STREAMFPM

FINDING FREQUENT PATTERNS

WITHIN TRANSACTION DATA STREAMS IN R

Approved by:

____

Dr. Michael Hahsler

____

Mark Fontenot

____

Eric Larson

STREAMFPM

FINDING FREQUENT PATTERNS WITHIN TRANSACTION DATA STREAMS IN R

A Thesis Presented to the Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Bachelors of Science

with a

Major in Computer Science

by

Derek S. Phanekham

May 17, 2015

ACKNOWLEDGMENTS

I would like to thank the Department of Computer Science of Southern Methodist University and my advisor for this thesis, Michael Hahsler.

Phanekham , Derek S.

Advisor: Professor Michael Hahsler

Bachelors of Science degree conferred May 17, 2015

Thesis completed April 9, 2015

Data streams and particularly data streams of transactions are everywhere from Twitter to customer purchases from point-of-sale systems. A data stream is a continuous flow of data points that has no foreseeable temporal end. These large, constantly updating streams are the subject of an increasing amount of research interest, particularly in the area of frequent pattern mining. This paper explores frequent pattern mining on data streams and various algorithms that have been developed. It then introduces **streamFPM**, an addition to the **stream** package that provides multiple transaction stream generators, two algorithms for frequent pattern mining, as well as a general framework that can be expanded upon in the future. **streamFPM** and the **stream** framework that it is a part of are all implemented in R, an open-source statistical computing language that is often used for data mining tasks. This provides a good basis for testing these algorithms against eachother, newly implemented algorithms, or non-streaming algorithms such as Apriori. Finally, this paper provides examples of how to use the various classes and functions in **streamFPM**.

TABLE OF CONTENTS

LIST OF FIGURES ... vii

CHAPTER

1. INTRODUCTION ... 1
   - 1.1. Frequent Pattern Mining ... 1
   - 1.2. Frequent Pattern Mining on Data streams ... 2
   - 1.3. stream Framework ... 2
   - 1.4. Organization of the Chapters ... 3
2. Related Work ... 4
   - 2.1. Lossy Counting ... 5
   - 2.2. estDec ... 6
   - 2.3. Other Algorithms ... 8
3. The streamFPM package ... 10
   - 3.1. DSD_Transactions ... 11
     - 3.1.1. DSD_Transactions_Random ... 12
     - 3.1.2. DSD_Transactions_TwitterStream ... 13
     - 3.1.3. DSD_Transactions_Twitter ... 17
   - 3.2. Frequent Pattern Mining DSTs ... 19
     - 3.2.1. DST_EstDec ... 19
     - 3.2.2. DST_LossyCounting ... 23
4. Example Application ... 26
   - 4.1. Setup ... 26
   - 4.2. Running the algorithm ... 28

5. Conclusion and Future Work 33

APPENDIX

A. APPENDIX 34

REFERENCES 45

LIST OF FIGURES

Figure Page

3.1. DSD Inheritance Diagram 12

3.2. DST Inheritance Diagram 19

3.3. estDec Sequence Diagram for `getPatterns()` 20

4.1. Frequent Itemsets over time 32

Chapter 1

INTRODUCTION

In this paper I will be focusing on the problem of mining for frequent patterns in data streams. The mining of static data is a field that has been well researched for many years, but the same task when applied to a rapidly updating and unbounded stream of data is less well researched.

1.1. Frequent Pattern Mining

A transaction is a set of discrete values, or items. Frequent pattern mining (FPM) involves looking at transactional sequences and finding items that appear together frequently, or itemsets [2]. These sequences can be anything from stock tickers to market basket data to a word document, as long as there are, preferably repeated, instances of items. A FPM algorithm will check to see if any items appear together often enough to meet a minimum support threshold defined by the user. If any sets of items are above this threshold, they are called frequent itemsets. These frequent itemsets, and the association rules that can be generated from them, are useful in many applications such as product placement and intrusion detection. Perhaps the best known and most influential algorithm in this area is Apriori, which like many similar algorithms, considers all possible itemsets in a transactional sequence to find the ones that meet the support requirements. Traditionally, this is done on a static dataset since Apriori requires many passes over the data to develop a complete understanding of it.

1.2. Frequent Pattern Mining on Data streams

Frequent Pattern Mining in datastreams can be a more difficult task, as it involves finding the most common itemsets in a continuous stream of data. Using a stream places restrictions on what an algorithm is able to do. Since a stream's life might have no definitive end, the volume of data a stream might be difficult for a static algorithm to handle [1]. Because storage space is finite and more data is constantly arriving, datastream mining faces the unique challenge of only being able to pass over the data once since we are not able to store all of the data arriving in the stream.

When adapting frequent pattern mining to perform on a data stream, one would either have to use an algorithm like Apriori over a sliding window, which would be less than optimal in terms of time, or use an algorithm specially constructed for the task. Algorithms for finding frequent patterns in data streams must consider several factors that are less important for a static dataset such as speed, memory, how to determine what data to store, and how to deal with concept drift. Concept drift refers to change in usage over time. Itemsets that were once frequent sometimes become infrequent and the algorithm must be able to recognize this and deal with it appropriately. There is also the additional factor of error that is not found in algorithms that can fully learn a dataset over multiple passes. In data streams, an algorithm may miss a few instances of an itemset before it is deemed significant enough to be tracked, so many of these algorithms keep a possible margin of error in addition to the count of the itemset.

1.3. stream Framework

stream is an R package that provides a framework for data stream modeling or clustering, classification and related tasks on data streams [7]. This framework has two main abstract classes that most other classes are descendents of: Data Stream

Data (DSD), which simulates a data stream and produces new data, and Data Stream Task (DST), which performs some type of task on the data recieved from a DSD. The framework is very open-ended and extensible for other data mining tasks involving data streams.

1.4. Organization of the Chapters

Chapter 2 explores existing research in the field of frequent pattern mining on data streams. Chapter 3 goes into the implementation details of streamFPM and how it relates to the stream framework. Chapter 4 presents an example using estDec and a data stream generator. Chapter 5 concludes the paper.

Chapter 2

Related Work

The first step to creating streamFPM was to research existing algorithms and theories. Every algorithm that I researched had some basic characteristics in common that are unique to working with data streams. In a data stream, new transactions are constantly being generated. For an algorithm to keep up with the flow of data, they can only pass over every transaction once or within the period of a sliding window. Unlike non-streaming algorithms such as Apriori, these algorithms do not store all of the information about every possible itemset they encounter. If they did this, the memory requirements would be infeasible for any real world scenario involving a number of transactions that is constantly increasing in size. Instead, they only store items or itemsets that meet a specified minimum support. Many of the algorithms also have a concept of change over time because what was frequent once may not be frequent currently or in the future. In doing so, they utilize some method of support decay. Old itemsets that are no longer considered frequent are thrown out and forgotten. Also, because they do not store all the data, these algorithms use some method of estimation, like a margin of error. They store a count of how many times they know the item or itemset has been seen in the stream and a estimation of how many times the itemset could have been in the stream but went unnoticed.

Of the various algorithms either theorized, defined, or implemented I found, below I outline and describe a few that cover several broader categories.

2.1. Lossy Counting

This algorithm counts the frequency of individual items in a data stream. It is a deterministic algorithm that is guaranteed to take at most $\frac{1}{\epsilon}\log(N)$ space, where N is the length of the stream and ϵ is a maximum allowable degree of error specified by the user [1]. For each item that may be frequent, it tracks that items's count since discovery, c_i, and the maximum possible error, e_i of this count. Tracking error is not unique among frequent pattern mining algorithms for data streams. Most have a possible error for each element simply because all elements cannot be tracked due to space constraints. Some instances of an element may go uncounted before the algorithm realizes that it is frequently occuring, and this is what the possible error, e_i, accounts for.

Lossy Counting uses the concept of buckets with width $w = ceil(1/\epsilon)$. The current bucket, b_{cur} begins at 1 and every time a bucket boundary is reached, when $N mod w = 0$, b_{cur} is incremented by 1, where N is the number of transactions seen so far. As N increases, we look at each item in each transaction and add it to the data structure if it is not already present. We set $c_i = 1$ and $e_i = b_{cur} - 1$. If the item is already present, we increment c_i by 1. At the bucket boundary the data structure is pruned of all infrequent items, i.e. when for an item, $c_i + e_i <= b_{cur}$. By taking this approach where all items are inserted and held in the data structure for at least the width of one bucket, or w transactions, the algorithm ensures that all items have a chance to become frequent [1].

The user also specifies a minimum support, s, that items must meet to be deemed frequent. This is used along with the error, ϵ, when extracting frequent items from the data structure. An item is frequent if for that item $c_i \geq (s - \epsilon)N$. When an item becomes infrequent, Lossy Counting prunes it from the data structure to save space. This means that at any point in time, the algorithm only is counting recent frequent

items and new items that may become frequent. This is useful for tracking concept drift, as the items that are frequent may completely change over time.

Lossy Counting runs quickly in comparison to some of the other algorithms, but it has the disadvantage that it only finds recent frequent items, which can be useful, but what we really want are recent frequent itemsets so we can mine for association rules.

2.2. estDec

estDec is the second alogorithm that I implemented. This algorithm keeps track of itemsets that it considers frequent or may become frequent soon, when adding a new itemset, it estimates a count and error by looking at its subsets that are already stored in a tree. EstDec keeps a prefix tree, or trie, of all the itemsets that it is tracking. Prefix trees are a staple of frequent pattern mining algorithms. Static algorithms that use this data structure tend to have very large trees, because they keep track of every item in the data set, while estDec's prefix tree only contains frequent or almost frequent itemsets. This reduces the amount of space the data structure occupies while also making it quicker to perform operations such as itemset lookup and pruning the tree of itemsets that become infrequent. That being said, these trees can still become very large when the decay rate and minimum support are too low and can slow the algorithm down signficantly.

All operations when working with a prefix tree become a matter of exploring the tree. Leaf nodes are the largest itemsets and every node on the path from the root to a leaf is a subset of the itemset represented by the leaf node. When pruning the tree, it is easiest to check the nodes closest to the root first, and if any of those are determined to be infrequent all of its children are also infrequent and subsequently removed [4].

Whenever a transaction is received from a data stream, the itemset and all of its subsets are updated in the prefix tree. When their count is updated, a decay rate is also applied

$$c_k = c_{k_{old}} * d^{k-k_{old}} + 1 \tag{2.1}$$

Where c_k is the new count of the itemset, $c_{k_{old}}$ is the previous count, d is the decay rate, k is the ID of the current transaction, k_{old} is the ID of the last transaction that updated this itemset. Transaction IDs simply increase sequentially [6].

When a new itemset might be inserted into the prefix tree, the algorithm checks for all subsets of that itemset in the tree. If it finds all the subsets, it uses their counts to estimate a maximal count and error for the itemset it is inserting. If the count for this itemset meets a minimum support, then it is inserted into the tree. Inserting an itemset, e, uses the following process.

$$C^{max}(e) = min(P^C_{n-1}(e)) \tag{2.2}$$

$$C^{min}(e) = max(C^{min}(\alpha_i \cup \alpha_j)|\forall \alpha_i, \alpha_j \in P_{n-1}(e) \, and \, i \neq j) \tag{2.3}$$

$$C^{min}(\alpha_1 \cup \alpha_2) = \begin{cases} max(0, C(\alpha_1) + C(\alpha_2) - C(\alpha_1 \cap \alpha_2)) & if \ \alpha_1 \cap \alpha_2 \neq \emptyset)) \\ max(0, C(\alpha_1) + C(\alpha_2) - |D|) & if \ \alpha_1 \cap \alpha_2 = \emptyset)) \end{cases}$$

(2.4)

where $C^{max}(e)$ is the maximum possible count of itemset e. It is calculated by taking the minimum count of all the $(n-1)$ subsets of e. $C^{min}(e)$ is the guarenteed count of e. It finds this by comparing counts for the unions and intersections of the $(n-1)$ subsets of e, where $|D|$ is the current number of items in the stream. D also decays over time according to the decay rate.

What we end up with is an algorithm that has the ability to track the frequency of recent itemsets and their estimated error, while only examining each transaction once before moving to the next transaction. While it does this, the data structure should stay roughly the same size due to the pruning of infrequent itemsets.

2.3. Other Algorithms

There are also a variety of other FPM algorithms that I researched but have not implemented, including Weighted Sliding Window, StreamMining, and Moment.

Weighted Sliding Window (WSW) functions using multiple user defined windows with different weights. Each transaction is divided between these windows and a support is calculated for each item in the transaction. The support is weighted depending on the window. From this support, Frequent items are found, and using these frequent items, frequent itemsets can be derived for the transaction [11].

StreamMining is an algorithm that does not rely on windowing. Instead, it uses an algorithm called KPS to find all frequent 1-itemsets. KPS does this by using a threshold $\Theta(0 < \Theta < 1)$. It finds all items that occur more frequently than $N\Theta$, where N is the length of the sequence. It does this by eliminating $1/\Theta$ distinct items in the transaction, leaving items that appear more than $N\Theta$ times. This is more efficient than other methods of finding frequent 1-itemsets. From here, StreamMining can

calculate frequent supersets from the 1-itemsets and store them in a prefix tree [9].

Moment mines for closed frequent itemsets over a sliding window that contains a sampling of the data stream [3]. It specifically mines for only closed frequent itemsets because there are usually significantly fewer of these than just frequent itemsets, and by tracking fewer itemsets, the algorithm uses less memory. A closed frequent itemset is an itemset for which there is no superset with the same or higher support. Similar to estDec, Moment tracks itemsets using a specialized prefix tree that they call a Closed Enumeration Tree. This algorithm can be found in MOA (Massive Online Analysis), a Java package for data stream generation and mining. Since this was one of the few frequent pattern mining algorithms I found already implemented and publically avaliable, I chose to not implement it.

Chapter 3

The streamFPM package

In this chapter I will be covering the implementation details of **streamFPM**. As I mentioned previously, **streamFPM** extends both the main classes of **stream**, DST (Data Stream Task) and DSD (Data Stream Data). The extensions are made for the purpose of providing the functionality to perform frequent pattern mining. This means creating data stream generators (DSDs) that produce transactions and implementing algorithms that can use those DSDs to mine frequent patterns.

The general flow of code is usually as follows. Create a new instance of a DSD_Transactions class and a new instance of a frequent pattern mining DST.

```
> dsd <- DSD_Transactions_Example(parameters...)
> dst <- DST_Example(parameters)
```

When you are ready to start generating data for the DST, call `update(dst, dsd, n = 1)`, with n set to the number of transactions you would like to generate.

```
> update(dst, dsd, n = 500)
```

After the transactions are processed, you can get the found frequent patterns from the dst.

```
> get_patterns(dst, ...)
```

Once you have these patterns, you can examine the frequent itemsets and the count of each itemset. You can also convert them into itemsets from the arules package for further analysis.

3.1. DSD_Transactions

To start with, I created a new abstract class called (DSD_Transactions) that extends DSD and is the superclass to the data stream generators that I implemented. I make this distinction because this abstract class and its subclasses produce transactions, unlike the other types of DSDs already in stream. The DSTs in streamFPM only function with transaction data, so I implemented them so that they can only use any subclass of type DSD_Transactions for data generation. This was made simply by the S3 class system in R, as you can assign a class to be of multiple types to establish a hierarchy. For example, DSD_Transactions is of type ("DSD", "D_Transactions"), while its subclass DSD_Transactions_Random is of type ("DSD", "DSD_Transactions", "DSD_Transactions_Random"). Figure 3.1 below, shows the created class hierarchy.

DSD_Transactions has multiple subclasses including one that creates random data, one that connects to the Twitter REST API, and one that connects to the Twitter stream API. All subclasses of DSD_Transactions use the `get_points(dsd, n=1)` request for more transactions. Transactions are returned as a list of transactions (by default there is only one transaction in the list), where each transaction is represented as an array of either type integer or character, depending on the specific DSD and settings. It should be noted that natural language is not the best candidate for streams and frequent pattern mining, as there can be an extremely large number of unique items (words).

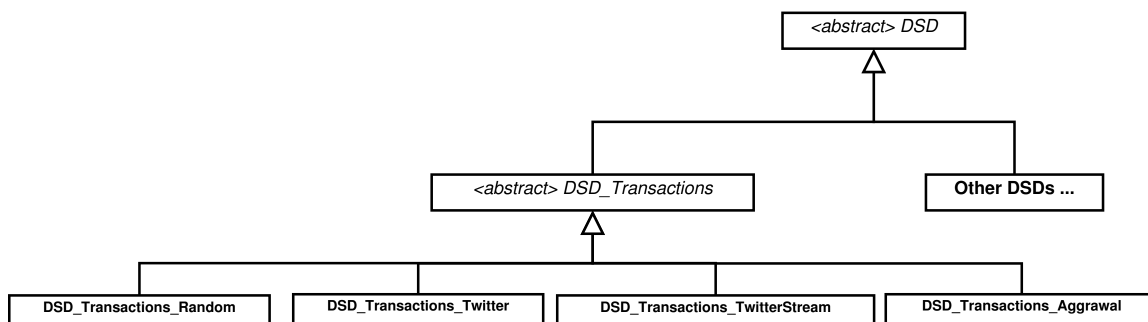

Figure 3.1. DSD Inheritance Diagram

3.1.1. DSD_Transactions_Random

This DSD produces pseudo-random transactions with items represented by integers. The user can specify the size of the set of items and the maximum size of a single transaction. By default, it uses an uniform probability distribution for selecting items, but the user can also pass their own probability and size functions, as can be seen in the constructor definition below.

```
> DSD_Transactions_Random <- function(type=c("integer"),
+   setSize=50, maxTransactionSize=10,
+      prob = function(set) rep(1/set, times=set),
+      size = function(maxSize) sample(1:maxSize, 1)
+ )
```

The following example creates a DSD_Transactions_Random object that will generation transactions that contain between 1 and 5 items, each of which is represented by a random integer between 1 and 10. It then uses get_points() to fetch 3 random transactions.

```
> rand <- DSD_Transactions_Random(setSize = 10,
```

```
+  maxTransactionSize = 5)
> get_points(rand, n = 3)

[[1]]
[1] 5 9

[[2]]
[1] 1 5

[[3]]
[1] 8 2 9 5
```

3.1.2. DSD_Transactions_TwitterStream

This DSD uses Twitter's stream API via the R package **streamR** to fetch realtime tweets. To use the stream API, one must register with Twitter and receive a consumer key and a consumer secret. Once these are obtained, you are free to search the Twitter stream, albeit with some small limitations on quantity and time. There is a limit on the number of tweets that you are allowed to download in a given time frame, but that limit is large enough and the time frame short enough as to not be much of an issue for small applications. The way the stream API works is that it samples all incoming tweets given a set of parameters, such as a search term, language, and how long to search. The API will return all relevant tweets found within this period of time. If no search term is specified, a sample of all new tweets will be returned. The following is the constructor definition. The last parameter is a parser function that takes in a single string and outputs a list of strings representing the items. This is for

splitting a tweets into individual words. It is listed as a parameter so that the user can pass their own parser function if they want to do something more complex, such as remove stop words.

```
> DSD_Transactions_TwitterStream <- function(consumer_key,
+ consumer_secret,
+ RegisteredOAuthCredentials = NULL,
+ search_term = "", timeout = 10,
+ language = "en",
+        parser = function(text)
+        unique(strsplit(
+        gsub("[^[:alnum:][:space:]#]", "", text),
+          " ")[[1]]))
```

Next we look at an example of how to use this data stream generator. To use DST_Transactions_Twitter or DST_Transactions_TwitterStream, you have to register with Twitter on their developers page. Once you have the key and the secret you have to create an OAuth object, for which the constructor should be called exactly as below.

```
>library(ROAuth)
> consumer_key <- "*****************"
> consumer_secret <- "********************************"
> cred <- OAuthFactory$new(consumerKey=consumer_key,
+                 consumerSecret=consumer_secret,
+                 requestURL='https://api.twitter.com/oauth/request_token',
+                 accessURL='https://api.twitter.com/oauth/access_token',
```

```
+                 authURL='https://api.twitter.com/oauth/authorize')
```

The handshake() function registers your credentials with twitter, which opens a webpage that contains a code to paste into the console. If you don not do this step, the Twitter API will throw an error. You can then save OAuth object and load it at a later time and it will stil be registered. If it has already been saved and registered previously, you can skip to the load step.

```
> cred$handshake()
To enable the connection, please direct your web browser to:
https://api.twitter.com/oauth/authorize?oauth_token=*************
When complete, record the PIN given to you and provide it here: *******
>
>save(cred, consumer_key, consumer_secret, file = "cred.RData")
```

The following loads a OAuth object if it was previously saved and creates the actual DSD_Transactions_TwitterStream object. It requires the key, the secret, a timeout, and a search term. RegisteredOAuthCredentials is not required, but if you do not use it, the DSD will create a new OAuth object and will have to register it with Twitter, as above. By passing an already registered OAuth object, you are skipping this step.

```
> load(file = "cred.RData")
>
> twitter <- DSD_Transactions_TwitterStream(consumer_key,
+  consumer_secret,
+  RegisteredOAuthCredentials = cred,
+   timeout = 10, search_term = "#yolo")
```

Calling `get_points()` on this object will prompt it to fetch tweets from the Twitter stream until the timeout is reached and it has found more than or exactly n tweets, the number of tweets that `get_points()()` requested. If less than n tweets are found, it doubles the search time and continues gathering tweets. It does this until the requested number is found. It then returns the tweets as a list of vectors, where each vector is a tweet. If you call `get_points()` again, it will not have to search for more tweets until all the ones it downloaded previously have been used. It should also be noted in the example below that we ask for 1 tweet, but 98 tweets were downloaded and 76 were parsed and stored. It downloaded 98 tweets because of the timeout that was set when the object was instantiated. From those 98, it removes tweets which contains characters that are incompatible with R leaving us with 76 tweets. It then returns a single tweet and stores the other 75 for later. When `get_points(twitter)` is called again, it won't have to connect to twitter until the list of stored tweets has been exhausted.

```
> get_points(twitter, n = 1)
Capturing tweets...
Connection to Twitter stream was closed after 10 seconds with
up to 98 tweets downloaded.
76 tweets have been parsed.
[[1]]
 [1] "#YOLO"      "means"      "you"        "only"       "live"       "once"
 [7] "Ive"        "clearly"    "perfected"  "the"        "term"       "better"
[13] "than"       "any"        "rapper"     "that"       "has"        "every"
[19] "lived"      "Sorry"      "Drake"

> get_points(twitter, n = 1)
```

```
[[1]]
 [1] "#followtrain"     "#TeamFollowBack" "#follows"         "#ialwaysfollow"
 [5] "tree"             "#Follow"         "#swag"            "#yolo"
 [8] "#Retweet"         "#followtoday"    "#followforfollow"
```

3.1.3. DSD_Transactions_Twitter

This DSD uses Twitter's REST API via the R package **twitteR** to retrieve tweets from an archive of stored tweets from the last several days. To use Twitter's REST API, one must register with Twitter and receive a consumer key and a consumer secret. This is the same key and secret that is used for DSD_Transactions_TwitterStream. When you have a key and secret, you are free to search all public tweets from the last several day. There is a limit on the number of tweets that you are allowed to download in a given time frame. You can search through past tweets by language, date, and a search term. The API will return all relevant tweets given the parameters. If no search term is specified, a sample all tweets from the specified time will be returned. The constructor is very similar to DSD_Transactions_TwitterStream, except instead of a `timeout`, there is a `desired_count` for how many tweets you would like to retrieve every time the DSD needs more tweets.

```
DSD_Transactions_Twitter <- function(consumer_key, consumer_secret,
 search_term, desired_count, since = NULL,
 until = NULL, sinceID = NULL, maxID = NULL,
 language = NULL, geocode = NULL,
 resultType = NULL, strip_retweets = FALSE,
     parser = function(text)
```

```
unique(strsplit(gsub("[^[:alnum:][:space:]#]", "", text), " ")[[1]]))
```

Now we will look at an example of how to use DSD_Transactions_Twitter. Here, the registration works differently from DSD_Transactions_TwitterStream. Instead of an ROAuth object, it uses the function `setup_twitter_oauth()`. This function only has to be called once per session with the user's Twitter API credentials, which can be obtained from the Twitter developer page. This function can be called before creating the DSD_Transactions_Twitter object, or the credentials can passed in the constructor, and it will be taken care of there.

```
> library(twitteR)
> consumer_key <- "*************"
> consumer_secret <- "********************"
> access_token <-  "*******************"
> access_secret <- "********************"
> setup_twitter_oauth(consumer_key, consumer_secret, access_token, access_secret)
[1] "Using direct authentication"
> twit3_28 <- DSD_Transactions_Twitter(search_term = "SMU", desired_count = 500,
+                                      language = "en")
```

Calling `get_points()` on this object will prompt it to retrieve tweets from Twitter's archive until it can find no more tweets or the desired count is met. The tweets returned are a list of string arrays, where each array is a tweet and each member of an array is a word in the tweet. When `get_points(dsd, n=1)`, it does not need to retrieve more tweets until all of the stored ones are used.

```
> get_points(twitterDSD)
[[1]]
```

```
[1] "#twitter"          "#love"              "#follows"
[4] "#AlwaysFollowBack" "tree"               "#Follow"
[7] "#THECAT"           "#yolo"              "#follownow"
[10]"#followtoday"      "#autofollow"
```

3.2. Frequent Pattern Mining DSTs

For this package, I have implemented two Frequent Pattern Mining algorithms that are completely unique from each other in terms of operation. The algorithms are estDec and Lossy Counting, both of which are described at length in the previous chapter. Just like DSD_Transactions, these classes are implemented using R's S3 object system. They inherit directly from the abstract DST class. See 3.2.

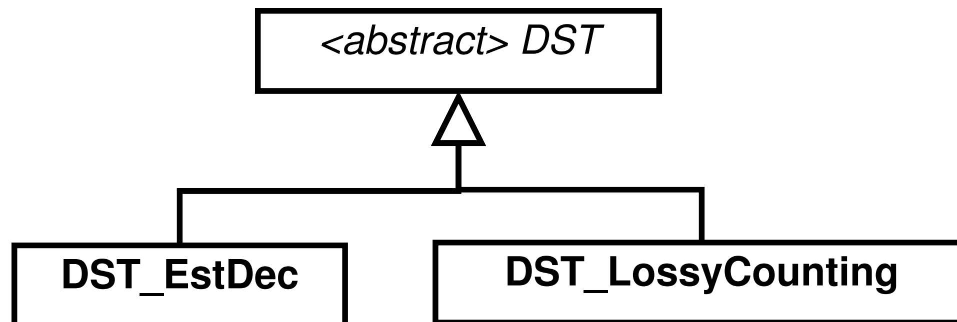

Figure 3.2. DST Inheritance Diagram

3.2.1. DST_EstDec

This class is the implementation of the estDec alogorithm in streamFPM. Most of the algorithm, including the prefix tree and all of the logic for inserting, pruning, etc

is actually written in C++ and is called using the Rcpp library in R. The R side of this algorithm is mainly responsible for containing various parameters and settings, as well as to handle to the C++ object. Also, DSD_EstDec works with both strings and integer transactions, but the prefix tree only supports integers, so I maintain a hash table in R that maps strings to integers.

The interface between R and C++ code works as follows. When a DSD_EstDec object is created in R, it also creates two C++ objects, one of class RTrie, and one of class Trie. Trie is the prefix tree, with functions for inserting, updating, and pruning. RTrie is an adapter class built to the specifications of the **Rcpp** package to map between R function calls and their C++ equivalent. The R code only ever interacts with RTrie, which then calls the appropriate function in Trie. Results are then returned to RTrie, converted to an R compatible datatype and then returned to the user. Figure 3.3 below shows a simple representation of how the classes interact.

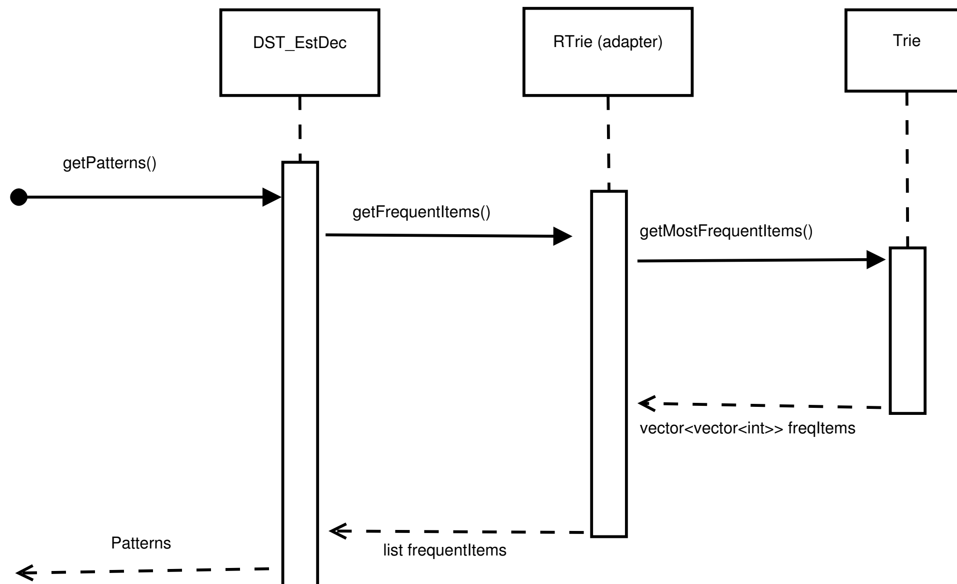

Figure 3.3. estDec Sequence Diagram for `getPatterns()`

Below is a segment of code from each class, showing how the getFrequentItemsets() function is called between classes.

```
//DST_EstDec.R
patterns <- dst$RObj$rt$getFrequentItemsets()


//RTrie.cpp
SEXP RTrie::getFrequentItemsets()
{
 std::vector<std::vector<int>> freqItems = this->getMostFrequentItemset();
 return Rcpp::wrap(freqItems);


}


//Trie.cpp
void Trie::getMostFrequentItemset()
{
//logic to find all frequent itemsets
...
return frequentItemsets
}
```

Here is an example of how to use DST_EstDec. First we make sure that we create a DSD_Transactions object that we can use to generate data. Then we create our DST_EstDec object. We are instantiating it with a minsup of .9 and a datatype of integer, because DSD_Transactions_Random returns integer transactions. After it is created, we call update() with the DST and DSD we want to use and n, the number of transactions to generate. Two more parameters, pruningSupport and

`insertSupport` can also be set to specify the support an itemset must have to not be pruned from the prefix tree and the support an itemset must have to be inserted into the prefix. As a default, these are both set at 60% of `minsup`.

```
> rand <- DSD_Transactions_Random(setSize = 100, maxTransactionSize = 10)
> estDec <-  DST_EstDec(datatype = "integer", minsup = .1)
> update(dst = estDec, dsd = rand, n=500)
```

Now that estDec has seen some transactions, we can call `get_patterns(dst)` to check which itemsets are currently frequent. It returns an object of class patterns which contains all the information about frequent items that estDec found. We can see that there are 92 different itemsets present. Now, we can see the top 5 frequent itemsets and their counts by using the `topN()` function, with n = 5. We can also convert patterns into itemsets, from the arules package [8].

```
> patterns <- get_patterns(estDec)
> patterns
Class: DST_Patterns
Set of 65 Patterns

> patterns <- get_patterns(estDec)
> topN(patterns, n= 5)
{93} {44, 59, 74, 93} {81} {81, 93} {59, 74, 93}
  6                5    5        5            5

> as.itemsets(patterns)
set of 39 itemsets
```

3.2.2. DST_LossyCounting

The implementation of the Lossy Counting algorithm is much simplier than estDec. Implemented entirely in R, it uses a hash table to track and store frequent items. Just like estDec, it is implemented for both character and integer transactions, which can be specified in the arguments. The only other argument that needs to be passed is error, which sets the the degree of error allowed. By default, error is set to 0.1.

```
DST_LossyCounting <- function(error=0.1, datatype = "integer")
```

I used a hash table for this algorithm because it is simple and and has fast lookup times. Specifically, I used the hash table implement is R's hash package. The hash tables implemented in this package utilize R environments, which have an native internal hash table. For large data structures, these hash tables perform much better than using other data structures native to R, such as a list [5].

The algorithm simply looks up the item using its name as a key and updates its values. However, deleting is slightly more complex since we must examine all the entries in the table and determine whether or not they are frequent. To do this, I get a list of the keys from the hash table and then do an sapply() with an anonymous function to remove infrequent items.

```
#dst is the DST_LossyCounting object
#dst$DH is the hash table containing frequent items

keys <- keys(dst$DH)

sapply(1:length(keys),
    function(x, b_current) {
```

```
      if(dst$DH[[keys[x]]][1] + dst$DH[[keys[x]]][2] <= b_current) {
        delete(keys[x], dst$DH)
        return(keys[x])
      }
    }, b_current = dst$RObj$b_current)
```

When `get_patterns(dst, ...)` is called, it does a similar operation, but instead of deleting infrequent items, it returns the items that meet the minimum support. It then formats these frequent items as DST_Patterns. It is different from estDec in that the minimum support does not have to be set when the object is instantiated. Instead, it is a parameter for `get_patterns()`. This can be useful when you want to try out several different supports without having to run the entire algorithm multiple times.

In the example below, we follow the same procedure as we did for DST_EstDec, except now we are using the DSD_Transactions_TwitterStream object we created earlier as our DSD. This means that the dataType has to be set to ćharacter، I have found that Lossy Counting tends to work better with smaller values for error and minsup, and this is also reflected in the example below. After we create the DST, we update it with 300 values from the DSD. Then we find frequent patterns using a minsup of 0.01.

```
>lc <- DST_LossyCounting(error = 0.001, datatype = "character")
>update(lc, twitter, n=300)
>patterns <- get_patterns(lc, minsup = 0.01)
> patterns
Class: DST_Patterns
Set of 6 Patterns
> topN.DST_Patterns(patterns)
```

RT	#yolo	YOLO	yolo	I	a
86	70	63	61	31	28

Chapter 4

Example Application

In this chapter will primarily focus on an example using DST_EstDec and DSD_Transactions_Twitter to find frequent patterns in tweets. The tweets are spread out over several days and all contain the search term “SMU”.

4.1. Setup

To start mining for frequent patterns, we will need to load the appropriate packages from R and make sure our credentials are registered with Twitter. There is a more detailed example about exactly how to do this in the previous chapter.

```
> library(streamFPM)
> consumer_key <- "*************"
> consumer_secret <- "**********************"
> access_token <-  "******************"
> access_secret <- "*********************"
> setup_twitter_oauth(consumer_key, consumer_secret, access_token, access_secret)
[1] "Using direct authentication"
```

Then I create DSD_Transactions_Twitter objects. To do this I created one for each day that I wanted tweets from. Here I created 4 objects, for each day from March 25 to March 28, 2015. I set the `since` and `until` parameters so that each object would only retrieve tweets from one particular date and I set `desired_count = 500`

so each one will retrieve about 500 tweets. The `since` and `until` parameters work from 12:00 AM on the specified date, so to get tweets from just March 28th, you would set them to `since = '2015-03-28'` and `until = '2015-03-29'`. Since we have already called `setup_twitter_oauth()`, we do not need to pass the keys and secrets as parameters.

```
> twit3_28 <- DSD_Transactions_Twitter(search_term = "SMU", desired_count = 500,
+                                         language = "en", since = '2015-03-28',
+                                         until = '2015-03-29')
>
> twit3_27 <- DSD_Transactions_Twitter(search_term = "SMU", desired_count = 500,
+                                         language = "en", since = '2015-03-27',
+                                         until = '2015-03-28')
>
> twit3_26 <- DSD_Transactions_Twitter( search_term = "SMU", desired_count = 500,
+                                        language = "en", since = '2015-03-26',
+                                        until = '2015-03-27')
>
> twit3_25 <- DSD_Transactions_Twitter(search_term = "SMU", desired_count = 500,
+                                       since = '2015-03-25', until = '2015-03-26',
+                                       language="en")
```

Next, I created the DST_EstDec object. I set the minsup low, as there will be a lot of itemsets in our stream and I want to keep the size of the prefix tree somewhat reasonable. Also, since there are only 500 tweets a day, I set the function to decay at

a relatively quick rate (a lower `decayRate` means the data will decay faster) so that we can better see the change in frequent itemsets from day to day. I also made sure that `datatype = c̈haracter¨` because we are dealing with tweets.

```
> estDec <- DST_EstDec(minsup = 0.001,
 decayRate = 0.99, datatype = "character")
```

4.2. Running the algorithm

When running the algorithm, I ran one day at a time and then retrieved the frequent patterns after each day of tweets. Every time I request the frequent patterns, it returns a snapshot of what the frequent patterns are at that moment in time. The total set of frequent itemsets can change, if only by a few itemsets, just by updating estDec with a couple more tweets from the stream. By looking at the frequent itemsets over several different days, we can see the changing ideas, or concept drift, in the data stream [10].

```
> update(estDec, twit3_23, n=500)
> patterns3_25<- get_patterns(estDec)
>
> update(estDec, twit3_24, n=500)
> patterns3_26 <- get_patterns(estDec)
>
> update(estDec, twit3_25, n=500)
> patterns3_27 <- get_patterns(estDec)
>
> update(estDec, twit3_26, n=500)
```

```
> patterns3_28 <- get_patterns(estDec)
```

These patterns are a little large to work with as is, so we can take the top 10 itemsets from each and work with those much smaller itemsets.

```
> patterns3_28
Class: DST_Patterns
Set of 2310837 Patterns
> topN3_28 <- topN.DST_Patterns(patterns3_28, n = 10)
> topN3_28
> length(topN3_28)
[1] 10
> topN3_28
with
  113
SMU
  104
Odobulu
  77
WR
  76
RT,goal,Odobulu
  74
the,SMU,from,#PonyUp,State,over,off,10,with,away
,comes,victory,Mississippi,goal,Odobulu
  74
the,a,SMU,from,#PonyUp,State,over,off,10,with,away,
```

```
comes,victory,Mississippi,goal,Odobulu
  74
the,a,from,#PonyUp,State,over,off,10,with,away,
comes,victory,Mississippi,goal,Odobulu
  74
the,#PonyUp,State,over,off,10,with,away,comes,
victory,Mississippi,goal,Odobulu
  74
from
  74
```

In the last command above, we look at the top 10 itemsets and their counts at the end of March 28th. The items in the itemsets here are separated by commas. You can see that several of the top itemsets are very similar, contain more items than you might expect and all have the same frequency. I have found that this is the case when a tweet is retweeted many times and they all show up in the stream. There were several more patterns with a count of 74, all different combinations of the words from one retweeted tweet.

With some manipulation, I coerced the 4 different top-10 lists into a single data frame of counts, where the columns are the different itemsets and each row is a different day. I then plotted all of these to show the change of all the itemsets over the four days.

```
> itemset_names <- unique(c(attr(topN3_25, "names"), attr(topN3_26, "names"),
+ attr(topN3_27, "names"), attr(topN3_28, "names")))
> itemset_df <- as.data.frame(matrix(ncol = length(itemset_names), nrow = 4))
> colnames(itemset_df) <- itemset_names
```

```
> for(i in 1:10) {
+
+   itemset_df[1,attr(topN3_25[i], "names")] <- topN3_25[i]
+   itemset_df[2,attr(topN3_26[i], "names")] <- topN3_26[i]
+   itemset_df[3,attr(topN3_27[i], "names")] <- topN3_27[i]
+   itemset_df[4,attr(topN3_28[i], "names")] <- topN3_28[i]
+ }
> dates <- as.Date(c("2015-03-25", "2015-03-26", "2015-03-27", "2015-03-28"))
> itemset_df[, "date"] <- dates
> itemset_df[is.na(itemset_df)] <- 0
> items_melt <- melt(itemset_df, id.vars = "date")
> ggplot(items_melt, aes(x = date, y = value, colour = variable)) + geom_line()
```

In Figure 4.1 you can clearly see the change of frequent itemsets over time. For some of the itemsets, the count decreases because because of the decay function in estDec. If no additional tweets contain those itemsets, they will decay until they are no longer in the top 10 itemsets. A few of the itemsets also managed to become frequent and remain in the top 10 for several days, while others where frequent for only a single day. This very much reflects the dynamism of Twitter. One tweet may be retweeted hundreds of times over the course of 1 or 2 days and then forgotten about. What is trending one day, very well might not be trending the next. With a large amount of time, a longer analysis might be done, gathering new tweets using DST_Transactions_TwitterStreamAnalysis every day and continually feeding them into a frequent pattern mining algorithm. Of course, analysis of Twitter trends is just one application of Frequent Pattern Mining on data streams, as there are as many diverse applications as there are data streams in the world.

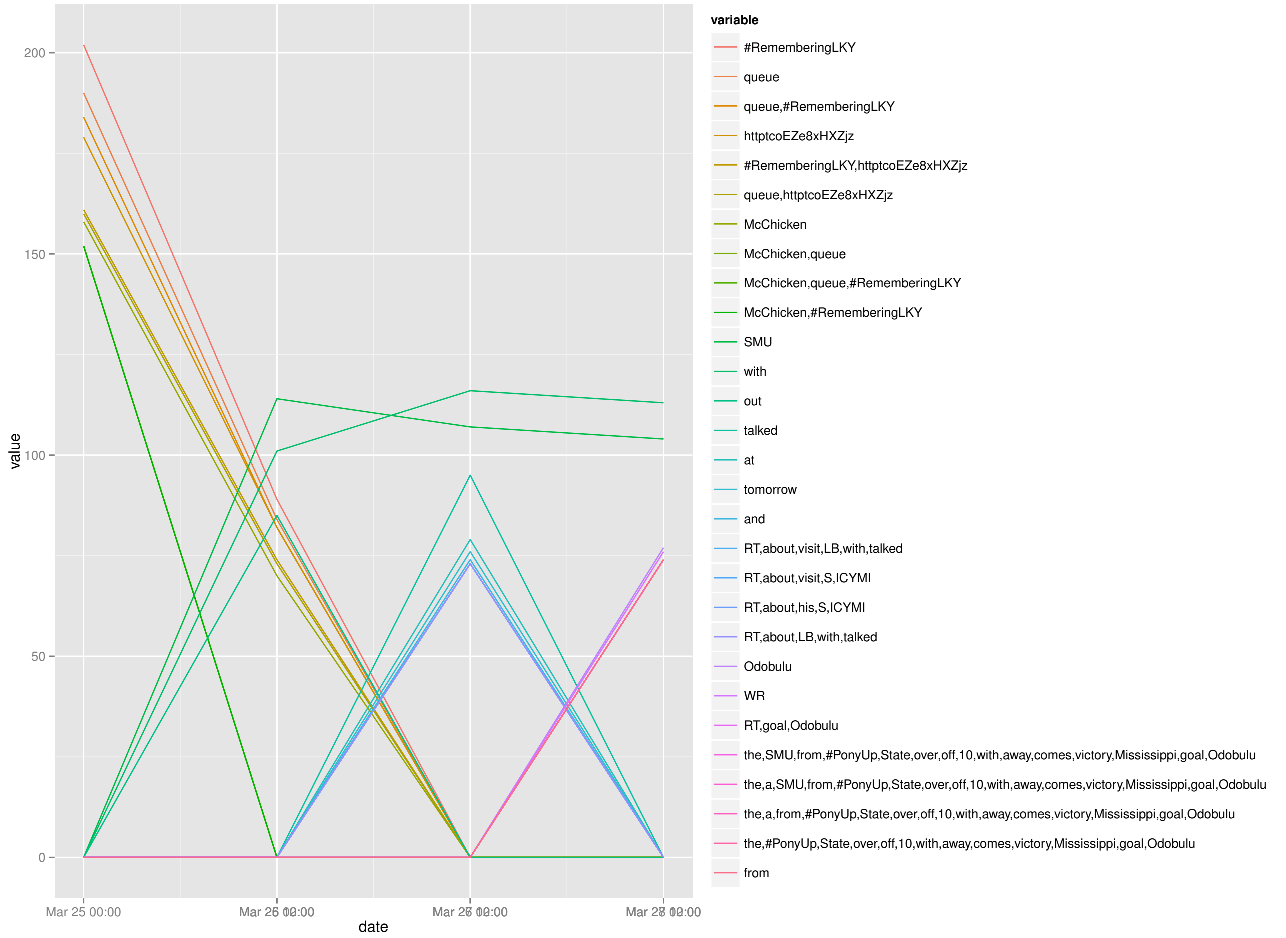

Figure 4.1. Frequent Itemsets over time

Chapter 5

Conclusion and Future Work

streamFPM is a valuable extension to **stream**, as it provides a solid foundation for frequent pattern mining on data streams in R. This is a niche that is filled by few other packages currently. Since it directly extends upon the **stream** framework, it is simple to add more features. Going forward, **streamFPM** has the opppurtunity to be expanded greatly. It could use a few more good algorithms implemented. I would also like to try to implement some parallelization into estDec and evaluate its performance.

Appendix A

APPENDIX

R documentation

of ‘streamFPM-package.Rd’ etc.

April 5, 2015

`streamFPM-package` *Stream Frequent Pattern Mining*

Description

Algorithms and data generators for mining for frequent patterns on data streams

Details

Package:	streamFPM
Type:	Package
Version:	1.0
Date:	2015-01-05

streamFPM expansions of the functionality of stream. It introduces several Frequent Pattern Mining Algorithms that can be used on streams of Transaction data.

Author(s)

Derek Phanekham <ds.phanekham@smu.edu>

`DSD_Transactions` *DSD Transactions*

Description

Abstract class. Parent class to DSDs that generate transactional data.

Usage

```
DSD_Transactions(...)
```

Arguments

...

Examples

```
stop("DSD_Transactions is an abstract class and cannot be instantiated!")
```

DSD_Transactions_Agrawal   *DSD Agrawal*

Description

A datastream generator that produces semi-random transactions within a specified range. Some items will be more frequent by design. Algorithm originally developed for IBM by Agrawal.

Usage

```
DSD_Transactions_Agrawal(type = c("integer"), setSize = 50,
  maxTransactionSize = 10, ..., verbose = FALSE)
```

Arguments

type	The type of transaction data to produce. Currently integer is the only option
setSize	The size of the set to create transactions from
maxTransactionSize	The maximum size for any one transaction
verbose	How much information the algorithm prints to the console.

Value

Returns a DSD_Transactions_Agrawal object (subclass of DSD_Transactions, DSD) which is a list of the defined params. The params are either passed in from the function or created internally

Examples

```
#create agrawal transaction datastream
dsd <- DSD_Transactions_Agrawal()

#create with a set size of 50 and a max transaction size of 10
dsd <- DSD_Transactions_Agrawal(setSize=50, maxTransactionSize=10)

#create with custom size function
dsd <- DSD_Transactions_Agrawal(setSize= 50, maxTransactionSize= 10 , size=function(max) rexp(max))

#get a transaction from the datastream
transaction <- get_points(dsd)

transaction
```

DSD_Transactions_Random

Random Transaction Data Stream Generator

Description

A datastream generator that produces random transactions within a specified range.

Usage

```
DSD_Transactions_Random <- function(type=c("integer"), setSize=50, maxTransactionSize=10,
                                        prob = function(set) rep(1/set, times=set),
                                    size = function(maxSize) sample(1:maxSize, 1) )
```

Arguments

`type`	The type of transaction data to produce. Currently integer is the only option
`setSize`	The size of the set to create transactions from
`maxTransactionSize`	The maximum size for any one transaction
`prob`	function describing the probability for each item in the itemset
`size`	function describing the probably size of any individual transaction

Details

DSD_Transactions_Random creates a DSD that generates random transactions with a specified set size and max transaction size. It allows for the use of custom functions for the probability of each number in that range, and the probable length of each transaction.

Value

Returns a `DSD_Transactions_Random` object (subclass of `DSD_Transactions`, `DSD`) which is a list of the defined params. The params are either passed in from the function or created internally

Examples

```
#create random transaction datastream
dsd <- DSD_Transactions_Random()

#create with a set size of 50 and a max transaction size of 10
dsd <- DSD_Transactions_Random(setSize=50, maxTransactionSize=10)

#create with custom size function
dsd <- DSD_Transactions_Random(setSize= 50, maxTransactionSize= 10 , size=function(max) rexp(max))

#get a transaction from the datastream
transaction <- get_points(dsd)

transaction
```

DSD_Transactions_TwitterStream
Twitter Stream transaction generator

Description

A datastream generator that uses the streamR package to create transactions by sampling the twitter stream

Usage

```
DSD_Transactions_TwitterStream <- function(consumer_key, consumer_secret,
                              RegisteredOAuthCredentials = NULL, search_term = "",
                                    timeout = 10, language = "en",
                                    parser = function(text)
                              strsplit(gsub("[^[:alnum:][:space:]#]", "", text), " ")[[1]])
```

Arguments

consumer_key
: The consumer key, provided by the twitter api

consumer_secret
: The consumer secret key, also provided by the twitter api

RegisteredOAuthCredentials
: Optional. The user can pass already registered twitter credentials using the **ROAuth** package. If the user does not pass this argument, it will attempt to register the twitter credentials using the consumer key and secret.

search_term
: If the user specifies a search term, it will return only tweets that contain that term.

timeout
: How long (in seconds) the data stream generator will sample the twitter stream when it runs out of stored tweets.

language
: When a language code is specified, all of which can be found in the twitter API documentation, it will only return tweets in that language. The default is 'en', or english.

parser
: The user can specify their own function for parsing tweets into individual words. The function must accept a string and return a list of strings. The default function removes punctuation except hashtags and splits the string by spaces.

Details

DSD_Transactions_TwitterStream creates a dsd that generates transactions from tweets sampled from the Twitter stream in real time, meaning unlike DSD_Transactions_Twitter, it receives a sampling of tweets that are happening right when the get_points() is called, rather than an archive of tweets from the last several hours or days

Value

Returns a `DSD_Transactions_TwitterStream` object (subclass of `DSD_Transactions`, `DSD`) which is a list of the defined params. The params are either passed in from the function or created internally

Author(s)

Derek Phanekham

References

https://dev.twitter.com/rest/public/search http://cran.r-project.org/web/packages/streamR/streamR.pdf

Examples

```
#create Twitter stream generator
## Not run:
key <- "************************"
secret <- "**********************"

dsd <- DSD_Transactions_TwitterStream(consumer_key = key, consumer_secret = secret, search_term="#stream", t

#create with already registered credentials

library(ROAuth)

cred <- OAuthFactory$new(consumerKey=key,
                         consumerSecret=secret,
                         requestURL= https://api.twitter.com/oauth/request_token ,
                         accessURL= https://api.twitter.com/oauth/access_token ,
                         authURL= https://api.twitter.com/oauth/authorize )
cred$handshake()

dsd <- DSD_Transactions_TwitterStream(consumer_key = key, consumer_secret = secret,
  RegisteredOAuthCredentials = cred, search_term="#stream", timeout=10)


#get a transaction from the datastream
transaction <- get_points(dsd)

transaction


## End(Not run)
```

`DSD_Transactions_Twitter`

Twitter Data Stream Generator

Description

A data stream generator that gathers and returns tweets using the Twitter REST API through the **twitteR** package. This API has access to an archive of tweets from the last several days.

Usage

```
DSD_Transactions_Twitter <- function(consumer_key,
      consumer_secret, RegisteredOAuthCredentials = NULL, search_term,
      desired_count, since = NULL, until = NULL, sinceID = NULL, lang = NULL,
      parser = function(text)
          strsplit(gsub("[^[:alnum:][:space:]#]", "", text), " ")[[1]])
```

Arguments

consumer_key	The consumer key, provided by the twitter api
consumer_secret	The consumer secret key, provided by the twitter api
access_token	The access token, provided by the twitter api
access_secret	The access secret key, provided by the twitter api
search_term	If the user specifies a search term, it will return only tweets that contain that term.
desired_count	The desired number of tweets to fetch whenever the stream is empty
lang	A language code. When a language code is specified, all of which can be found in the twitter API documentation, it will only return tweets in that language.
since	A date in the format: '01-31-2015'. It will only retrieve tweets since this date
until	A date in the format: '01-31-2015'. Must be used in conjunction with since. It will only retrieve tweets up until this date.
sinceID	A character ID in the form '123456'. It will only retrieve tweets with IDs higher than this value.
parser	The user can specify their own function for parsing tweets into individual words. The function must accept a string and return a list of strings. The default function removes punctuation except hashtags and splits the string by spaces.

Details

DSD_Transactions_TwitterStream creates a DSD that generates transactions from tweets retrieved from an archive of tweets from the past several days. To get current tweets in real time, use DSD_Transactions_TwitterStream.

Value

Returns a DSD_Transactions_Twitter object (subclass of DSD_Transactions, DSD) which is a list of the defined params. The params are either passed in from the function or created internally

Author(s)

Derek Phanekham

References

https://dev.twitter.com/rest/public/search

Examples

```
## Not run:

#create Twitter data stream generator using a key and secret from the Twitter API
consumer_key <- "************************"
consumer_secret <- "**********************"
access_token <- "************************"
access_secret <- "**********************"

twitterDSD <- DSD_Transactions_Twitter(consumer_key, consumer_secret,
  access_token, access_secret, search_term = "SMU", desired_count = 500,
  lang = "en", since =  2015-03-28 , until =  2015-03-29 )

#create with already registered credentials
library(twitteR)

setup_twitter_oauth(consumer_key, consumer_secret, access_token, access_secret)

twitterDSD <- DSD_Transactions_Twitter(search_term = "streamFPM", desired_count = 500,
lang = "en", since =  2015-03-28 , until =  2015-03-29 )

#get a transaction from the datastream
transaction <- get_points(twitterDSD)

transaction

## End(Not run)
```

DST_EstDec *estDec Recent Frequent Pattern Miner*

Description

Implements the estDec algorithm for finding recent frequent patterns in transactional datastreams.

Usage

```
DST_EstDec <- function(decayRate = 0.99, minsup = 0.1,
                        insertSupport = NULL,
                        pruningSupport = NULL,
                        datatype="character")
```

Arguments

decayRate	The decay rate for patterns found. As this number increases, the patterns found by estDec get more recent, and if it is close to 0, patterns found cover more of the transaction stream's history.
minsup	The minimum support needed for a itemset to be considered frequent.
insertSupport	The minimum estimated support an itemset needs to be inserted into the tree. If no value is specified, it is set at 60% of minsup.

`pruningSupport`	The minimum support an itemset needs to not be removed from the tree. If no value is specified, it is set at 60% of `minsup`.
`datatype`	The datatype used to represent items. The options are "integer" and "character". This depends on the kind of data the DSD used produces.

Details

estDec finds frequent patterns over a transaction or list datastream. It only keeps track of itemsets that it considers recently frequent based off of the given decayRate and minsup.

Value

An object of class `DST_EstDec` (subclass of `DST`

References

Chang, J., Lee, W. (2003) Finding Recent Frequent Itemsets Adaptively over Online Data Streams.

Examples

```
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--or do  help(data=index)  for the standard data sets.

#create datastream
dsd <- DSD_Transactions_Random(setSize=10, maxTransactionSize=4)

#create estDec
dst <- DST_EstDec()

#update estDec with new values from data stream
update(dst, dsd, 5)

#get most frequent itemsets
patterns <- get_patterns(dst)

#get the top ten most frequent patterns
topN.DST_Patterns(patterns, n=10)
```

`Patterns-class` *Class "Patterns" —*

Description

The `Patterns` class represents frequent itemsets found by a frequent pattern mining `DST`s, such as `estDec`.

Slots

`transactionInfo`**:** a data.frame with vectors of the same length as the number of transactions. Each vector can hold additional information, e.g., store transaction IDs or user IDs for each transaction.

`data`**:** object of class `ngCMatrix` to store the binary incidence matrix (see `itemMatrix` class)

`itemInfo`**:** a data.frame to store item labels (see `itemMatrix` class)

Methods

as.itemsets `signature(patterns, ...)`; produces a \classitemsets object from the **arules** package.

as.vector `signature(patterns, ...)`; produces a vector of counts with itemsets as names.

getSets `signature(patterns)`; produces a list of the itemsets countained in the \classPatterns class, with each itemset represented by a vector of item.

summary `signature(patterns)`; shows a summary of the \classPatterns class.

topN `signature(patterns, n = 1)`; returns a vector of the top `n` most frequent itemsets.

print `signature(patterns)`; represents the itemsets in a printable form

Examples

```
#create datastream
dsd <- DSD_Transactions_Random(setSize=10, maxTransactionSize=4)
#create estDec
dst <- DST_EstDec()
#update estDec with new values from data stream
update(dst, dsd, 20)

#get most frequent itemsets from the DST
patterns <- get_patterns(dst)

#show a summary of the patterns
summary(patterns)

#coerce the patterns into a list of vectors
as.vector(patterns)

#coerce into arules itemsets
library(arules)
as.itemsets(patterns)

#get the top ten most frequent patterns
topN(patterns, n=10)
```

Index

∗Topic **classes**
Patterns-class, 8
∗Topic **package**
streamFPM-package, 1

as.vector,Patterns-method (Patterns-class), 8

DSD_Transactions, 1
DSD_Transactions_Agrawal, 2
DSD_Transactions_Random, 3
DSD_Transactions_Twitter, 5
DSD_Transactions_TwitterStream, 4
DST, *8*
DST_EstDec, 7

estDec, *8*

itemMatrix, *9*

ngCMatrix, *9*

Patterns (Patterns-class), 8
Patterns-class, 8
print,Patterns-method (Patterns-class), 8

streamFPM (streamFPM-package), 1
streamFPM-package, 1
summary,Patterns-method (Patterns-class), 8

REFERENCES

[1] Manku, G., Motwani, R. *Approximate Frequency Counts over Data streams.* 28th VLDB Conference. Hong Kong, China. (2002).

[2] Aggarwal, Charu C and Han, Jiawei *Frequent Pattern Mining*. Springer (2014).

[3] Bifet, A., Holmes, G., Kirkby, R., and Pfahringer, B. *MOA: Massive online analysis.* The Journal of Machine Learning Research, vol 11. (2010). 1601–1604.

[4] Brin, S., Motwani, R., Ullman, J. and Tsur, S. *Dynamic itemset counting and implication rules for market basket data.* (1997).

[5] Brown, C. *Package 'hash'*. (2003).

[6] Chang, J. and Lee, W. *Finding Recent Frequent Itemsets Adaptively Over Online Data Streams.* (2003).

[7] Hahsler, M., Bolanos, M. and Forrest, J. *Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R* (2013).

[8] Hahsler, Michael and Grün, Bettina and Hornik, Kurt *Introduction to arules–Mining Association Rules and Frequent Item Sets.* (2007).

[9] Jin, R., and Agrawal, G. Frequent pattern mining in data streams. In *Data Streams*. Springer, 2007, pp. 61–84.

[10] Tsymbal, A. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin 106* (2004).

[11] Vijayarani, S., and Sathya, P. *A survey on frequent pattern mining over data streams.* International Journal of Computer Science and Information Technology & Security 2, 5 (2012), 1046–1050.