

Kernel-Tree: Mining Frequent Patterns in a Data Stream Based on Forecast Support

David Tse Jung Huang¹, Yun Sing Koh¹, Gillian Dobbie¹, and Russel Pears²

¹ Department of Computer Science, University of Auckland, New Zealand
dhua035@aucklanduni.ac.nz, {ykoh,gill}@cs.auckland.ac.nz

² School of Computing and Mathematical Sciences, AUT University, New Zealand
rpears@aut.ac.nz

Abstract. Although frequent pattern mining techniques have been extensively studied, the extension of their application onto data streams has been challenging. Due to data streams being continuous and unbounded, an efficient algorithm that avoids multiple scans of data is needed. In this paper we propose Kernel-Tree (KerTree), a single pass tree structured technique that mines frequent patterns in a data stream based on forecasting the support of current items in the future state. Unlike previous techniques that build a tree based on the support of items in the previous block, KerTree performs an estimation of item support in the next block and builds the tree based on the estimation. By building the tree on an estimated future state, KerTree effectively reduces the need to restructure for every block and thus results in a better performance and mines the complete set of frequent patterns from the stream while maintaining a compact structure.

Keywords: Data Streams, Kernel Regression, Frequent Pattern Mining.

1 Introduction

The introduction of FP-Tree by Han et al. [1] directed focus in frequent pattern mining from the traditional Apriori-like candidate generation-and-test approach to tree structured approaches. Most of the tree-based frequent pattern mining techniques are designed to work in static databases which only provides snapshots in time of the patterns found. As the generation of data streams increases, the ability to derive useful information in an efficient manner becomes more necessary.

Ever since its inception, data stream mining has remained one of the challenging problems in data mining. This is mainly due to the nature of data streams being continuous and unbounded in data flow as opposed to traditional databases where data is static and stable. Examples of situations where streams of data can be generated range from online retail such as auctions, book stores, telecommunications calling data to credit card transactions and weather/climate data. Because these data arrive at a fast rate and in a continuous manner, traditional techniques designed for databases which perform multiple scans on the data are

not suitable for mining these streams of data. The other issue is with memory usage during mining of such unbounded data. The capability of any technique that mines a stream is bounded by the memory space (buffer) available for storing the most recent transactions in order to update item support. If the data stream miner cannot process the data fast enough, the buffer will eventually be filled and data from the stream will inevitably be lost. This restriction requires that the mining of data streams be single pass and as fast as possible.

Techniques that mine data streams based on a tree structure have been studied recently. An example is CPS-Tree [2], which builds its tree based on the frequency-descending order of items from the previous block and thus maintains a compact tree and is an ideal single scan approach. A disadvantage of CPS-Tree is that it assumes that the state of the data stream in the next block is the same as the current block and that there is minimal to no concept drift in item support. In this paper we propose an approach called KerTree where we implement a concept drift detection mechanism to signal when item support changes from one block to the next and present an inexpensive but accurate extrapolation function that is capable of estimating item support in the future states of the data stream. An accurate assessment of the next state has a flow-on effect on subsequent states, minimizing the changes required to the tree, thus improving the overall tree maintenance time.

The key contribution of this paper is a tree structure mining technique called the KerTree that finds the complete set of frequent patterns in a data stream using a sliding window. Specifically in our approach, we use an extrapolation function to estimate the state (support) of items in a future block; thus, our extrapolation function allows us to compensate for the concept change (i.e. shifts in the pattern of support amongst items) in incoming transactions while keeping a compact tree structure.

The paper is organized as follows. In Section 2 we discuss related work. Section 3 presents the preliminaries and the detail of our proposed algorithm. Our experimentation and evaluation of KerTree and comparison with CPS-Tree is presented in Section 4. Finally in Section 5 we conclude our work and discuss future directions.

2 Related Work

In this section, we discuss some previous work that is relevant to our work. FP-Tree [1] is a widely known frequent pattern mining technique that uses a tree structure. Prior to FP-Tree, most approaches adopt an Apriori-like candidate generation-and-test approach which is usually costly and less efficient. FP-Tree is efficient at finding the complete set of frequent patterns in a database, but its multi-pass approach is unfavorable and considered too memory intensive to be used in a data stream environment.

To resolve this, Giannella et al. [3] proposed a technique called FP-Stream which adapts the FP-Tree for mining data streams. The major disadvantage of the FP-Stream approach is that it finds an approximate set of frequent patterns instead of the complete set of frequent patterns.

A tree structure that is developed by Leung et al. [4] called DSTree is a technique that finds the complete set of frequent patterns in a data stream. This particular structure is adapted from CanTree [5] which is a technique that is designed for incremental mining. DSTree, like CanTree, builds its tree structure based on a canonical-ordering of items that is determined by the user prior to the mining process or at runtime during mining.

Tanbeer et al. [2] proposed a tree structure called CPS-Tree that builds a compact tree and mines the complete set of frequent patterns. CPS-Tree has a frequency descending structure that captures data from the dataset and dynamically restructures itself. The tree construction of the CPS-Tree has two phases: insertion phase and restructure phase. In the insertion phase, the items in the currently processed transaction are inserted into the tree based on the current sorted order of the item list and the frequency counts of the items are updated. During the restructure phase, the item list is sorted based on frequency descending order and the tree is reorganized based on the newly sorted item list. The two phases are executed consecutively. CPS-Tree works under the assumption that drift within the stream remains fairly constant, an assumption which is not always true in practice. The relaxation of this assumption requires both a drift detection mechanism as well as an estimation mechanism which our proposed approach implements.

Koh et al. [6] proposed the Extrap-Tree, which builds the prefix tree based on a forecast support, like that of KerTree. However, Extrap-Tree uses a landmark model where linear interpolation is used as the estimation mechanism. In a landmark model, the support of items increases monotonically with time, making linear interpolation the suitable estimation function whereas the KerTree introduced in this paper assumes a sliding window approach where the support of items can both increase and decrease with time.

3 Kernel-Tree (KerTree)

In this section we will first give the general framework of KerTree then later in the section we will detail each of the steps taken to construct KerTree for mining frequent patterns.

3.1 Preliminaries

In this section we provide definitions of key terms that explain the concepts of frequent pattern mining in a data stream. A transaction data stream $\mathcal{S}$ can be formally defined as an infinite sequence of transactions, $\mathcal{S} = [t_1, t_2, t_3, \dots, t_m]$, where t_i is the i th transaction in the data stream and $i \in [1, m]$. A window W is a set of all transactions between the i th and j th (where $j > i$) transactions and the size of W is $|W|$ calculated by $j - i$. The frequency of an itemset X denoted as $count_W(X)$ in a window W , is the number of transactions in W that contain X . The support of X in W , denoted as $supp_W(X)$, is defined as

$\frac{count_W(X)}{|W|}$, where $|W|$ is the size of the window W . X is a frequent itemset in W if $supp_W(X) \geq minsup$ where $minsup$ is the minimum support threshold.

Given a transaction data stream and a $minsup$, the problem of frequent itemset mining over a window W in the transaction data stream S , is to find the set of all itemsets whose support is greater than $minsup$. A block B is a set of transactions with the size of $|B|$. A data stream is consisted of a continuous sequence of blocks, $\mathcal{S} = [B_1, B_2, \dots]$.

3.2 Framework and Overview of KerTree

KerTree, like many other tree structured pattern mining techniques, goes through two phases: construction phase and mining phase. During the construction phase, a tree is storing the items in the transactions is built. The mining phase follows the construction phase where FP-Growth proposed in the FP-Tree [1] is used to mine the tree for frequent itemsets. FP-Growth is also used in some previous work [5,2,7,6].

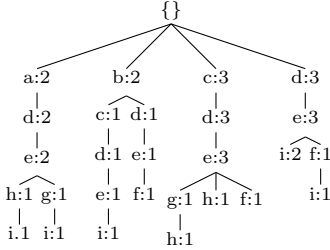
The construction of KerTree consists of several steps. First, an initial tree is built as transactions are read from the stream. At the end of each block, KerTree uses a delay condition to determine whether the initial tree built requires re-structuring. If re-structuring is deemed unnecessary, all following construction steps are skipped and KerTree goes into mining phase. If re-structuring is deemed necessary, KerTree continues with the construction phase by determining whether there is drift in the support of items from the previous block to the current block. Then, the ordering of items in the tree is adjusted based on the results using kernel regression. Lastly, the tree is re-structured based on the adjusted ordering.

Figure 1 is an example of KerTree showing the construction of the tree in the first block. Shown in Figure 1 (a) is the result of the initial tree built at the end of the block. Because this is an example of the first block, items are initially ordered in the canonical ordering (alphabetical). For all blocks after the first, the initial tree is built using the forecast ordering of items from the previous block. This ordering is determined using kernel regression. Using kernel regression as the extrapolation function to forecast item supports in the future is also one of the major contributions of this paper. The actual technique on how we perform kernel regression is detailed in Section 3.3. For the first block of the stream, we always deem the tree needing re-structuring. After the first block, whether we decide to re-structure and continue with construction is determined by the delay condition presented in Section 3.4. If re-structuring is needed, KerTree first uses a drift detection mechanism to detect changes in item supports. The details of this mechanism is presented in Section 3.5. After determining drift and forecasting item supports using kernel regression, we continue with re-structuring the tree based on the forecast order and the resulting tree is shown in Figure 1 (b). Notice that the re-structured tree is more compact (fewer nodes). When the second block of transactions come in, the tree will be updated and built according to the forecast order determined in block one. At the end of block two, KerTree will then go through the same procedures as presented above.

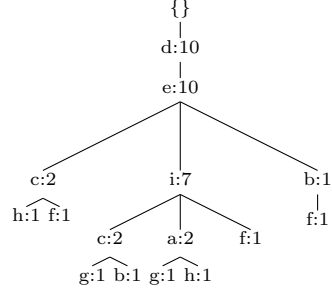
Transactions for the first block			
tid	transactions	tid	transactions
1	a d e g i	6	d e i
2	a d e h i	7	c d e h
3	b c d e i	8	d e f i
4	b d e f	9	d e i
5	c d e g i	10	c d e f

Canonical Order	Forecast Order
a	d
b	e
c	i
d	a
e	c
f	g
g	b
h	h
i	f

(a) Initial tree based on canonical order



(b) Reorganization based on forecast order

**Fig. 1.** Example of KerTree

3.3 Kernel Regression

Kernel regression is a non-parametric regression technique. Unlike linear regression or polynomial regression, the underlying distribution to estimate the kernel function is not assumed.

The problem is: given a set of data points (t, y_t) where t indicates a time point in the stream and y_t the support of a given item, we find a regression function that best fits the given data. Given the best fit function, we can determine the estimated support $\hat{y}_{t+1}$ at time $t+1$. The set of data points (t, y_t) used in KerTree is obtained through tracking the support of items in the current block at various points. The amount of points tracked is user defined and can be increased to improve estimating accuracy (but reducing efficiency). The points are assigned in a manner where the distance from one point to the next is uniform. For example, given a block B of 10 transactions with 5 points defined, the support of an item would be tracked at transactions 2, 4, 6, 8, and 10. Generally about five to ten points is enough to provide the regression technique with sufficient data for an accurate estimation.

In kernel regression, a weight function called a kernel is first associated at each time point, t . The kernel then estimates the probability of point x occurring based on the weight function of the kernel (ie. Radial Basis Kernel estimates the probability using a Gaussian function). Apart from the probability calculated at each kernel, a weight is also assigned to each kernel. In our implementation we used the KRLS algorithm [8] to calculate the optimal weight vector α . The optimal weight vector is obtained through minimizing the sum of the squared errors calculated as the difference between the estimated and actual support values. The estimated value $\hat{y}_{t+1}$ at the $(t+1)^{th}$ observation is given by:

$$\hat{y}_{t+1} = \frac{\sum_{j \in s} w_j y_j}{\sum_{j \in s} y_j}$$

where s is the set of data points or kernels; w_j is the kernel weight of the j^{th} kernel data point. Each $\hat{y}_j$ is fitted by the Polynomial kernel function of degree 2: $(z \cdot y_j + 1)^2$, where z takes values in the range $[0, 1, 2, \dots, (\frac{s}{dX} + 1)]$; dy is the sampling rate for each observation y_j in the current block t .

Thus in KerTree, by calculating the estimated support for each item at a future block, we can generate a forecast order of items at which the tree in the next block can be built from.

3.4 Delay Condition

The reason that sometimes restructuring of the tree could be delayed at some blocks is because we are constructing the tree based on forecast support of items from the previous block; therefore, if the prediction is accurate, the initial tree built would already have been representative of the state of items in the current block and is usually a compact tree. However, even though our proposed kernel regression extrapolation function is capable of predicting the support of items in the near future, the estimation function will eventually lose its predicting strength as time progresses and a tree reorganization will be inevitable at some stage. We deem whether reorganization of the tree is necessary based on a tree growth threshold (γ). The growth threshold is measured using the number of nodes in the tree at the previous block and comparing it with the number of nodes in the tree at the current block. We then calculate the percentage of increased nodes which we call percentage of growth. If the percentage of growth is less than γ , then we delay reorganization at the current block and skip drift detection and extrapolation in the rest of the construction phase.

3.5 Drift Detection

After deciding to re-structure the tree based on growth threshold γ , we go into drift detection where we determine whether a drift in the support of items is present or absent. We then use the extrapolation function (kernel regression) to forecast the support of items if a drift is deemed present. A drift is a change in the support of an item from the previous block to the current block. For example, if item X is seen with a support of 0.8 in the previous block and 0.2 in the current, then we say that there is a drift. We use the drift detection mechanism proposed in Extrap-Tree called the Hoeffding Bound [9] in KerTree.

In order to determine whether there is a concept drift in the support of items from the previous block to the current block, we utilize the change in support of an item from the previous block to the current block given by:

$$\Delta supp(X) = supp_{B_{n-1}}(X) - supp_{B_n}(X) \quad (1)$$

We use the Hoeffding Bound to determine whether a drift is present. The Hoeffding bound states that, with probability $1 - \delta$, the true mean of a random variable r is at least $\bar{r} - \epsilon$ when the mean is estimated over t samples, where

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2t}}$$

R is the range of r . In our case the variable r is denoted by $|\Delta supp(X)|$, which has a range value R of 1, and the number of samples $t = |B|$, the block size. There are two possible scenarios that we need to consider when using $|\Delta supp(X)|$ to determine drift. In the first scenario, we assume that there is no concept drift when $|\Delta supp(X)| \leq \epsilon$ and the previously forecasted support holds the same and we do not need to re-calculate the support of item X using kernel regression. In the second scenario, we assume that concept drift is present when $|\Delta supp_{B_n}(X)| > \epsilon$ and we need to re-calculate and forecast the support of item X .

To obtain $supp_{B_{n+1}}^{extrap}(X)$ when concept drift is assumed present, we use kernel regression to estimate and extrapolate item support at the next block as specified earlier in Section 3.3.

4 Experimental Results

We have conducted various experiments on both real-world and synthetic datasets to test the efficiency and efficacy of our proposed technique. The algorithms are all coded in Microsoft Visual C++ and run on a Intel Core i5-2400 CPU @ 3.10 GHz with 4GB of RAM running Windows 7 x64. In all experiments, runtime excludes I/O costs.

4.1 Synthetic Data Generator

In order to evaluate the performance of KerTree with relation to the effects of concept drift, we modified the IBM synthetic data generator to inject drift patterns into the generated datasets based on known polynomial equations shown in Table 1. These equations allow us to model and control different item distributions and produce different datasets of varying characteristics. The stream is based on a Gaussian distribution with the polynomial equations giving the probability at which we introduce or remove itemsets. The generator produces large and small itemsets with the items in the large set having higher support than those in the small set. We select x random points at which we introduce or remove large itemsets. Through introducing and removing these itemsets at these points, we are able to simulate drifts in patterns in the dataset. The introduction of large itemsets at a drift point corresponds to the simulation of emerging frequent patterns in the dataset whereas the removal of large itemsets corresponds to the simulation of disappearing patterns in the dataset.

The eight synthetic datasets generated in our experiments used the same parameters where the average transaction length was 10, the number of frequent

Table 1. Kernel equations

Dataset	Equation	Dataset	Equation
Synthetic 1	x^2	Synthetic 5	$-4x^3 - 3x^2 + 5x - 10$
Synthetic 2	$3x^2 - 4x + 10$	Synthetic 6	$-3x^2 + x$
Synthetic 3	$-2x^2 + 10x - 4$	Synthetic 7	$4x^3 - 4x + 4$
Synthetic 4	$3x^3 + 4x^2 - x + 1$	Synthetic 8	$3x^2 - 4x - 10$

items was 100, the average length of a large item was 4, the number of drift points is 50,000 and the number of transactions was 1 million using 1000 unique items.

4.2 Experimentation on Synthetic Data

In this section we present our experimentation and evaluation on the eight synthetic datasets. We evaluate our approach against CPS-Tree based on the following features: (1) Overall execution time, (2) Average number of nodes in tree, and (3) Correlation of the extrapolated support vs. the actual support. In the first evaluation, we primarily look at the performance of KerTree against the CPS-Tree through comparing the execution time. In the second evaluation, we aim at comparing the compactness of the trees built between KerTree and the CPS-Tree by looking at the average number of nodes in the trees built by the two techniques. In the third evaluation, we particularly wanted to look at the accuracy of our kernel extrapolation function through the correlation measure of the extrapolated support vs the actual observed support.

For all the synthetic dataset experiments we used varying block sizes of 50K, 100K, 150K and 200K, minsup of 0.1, tree growth threshold (γ) of 0.01, Hoeffding bound (δ) of 0.001, and a polynomial kernel of degree 2. Table 2 reports the comparison based on execution time. For ease of comparison, we have included the relative time measure of KerTree vs. CPS-Tree, which is calculated as follows:

$$\text{Relative Time (Rel. Time)} = \frac{\text{Time taken by KerTree}}{\text{Time taken by CPS-Tree}}$$

Table 2. Comparison based on Execution Time

Dataset	50K			100K			150K			200K		
	Ker-Tree	CPS-Tree	Rel. Time	Ker-Tree	CPS-Tree	Rel. Time	Ker-Tree	CPS-Tree	Rel. Time	Ker-Tree	CPS-Tree	Rel. Time
Synthetic 1	14.57	18.24	0.8	13.73	17.16	0.8	13.45	16.07	0.84	13.43	15.94	0.84
Synthetic 2	13.67	16.88	0.81	13.26	15.79	0.84	13.2	15.01	0.88	13.04	14.96	0.87
Synthetic 3	13.85	16.77	0.83	13.48	15.79	0.85	12.93	14.96	0.86	12.87	14.87	0.87
Synthetic 4	13.85	17.32	0.8	13.31	16.47	0.81	13.14	15.51	0.85	13.1	15.69	0.83
Synthetic 5	14.87	17.32	0.86	13.67	16.33	0.84	13.57	15.59	0.87	13.67	15.69	0.87
Synthetic 6	14.24	16.6	0.86	13.21	15.76	0.84	12.84	14.91	0.86	12.92	14.99	0.86
Synthetic 7	13.56	16.65	0.81	13.37	15.68	0.85	12.82	14.79	0.87	12.82	14.79	0.87
Synthetic 8	14.42	17.3	0.83	13.71	16.43	0.83	13.78	15.8	0.87	13.68	15.51	0.88

Looking at Table 2 we can see that KerTree is about 12%-20% faster for all experiments against CPS-Tree. This is shown in Figure 2 where the execution time for KerTree is plotted against the execution time of CPS-Tree to show a comparison. The reduction in execution time is due to the ability of KerTree to delay the reorganization phase in a majority of the blocks by accurately predicting the future support of items.

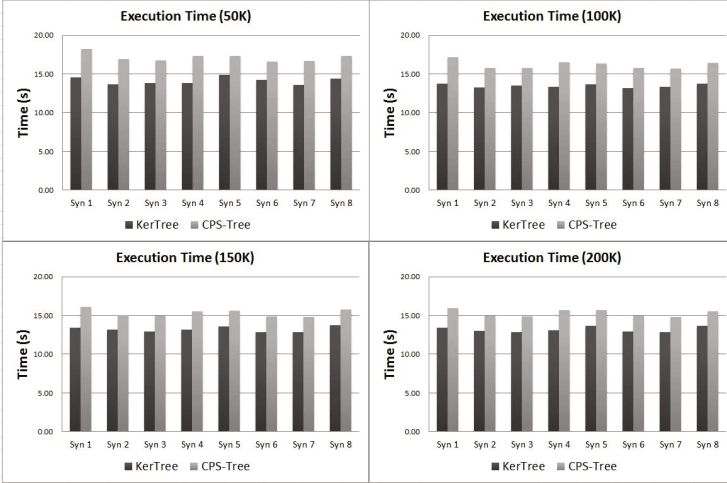


Fig. 2. Execution Time

For the comparison based on the average number of nodes in tree, we calculate the percentage variance of the trees constructed by KerTree and CPS-Tree. We observed that the percentage variance ranges from 0.001% to 0.261%. This shows that the two techniques end up with tree structures that are similar in the number of nodes and therefore similar in compactness.

Table 3 shows the comparison of KerTree against the CPS-Tree based on the number of nodes in the tree constructed for each block for synthetic dataset 6. The table also presents whether our KerTree delays reorganization and the correlation associated with the actual and forecasted support of items in the block. We can see that our technique is capable of delaying a majority of the reorganization phase which contributes to the faster execution time. The correlation of the forecasted supports and actual supports of items observed ranges from 0.9995 to 0.9997 which signifies that the predictions based on our kernel regression extrapolation is accurate at predicting item supports in future blocks. Note that the first block has no associated correlation value because the item support in the first block is not extrapolated prior to that and the tree is constructed based on canonical ordering.

Table 3. Number of Nodes from Synthetic Dataset 6

Window	KerTree		CPS-Tree		Correlation
	After Reorganization	Current Nodes in Tree	After Reorganization		
50000		80301		80358	-
100000	Delayed			77790	0.9997
150000	Delayed			77054	0.9996
200000		78607		78527	0.9995
250000	Delayed			77593	0.9997
300000	Delayed			77886	0.9996
350000	Delayed			77663	0.9996
400000	Delayed			77385	0.9996
450000	Delayed			78073	0.9996
500000	Delayed			77400	0.9995
550000	Delayed			76736	0.9996
600000		78436		78254	0.9995
650000	Delayed			77797	0.9997
700000	Delayed			78315	0.9996
750000	Delayed			77858	0.9996
800000	Delayed			77024	0.9996
850000		78305		78214	0.9995
900000	Delayed			77834	0.9997
950000	Delayed			77867	0.9996
1000000	Delayed			77585	0.9995

4.3 Experimentation on Real World Data

This section presents the results of comparing our KerTree to CPS-Tree on real world datasets from the FIMI repository (<http://fimi.ua.ac.be/>) and UCI repository (<http://archive.ics.uci.edu/ml/>). Table 4 shows the results based on execution time and relative time and Table 5 and Table 6 shows the number of nodes in the tree at each block and the correlation of forecasted supports and actual supports of items.

The table also identifies blocks at which KerTree decides to delay the reorganization phase. The results produced by these datasets follow the same trend as the results produced by the synthetic datasets. Both KerTree and CPS-Tree produce a similar number of nodes, whereas the total time taken by KerTree remains lower than that of CPS-Tree. A graph representation of the execution time for the real world datasets is shown in Figure 3.

From Table 5 and Table 6 we can see that for real world datasets, the correlation values we observed are still very promising with lowest in BMS-POS being 0.9634 and Kosarak being 0.9941. This signifies that our kernel regression estimation function is capable of predicting item supports in these real-world datasets.

Table 4. Results based on Real-World Dataset

Dataset	Block Size	KerTree		CPS-Tree		Rel. Time
		Avg. Nodes	Time(s)	Avg. Nodes	Time(s)	
BMS-POS	25K	177575	6.63	177247	8.00	0.83
Kosarak	25K	271999	26.57	271667	29.47	0.90
Poker-hand-testing	25K	241894	15.04	242045	22.07	0.68
T10I4D100K	10K	79536	2.20	79501	2.76	0.80
T40I10D100K	10K	365386	7.41	365380	9.92	0.75

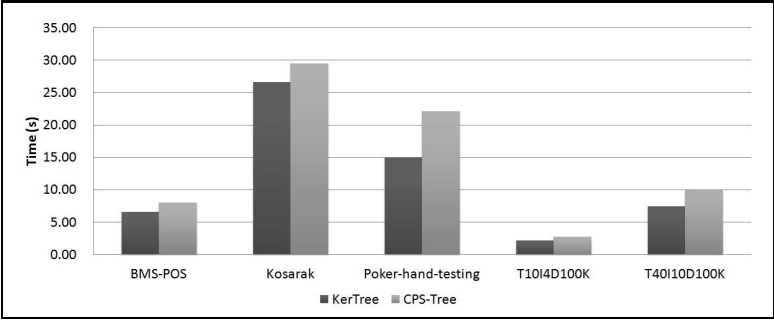


Fig. 3. Real World Dataset Execution Time

Table 5. Number of Nodes from BMS-POS

Window	KerTree		CPS-Tree	Correlation
	After Reorganization	Current Nodes in Tree	After Reorganization	
50000		206419	206440	-
100000	Delayed	187126	186689	0.9978
150000	Delayed	178284	178091	0.9936
200000	Delayed	171222	171095	0.9943
250000	Delayed	139596	138964	0.9939
300000	Delayed	129738	129128	0.9766
350000	Delayed	119447	118496	0.9820
400000		199553	199488	0.9634
450000		224030	223989	0.9971
500000	Delayed	220338	220085	0.9969

Table 6. Number of Nodes from Kosarak

Window	KerTree		CPS-Tree	Correlation
	After Reorganization	Current Nodes in Tree	After Reorganization	
50000		268589	268456	-
100000		272690	272580	0.9949
150000	Delayed	269602	269421	0.9946
200000		274966	274818	0.9946
250000		282401	282285	0.9946
300000	Delayed	280351	280242	0.9942
350000	Delayed	278955	278740	0.9942
400000	Delayed	278764	278681	0.9945
450000	Delayed	280405	280317	0.9943
500000	Delayed	266822	266717	0.9941
550000		272464	272283	0.9945
600000	Delayed	269292	269216	0.9944
650000		274358	274216	0.9945
700000	Delayed	276249	276100	0.9944
750000	Delayed	276956	276827	0.9944
800000		282049	281936	0.9943
850000	Delayed	269211	269169	0.9942
900000	Delayed	270788	270657	0.9945
950000		275478	275392	0.9945
1000000	Delayed	215350	215253	0.9941

5 Conclusions and Future Work

In this paper we have proposed the Kernel-Tree algorithm that mines frequent patterns from a data stream based on forecast support of items. From the results of our experiments we have shown that kernel regression is a suitable technique that is capable of accurately predicting item supports in a data stream at a future time. Because KerTree can preemptively build trees based on a predicted future state, the need to restructure the tree is largely reduced resulting in a faster run-time. KerTree also maintains a similar level of tree compactness as the CPS-Tree while capable of achieving a faster overall run-time. Our future work includes testing the effectiveness of kernel regression using different types of kernels (weight functions) and also looking at cost functions.

References

1. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. *SIGMOD Rec.* 29, 1–12 (2000)
2. Tanbeer, S.K., Ahmed, C.F., Jeong, B.S., Lee, Y.K.: Efficient frequent pattern mining over data streams. In: *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008*, pp. 1447–1448. ACM, NY (2008)
3. Giannella, C., Han, J., Pei, J., Yan, X., Yu, P.S.: Mining frequent patterns in data streams at multiple time granularities (2002)
4. Leung, C.K.S., Khan, Q.I.: DSTree: A tree structure for the mining of frequent sets from data streams. In: *Sixth International Conference on Data Mining, ICDM 2006*, pp. 928–932 (December 2006)
5. Leung, C.K.S., Khan, Q.I., Li, Z., Hoque, T.: CanTree: a canonical-order tree for incremental frequent-pattern mining. *Knowl. Inf. Syst.* 11, 287–311 (2007)
6. Koh, Y.S., Pears, R., Dobbie, G.: Extrapolation Prefix Tree for Data Stream Mining Using a Landmark Model. In: Cuzzocrea, A., Dayal, U. (eds.) *DaWaK 2012*. LNCS, vol. 7448, pp. 340–351. Springer, Heidelberg (2012)
7. Koh, Y.S., Dobbie, G.: SPO-Tree: Efficient Single Pass Ordered Incremental Pattern Mining. In: Cuzzocrea, A., Dayal, U. (eds.) *DaWaK 2011*. LNCS, vol. 6862, pp. 265–276. Springer, Heidelberg (2011)
8. Engel, Y., Mannor, S., Meir, R.: The kernel recursive least-squares algorithm. *IEEE Transactions on Signal Processing* 52(8), 2275–2285 (2004)
9. Hoeffding, W.: Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association* 58(301), 13–30 (1963)