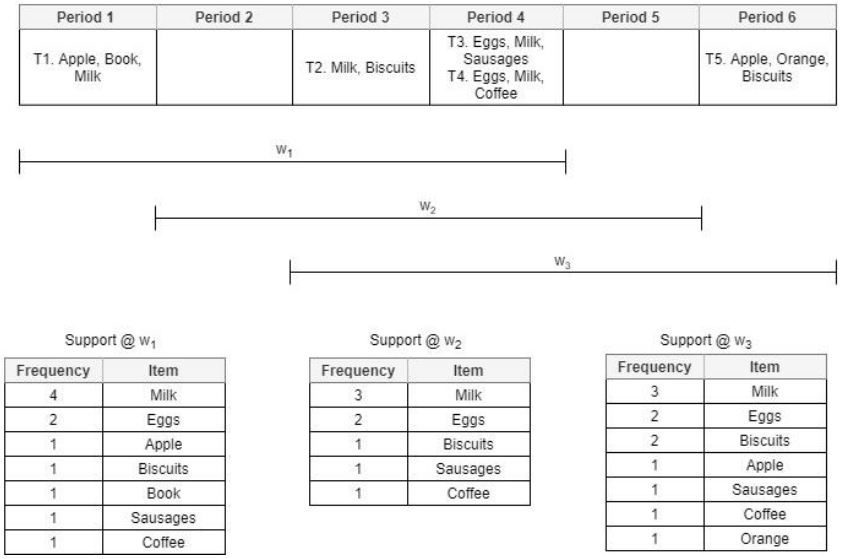

Figure 2: Support Calculation within Sliding Window

The FIMoTS algorithm [8] takes a novel approach of calculating exact frequent itemsets by only computing supports for itemsets which may have a frequency change as transactions change within the datastream. For example, if an itemset has a high support within the transactions of a data stream the arrival of a relatively small number of new transactions cannot change the itemset to infrequent and so it is unnecessary to calculate its support again. Clearly this approach implies that the stream of data must be recorded within memory scans over the entire active stream and to improve performance a Spark implementation has been proposed within [9].

The research question posed within this document is whether an element of approximation can be added to this algorithm to further improve the performance at the cost of recall or precision. This algorithm uses transforming bounds to determine when an itemset may start becoming frequent from infrequent and vice versa which are conservative (e.g., if all new items from the stream include this itemset could it start becoming frequent). Statistically this is very unlike to happen and incurs many redundant support checks. The aim of this analysis is to determine whether the transforming bounds may be parameterised to reduce the number of checks which are performed and while doing so what is the reduction in accuracy.

This paper is arranged as follows: Section 2 discusses related work in this topic and highlights why this is an area of interest, Section 3 introduces the datasets upon which the analysis is performed and how the results will be evaluated, Section 4 details the FIMOTS algorithm and the parameterisation implemented for these tests, Section 5 details the implementation of the algorithm and what changes we applied from the original, Section 6 presents the test results and analysis and finally Section 7 provides a summary of the work and what future work could be undertaken.

2 Related Work

2.1 Frequent Pattern Mining Algorithms

While the algorithms mentioned earlier focus on the topic of frequent pattern mining on a static set of transactions, within modern applications the need for flexibility to provide real-time response to changes in the dataset require a change in focus in the approach. Constraints exist such as:

- Storage capacity to retain the amount of data
- Processing capacity to be able to calculate results as new data arrives
- Expectation on response times for queries

This section introduces more modern algorithms which aim to tackle this problem taking different approaches – providing approximate or exact results.

2.1.1 Approximate Algorithms

One of the earliest proposals to treat this challenge was proposed in [10] which introduced the *Lossy Count* algorithm. This approach accepts that the memory required to track all items which may become frequent in the future is too large and uses a parameter to define the error threshold of results. There are no false negatives reported however false positives between the error threshold and the minimum support can exist, which correspond to items which were not tracked previously due to their low support possibly becoming frequent later. To maintain efficient storage of patterns a tree structure is used as in [4].

Another approximate method, *estDec* was introduced in [11] which once more seeks to provide a best case scenario for frequent patterns. As in *LossyCount*, an error parameter is introduced which to eject existing non-frequent items from the monitoring memory, and another for when to start inserting non-frequent but possibly frequent in the future items. Assumptions on the max occurrence of new items are taken (e.g., support of *n-itemset* cannot be greater than supports of it is *n-1* ancestor itemsets) to reduce the amount of error from this approach.

The FP Stream algorithm [12] is an application of FP-Tree aimed towards handling information within data streams. This approach takes a time-tilted windowing to compress information further back in time. As with other techniques in this section, it uses an error factor to determine when to prune non-frequent items which may become frequent in the future. As this method is very closely linked to FP-Tree which builds a frequent pattern tree based on the frequent patterns in the original data, it can suffer from inefficiencies if the content of the stream changes substantially over time. Also, as it creates an FP-Tree and FP-Stream structure its memory requirements are increased.

The GFCP method [13] constructs a pattern summarization tree (PS-Tree) to summarise the exists patterns within the stream efficiently without holding all transactional data. Frequency changing points and support estimation methods are used to determine historical support for newly frequent itemsets which were not previously monitored. Afterwards an FP-Growth-like algorithm is run to retrieve frequent itemsets from the PS-Tree structure. Despite efficiencies in maintaining the PS-Tree via a single scan for new transactions, the memory costs are still large and the support estimation methods result in reduced precision.

All approaches discussed thus far in the section are approximate algorithms which allow false positives. In [14], the authors argue the rationale for such an approach and favour of false-

negatives. A statistical approach is taken to determining the size of the infrequent items to keep given a supplied reliability parameter. This approach does rely on an assumption on the independence of the input data which may not be valid at all and so could limit its applications.
The approach in [15] allows either a false-positive or false-negative approach but keeping running totals of actual counts and possible counts of all potentially frequent itemsets. Any itemset which becomes potentially frequent may have counts from earlier in the stream which are not recorded which impact the possible count. This paper also introduces a merging operation to maintain memory size in its potential frequent itemset structure by merging similar itemsets. The impact of these techniques can reduce considerably the accuracy of results.
SWP Tree [16] is another approximate algorithm which introduces a decay factor to reduce the frequency of older items in the data stream and thus achieve a sliding window approach without necessarily needing to remove the counts of older transactions. The decay factor may be parameterised to achieve 100% recall. Like DSTree and CPSTree a FP-like structure is used which requires frequent pruning for itemsets becoming infrequent. Like FP-Tree an algorithm is needed when building the list of frequent itemsets as the raw data is not readily available within the SWP-Tree. A very similar approach to this is taken within [17] in the WMFP-SW-Tree algorithm although with the caveat of operating over *weighted* frequent patterns.
Within [18], a CP-Tree (compressible prefix tree) is introduced to address the memory issues concerns from the *estDec* method. The algorithm seeks to merge similar itemsets together to reduce the size of the tree structure, while maintaining a high degree of accuracy. A merging threshold is introduced to allow control over the degree of differences between items when merging occurs, which in turn impacts memory usage. Sizable memory reductions from estDec are attained for a moderate reduction in accuracy.

2.1.2 Exact Algorithms

Unlike the above, an exact algorithm (*SWF*) was introduced in [19] which uses an *Apriori*-based approach for candidate generation after iteratively determining the 2-itemsets frequent patterns. Each period in a sliding window is partitioned and processed separately. When a partition is too old to be including in the window the running counts of the 2-itemsets are updated and similar for new items in the most recent window. As with *Apriori* the candidate generation and evaluation for $n > 2$ itemsets can have a prohibitively large impact on performance for large datasets.
Another exact algorithm is DSTree [20], which constructs a data stream tree which includes all possible combinations within the data stream. The structure of the tree can be more concise than an FP-Tree however as there is no pruning due to minimum support the overall size can be very large. A key assumption being made is that with increases in size of memory available fitting the tree in main memory should not be prohibitive. The main technique applied is to maintain a list of counts over different time periods within the tree and as a sliding window is used, the first in the list can be removed as a new period is added. Unlike the FP-Tree, there is no arrangement of the tree according to which is most frequent so

another inefficiency can be in sparser tree structures. CPSTree [21], was introduced to improve upon some of the limitations of DSTree. In an effort to construct a denser tree structure, the CPSTree enforces a re-structuring after every new window which places the most frequent nodes at the top of the tree. While this operation can be time-consuming, this approach leads to less memory usage and improved processing time to the DSTree.
FIMoTS – the subject of this research paper – was introduced in [8] with the aim of reducing the number of queries required on sliding window data by determining which frequent patterns could become infrequent and vice-versa. Only itemsets which may change in nature are checked and then may trigger a cascade effect of pruning or candidate generation depending on whether they are determined to be frequent or infrequent. This approach assumes that full window data can exist within memory as unlike other approaches scans of all data in the sliding window are performed. To efficiently manage this data an extended FP tree maintains a timestamped version of the stream data, although little detail is given on how to efficiently delete expired transactions from this tree. Also, worth noting that while the accuracy of the itemsets declared frequent is exact, the support of each itemset while the algorithm is running is not as only mandatory itemsets have their supports re-calculated.
Based on the Eclat approach from [22], SWEclat follows a similar approach of constructing a vertical database structure and performing mining based on this. To do this, conditional vertical databases are constructed and distributed across a Spark grid. The content of these conditional databases is based on the frequently occurring items and it benefits from a high degree of parallelization. However, it is not clear how infrequent 1-itemsets can become frequent from the proposed algorithm as the vertical database only includes frequent items, not potentially frequent ones.

2.2 Windowing for Streaming Data

Some thought must be put into the windowing system which is adopted for treating incoming data to the data stream. Generally, the following are used:

- Landmark window
    - All data is taken from a fixed time point onwards.
- Decayed window
    - Earlier occurrences of itemsets have their weights reduced as the stream progresses, thereby favouring more recent trends.
- Sliding window
    - The start and end points of the window move together over time. Usually they are either time oriented (e.g., a window of 1 hour) or data oriented (e.g. 1000 transactions).

The type of window used is closely linked to the approach of the algorithm being adopted as often the methods of reducing memory or processing requirements are linked with the data is being maintained.
Landmark windows are used in Lossy Count [10] and [14] wherein it is considered that all data within a stream is of equal importance. These approaches can be more simplistic as they do not require the management of older data however can have functional limitations in applications where determine new frequent trends are required. The Landmark window was

the initial approach to these problems but is less common in more modern algorithms. A minor divergence to the Landmark window is the Time-tilted window introduced in [12] which keeps all past date but has a compression phase which merges earlier time periods together (for example 24 hours being managed as distinct time points becoming 1 day).
Decayed (or dampened) windows can be used to reduce the support counts for older items and thus prioritising itemsets which are new in the stream. Usually a dampening factor (<1) is applied to existing supports every time a new batch of transactions are received which implies eventually older occurrences of itemsets provide little benefit to support calculations. Approaches using this structure include [11], [16], [18], [23] and [24].
The most common approach is the usage of sliding windows where each subsequent time period or batch of transactions causes the oldest to be removed. Usually sliding windows using fixed time periods are preferred over those where the window is based on a number of transactions as this can lead to delays in new trends being raised when frequencies of arrivals are low. Sliding windows are used in all [15], [8], [17], [19], [20] and [21] and generally each new batch must include an update and a delete phase to manage both transaction inserted to the stream and those leaving.

2.3 Management of Stream Data

As indicated in [6], the expectation for modern algorithms is to have only one pass over streaming data without incurring preventative storage and/or processing difficulties. Older techniques on static datasets can run multiple iterations however in all cases here the assumption is that the raw data from which frequent itemsets are derived can be stored at once. Modern algorithms on streams must perform a trade-off between storing all data necessary for being exact and being able to process changes quickly.
Broadly there are the following approaches:

- Store all data within a tree structure
    - Generally exact algorithms follow this pattern [8], [20] and [21]
    - There can be divergences between storing all combinations (as in [20] and [21]) and all data as in [8]
    - The understanding that a tree structure is the most efficient method of storing the required information which is always necessary in case an infrequent pattern becomes frequent in the future
    - Can require extra work or structure to determine frequent itemsets from maintained data
- Store all frequent and some non-frequent patterns within a tree structure
    - Favoured by approximate algorithms - [10], [11], [12], [14] and [13]
    - Do not store the raw data in the stream, but rather immediately record it on a tree of patterns and use this tree as a running total of frequency
    - Issues can arise in how non-frequent itemsets can become frequent and often assumptions on the occurrences of itemsets before they started becoming monitored
    - Difficulties to ensure trees are well structured as frequent itemsets change over time

2.4 Summary

The choice of algorithm when performing frequent itemset mining on a data streaming should be closely linked to the business drivers for running this analysis. Most approaches accept an element of uncertain in exchange for efficiencies in their data storage. A large amount of effort is required in maintaining structure in memory which can provide frequent itemsets and be able to react quickly to new items arriving to the data stream.
The analysis conducted within this paper is inspired by most techniques in this space which consider approximate results acceptable and so taking an exact algorithm (FIMoTS) and parameterise the conditions for performing infrequent pattern checks so less effort is required when determining is an itemset may become frequent. The resulting approach is expected to provide a high degree of scalability as the necessary support checks may be performed across a spark cluster and ideally not diverge substantially for the exact nature of the FIMoTS approach.

3 Research Methodology

3.1 Datasets

In this analysis I will examine the impact of altering the FIMoTS algorithm across seven datasets. Five of these datasets are very standard in this area and are used in a variety of recent FPM studies [9-21], to which a Twitter and an online retail dataset have been added.
In below table provide a summary of the key aspects of the datasets being studied, similar to what was performed in [8] and [22]:

Dataset	# Transactions	#Distinct Items	# Total Items	# Average Transaction Length	Expected Frequency
T40I10D100K	100,000	942	3,960,507	39.6	23.8
Twitter	26,838	47,875	203,494	7.6	6314.0
Online Retail	25,900	4,070	541,909	20.5	198.4

Table 1: Dataset Analysis

The columns show:

- Transactions: total number of transactions in the dataset (across the full-time horizon)
- Distinct Items: number of distinct items within all the transactions
- Total Items: sum of items across all transactions
- Average Transaction Length: average number of items in a transaction
- Expected Frequency: given the number of distinct items and the average transaction length a measure of how frequently it is expected for an item to appear

Datasets with a high expected frequency are less dense and unlikely to have results for a high value of a minimum support. It is also more unlikely that large itemsets will be present within sparser datasets, rather it is expected than the majority will be 1-itemsets. As 1-itemsets do not benefit from the tree structure of the algorithm it is interesting to analyse how datasets with less or greater level of depth in their datasets are impacted differently by the altered algorithm.

Twitter

FPM can be used on streams of Twitter data for determining trending topics [25] and so a Twitter dataset is used within this analysis. The Spritzer[1] version of data for a period in the morning of 01/01/2021 from Archive.org[2] is used. The pre-processing on this data consists of:

- Filtering on English language tweets
    - Part of the twitter schema provide the language of each tweet
    - To reduce the complexity in processing with different languages and different character encoding, only items in English are used
- Removal of special characters and emojis
    - While emojis and emoticons are standard within twitter communications their encoding makes them difficult to parse and so have been omitted with the rest of the special characters from this analysis
- Stopword removal
    - Using the standard Pyspark ML implementation[3]
    - Additionally, removing technical text appearing frequently (http, https, re, co)

As can be identified from Table 1, this dataset has the lowest average length and the highest expected frequency per item. It follows that we expect to require a low minimum support to get a large set of frequent itemsets.

T40I10D100K

Synthetic data generated by the IBM Almaden Quest research group[4] which is a common dataset among FPM projects such as [9] and [26]. No time dimension was provided within this dataset so 2 approaches have been taken during this project to determine the impact of variance within the time structure on the efficacy of the approach. The initial tests are performed with a uniform number of transactions per time period and a subsequent test is performed with a randomization of the number of transactions per time period, although with the same number of expected transactions in each period.

Figure 3, shows the number of transactions within the time randomised T40 dataset. In contrast, the uniform dataset has a constant number of 100 transactions per time period. As one of the attributes of the FIMoTS algorithm is its capacity to handle different number of transactions within different time periods in a data stream it is of interest to determine the difference in efficiency of the algorithm across both these cases.

[1] https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data

[2] https://archive.org/details/twitterstream

[3] https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StopWordsRemover.html

[4] http://fimi.uantwerpen.be/data/

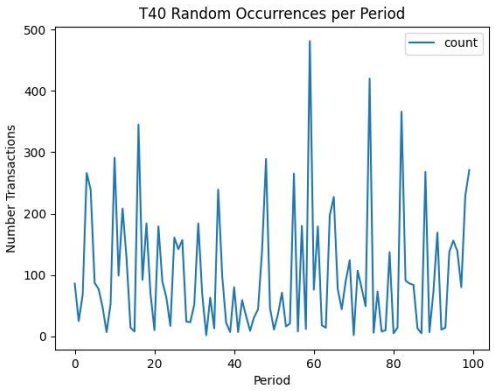

Figure 3: Occurrences per Period for T40 Random Approach

Online Retail

Transnational dataset which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for an online retailer.[5]

3.2 Trends in Datasets

FP Stream was run on the dataset in order to discover trends which existed prior to running tests on the altered FIMoTS algorithm. This consisted of running the Spark ML implementation of FP Stream recursively over the sliding windows in the datasets to determine baseline results for frequent itemsets and thus also point to notable attributes within the data. This section presents the results of this analysis across the 3 datasets used in this study.

3.2.1 Twitter Dataset

Figure 4 shows the variation in the number of frequent patterns over time. The tests within this project were completed using a period length of 30 seconds and a sliding window length of 10 minutes (thus 20 periods per window). The support value was 2.5%, over which it was difficult to get meaningful results. As can be determined from Figure 4, there is a clear decrease in this number after the first quarter for the streaming period. A similar decrease can be observed in Figure 5, which indicates the maximum length of a frequent pattern within each time period. As may be intuitive the maximum size reduces in a similar fashion to the reduction in the number of frequent itemsets.

Figure 6 indicates the frequency with which certain itemsets appear. Certain itemsets can appear within all time periods and so little work may need to be performed in calculating these during each movement of the sliding window. From the above it is noticeable at 10%-15% of itemsets are frequent in every time period but the majority are frequent for < 30% of time periods. This indicates a high degree of movement between itemsets becoming frequent and then infrequent again.

[5] https://archive.ics.uci.edu/ml/datasets/online+retail

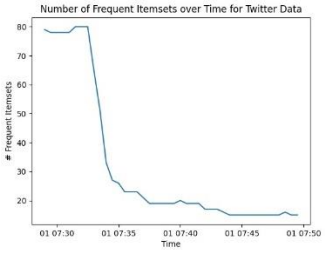

Figure 4: No. Frequent Patterns

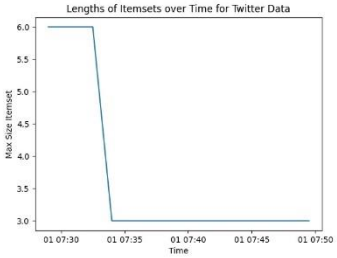

Figure 5: Size of Frequent Patterns

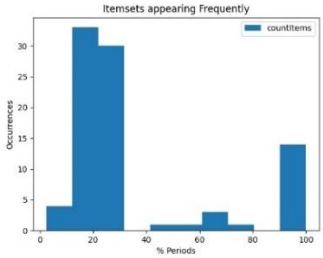

Figure 6: Frequency of Frequent Patterns

3.2.2 Online Retail Dataset

The Online Retail Dataset displays a slight increasing trend in the number of frequent patterns appearing over times as in the figure on the below left. For this dataset, the sliding window length is 28 days with the period being 1 day. The frequency of patterns again is low so a support value of 5% was used (similar to that used for this same dataset within [22]). Unlike the Twitter dataset this displays very little correlation between frequent patterns and so the maximum length of an itemset is 2.

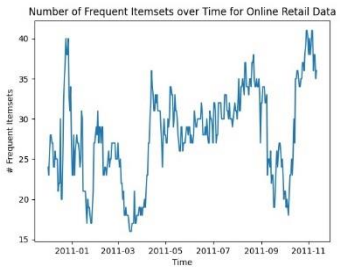

Figure 7: No. Frequent Patterns

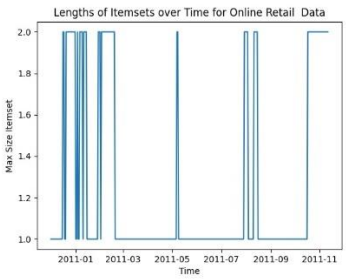

Figure 8: Size of Frequent Patterns

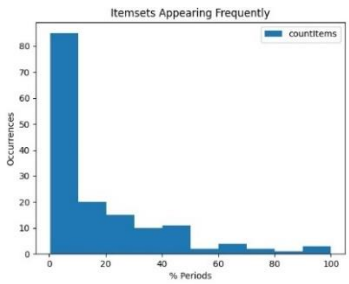

Figure 9: Frequency of Frequent Patterns

The frequent histogram indicates that a large proportion of itemsets are present in less than 10% of time periods.

3.2.3 T40I10D100K Dataset

As outlined, there are 2 approaches taken in this study for this dataset – one keeping each time period with a uniform number of transactions and another using a random number of transactions. In both approaches the sequence of transactions is maintained and so the trend characteristics are similar. The tests are performed using a sliding window of 10 periods and with a minimum support of 10%.

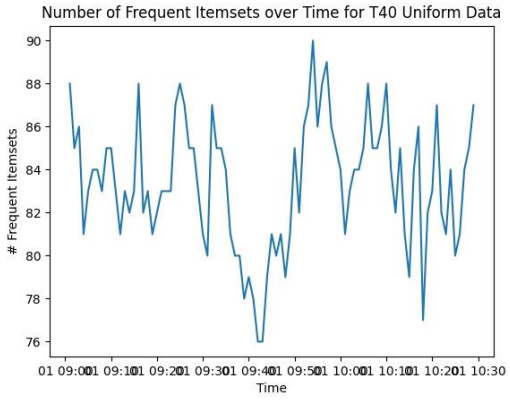

Figure 10: No. Frequent Patterns for Uniform Data

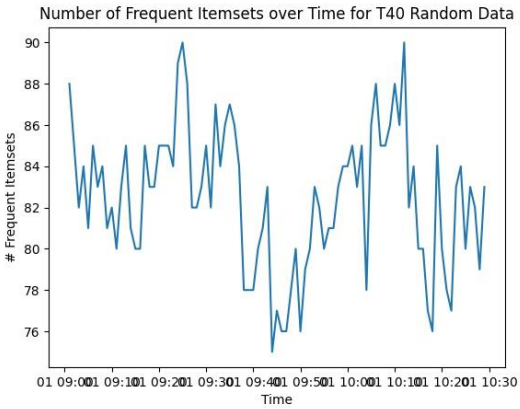

Figure 11: No. Frequent Patterns for Randomised Data

Across both approach the degree of correlation between items was small which led to no > 1-itemset frequent pattern. Notable in the below figures is the high degree of consistency in the frequent itemsets. Greater than 50% of itemsets are present in each period.

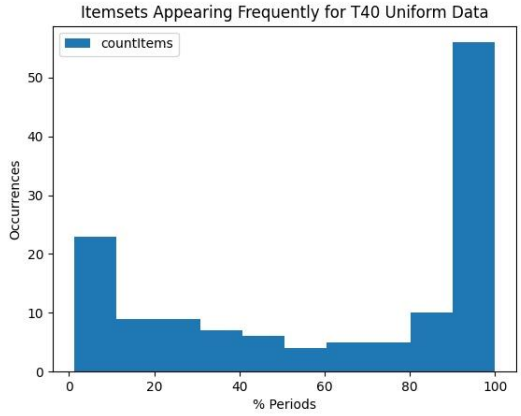

Figure 12: Frequency of Patterns for Uniform Data

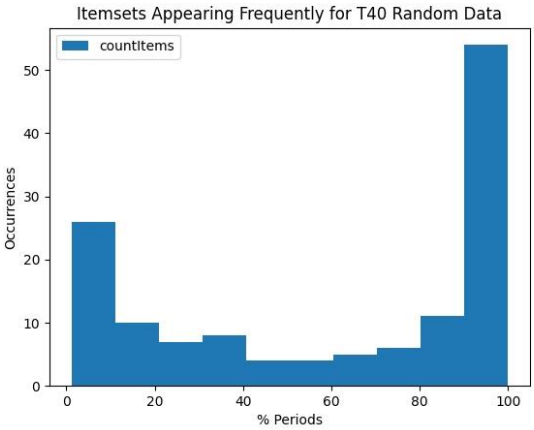

Figure 13: Frequency of Patterns for Randomised Data

3.3 Evaluation of Results

The success of the algorithm will be measured as the correct frequent patterns (as retrieved via the static FP Stream implementation). For analyses on approximate algorithms, usually Precision and Recall are measured as in [16] and [8]. As will be discussed further in Section 4.2, Precision is always 100% for the approximate algorithm used in this study and thus only the Recall will be measured for different values of the multiplier parameter.

4 Design Specification

As explained earlier, an altered version of the FIMoTS algorithm [8] is being used in this analysis. In this section, we shall introduce this algorithm and briefly discuss the points of interest in the approach before moving to the novel aspects of how we wish to change it to become more efficient.

4.1 FIMoTS Algorithm

The FIMoTS algorithm [8] was created with the consideration of reducing the number of times which the raw transactions are checked as items change within a data stream. For example, take an item which has an occurrence rate of 1% in a data stream where the minimum support is 10%. If a time increment only changes the content of the stream by 2% (some items leaving, some items entering) then there is no likelihood that this item can become frequent after this single period and no calculations should be performed for it. Analogously there could exist an itemset with 20% support which cannot become infrequent due to the 2% change and so should be omitted from consideration.
The concept of type transforming bounds have been introduced to determine if a change in the frequency of an itemset may occur as the time period of the data stream shifts.

Type Transforming Upper Bound: For an itemset I, when adding ub transactions to the stream may change I from frequent to infrequent (or vice versa) but adding $ub - 1$ transactions may not, ub is the type transforming upper bound of I, denoted $ub(I)$.
Type Transforming Lower Bound: For an itemset I, when removing lb transactions from the stream may change I from frequent to infrequent (or vice versa) but removing $lb - 1$ transactions may not, lb is the type transforming lower bound of I, denoted $lb(I)$.

The procedure for the FIMoTS algorithm is not to record exactly the support for each itemset at all times, but instead to perform the minimum effort to ensure that itemsets determined to be greater than the minimum support are correct. As per the above definitions, only when the number of transactions arriving to and leaving the dataset exceed the transforming bounds for an itemset should a support calculation be performed.

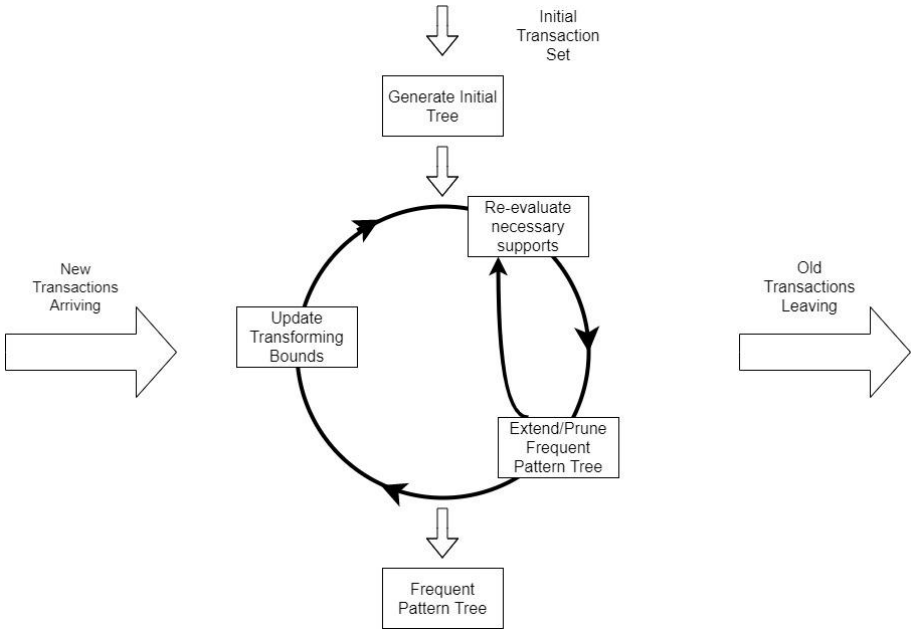

Figure 14: FIMoTS Algorithm

A graphical summary of the operations of the algorithm are provided in Figure 14.

4.1.1 Initial Step

Construct an enumeration tree where each node in the tree represents an itemset. Therefore, each parent node is a subset of a child node and two child sibling n-itemsets each have $n-1$ items in common. All infrequent items will be leaf nodes as by definition no child itemset of an infrequent itemset can be frequent. By default, items are sorted in lexicographical order and child nodes must always be greater than their parents to avoid duplication of itemsets. The below is an example of this tree generation for the 3 transactions where minimum support is 50% (itemsets with blue backgrounds are frequent):

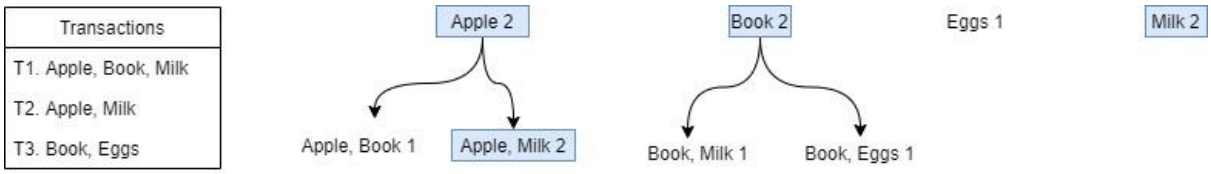

Figure 15: Construction of Enumeration Tree

In the above note:

- Apple, Book, Milk is an itemset which exists in the dataset however as its only viable parent is "Apple, Book" which is infrequent this is not generated
- Apply, Milk is a frequent leaf node. We do not seek to generate "Apple, Milk, Book" from it as Book > Milk
- Milk is a frequent root node but as it has the highest order there are no possible child nodes

In this tree structure it is straightforward to determine as items become frequent and also to generate potentially frequent child nodes. For example, if another transaction arrived to the stream containing "Apple, Book" then this node would become frequent

4.1.2 Creation and Update of Transforming Bounds

The transforming bounds are an indication of how close an itemset is to changing its frequency. Initially an infrequent itemsets *I* will have transforming bounds as:

$$Lower\ Bound(I) = \frac{|D| * s - count(I)}{s} + 1$$

$$Upper\ Bound(I) = \frac{|D| * s - count(I)}{1 - s} + 1$$

And a frequent itemset will have transforming bounds:

$$Lower\ Bound(I) = \frac{count(I) - |D| * s}{1 - s} + 1$$

$$Upper\ Bound(I) = \frac{count(I)}{s} - |D| + 1$$

As new transactions (*nt*) arrive to the stream and old transactions (*ot*) leave the stream the transforming bounds must be updated. For infrequent itemsets, the assumption is that all transactions leaving the dataset do not have the itemset and all transaction arriving may have it which provides these formulae:

$$Infrequent\ Lower\ Bound(I) = IFLB(I) - ot - nt * \frac{1-s}{s}$$

$$Infrequent\ Upper\ Bound(I) = IFUB(I) - ot * \frac{s}{1-s} - nt$$

For frequent itemsets the assumption is that all transactions leaving the dataset contain the itemset and none of those arriving to the dataset contain it. Once again this is worst case scenario to have full certainty there is no chance an item may incorrect be determined still frequent. This logic provides the follow for updates for frequent bounds:

$$Frequent\ Lower\ Bound(I) = FLB(I) - ot - nt * \frac{s}{1-s}$$

$$Frequent\ Upper\ Bound(I) = FUB(I) - ot * \frac{1-s}{s} - nt$$

As new transactions reach the stream, the algorithm only requires the number of new transaction (the number being removed should already be known) to perform this calculation to determine which itemsets result in a negative transforming bound. Those with a negative transforming bound must have their supports re-evaluated as they may have become frequent/infrequent.

4.1.3 Evaluation of Supports

The update of the transforming bounds steps results in a list of:

- Previously frequent itemsets which may now be infrequent
- Previously infrequent itemsets which may be frequent

Scans are performed over the full dataset which is stored in memory to retrieve exact values of the support for each of these itemsets. It is worth noting that only ever after this step does the algorithm provide an exact value of the support of an itemset, the efficiencies lie in the reduction of the itemsets to require this step.

4.1.4 Extending and Pruning the Enumeration Tree

As the frequencies of itemsets are changing, the structure of the enumeration tree must change as well. As mentioned in section 4.1.1, child nodes of infrequent itemsets are not stored in the enumeration tree so once this occurs a pruning operation is required to remove nodes which are no longer necessary from the tree. Clearly these nodes cannot be frequent and must already have been declared so by the support update step. Each time a former frequent itemset becomes infrequent and child nodes are pruned from the enumeration tree and transforming bounds for them will no longer be calculated.

Itemsets moving in the opposite direction i.e., from infrequent to frequent require slightly more attention as they can create child nodes which have so far not been tracked and so may be frequent.

Itemsets becoming frequent follow the below steps:

1. For each frequent right sibling (J) of I
    a. Create a child of I as $K = I \cup J$
    b. Check if K is frequent
2. Repeat step 1 with all new frequent nodes generated in this way

These recursive procedure for generating new itemsets from newly frequent ones is why in Figure 14 there is a loop between the prune and extend step and the support evaluation. These two processes can repeat until no new frequent itemsets are found.

4.2 Approximate FIMoTS Algorithm

The purpose of this analysis is to determine if and when the FIMoTS algorithm can be altered to reduce the number of candidates to be sent to the support calculation step, as this is the most computationally intensive phase [9]. Remember that when calculating the transforming bounds for infrequent itemsets, the algorithm assumes that all new transactions will contain the itemset and none of those leaving. If the minimum support used for FPM is low (for example 5%) this means that every time there is a 5% change in the dataset a support check is required. Usually if the support query is this low it indicates a sparse dataset which clearly would not have 100% of transactions containing a formerly infrequent itemset [8].
Therefore, the premise of the analysis is to introduce a dampening factor to reduce the extent to which the transforming bounds are changed every time period and so avoid redundant support calculations. The analysis will only focus on the infrequent transforming bounds as the number of frequent bound calculations are usually minor relative to the infrequent calculations and it is considerably more likely for a frequent itemset to missing from all new transactions than an infrequent itemsets to be present in all new transactions.
The formulae for this approach are changed to:

$$Infrequent\ Lower\ Bound(I) = IFLB(I) - (ot + nt * \frac{1-s}{s}) * m$$

$$Infrequent\ Upper\ Bound(I) = IFUB(I) - \left(ot * \frac{s}{1-s} + nt\right) * m$$

Where m is the dampening factor. The analysis seeks to answer:

- How the accuracy (recall) of the results differ with changing values of m
- Relationship between s and m
- The changing impact of different values of m on different datasets

In this instance the analysis need only focus on the recall and not the precision of the results. As the Frequent Bounds are not subject to the dampening factor there is certainty any frequent item approaching infrequency will be verified. Uncertainty may only exist on whether sufficient itemsets are being verified for changing from infrequent to frequent. Therefore, regardless of the value for m the precision will be 100% and thus it will not be discussed further in the Evaluation section.

5 Implementation

This project has been implemented in Python using PySpark to perform the more computationally intensive operations. The implementation largely follows the approach from [9] however there are differences in the technical implementation due to difficulties encountered using Spark Streaming in this manner.

From the research performed there is no freely available implementation of FPM on streaming datasets. There is an FP-Growth implementation in the Spark ML library[6] however this is for static datasets only. In order to perform a comparison of the results between the generic FIMoTS algorithm and the approximate version proposed in this study, it was mandatory to first implement this algorithm and afterwards update to apply the dampening factors. Spark Streaming is still a recent technology with its feature set still expanding however the limitation on multiple aggregation within streaming data not being supported[7] resulted in complications within this approach which were considered beyond the scope of this project.
Instead, a simulated process is adopted wherein each dataset is read in full to a Spark dataframe, this is divided into periods and a sliding window procedure performed on these periods. The code iterations over the windows in the dataset as if it were receiving a new set of data at each iteration. In this manner the implementation benefits from the application of Spark and does not suffer from any of the limitations of Spark Streaming.

5.1 Building Enumeration Tree

The enumeration tree is structured to represent all possible patterns from the transactions present within the initial window of the data stream [8]. It may become very broad and long if all possibilities are initially built before the count phase to determine which are within the minimum support threshold. Particularly if the length of transactions in the stream are large (N) then the creation of the tree will be $O(N)!$. The approach taken is therefore to perform 2 initial passes of the data to count the frequent items and frequent 2-itemsets and just extend the tree when a pair is frequent.
The process of building the enumeration tree is thus as follows:

1. Attain all frequent items
2. Attain all frequent 2-itemsets
3. For each transaction
    a. Sort items
    b. Run **Extend Procedure** with (Root, transaction)
4. Prune tree by removing nodes below infrequent itemsets

Extend Procedure (node, transaction)

1. If node has a parent of Root and is frequent
    a. Create child of node with first item from transaction
        i. If already exists increment count
        ii. Run Extend Procedure with (child, transaction from second item)
    b. Else Create node with item and stop
        i. If already exists increment count
2. If node and first item in transaction are a frequent 2-itemset
    a. Create child of node with item
        i. If already exists increment count

[6] https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html
[7] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#unsupported-operations

ii. Run Extend Procedure with (child, transaction from second item)
b. Else Create node with item and stop
i. If already exists increment count
3. Run Extend procedure with (node, transaction from second item)

While this approach results in extra counts of the initial trade population to that in the FIMoTS Algorithm, it prevents a large redundancy in extending a broad and long tree and pruning it later.

5.2 Growth of Tree for New Frequent Itemsets

The enumeration tree should be extended as an itemset becomes frequent as this could imply child itemsets are frequent. In Figure 16, an example is provided for the enumeration tree as a new transaction arrives which makes the itemset [Apple, Eggs] frequent. According to the algorithm right siblings of a newly frequent itemset should be used to extend the enumeration tree, in which case [Apple, Eggs, Milk] is created as a child itemset which may be frequent (the green background implies a new check needs to be performed). As [Apple, Book] is a left itemset it is not considered for extension (as there cannot be [Apple, Eggs, Book], where the sorted order of Eggs > Book).

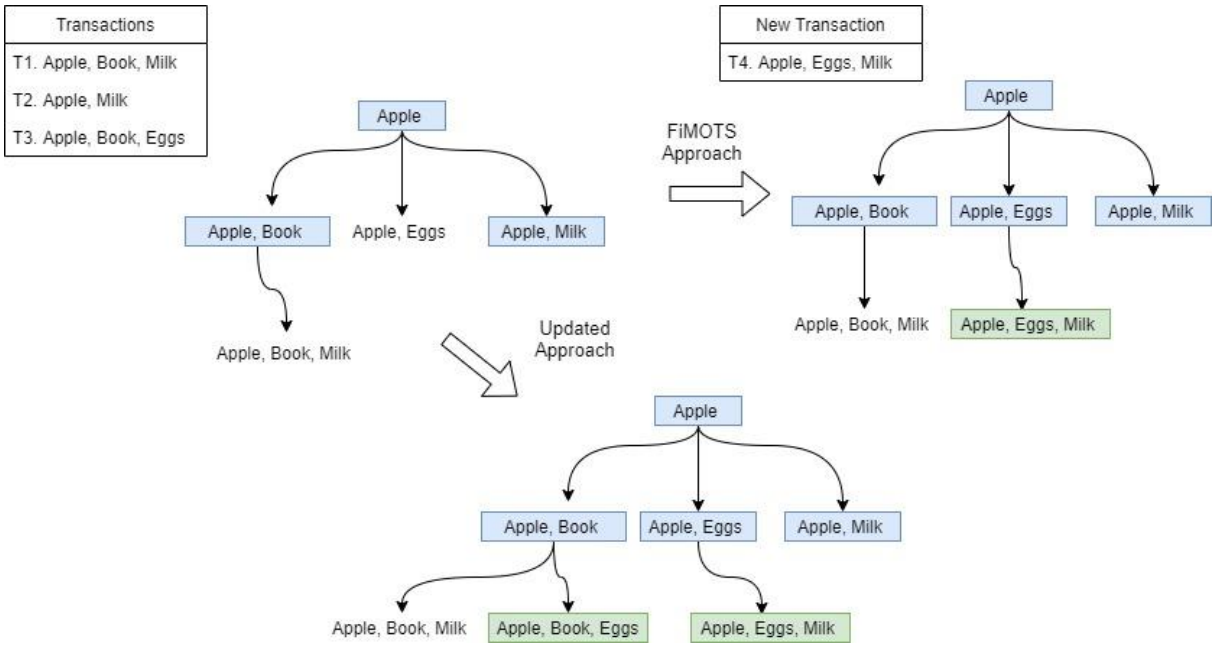

Figure 16: Update to Enumeration Tree Growth

However, [Apple, Book, Eggs] may be a frequent itemset and thus the altered FIMoTS algorithm used in this study both appends right frequent siblings to a newly frequent item but also appends the new item to its left frequent siblings.

5.3 New Items within Stream

The original FIMoTS does not make any allowances for items which arrive to the stream after the initial tree has been created. The focus is on updating the transforming bounds for the number of transactions leaving and entering the stream, not on how the contents of the

stream are changing. Using the standard approach, it was found that several inaccuracies between the baseline results were identified due to new items arriving to the stream. Unless the algorithm is always expected to operate over a closed set of items allowances should be made for new items entering the stream. In this study each new block of transactions entering the stream is queried for new items and those found are added to the infrequent bounds table and included for count computation within the current sliding window.

5.4 Python and Spark Approach

Taking these elements into account, the high-level implementation of the algorithm consists of:

1. The initial dataset is read from source files into a Spark Dataframe
2. This Dataframe broken down into a series of periods of equal durations
3. Panda Dataframes are constructed for *counts per periods* and *counts per sliding window* (groups of sequential periods)
4. Python is used to generate the initial tree from the first sliding window and pandas frequent and infrequent bounds are generated
5. For Window #2 until end of stream
    a. Frequent and infrequent bounds are updated from values stored in Step 3 for current window. This is a basic calculation and so would not benefit from being sent to a Spark cluster.
    b. Itemsets which require evaluation are grouped and verified against the Spark Dataframe filtered on the current sliding window.
    c. Tree is extended or pruned according to frequencies. Extended trees can require further frequency validations against the Spark dataframe as in the example from Section 5.2.
    d. The frequent and infrequent bounds for itemsets which have been checked are updated for actual values.
    e. Frequent itemsets for window recorded.

6 Evaluation

The evaluation within this project consists of reporting on the efficiency of the implementation across both the recall of the results and the reduction in checks and timing improvements due to this. This section will provide these results across the different datasets identified and analyse the reasons for these findings.

6.1 Twitter Dataset

The number of frequencies of itemsets in this dataset was relatively low and so a 2.5% minimum support was used to ensure there were an acceptable number of items frequent at any one time. Given this low minimum support it is expected that the default algorithm had a large amount of redundancy as it is not very likely for an itemset represented in less than 2.5% of transactions in one period to be present in a high percentage of new transactions. For

these tests, the multiplier has been varied from 1% to 100% (default) to determine the impact across number of checks and recall and precision.
The below graphs (Figure 17 and Figure 18) are created using a 10-period moving average of the results. Moving averages are used in the display as there can be a lot of variation between different periods which makes it difficult to assess the trends in the results.

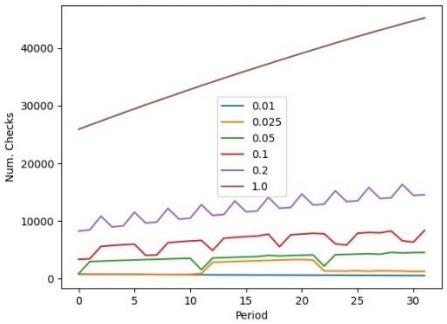

Figure 17: Number of Checks for Twitter

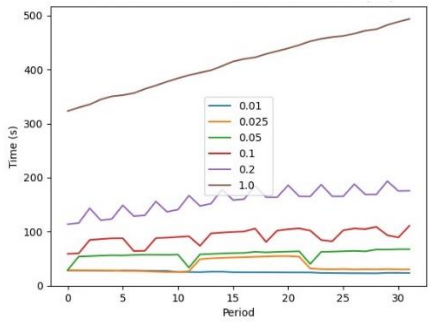

Figure 18: Time per Period for Evaluation

As can be determined, both graphs display almost identical trends which indicates the strong correlation between number of checks to perform and the time for completion. All graphs display a slightly upward trends which is due to the new single itemsets arriving to the data stream. As new items are continuously arriving the number of checks which must be performed increases – clearly this raises questions about the sustainability of such an approach in a variable source of data like Twitter.
A graph of the change in accuracy of results is displayed in Figure 19. It is expected that when using a multiplier less than the minimum support (e.g., 1%) the accuracy of the results is reduced however largely this does not appear to be the case. In all cases the precision is 100% and the average recall is 98.8% even for the 1% multiplier.

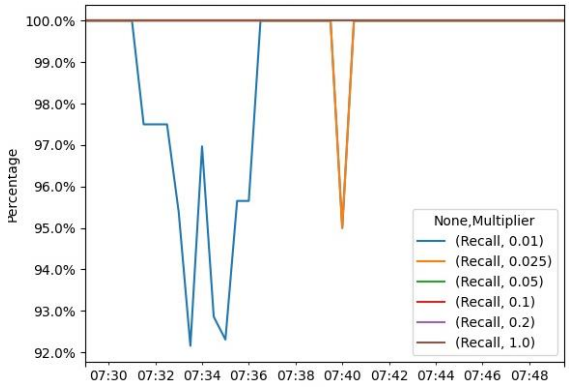

Figure 19: Precision and Recall over Time for Twitter Dataset

The below table summarise the accuracy results for this dataset as well as the reduction in number of checks required and total time to process.

	0.01	0.025	0.05	0.1	0.2	1
Recall	98.8%	99.9%	100%	100%	100%	100%
Checks (percentage of m = 1)	2%	4%	9%	18%	33%	100%
Time (percentage of m = 1)	6%	8%	13%	22%	37%	100%

Table 2: Twitter Results

Note the aim of the project is to determine the impact of the reduction in the number of checks performed on the accuracy of the results. The timing results are here for indicative purposes only as the technical implementation of the algorithm could certainly be optimised.

6.2 Online Retail Dataset

The online retail dataset displays less favourable results for the approximate approach as show in the below table:

	0.01	0.025	0.05	0.1	0.2	0.4	0.75	1
Recall	60.1%	84.6%	95.9%	99.2%	99.7%	99.9%	100%	100%
Checks	2%	4%	8%	15%	28%	50%	77%	100%
Time	42%	47%	52%	53%	58%	68%	84%	100%

Table 3: Online Retail Results

The minimum support values used when running these tests was 5% and when running tests with this value of multiplier the recall is 95.9 (unlike Twitter which was 99.9% for multiplier equal to minimum support). This likely indicates itemsets becoming frequent quickly – in which case a high percentage of transactions appearing in new periods of the sliding window containing infrequent itemsets may not be checked by the algorithm.
Also notable in the above is the reduction in time spent performing these checks. Unlike the Twitter example the maximum reduction was to 42% of the total time. This is likely because for this dataset the overall time for running the checking operation is not very large and each iteration there is time spent other places in that algorithm to what is optimized via the multiplier. Therefore, in this case the proportional speedup is not very high.

6.3 T40 Datasets

As explained previously, for the T40 dataset a synthetic time element was applied with a uniform approach and a randomised approach (where the number of transactions in a time period can vary). Given the underlying data is the same and it is just the grouping which differs it is expected to return similar results.

	0.025	0.05	0.1	0.2	0.4	0.75	1
Recall	97.1%	99.3%	100%	100%	100%	100%	100%
Checks	4%	7%	12%	23%	39%	59%	100%
Time	20%	23%	27%	33%	44%	58%	100%

Table 4: T40 Uniform Results

	0.025	0.05	0.1	0.2	0.4	0.75	1
Recall	95.6%	98.1%	99.2%	99.8%	100%	100%	100%
Checks	6%	11%	20%	36%	63%	90%	100%
Time	31%	35%	42%	52%	71%	90%	100%

Table 5: T40 Random Results

As can be seen, in all cases the results on the randomised set are worse than the uniform set. The multiplier required to receive 100% recall is 0.1 in the uniform set compared with 0.4 in the randomised set. When returning 100% recall the timing for the uniform results is 27% of basic algorithm whereas in the random test it is 71% which would indicate that the approximation is of more benefit when the number of transactions in each time period are consistent.

6.4 Discussion

The key findings from this study:

- The initial FIMoTS algorithm proposed may be improved upon to deliver similar levels of accuracy with a large reduction in time and number of checks to be performed.
- This is however an approximate approach in which false negatives can occur. The likelihood of this occurring is proportional to the extent of the approximation taken and the degree to which new itemsets become frequent in the dataset.

The Twitter dataset displays the most impressive results with a large saving in time from applying low value multipliers which still returning strong results. This would indicate that for datasets like this, the build-up for itemsets to become frequent is gradual and taking the approximate approach is very worthwhile. However, the upward trend in the number of checks required would indicate that the algorithm lacks an ability to prune single items which no longer appear in the stream and so long term would not be suitable on a dataset which can have such a variety of items.

The Online Retail dataset displays an opposite view wherein a high number for the multiplier is required to achieve confidence in 100% recall. This dataset is quite sparse and consists of few frequent itemsets which may not lend itself to benefit from an approach such as this.

The T40 dataset analysis provides more insight into an aspect of the datasets which appear to impact the effectiveness of the approximate approach. Uniform numbers of transactions per period give better results than variable. It is worth noting that the Online Retail dataset also exhibited a high degree of variability in number of transactions per period which could have contributed to the relatively poor results for that test.

7 Conclusion and Future Work

In this study it has been demonstrated that the FIMoTS algorithm for determining frequent itemsets from streams of data could be approximated to operate with less effort when accepting a relatively minor reduction in accuracy. The approximation was tested on different dataset to judge the variation of effectiveness according to the underlying data. It has been

found to operate most effectively upon a Twitter dataset where the timings were reduced by 96% for 100% precision and 99.9% recall. In all cases the number of checks required may be reduced by 80% with less than 1% reduction in recall. This reduction in recall compares favourably with other accuracy results for approximate algorithms for example in [16] and [8]. More testing could be run on the datasets varying parameters such as minimum support and sliding window length to determine their impact on appropriate multipliers to use and their effectiveness.

The appropriate multiplier to use is directly related to the speed within which new trends emerge within the dataset so it may be impractical to determine a multiplier in advance or one which can operate long term over a datastream. Further investigation may be performed to seek to determine a self-adjusting multiplier which could interpret the extent to which data within the stream is changing and varying the multiplier appropriately. As apparent from the Twitter tests the upward trend in number of checks required would lead to poor performances and so the algorithm could be improved to remove single items which have no occurrences in the stream anymore.

Furthermore, the technical implementation of the algorithm would ideally be improved upon to leverage directly off Spark Streaming instead of looping method implemented here. It would likely yield much performance optimizations so thus impact level of time reductions from these tests. It would have been preferable to have other implementations of FPM on data streams available for comparisons of results from these tests. As these are made available in the future it would be of interest to understand how they comparison against the algorithm proposed here both in terms of time to run and accuracy of results.

References

[1] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, “Mining Data Streams: A Review,” *SIGMOD Rec*, vol. 34, no. 2, pp. 18–26, Jun. 2005, doi: 10.1145/1083784.1083789.

[2] H.-G. Kim, S. Lee, and S. Kyeong, *Discovering hot topics using Twitter streaming data: social topic detection and geographic clustering*. 2013, p. 1220. doi: 10.1145/2492517.2500286.

[3] R. Agrawal, T. Imieliński, and A. Swami, “Mining association rules between sets of items in large databases,” in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, New York, NY, USA, Jun. 1993, pp. 207–216. doi: 10.1145/170035.170072.

[4] R. Agrawal, R. Srikant, H. Road, and S. Jose, “Fast Algorithms for Mining Association Rules,” *Proc. 20th VLDB Conf.*, p. 13, 1994.

[5] J. Han, J. Pei, and Y. Yin, “Mining Frequent Patterns Without Candidate Generation,” in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2000, pp. 1–12. doi: 10.1145/342009.335372.

[6] M. Garofalakis, J. Gehrke, and R. Rastogi, “Querying and mining data streams: you only get one look a tutorial,” in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, New York, NY, USA, Jun. 2002, p. 635. doi: 10.1145/564691.564794.

[7] G. S. Manku and R. Motwani, “Approximate frequency counts over data streams,” *Proc. VLDB Endow.*, vol. 5, no. 12, p. 1699, Aug. 2012, doi: 10.14778/2367502.2367508.

[8] H. Li, N. Zhang, J. Zhu, H. Cao, and Y. Wang, “Efficient frequent itemset mining methods over time-sensitive streams,” *Knowl.-Based Syst.*, vol. 56, pp. 281–298, Jan. 2014, doi: 10.1016/j.knosys.2013.12.001.

[9] C. Fernandez-Basso, A. J. Francisco-Agra, M. J. Martin-Bautista, and M. Dolores Ruiz, “Finding tendencies in streaming data using Big Data frequent itemset mining,” *Knowl.-Based Syst.*, vol. 163, pp. 666–674, Jan. 2019, doi: 10.1016/j.knosys.2018.09.026.

[10] G. S. Manku and R. Motwani, “Chapter 31 - Approximate Frequency Counts over Data Streams,” in *VLDB ’02: Proceedings of the 28th International Conference on Very Large Databases*, P. A. Bernstein, Y. E. Ioannidis, R. Ramakrishnan, and D. Papadias, Eds. San Francisco: Morgan Kaufmann, 2002, pp. 346–357. doi: 10.1016/B978-155860869-6/50038-X.

[11] J. H. Chang and W. S. Lee, “Finding recent frequent itemsets adaptively over online data streams,” in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, New York, NY, USA, Aug. 2003, pp. 487–492. doi: 10.1145/956750.956807.

[12] C. Giannella, J. Han, J. Pei, X. Yan, and P. Yu, “Mining Frequent Patterns in Data Streams at Multiple Time Granularities,” Jan. 2003.

[13] J.-L. Koh and Y.-B. Don, “Approximately Mining Recently Representative Patterns on Data Streams,” in *Emerging Technologies in Knowledge Discovery and Data Mining*, 2007, pp. 231–243.

[14] J. X. Yu, Z. Chong, H. Lu, and A. Zhou, “False positive or false negative: mining frequent itemsets from high speed transactional data streams,” in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, Toronto, Canada, Aug. 2004, pp. 204–215.

[15] C.-H. Lin, D.-Y. Chiu, Y.-H. Wu, and A. Chen, *Mining Frequent Itemsets from Data Streams with a Time-Sensitive Sliding Window*. 2005. doi: 10.1137/1.9781611972757.7.

[16] H. Chen, L. Shu, J. Xia, and Q. Deng, “Mining frequent patterns in a varying-size sliding window of online transactional data streams,” *Inf. Sci.*, vol. 215, pp. 15–36, Dec. 2012, doi: 10.1016/j.ins.2012.05.007.

[17] G. Lee, U. Yun, and K. H. Ryu, “Sliding window based weighted maximal frequent pattern mining over data streams,” *Expert Syst. Appl.*, vol. 41, no. 2, pp. 694–708, Feb. 2014, doi: 10.1016/j.eswa.2013.07.094.

[18] S. J. Shin, D. S. Lee, and W. S. Lee, “CP-tree: An adaptive synopsis structure for compressing frequent itemsets over online data streams,” *Inf. Sci.*, vol. 278, pp. 559–576, Sep. 2014, doi: 10.1016/j.ins.2014.03.074.

[19] C.-H. Lee, C.-R. Lin, and M.-S. Chen, “Sliding-window filtering: an efficient algorithm for incremental mining,” in *Proceedings of the tenth international conference on Information and knowledge management*, New York, NY, USA, Oct. 2001, pp. 263–270. doi: 10.1145/502585.502630.

[20] C. K.-S. Leung and Q. I. Khan, “DSTree: A Tree Structure for the Mining of Frequent Sets from Data Streams,” in *Proceedings of the Sixth International Conference on Data Mining*, USA, Dec. 2006, pp. 928–932. doi: 10.1109/ICDM.2006.62.

[21] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong, and Y.-K. Lee, “Sliding window-based frequent pattern mining over data streams,” *Inf. Sci.*, vol. 179, no. 22, pp. 3843–3865, Nov. 2009, doi: 10.1016/j.ins.2009.07.012.

[22] W. Xiao and J. Hu, “SWEclat: a frequent itemset mining algorithm over streaming data using Spark Streaming,” *J. Supercomput.*, vol. 76, no. 10, pp. 7619–7634, 2020, doi: 10.1007/s11227-020-03190-5.

[23] C. K.-S. Leung and F. Jiang, “Frequent itemset mining of uncertain data streams using the damped window model,” in *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC ’11*, TaiChung, Taiwan, 2011, p. 950. doi: 10.1145/1982185.1982393.

[24] K. K. Parameswari and D. A. S. Thanamani, “Frequent Item Mining Using Damped Window Model,” vol. 3, no. 9, p. 4, 2014.

[25] L. M. Aiello *et al.*, “Sensing Trending Topics in Twitter,” *IEEE Trans. Multimed.*, vol. 15, no. 6, pp. 1268–1282, Oct. 2013, doi: 10.1109/TMM.2013.2265080.

[26] F. Gui *et al.*, “A distributed frequent itemset mining algorithm based on Spark,” in *2015 IEEE 19th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, May 2015, pp. 271–275. doi: 10.1109/CSCWD.2015.7230970.