Mining Time-Changing Data Streams

by

Yingying Tao

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2011

Abstract

Streaming data have gained considerable attention in database and data mining communities because of the emergence of a class of applications, such as financial marketing, sensor networks, internet IP monitoring, and telecommunications that produce these data. Data streams have some unique characteristics that are not exhibited by traditional data: unbounded, fast-arriving, and time-changing. Traditional data mining techniques that make multiple passes over data or that ignore distribution changes are not applicable to dynamic data streams. Mining data streams has been an active research area to address requirements of the streaming applications. This thesis focuses on developing techniques for distribution change detection and mining time-changing data streams. Two techniques are proposed that can detect distribution changes in generic data streams. One approach for tackling one of the most popular stream mining tasks, frequent itemsets mining, is also presented in this thesis. All the proposed techniques are implemented and empirically studied. Experimental results show that the proposed techniques can achieve promising performance for detecting changes and mining dynamic data streams.

Acknowledgments

I would like to thank all people who have helped and inspired me during my Ph.D study.

It is difficult to overstate my appreciation to my supervisor, Dr. M. Tamer Özsu, who shared with me a lot of his expertise and research insight. His wide area of knowledge, thoughtful advises, understanding and encouraging have laid a good basis for this thesis. I thank him for all the support during my Ph.D study.

I am heartily thankful to Dr. Wayne Oldford for his guidance and support during the year of my thesis revision. His insights in statistical analysis and testing is second to none. He enabled me to develop an deeper understanding on the subject.

I am grateful Dr. Grant Weddell, Dr. Ajit Singh, and Dr. Raymond Ng for their willingness to serve my committee. I also like to warmly thank Calisto Zuzarte and IBM Toronto Software Lab. for their support.

I would like to express my deep gratitude to Dr. Qiang Zhu, my former supervisor during my Master study, for bringing me into the world of research and guiding me through many difficulties.

I owe my sincere thanks to everyone whom I have shared experiences in life. This is a long and never ending list so I won't be able to put all the names down.

Lastly, and most importantly, I wish to thank my parents, Tao Jiang and Song Heng. They bore me, raised me, taught me, and loved me. To them I dedicate this thesis.

Contents

List of Figures

List of Tables

List of Symbols

Δ	size of window in terms of time
ϵ_1, ϵ_2	stopping rule thresholds
UCL, LCL	upper and lower control limit of control chart C
κ	distance of control limits from center line
Λ	weight matrix
λ	threshold for calculating frequent itemset candidates
$\mu(S)$	mean value of S
ν	threshold for calculating the support of frequent itemsets
ρ	probability that a T^2 control chart has chi-square distribution
$\sigma(S)$	standard deviation of S
τ	significance level
ω	weight for calculating moving mean and standard deviation
$\mathcal{A}_i$	frequent itemset
$\mathcal{A}$	complete set of frequent itemsets in S
$\mathcal{A}^C$	cover set of $\mathcal{A}$
$\mathcal{A}^{SC}$	smallest cover set of $\mathcal{A}$
$Acc(R_i)$	the accuracy of the mining result R_i
C	control chart
$\mathcal{C}$	candidate list for frequent itemsets mining
d	number of dimensions in multi-dimensional streams
G_i	partitions of a distribution
g_i^S, g_i^E	start and end point of partition G_i
h	bandwidth parameter for kernel density estimation
H_0	Null hypnosis that asserts distribution does not change
H_1	Alternative hypnosis that asserts distribution has changed

$\mathcal{I}, I_i$	set of items for transactional stream
$K()$	kernel function for kernel density estimation
k	number of partitions in the distribution
M	covariance matrix
N_t	number of transactions received by time t
$n(t', t]$	number of data received within time $(t', t]$
occ_i	number of times distribution P_i occur in S
$\mathcal{P}$	set of important distributions observed in S
P, P_A, P_i	probability distribution of S, S_A, S_i
$\mathcal{P}(\mathcal{A})$	power set of itemset $\mathcal{A}$
$p(v_i)$	probability of $v_i \in v(s)$ occurring in S
R_i	mining results for S under distribution P_i
S	data stream
S^d	d-dimensional data stream
s, s_i	data element in S
S_A, S_i	substream in S
S_r	substream representing current distribution
StR	Stopping rule
$sup(\mathcal{A})$	number of transactions that support $\mathcal{A}$
$SUP(\mathcal{A})$	the support of $\mathcal{A}$
t, t_i	timestamp
t_a	timestamp of the last distribution change
$\mathcal{T}_i$	a transaction that access itemset I_i
$\mathcal{T}$	set of transactions
$v(s)$	value domain of the data s
W	window on S
$\|W\|$	size of window in terms of number of elements
W_M	maintenance window containing frequent itemsets
W_P	prediction window containing a candidate list
W_r	reference window containing representative set S_r
W_t	observation window containing latest data

Chapter 1

Introduction

1.1 Data stream environment

1.1.1 Data streams

Traditional database management systems (DBMSs) are successful in many real-world applications where data are modeled as persistent relations. However, in the past decade, a set of applications has emerged that involve processing large volumes of continuous data. The data involved in these applications come in the form of *streams*. They are generated continuously and in fixed order; the large volume (often assumed to be unbounded) of the data that arrive in the stream makes it impossible to store the entire stream on disk, and in many applications the data arrival rate is high (e.g., hundreds or even thousands data per second). The following are some typical examples of such applications:

- Sensor networks are becoming increasingly popular for environmental and geophysical monitoring [10, 165], traffic monitoring [97], location tracking [64], surveillance [162], and supply-chain analysis [58]. The measurements produced by sensors can be modeled as a continuous and unbounded stream of data.

- Financial and market activities continuously generate data such as point-of-sale purchase transactions, stock tickers, real time prices, and foreign exchange rates [88, 118]. On-line analysis of these data can identify important market activities and economic patterns.
- In telecommunications, an overwhelming amount of telephone call records is generated every minute [17]. Analyzing such telephone logs in real-time may reveal interesting customer spending patterns and may improve service quality as a result.
- In the networking community, much recent interest has focused on on-line monitoring and analysis of network traffic [112, 126]. Tasks in this context include bandwidth usage tracking, routing system analysis, and server attack detection. The IP packet headers collected from a web site can be modeled as a data stream.

Traditional DBMSs are not well equipped for such data streams. For example, many techniques used in conventional DBMSs require multiple scans over the entire data set; however, since data streams are unbounded, in general we can only have one look at the data. Once the data is processed and discarded, we will not be able to get it back. Therefore, a new class of systems known as *Data Stream Management Systems (DSMSs)* are being studied by the database research community to fulfill the needs of managing and processing data streams.

1.1.2 Data stream management systems

Recently, many DSMS prototypes have emerged to specifically support stream processing applications. Stonebraker et al. summarized the eight requirements that any DSMS must fulfill [148]:

1. Keep the data moving.
2. Query using SQL on streams (e.g., StreamSQL).
3. Handle stream imperfections (delayed, missing and out-of-order data).

4. Integrate stored and streaming Data.
5. Generate predictable outcomes.
6. Guarantee data safety and availability.
7. Partition and scale applications automatically.
8. Process and respond instantaneously.

Figure 1.1 illustrates the abstract system architecture of most proposed DSMSs. The high-volume, low-latency, and time-changing streams arrive for processing in real time. An input monitor then regulates the input rates by dropping some data when the DSMS is unable to keep up with the stream arrival speed. An allocated memory space is used as working storage for processing the input streams. This memory space contains only a portion of the stream that usually consists of the latest data, and is maintained automatically by expiring oldest data when new data arrive. Additional storage (usually allocated on disks) can be used for maintaining important data in the stream, or for storing some auxiliary information such as query plans and indexes. The output of the DSMS can have different forms depending on the stream processing applications. This can be a subset of data in the stream (e.g., for queries), an alert (e.g., for fraud detection), or some extracted information (e.g., for association rule mining). Some DSMSs are built on top of existing DBMSs, so that they may exchange information and take advantage of some components in the comparatively mature DBMSs.

1.1.3 Brief history of data streams

Although database research community's interest in streaming data is recent, the idea of data streams can be traced back to almost half a century ago. In the 1960s, Landin formulated the term *stream* for the use of modeling the histories of loop variables when designing unimplemented computing languages [84]. However, within the next few decades, data stream and its processing techniques were mostly identified and developed within

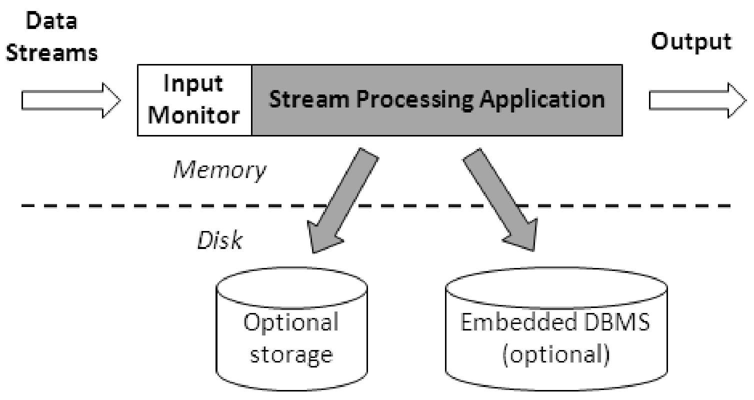


Figure 1.1: Abstract architecture of a DSMS

the literature of *data flow* [2, 43, 79, 161]. Data flow is predominately concerned with the development of parallel processing techniques and the evaluation of potential concurrency in computations. It can be considered as a canonical example of data streams. Stephens presented a detailed survey on the history and on the related work of data flow processing [147]. Although not always in the form that is immediately recognizable today, stream processing and analysis have been an active area of research in computer science, such as in neural networks, in cellular automata, and in safety critical systems.

Instigated by the trends and applications of the World Wide Web, wireless communications, sensor networks, and many others, data stream management and processing have become a hot topic in the past decade. Many DSMS systems have been developed. Academic systems include Aurora [110], Atlas [158], Borealis [103], CAPE [108], MAIDS [104], NiagaraCQ [26], Nile [114], STREAM [96], PIPES [80], PSoup [19], TelegraphCQ [121], and Tribeca [149]. Commercial DBMSs have also started to incorporate new features to support streaming data. Many new techniques for tackling issues in all aspects of data stream processing have been proposed. Good overviews on these topics can be found in [100, 56].

1.1.4 Distribution changes in data streams

In traditional DBMSs, it is reasonable to assume that the data set is static, i.e., the data elements are samples from a static distribution. However, this does not hold for many real-world data stream applications. Typically, fast data streams are created by continuous activity over long periods of time. It is natural that the underlying phenomena can change over time and, thus, the distribution of the values of the data in the stream may show significant changes over time. This is referred to as *data evolution*, *dynamic stream*, *time-changing data*, or *concept-drifting data* [3, 69, 78, 157].

The distribution change in the stream can be either a slow and gradual long term process (we refer to this as *distribution drift* in the rest of the thesis), or a significant sudden change (we refer to this as *distribution shift*). Both types of changes can be commonly observed in many stream-based applications.

Example 1 (distribution drift). Scientists have been using temperature and precipitation detecting sensors to monitor annual hydrologic processes [33]. A study of the climate trends in California shows that, due to the increasing concentrations of atmospheric carbon dioxide, the mean value of the annual runoff shows a 37% decrease over a 100 year period [10].

Example 2 (distribution shift). To gain a greater share of consumer expenditures, a special "loyalty program" is proposed by a retail brand. By monitoring the daily transactions of multiple outlets, a 14% increase in gross sales is observed immediately after the introduction of the loyalty program, indicating a positive impact of this proposed program [128].

Distribution changes over data streams have significant impact on most of the DSMSs. A stream processing model built previously may not be efficient or accurate after the data evolve, since some characteristics observed earlier in the data will no longer hold. Hence, if a distribution change occurs in the stream, it is important for the user to be notified of this change. The DSMS needs to be adjusted to reflect this change and new results should be generated for the data under the new distribution.

There is a considerable amount of work that focus on distribution change detection and incremental maintenance of stream processing models [4, 22, 52, 62, 123]. Change detection techniques should fulfill two goals: 1) To find whether a particular stream processing model has become stale because of the distribution change; and 2) to provide rich information for users to understand the nature of the data changes. An ideal solution should not make any assumptions regarding stream characteristics (past or present), and should be able to provide descriptive results that can be used to interpret the trend of the changes. Such a solution can be incorporated into any existing stream processing engine and, thus, could make many stream processing techniques previously designed only for static streams suitable for processing time-changing data streams. Furthermore, for streams whose distribution changes follow certain patterns, by analyzing the distribution during different time periods, it may be possible to predict their upcoming distribution changes.

1.2 Data stream mining

In his 1982 book *Megatrends*, John Naisbitt wrote: "We are drowning in information, but we are starved for knowledge." In past decades, available data volume has doubled almost every year, however, the knowledge we have learned from these has not increased at the same speed. The area of data mining arose partially to address this problem, and data mining techniques have proven to be extremely useful in almost all real-world application domains.

With the emergence of data streams, the amount of data generated and accumulated is rapidly increasing. Traditional data mining that are designed for static and well structured data with comparatively low efficiency are not suitable for mining such streaming data. Therefore, a new type of mining application that accepts data streams as input has emerged. Stream mining technology can dramatically change the way corporations, governments, and even individuals process data.

An example of an email stream mining application process is illustrated in Figure 1.2. This application classifies emails according to certain criteria, such as types of email (e.g., business or personal), email topics, or the email sender/receiver locations. The contents of emails in each class are then extracted, aggregated, and reconstructed. The mining results are then continuously generated after analyzing the contents.

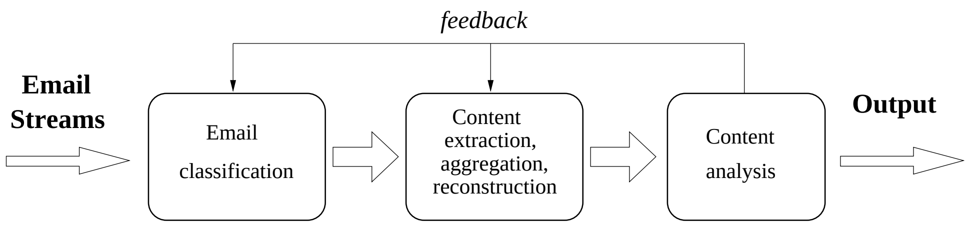


Figure 1.2: Example of mining an email stream

Since there are many similarities and common processes between traditional data mining and data stream mining, in Section 1.2.1, an overview of the models and issues in traditional data mining is provided. The new challenges and requirements for mining applications with streaming input is then discussed in Section 1.2.2.

1.2.1 Models and issues in data mining

This thesis focuses on the techniques for detecting distribution changes in time-changing streams and performing popular data mining tasks over these dynamic streams. Although data stream mining has gained attention only in the past few years, data mining over relational databases has been one of the key features in many real-world applications, such as fraud detection, risk assessing, retail marketing, and scientific discovery.

Data mining is a process of data collection, analysis, and prediction. Data mining tools use sophisticated analysis methods to discover previously unknown relationships and patterns in large (multi-dimensional) data sets, and predict their future trends and behaviors. These mining results enable the users to make proactive and knowledge-driven decisions. The typical data mining process consists of the following three stages:

Stage 1: Exploration. This stage usually starts with data preparation that involves data cleaning, data transformation, and performing some preliminary analysis over the data to identify the most relevant variables and to determine the complexity and the general nature of the mining techniques that can be taken into account in the next stage.

Stage 2: Model building. This stage involves applying various techniques for the same task on a test data set, and choosing the one with the best performance.

Stage 3: Deployment. In this final stage, the selected technique is applied to new data to generate expected output.

Depending on the application domain and user requirements, data mining applications can fulfill different functions/tasks. Data mining tasks are quite diverse and distinct because different types of data sets consist of many patterns. Among all the data mining tasks, the following are widely used in various real-world applications:

- *Clustering.* This task seeks to identify a finite set of categories (clusters) to describe the data. The clusters can be mutually exclusive or consist of a richer representation, such as a hierarchy.
- *Classification.* This task aims to find a function that can map (classify) each data item in the data set into one of the several predefined classes.
- *Frequency counting and association rule learning.* This task searches for relationships between variables, or identifies the most significant values in the data set.
- *Summarization and regression.* This task involves methods for finding a compact description for the entire data set or a subset of it. A simple example is tabulating the mean and standard deviations for all the fields. Another example is to find a function that models the data with the least error.

1.2.2 Mining data streams: New challenges

Data stream mining applications address the same tasks as traditional data mining but over unbounded, continuous, fast-arriving, and time-changing data streams. These characteristics impose many new challenges for even the simplest task in traditional data mining. Most of the existing techniques cannot be adopted for the data stream environment. The following is a summarization of major issues and challenges that differentiate data stream mining from conventional data mining.

- *Unbounded data sets.* Conventional data are completely stored on disk. When the data mining application initiates, it can retrieve the entire data set, whereas data streams are unbounded. Once the storage space is full, old data must be evicted to make room for the newly arrived data. Hence, at each time, only a portion (usually the latest part) of the data is available, and old data that have been discarded cannot be retrieved. This issue renders data mining techniques that require multiple scans over the entire data set to be useless.

- *"Messy" data.* Traditional data mining techniques are designed for data sets that typically come in relational form, such as tuples in relational tables. The tuples in a relational table may be organized on disk in specific layout patterns, e.g., tuples with identical values of an attribute can be clustered for efficient access. Furthermore, auxiliary structures such as indexes can be built on the relations to enable efficient retrieval of individual tuples. However, data in a stream arrive continuously, usually at a high rate. The DSMSs typically have no control over the arrival order, rate, or data distribution of the input streams. Hence, an efficient mining technique for relational data may no longer have similar performance when handling streams.

- *Efficiency.* Data mining over relational data sets can be considered as one-time tasks. The process starts when the mining process is triggered and ends once the results are generated. The results are then used for some time until the re-execution of the data mining task. Therefore, efficiency is usually not the primary concern in traditional data mining. Some sacrifices of efficiency is reasonable to gain a higher accuracy or generate exact results. Mining data streams is a continuous process and will not end unless manually terminated by user. Hence, the results are approximations in many stream mining applications. Since the data arrival rate can be extremely high, and real-time response is required, efficiency of data mining algorithms is critical for stream mining techniques. Techniques such as load-shedding and sampling are not commonly used in traditional data mining, but are widely adopted for mining high-speed streams.

- *Data evolution.* Data streams are generated by real-world applications continuously, and the underlying distribution of a data stream may change over time. This problem does not exist in traditional data mining, since relational data are usually considered static. The distribution changes in a data stream can greatly affect the performance of the stream mining technique, because mining results generated for the previous distribution may not be accurate distribution changes, and, hence, new results need to be produced on the fly whenever a distribution change is detected.

  For example, frequent itemset mining is a common data mining task for transactional data streams. Mining frequent itemsets can be quite time-consuming, since the total number of itemsets is exponential. Hence, for traditional data mining, usually only the itemsets that are known to be frequent are monitored and infrequent itemsets are discarded. However, for data streams that change over time, an itemset that was once infrequent can become frequent if the distribution of the stream changes, and vice versa. Therefore, it is critical for a frequent itemset mining application to have the ability of detecting changes, of eliminating itemsets that are no longer frequent, and also of generating new frequent itemsets.

To address these challenges, any data stream mining application must fulfill the following requirements:

- *Time efficiency.* As discussed previously, stream mining applications must have real-time efficiency to mine fast-arriving data streams.

Unlike data mining algorithms that aim to gain high or even perfect accuracy, stream mining techniques should have the ability to be adjusted to achieve balance between the accuracy of the results and the response time.

- *Resource efficiency.* With the unbounded and pass-through features of data streams, two types of resources, memory space and computation power, are particularly valuable in streaming environments. Any stream mining application must have the capability of resource-awareness; resources should be adaptively allocated. Advanced scheduling and memory management techniques are important for mining data streams.

- *Ability to handle delayed, missing, and out-of order data.* A stream mining application runs in real-time and, hence, should not wait for certain data indefinitely. Any algorithm that can cause blocking must also have the ability to time-out, so that the mining application can continue with partial data.

- *Ability to detect changes.* Data streams can change over time. When the distribution changes, previously generated mining results may no longer be valid. Therefore, a stream mining technique must have the ability to detect changes in the stream and should automatically modify its mining strategy for different distributions.

- *Determinism.* A stream mining application must ensure that its mining process is both deterministic and repeatable. Therefore, remining the same input stream should generate the same output regardless of the time of execution. This ability is important from the perspective of fault tolerance and recovery.

1.3 Motivation

Mining time-changing data streams is crucial but difficult. A large number of existing stream mining techniques make the false assumption that the stream of interest has stable distribution over its entire lifespan. Although these approaches may achieve high performance by by-passing the difficulty of change detection, their practical values are questionable. Therefore, the focus of this thesis is on developing new techniques for detecting distribution changes and stream mining techniques that are suitable for dynamic data streams.

1.3.1 Mining data streams with distribution change

As discussed previously, distribution changes over data streams have considerable impact on most of the stream mining algorithms and, thus, techniques for detecting changes in streaming data are required. However, the importance of this challenge has only been recognized in recent years. There are basically two different approaches to detect changes. One looks at the nature of the data set in the stream and determines if that set has evolved while the other detects if an existing data mining model is no longer suitable for recent data, which implies concept drifts. The former approach leads to general techniques that are suitable for any type of stream, whereas the second group of approaches are task-specific that are designed for one specific stream mining technique and usually only perform well on certain types of streams.

The general approaches have the advantage of flexibility. They can be incorporated into any stream mining technique. Moreover, since they detect changes by directly analyzing the data, aside from generating alarms at the time of the distribution changes, these approaches may be able to provide richer information, such as the characteristics of the new distribution, or the type of the distribution change (i.e., whether it is a sudden and severe shift or a slow and gradual drift). This information is crucial for analyzing the streaming data and can help users choose the best mining technique for a given input stream. However, few distribution change detection approaches proposed in literature are independent of stream mining applications.

Although the task-specific approaches do not have as much flexibility as general approaches, the previous mining results may provide useful information that can help the change detection and mining techniques to be fine-tuned to improve performance efficiency and accuracy. For example, in an association rule mining application, a significant frequency change in some of the itemsets may indicate a distribution change. In a stream classification application, a dramatic change of the size of one class may be a signal of the impending arrival of a new distribution. Since general approaches are not directly connected to the mining task, they cannot take advantage of such feedback information. Hence, for many streaming applications that only perform one specific mining task, task-specific techniques with outstanding performance may be more useful than general approaches.

There is a large number of task-specific techniques developed for each of the stream mining tasks. However, many of these techniques have problems in meeting all of the common requirements for mining streaming data: ability to process large volumes of data in real time, low memory usage, and ability to cope with time-changing data. This thesis looks into one of the most important mining tasks, which is frequency counting. An algorithm is proposed that meet all three common requirements and can out perform existing techniques for mining frequent itemsets in time-changing data streams.

1.3.2 Multi-dimensional streams

Most of the stream mining techniques for dynamic data streams only work over single-dimensional data, i.e., they assume there is only one attribute of interest in the stream. However, the data collected in a data stream from real-world applications usually contain several attributes. In practice, many stream processing applications need to take more than one attribute into consideration. For example, in modern quality control, several quality characteristics are usually monitored simultaneously. In e-commerce, where each data element in the stream is an order placed by customers, a positive linear correlation between items in the order may indicate similar purchase patterns. There has been little attention paid to the problem of extending change detection and mining to multi-dimensional streams.

Under the assumption that attributes of interest are not correlated with each other, the issue due to multi-dimensionality can be easily addressed by running a set of single-dimensional dynamic stream mining process simultaneously on each attribute. However, if the correlations among several attributes are taken into account, such a solution is no longer satisfactory. One of the challenges for mining multi-dimensional data set is known as “the curse of dimensionality” (or Hughes effect [15]): in high-dimensional space, data may be sparse, making it difficult to find any structure in the data. Under the streaming environment that requires efficient on-line techniques, it is more difficult to mine such multi-dimensional dynamic streams.

A practical stream mining technique should have the ability to handle streams with more than one dimension. Based on this insight, in this thesis, one of the proposed techniques that are primarily designed for mining single-dimensional streams is further extended to higher dimension.

1.4 Scope and Contributions

In the next chapter, we present the formal definition of the data stream model, and discuss the window models that are commonly used in stream processing and mining. A survey of issues and related works in data stream mining is introduced afterwards. This background is important for understanding the remainder of the thesis.

Since the focus of this thesis is on mining data streams with distribution changes, we begin with exploring the fundamental problem of this topic in Chapter 3: detection of distribution changes in data streams. Discussions in this chapter are not restricted by any specific stream mining application and the proposed approaches can be applied on any type of uni-dimensional streams.

Many existing generic change detection techniques adopt statistical tools to estimate the distributions of the data sets in the stream and to calculate the discrepancy between the current distribution and distribution of the newly arrived data. Since many existing statistical tools require a large amount of sample data to accurately estimate the distribution, under the streaming environment where the size of the data set is usually small due to memory constraint, the accuracy is impacted. Hence, we propose an approach to represent the distribution that generates the data in the stream using a small data set. Two windows, a reference window and a observation window, are used to maintain data sets that represent the current and new distributions of the stream. An intelligent merge-and-select sampling approach is proposed that can dynamically update the reference window. The small data set in the reference window can represent the current distribution of the stream with high accuracy.

Although it may not be able to estimate the distribution function of a small data set with high accuracy, some key features of this distribution, such as mean, range, and variance, can be easily obtained. For some types of data stream applications, these key features are the sole interest and are sufficient for generating mining results. Based on this, another general approach is proposed in Chapter 3 that detects mean and standard deviation changes in any dynamic data stream. This technique uses control charts [129] to monitor the input stream and generates alarms when there is a significant change in the mean and standard deviation of the data. Unlike most of the change detection techniques that require a huge number of data to get promising results, this approach can achieve high accuracy with only a small sample set. As the experiments demonstrate, the proposed approach has high efficiency, so that fast distribution changes in the stream can be captured.

Most of the existing stream mining approaches, including the ones presented in Chapter 3, are mainly designed for streams with single dimension. There is very little research on change detection and mining multi-dimensional streams. As noted earlier, for many practical applications, data collected in streams contain multiple attributes, and there can be more than one attribute of interest. In Chapter 4, a method is provided to extend one of the proposed techniques to higher dimensions. It is shown that, by modifying the proposed change detection and stream mining approach, this approach is suitable for multi-dimensional time-changing streams.

Chapter 5 examines one of the most important mining tasks: frequent itemsets mining. A task-specific stream mining technique is proposed.

Mining frequent itemsets in dynamic streams are particularly difficult because infrequent itemsets are usually not maintained due to limited memory. However, when a distribution change occurs, a once infrequent itemset may become frequent under the new distribution. Such new frequent itemsets are difficult to detect. Furthermore, even if these itemsets can be detected, their statistics can not be obtained, since mining a data stream is a one-pass procedure and historical information is irretrievable. In Chapter 5, a false-negative oriented algorithm is proposed that can find most of the frequent itemsets, detect distribution changes, and update the mining results accordingly. This technique uses two windows (window models will be discussed in Section 2.1.2), one for maintaining existing frequent itemsets and one for predicting new frequent itemsets. A candidate list is maintained that contains a list of itemsets that have potential to become frequent. Every time the two windows move, we check if new frequent itemsets and candidates should be added and if some existing ones need to be removed from the list.

The contribution of this thesis can be briefly summarized as follows:

- The issue of mining time-changing data streams is systematically studied. Data stream mining has only gained popularity recently, and many proposed techniques are based on the false-assumption that distribution in the data stream will always be static. In this thesis, we study this important but challenging issue both in depth and in width.

- Two generalized change detection algorithms are proposed. As has been previously discussed, most of the change detection techniques proposed in literature are task-specific. Generalized approaches are rare, but have greater potential since they can be applied to any type of stream and plugged in any stream mining applications.

- These proposed techniques are extended to data streams with higher dimensions. Multi-dimensional data mining is a difficult task even for traditional data mining. Few studies have been conducted for detecting changes and mining multi-dimensional data streams.

- A novel technique for mining frequent itemsets in data streams is proposed. The proposed technique has the ability of detecting distribution changes in real-time, and can out-perform others according to the experiments.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the background of data stream mining. In Chapter 3, the problem of distribution change detection is discussed and two change detection techniques are proposed. In Chapter 4, we discuss the potential of extending our proposed techniques to higher dimensions. A novel approach for mining frequent itemsets in transactional streams is presented in Chapter 5. Finally, Chapter 6 concludes this thesis with summarizations and suggestions for future work.

Chapter 2

Background

This chapter surveys the background and related work in data stream models and processing techniques. In keeping with the emphasis on data stream mining and distribution change detection, the following topics in stream processing are omitted.

- Query languages and query processing over streams. An overview of these topics can be found in [98, 11, 121, 57, 76, 81, 87, 107].
- Distributed stream processing. See [13, 35, 131, 116, 151, 160] for examples of recently proposed techniques on this issue.
- High availability and fault tolerance. See the following representative papers [12, 47, 59, 60, 111, 83] as examples.
- Application-specific DSMS issues, such as sensor streams [65, 122, 92], merging streams [105, 55, 115], grid computing [1, 72, 138], and geospatial streams [113, 64, 141].

Section 2.1 surveys data stream models. Section 2.2 gives a formal definition of the stream distributions. Reviews of recent work on data stream mining is given in Section 2.3.

2.1 The data stream model

2.1.1 Data models

A data stream S is a sequence of continuous, append-only, and ordered (usually by timestamps) data elements that arrives in real-time. Each element in S can be denoted as $\langle s, t \rangle$, where s is the actual data and t is the monotonically increasing timestamp attached to s[1], indicating the arrival time of the element. Depending on the underlying application that generates S, data s may have different forms. For example, s may take the form of relational tuples in relational-based stream models; in object-based models, s can be instantiations of data types with associated methods. Also, s can be modeled as a set of items (or tuples) in transactional-based DSMSs. Each stream processing application may only consider a few attributes or items of the stream. The unconsidered attributes or items can be filtered before data are fed to the process.

Timestamp t can be either explicit (as a traditional timestamp) or implicit (as a sequence number). Explicit timestamps are often used when the arrival time of the elements is significant, whereas implicit timestamps are mostly adopted when the general considerations of “recent” or “old” as sufficient for the DSMS. Note that in either case, timestamps may or may not be visible to users and the arrival order of the elements may not be identical to the order in which the elements were submitted by the application, due to unstable network speeds, especially for distributed systems.

Although the individual element s may take the form of relational tuples, the entire data stream differs from the relational models in several ways:

- Data elements in the stream arrive online. Although the system may attach implicit timestamps on the stream, it has no control over the order of data arrival.

[1]In the rest of the thesis, the data element may be referred as only s when there is no ambiguity.

- Due to performance and storage constraints, in general, once an element from a stream is processed, it may not be retrieved (backtracked). In particualr, online stream algorithms are restricted to only one look at the data [53].
- Data streams are potentially unbounded (often assumed to be infinite) in size and, hence, cannot be stored completely in bounded memory. Therefore, most stream processing applications can only produce approximate results, since the whole data set is not available at any time.
- Online applications that process streams in real-time must have high efficiency, sometimes at the cost of sacrificing some accuracy in overall performance.

Since a stream cannot be stored entirely in a DSMS, only a subset of the stream is available at any time. This subset can be a continuous "chunk" of the stream in terms of arrival time, or the subset can be made up of discontinuous elements selected by the application. We call such subsets *substreams*, defined as follows:

Definition 2.1 A substream $S_i = \{\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, ..., \langle s_k, t_k\rangle\}$ of stream S is a finite set of elements that occurs in S, i.e., $\langle s_j, t_j\rangle \in S$, $j = 1, ..., k$.

2.1.2 Window models

In many cases, new elements in a data stream are more relevant and usually more accurate than are the older ones. Therefore, for most stream applications, only a recent excerpt of a stream is of interest. This insight gives rise to window models. A window W over a stream S contains a continuous substream S_i of S. Substream S_i can be represented by using the timestamps of the oldest element $\langle s_1, t_1\rangle$ and the newest element $\langle s_k, t_k\rangle$ in S_i. That is, window W can be referred to as "window on S from time t_1 to t_k". Elements that are inside W are (temporarily) stored locally and, hence, can be scanned multiple times. Once an element is evicted from the window, it cannot be retraced.

Many window models have been proposed in literature. These models can be classified according to the following criteria:

- Window size.

  Time-based windows contain elements that arrive within a certain time period. For example, a time-based window with size Δ may contain elements with timestamps within range $[t, t + \Delta]$[2]. In contrast, *count-based windows* contain fixed number of elements. Therefore, the length in time for a count-based window is not stable, since data streams usually do not have even arrival speeds.

- Update intervals.

  Windows using eager re-evaluation strategies expire old elements upon the arrival of each new element. Such windows are referred to as *sliding windows*. In contrary, lazy re-evaluation strategies (batch processing) lead to a "jumping" of the windows when they update. These windows are referred to as *jumping windows*. If the update interval is larger than the window size, then this window is called a *tumbling window*. Figure 2.1 illustrates different types of windows in terms of update intervals. Note that in the rest of the thesis, we use W and W' to denote the same window before and after moving, separately, when there is no ambiguity.

- Movement of the windows endpoints.

  For a window with both endpoints (the timestamps of its oldest and newest elements) fixed, the window does not move over time. Such windows are defined as *fixed windows*. Fixed windows are often used as a reference for comparison. If one endpoint is fixed and the other is moving, then such windows are referred to as *landmark windows*. Landmark windows need to be monitored carefully, since they can easily drain the system memory. If both endpoints of a window are

[2]It can also be $(t, t+\Delta)$, $(t, t+\Delta]$, or $[t, t+\Delta)$, depending on the specific definition of the window.

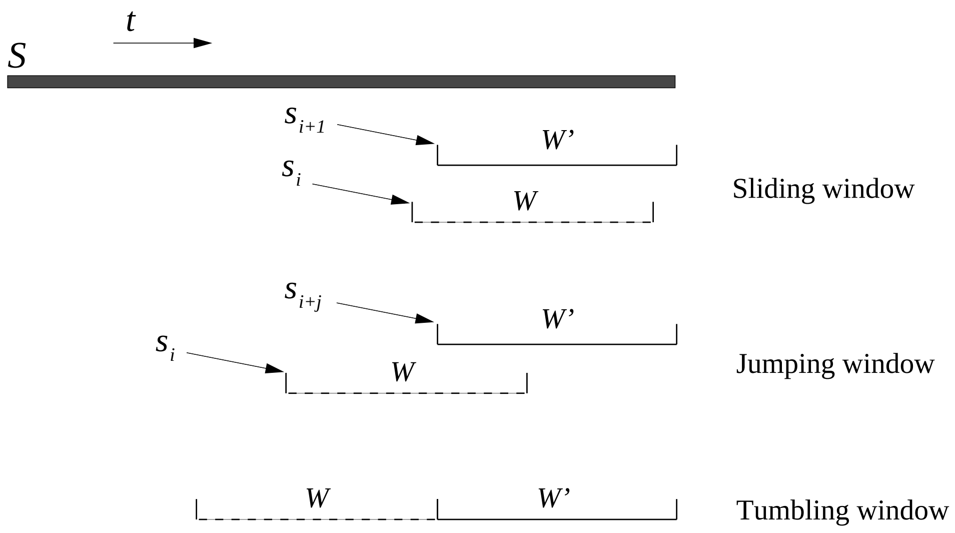


Figure 2.1: Windows update intervals

moving[3], then the window is a moving window. Figure 2.2 illustrates these three types of windows.

2.2 Stream distribution and change detection

If the timestamps of all data elements in a stream are discarded, then these elements form a data set. The probability distribution of such a data set can be defined conventionally: all the possible values (for discrete distribution) or the intervals (for continuous distribution) are described, along with the probabilities that a random variable can take within a given range [48].

However, as discussed in Section 1.1.4, data streams can change significantly over time and, thus, distributions in dynamic streams are temporal.

[3]Theoretically, the direction of the movement can be either forward or backward; however, only forward movement is used in most cases.

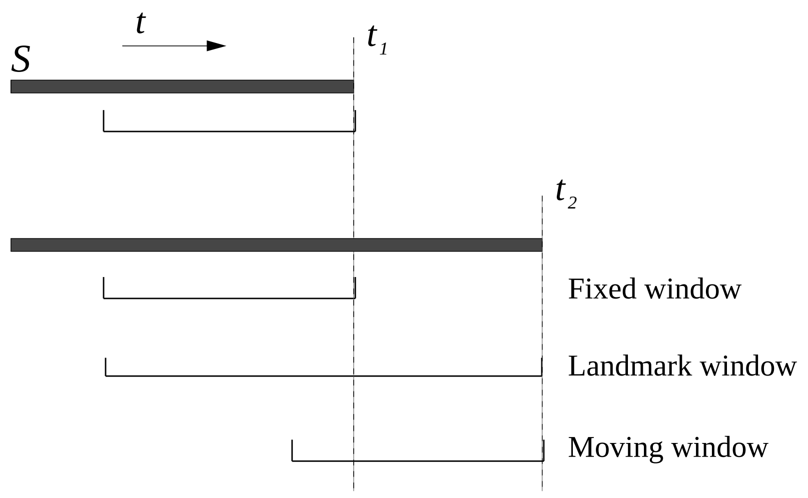


Figure 2.2: Movement of windows endpoints

Estimating the distribution over the entire data set of a dynamic stream has no practical meaning and cannot provide useful information for understanding and analyzing the stream. Therefore, in this thesis, the probability distributions of a stream are defined in the scope of substreams:

Definition 2.2 The probability distribution P_i of a substream S_i, under the assumption that all elements in S_i are generated by the same distribution P_i, is the frequency distribution of all values within this substream, without considering their arrival time. The value domain of elements in S_i is denoted as V.

Figure 2.3 gives an example of the density curve of a continuous distribution that generates current data in stream S. The x-axis is values domain, and y-axis is density. The density curve has total area 1 underneath it. The area under the curve within certain range of values $[v_i, v_j]$ indicates the probability that a data element s in stream S has its value $v(s)$ falls in this range. For example, if the area under the curve within range $[v_i, v_j]$ is 30%, then 30% of the data s in stream S generated by the current distribution will have values $v(s) \in [v_i, v_j]$.

Given two substreams $S_A = S(t_1, t_2]$ and $S_B = S(t_2, t_3]$ from data

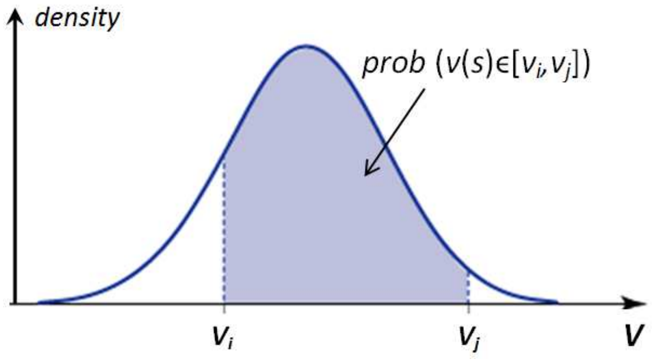


Figure 2.3: Example of continuous distribution of a stream

stream S, with data generated by probability distributions P_A and P_B, respectively, the similarity of P_A and P_B can be measured by conducting statistical tests on P_A and P_B. The choice of the statistical tests is application-based. For example, discrepancy in distributions can be calculated using distance functions. Also, similarities of two distributions can be estimated using key features of the data sets, such as the value range, the standard deviation, and the number of distinct values.

If P_A and P_B show high similarity according to the tests, then it is safe to assume that the distribution of S does not change during the time span of S_A and S_B. However, if the probability of $P_A \neq P_B$ is high (greater than a user defined threshold), then it is reasonable to believe that the distribution of S has changed. There are mainly two types of distribution changes: distribution *drifts* and distribution *shifts*. Distribution drifts refer to the slow and gradual changes in a stream, such as temperature and gas price changes; whereas distribution shifts represent abrupt and severe changes, such as machine malfunction and fraud.

The main tasks for designing a change detector are to design a statistical test with high accuracy and to develop a technique that can report distribution changes quickly. Change detection is not an easy task. The design of a change detector is a compromise between detecting true changes and avoiding false alarms. For data streams with high arrival rates, efficiency is highly important and, hence, the change detection technique over dynamic streams must find a "balance" between efficiency and accuracy. In some cases, some accuracy may be sacrificed to achieve higher efficiency, and vise versa. A more detailed survey of the related work on change detection will be given in Section 3.2.

2.3 Survey on data stream mining

Data stream mining is the process of extracting information and patterns from streaming data. It can be considered as an extension of traditional data mining and knowledge discovery from relational tables to the new type of continuous, unbounded, rapid, and time-changing data. Therefore, most of the issues identified in relational data mining must also be addressed for stream mining, with the additional difficulties introduced by new stream data as discussed in Chapter 1.

2.3.1 Data refining

Data refining approaches refine the data elements in the stream for the purpose of mining. These approaches do not extract complex information from streams, but a data stream after these processes will be cleaner, more compact, and better structured. Mining the stream after these processes may greatly improve the performance of the mining applications. Some information generated from the processes, such as data synopsis and distribution change alarms, can help the mining applications to adjust their parameters and strategies over time. Unlike relational data mining that process the entire data once, before the mining procedure starts, streaming data processing procedures continue during the entire life span of streams.

Sampling

Data stream sampling is the process of choosing a suitable representative subset from the stream of interest. The major purpose of stream sampling is to reduce the potentially infinite size of the stream to a bounded set of samples so that it can be stored in memory. There are other uses of stream sampling, such as cleaning "messy" data and preserving representative sets for historical distributions (see Section 3.3 as an example). However, since some data elements of the stream are not looked at, in general, it is impossible to guarantee that the results produced by the mining application using the samples will be identical to the most recent results returned on the complete stream. Therefore, one of the most critical tasks for stream sampling techniques is to provide guarantees about how much the results obtained using the samples differ from the non-sampling based results.

Sampling is a technique that has long been used in various domains. The size of the data set is always unknown when the data come in the form of a stream. Moreover, stream processing restricts only one pass through the data. These two conditions render many existing sampling approaches useless for streaming data. A number of stream sampling techniques have been proposed in recent years. Manku et al. developed a framework that is based on random sampling and includes several known algorithms as special cases [94]. Park et al. proposed a reservoir-based sampling algorithm using replacement technique to maintain the sample set [102]. Cormode and others proposed a method for maintaining dynamic samples that can be used in a variety of summarization tasks [38]. Chuang et al. explored a new sampling model, called feature preserved sampling, that sequentially generates samples over sliding windows [34].

Load shedding

The arrival speed of elements in data streams are usually unstable and many data stream sources are prone to dramatic spikes in load. Therefore, stream mining applications must cope with the effects of system overload. Maximizing the mining benefits under resource constraints is challenging. Load shedding techniques, which are techniques that discard some of the unprocessed data during peak time, are widely adopted for handling system overload. There are three major decisions in load shedding: 1) Determining when to shed load; 2) Determining how much load to shed; and 3) Determining which elements drop. Dropping elements earlier and dropping more elements can speed-up the system and avoid wasting work; however, the accuracy of the mining results may be affected. The quality of the decisions can be estimated by loss or gain ratios.

Some heuristic approaches have been proposed and implemented in DSMS prototypes [105, 120, 117]. Chi et al. developed a load shedding technique using the Markov model[4] and the quality of decision metric for classifying data streams [32]. A frequency-based load shedding approach is introduced by Chang and Kum [20]. A control-based load shedding approach adopted in the Borealis system is presented in [153].

Synopses maintenance

Synopsis maintenance processes create synopses or "sketches" for summarizing the streams. Synopses do not represent all characteristics of streams, but rather some "key features" that might be useful for tuning stream mining processes and further analyzing streams. It is especially useful for stream mining applications that accept various streams as input, or for input streams with frequent distribution changes. When streams change, some kind of re-computation, either from scratch or according to differences only, has to be done. An efficient synopsis maintenance process can generate summaries of streams shortly after changes, and stream mining applications can re-adjust their settings or switch to other mining techniques based on this valuable information. Which synopsis should be maintained is an application-based issue. Examples of synopsis include histograms, range estimation, quantiles, and frequency moments.

Cormode and Muthukrishnan introduced a sublinear space structure called the Count-Min sketch for summarizing data streams [37]. Lin and Xu developed an algorithm for continuously maintaining quantile summary of the most recent elements in a stream [90]. Pham et al. proposed a synopsis building approach by using a service and message-oriented architecture for data streams [137].

[4]A Markov model [95] is a state transition model that represent a changing set of states over time, where there is a known probability or rate of transition from one state to another.

Change detection

As discussed in Section 1.3.1, distribution changes over data streams have great impact on stream mining applications. When the distribution of the stream changes, previous mining results may no longer be valid under the new distribution and the mining technique must be adjusted to maintain good performance for the new distribution. Hence, it is critical that the distribution changes in a stream be detected in real-time, so that the stream mining application can react promptly.

Change detection on data streams has been recognized as an important problem in recent years. In general, there are two different tracks of techniques for detecting changes. One track looks at the nature of the dataset and determines if that set has evolved and the other track detects if an existing data model is no longer suitable for recent data, which implies concept drifting. The work proposed by Kifer et al. [78] and Aggarwal [3, 4] are representative of the first track and much work [69, 157, 49, 51] belongs to the second track. See Section 3.2 for a more detailed survey on change detection techniques.

2.3.2 Stream mining tasks

This section reviews some of the most popular stream mining tasks: clustering, classification, frequency counting and association rule mining, and time series analysis.

Clustering

Clustering is an important technique for both conventional data mining and data stream mining. Clustering groups together data with similar behavior. Clustering can be thought of as partitioning or segmenting elements into groups (clusters) that might or might not be disjointed. Note that in many cases, the answer to a clustering problem is not unique, that is, many answers can be found and interpreting the practical meaning of each cluster may be difficult.

Aggarwal et al. proposed a framework for clustering data streams [5]. This framework uses an online component to store summarized information about the streams and an offline component performs clustering on the summarized data. An extension of this framework, called HP Stream, was proposed the following year [6]. HP Stream finds projected clusters for high dimensional data streams.

Clustering algorithms proposed in literature can be briefly categorized into decision tree based and K-Median based approaches. Some examples of decision tree based techniques are [45, 51, 69, 150]. Representative papers that are based on the improvement of K-Median or K-Mean algorithms include [101, 24, 25, 61, 132].

Classification

Classification maps data into predefined groups (classes). Classification is similar to clustering but with the difference that the number of groups is fixed and classification results are not dynamic. Classification algorithms require that the classes based on data attribute values to be defined. Pattern recognition is a popular classification task. In pattern recognition, an input pattern is classified into one of the several predefined classes based on its similarity to these classes.

Similar to clustering approaches, the classification technique can also adopt the decision-tree model. Decision-tree classifiers, Interval Classifier [119], and SPRINT [143], have been developed for mining databases that do not fit in main memory using sequential scans and, thus, are suitable for data stream environments. The VFDT [45] and CVFDT [69] systems originally designed for stream clustering can be adopted for classification tasks. Techniques proposed by Ding et al. [44] and Ganti et al. [52] are other examples of decision-tree based classification approaches.

The classification system developed by Last [86] uses an info-fuzzy network as a base classifier. This system can automatically adjust itself for different distributions in dynamic streams. Wang et al. introduced a framework [157] that can also deal with time-changing streams by using weighted classifier ensembles. Aggarwal et al. built a distribution change sensitive system using the idea of microclusters [7].

Frequency counting and association rule mining

The problem of frequency counting and mining association rules (frequent itemsets) has long been recognized as important task. However, although mining frequent itemsets has been widely studied in data mining, it is challenging to extend it to data stream environment, especially for streams with non-static distributions. An overview of this issue is presented by Jiang and Gruenwald [74].

Mining frequent itemsets is a continuous process that runs throughout the life span of a stream. Since the total number of itemsets is exponential, it is impractical to keep a counter for each itemset. Usually, only the itemsets that are already known to be frequent are recorded and monitored and counters of infrequent itemsets are discarded. However, data streams can change over time and, hence, a once infrequent itemset may become frequent if distribution changes. Such (new) frequent itemsets are difficult to detect, since mining data streams is a one-pass procedure and history is not retrievable.

Despite these difficulties, a number of frequency counting and association rule mining techniques for data streams have been proposed. Earlier work simplifies the problem by counting only the frequent items. Techniques for mining frequent items [18, 36, 41, 63] commonly assume that the total number of items is too large for memory-intensive solutions to be feasible. Approaches developed for mining frequent itemsets in literature [22, 29, 31, 166] may or may not be applied on dynamic streams. See Section 5.3 for detailed discussions on related work about this topic.

Time series analysis

In general, a set of attribute values over a period of time is described as time series. Usually, a time series consists of only numeric values, either continuous or discrete. From this informal definition, it is natural to picture a data stream that contains only numeric attributes as a time series. Mining tasks over time series can be briefly classified into two types based on this criteria: pattern detection and trend analysis. A typical mining task for pattern detection would involve being given a sample pattern or a base time series with a certain pattern and to find all the time series that contain this pattern. Detecting trends in time series and predicting the upcoming trends are the tasks for trend prediction.

Zhu and Shasha proposed a system for computing measures over time series using an arbitrarily chosen sliding window [170]. Lin et al. proposed the use of symbolic representation of time series [89]. This representation allows both dimensionality reduction and distance measurement. Chen proposed two distance functions for time series pattern matching [27, 28].

Perlman and Java developed a two-phase approach for predicting trends in astronomical time series [136]. First, they attempt to represent the data set as a collection of patterns. In the next step, probabilistic rules of form "If pattern A occurs in time series 1, then pattern B may occur in time series 2 within time T". Indyk et al. proposed a technique for estimating the average trends and the relaxed periods of time series by using a so-called sketch pool [71].

2.3.3 Other related research issues

There are many open issues in stream mining area that have grand importance but do not receive sufficient attentions. Some of these issues are addressed in the following discussions.

Evaluation and benchmark

Although a large number of data stream mining techniques have been developed, there is not yet an open source benchmark specifically designed for data streams that can be used to evaluate and compare the performance of different stream mining systems. Data stream researchers are left with the options of either using the benchmarks designed for DBMSs or re-implementing others technique and conducting experiments on some sample data. Both solutions consume significant amount of time and resources, and the quality of the comparison results are usually not satisfying. Developing benchmarks for stream mining systems is a both challenging and important research issue.

Integration of stream mining systems with relational DBMSs

Developing and commercializing an independent data stream mining systems require large amount of time and resources. One of the alternative solutions is to integrate prototype of a stream mining system with an existing relational DBMS or a relational data mining system. Nowadays more and more commercial DBMSs start to incorporate new components to support streaming data. Research works need to be done on issues such as increasing the compatibility between both systems and sharing resources effectively.

Stream visualization

Data stream mining often involves many transformations. Some of the features of the data may be hidden after transformations. Visualization could help user asses whether important features of stream are captured by the mining application. Moreover, stream mining results, such as clustering structure and distribution differences, can be better presented and interpreted after visualization.

Chapter 3

Distribution Change Detection

3.1 Introduction

As discussed in previous chapters, the underlying distributions in some data streams can change over time. This is due to the evolving nature of many real-world applications that run for long periods. A change in the distribution that generates the data elements in the stream can cause the stream mining models to go stale and to degrade their accuracy. Hence, any stream mining technique that ignores distribution changes is not applicable to dynamic data streams. However, the problem of detecting distribution changes in dynamic data streams remains difficult and open due to the unbounded and fast-arriving characteristics of data streams.

Various change-detection techniques have been proposed in literature [4, 99, 68, 69, 75, 130, 168]. However, most of these techniques are ad-hoc. They are designed for one specific stream processing technique, such as stream query processing or association rule mining, and may perform well only on streams generated by particular streaming applications. These techniques cannot be directly applied to detect distribution changes in generic data streams.

Only a few distribution change-detection approaches proposed in literature are independent of a specific stream processing application. These techniques are discussed in detail in Section 3.2. Some of these solutions assume that the user already has knowledge of which type of distribution the sample set follows. Unfortunately, in real world applications this information is usually hard to obtain, especially for streams with changing distributions.

There exist a number of statistical tools that can estimate the distribution of given data set, such as kernel density estimation (KDE) [133] and empirical cumulative distribution function (ECDF) [145]. Using these statistical tools, distribution changes can be detected by calculating the discrepancy between the estimated distributions of previous data set and the newly arrived data set.

It is well-known that by using the ECDF, the estimated distribution function $F_n(x)$ (where n is the size of the given data set) converges to the underlying distribution $F(x)$. Indeed, the distribution of $F_n(x)$ is asymptotical $N(F(x), \frac{F(x)(1-F(x))}{n})$. As n increases, the asymptotic variance decreases and $F_n(x)$ improves as a more accurate estimation of $F(x)$. However, in the data stream environment, memory is usually limited and, hence, the number of data elements to determine $F_n(x)$ is also limited.

To address this problem, a new technique called merged-window method is proposed in Section 3.3. Suppose that there are m new data generated by the same distribution that arrive in the stream. The goal of the proposed technique is to select n data from the union of the n original data that are used to represent $F(x)$ and the m new data, so that the variance of ECDF can be reduced using the newly selected n data. This approach is investigated using several sets of experiments.

Although it is difficult to achieve high performance for detecting all kinds of distribution changes in streams with any type of distribution, for some stream mining applications, only certain key features of the data stream are of interest. These key features are sufficient for generating mining results for these applications. The performance of a change detection technique may be greatly improved if, instead of monitoring all types of distribution changes, it only focuses on changes in these key features. Based on this insight, a control chart based approach is proposed in Section 3.4 that detects mean and standard deviation changes in any dynamic data stream.

3.2 Related work

Distribution estimation and change detection are two problems that often arise in a number of research areas. The problem of estimating the distribution of a given data set has long been studied and several statistical tools have been proposed.

One of the most popular non-parametric distribution estimation techniques is kernel density estimation (KDE) [133]. The kernel density estimator has the form $\hat{f}_h(x) = \frac{1}{nh}\sum_{i=1}^{n} K(\frac{x-x_i}{h})$. KDE smooths out the contribution of each observed data point over a local neighborhood using a kernel function $K()$. The value of the density for each point of interest is estimated as the sum of the weighted values of all data in the sample. The kernel width h, usually referred as “bandwidth”, is a smoothing parameter. Silverman gives a good summary of many commonly used density estimation techniques [146].

KDE is a powerful tool that provides a descriptive output that assists in diagnosing the nature of the data set. In practice, however, the quality of a kernel estimate largely depends on the choice of the bandwidth h. A large bandwidth may over-simplify the distribution of a given data set, whereas a small bandwidth could make two data sets generated from the same distribution look very different. It is hard to determine which value of h provides the optimal degree of smoothness without some formal criterion. Furthermore, a given value of h does not guarantee the same degree of smoothness if used with different kernel functions $K()$.

Aggarwal address the data stream change detection problem by providing a framework that uses KDE [3]. The kernel function and bandwidth used in this framework are the Gaussian kernel and Silverman’s rule-of-thumb for choosing the bandwidth [146]. This classical choice has been proven to be promising. However, as will be shown in our experiments, this method is inclined to be conservative, i.e., it detects fewer true changes with fewer false alarms. Therefore, this approach may not be ideal for all real-world applications.

A set of algorithms known as *probabilistic model-building genetic algorithms* (PMBGA) [16, 85, 134] are also used for estimating distributions. In these algorithms, a set of candidate solutions are provided at the beginning. Initially, a random set of samples is selected and evaluated using an objective function. This objective function evaluates the accuracy of each candidate solution for that sample. The evaluation continues until the solution with optimal accuracy is found. These approaches suffer from one major problem: without any prior knowledge about the sample set, candidate solutions cannot be properly selected and, hence, the estimation result can be poor. Furthermore, the learning process is long and may not be suitable for streams with high arrival rates.

Kifer et al. propose an approach that detects changes by calculating the distances between the distributions of two data sets collected from a stream during different time periods (past and present). The distance is calculated using a set of novel distance functions [78]. A distribution change is reported if the distance calculated using these functions is beyond a pre-set threshold. As will be shown in our experiments, the proposed distance functions are "aggressive" in change detection: they can detect most true distribution changes, however, the false alarm rate is usually high.

Muthukrishnan and others [130] develop an algorithm that bases its detection decision on the classic *Sequential Probability Ratio Test* (SPRT) [156]. The probabilities of whether the new data set contain or do not contain a change point are estimated and the difference between the distributions is measured by the ratio of these two probabilities.

There are other statistical techniques that do not estimate the distribution function of the data set, but some of them are ad-hoc and cannot be applied to all types of data generated by different applications. Furthermore, most of the techniques cannot be directly adapted to the one-pass, continuous, and fast-arriving stream environment. They either require a large sample size, or suffer from a long delay from the time a distribution change occurs to the time it is detected.

3.3 Detecting Changes with Tumbling Windows

3.3.1 Motivation

In most change detection techniques, detecting distribution changes is a process of comparison: the distribution of newly arrived data is compared with the current distribution of the stream and distribution changes are detected by measuring the discrepancy between the two distributions. Since the data set continuously generated by a distribution can be unbounded, naturally, a subset of data for a distribution will be selected for comparison. This gives rise to window models. The data set in the window is a representative set representing the distribution that this set is generated from. This window is called the *reference window*.

Given a data set (substream) that arrives within a certain time period, there are many ways of selecting and updating (if required) its representative set. Hence, different reference window selection methods are proposed in literature. There are two popular ways of selecting the reference window: one is to select the first substream after a new distribution is detected; the other is to select the last substream of the current distribution. Because the former method fixes the window at the beginning of the distribution, we call it the *fixed window method*. For the latter method, because the window is continuously moving to capture the latest substream when new data generated by the same distribution arrive, we call it the *moving window method*.

Both methods may be problematic in practice. As mentioned in Section 3.1, the distribution of a large data set may not be properly captured by using a small subset that is selected based on only one criteria: data in the subset arrive within a certain time period. Furthermore, for a stream with slow distribution changes, it may be a while until the new distribution is stabilized. Hence, the substream captured immediately after the distribution change cannot reflect the real distribution. Therefore, this substream is not suitable to be the representative set of the current distribution.

Although the size of the representative set has to be limited due to memory and efficiency concerns, if, instead of blindly using the first or last set of data generated by the current distribution, a sample set with the same size can be chosen "intelligently" by considering the characteristics of the distribution, then the distribution of this selected set may be much closer to the true distribution.

Based on this idea, a new reference window selection method called *merged window method* is proposed for solving the problem of representing a (complicated) distribution using a small sample set accurately. Details of this proposed method are described in Section 3.3.3.

Distribution change detection is then performed by comparing the data set in the merged window with the newly arrived data. If there is a large discrepancy, then a distribution change is detected.

3.3.2 Tumbling window design

To detect changes, two time-based windows are maintained on a stream S: a *reference window* W_r with time interval Δ_r and an *observation window* W_t with time interval Δ_t. Substream S_r in W_r represents the current distribution, whereas the substream S_t in W_t records the set of data elements that have arrived in the last Δ time units. The change detection procedure is triggered every time W_t moves.

The observation window W_t is implemented as a tumbling window. Every Δ_t time units, W_t tumbles so that all the elements in it are deleted and a new empty window is opened. W_t is implemented as a tumbling window, rather than the more common sliding window, because of performance considerations. A time-based sliding window moves forward for each unit of time (when time moves forward), and, all items in the window with a timestamp less than $(t_{now} - \Delta)$ are evicted. Since the change detection process is triggered every time W_t moves, using a sliding window would greatly reduce the efficiency of the change detection process when the stream has a high arrival rate.

The time intervals Δ_r of reference window W_r and Δ_t of observation window W_t are pre-defined values. If the arrival speed of S is relatively slow and stable, then larger intervals Δ_r and Δ_t are chosen, indicating bigger window sizes $|W_r|$ and $|W_t|$, respectively. In contrast, if S has high arrival rate, or if the speed of S may change drastically (e.g., "bursts" in data), then smaller Δ_r and Δ_t are more proper.

Intuitively, the larger the representative set S_r in W_r, the closer its distribution is to the real distribution of S, and potentially the higher the accuracy of distribution change detection. However, memory limitation forces the size of S_r to be as small as possible. Furthermore, large S_r may reduce the efficiency of change detection technique because large amount of data is involved in the distribution discrepancy calculation. As mentioned above, the time interval Δ_t of W_t indicates the frequency with which the change detection procedure is triggered. Smaller Δ_t values would mean more frequent change detection and, thus, the number of delayed alarms will be fewer. However, smaller Δ_t values reduces the efficiency of the proposed change detection technique. Therefore, the values of Δ_r and Δ_t should be determined according to the accuracy and efficiency requirements of the relevant application.

3.3.3 Generating reference window

Definition 3.1 Let W_A and W_B be two windows on stream S containing substreams S_A and S_B, respectively. We define the *concatenation* of windows W_A and W_B, denoted as $W_A + W_B$, as the set union of the two substreams within them, i.e., $W_A + W_B = S_A \cup S_B$. Therefore, $\forall \langle s_i, t_i \rangle$ in $W_A + W_B$, $\langle s_i, t_i \rangle \in S_A$ or $\langle s_i, t_i \rangle \in S_B$.

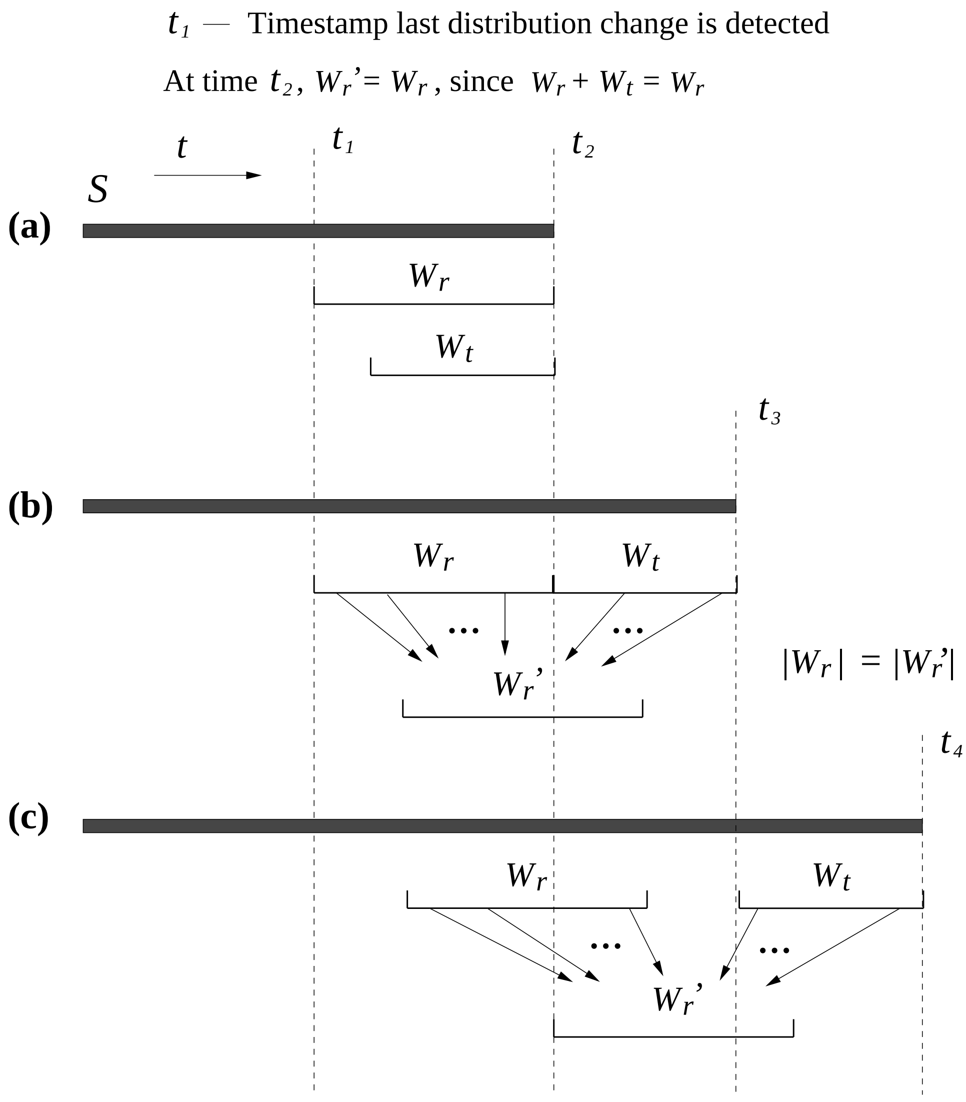
Notice that although by Definition 2.2, the arrival time of each element is not considered when calculating the distribution, the timestamps attached to the elements are not removed when concatenating two windows. Therefore, two elements $\langle s_i, t_i \rangle$ and $\langle s_j, t_j \rangle$ with the same values, i.e., $s_i = s_j$, are not considered duplicates in $W_A + W_B$. This guarantees that the probability of a value that occurs in many elements will be calculated correctly. Thus, $|W_A + W_B| = |W_A| + |W_B|$ even when W_A and W_B overlap.

The overview of generating the dynamic reference window W_r is illustrated in Figure 3.1. Let a distribution change be detected at time t_1. Starting at t_1, W_r records a substream S_r that contains the first $|W_r|$ observed elements. Observation window W_t contains substream S_t with the newest $|W_t|$ elements (Figure 3.1a). At time t_2, W_t is full and ready to tumble forward. If, at this point, a distribution change is not detected, W_r and W_t are concatenated/merged so that a larger substream with size $|W_r + W_t|$, denoted as $W_t + W_r$ as per Definition 3.1, is available (Figure 3.1b). Then $|W_r|$ elements are selected from the concatenated window $(W_r + W_t)$ as the latest representative set, and this replaces the substream in W_r[1] (Figure 3.1c). This merge-and-select process is triggered every time W_t tumbles. Hence, elements in W_r can be regarded as a "concentration" of all data from the current distribution that have been observed so far. In other words, this representative set is a careful selection from the large vault of observed elements since the beginning of the distribution. Thus, substream S_r in W_r is highly representative of the current distribution. This merge-and-select process continues until the next time a distribution change occurs.

Ideally, what is desired is to find the data set W'_r in $W_r + W_t$, such that its distribution is the closest to the true distribution of S, i.e., $Discrepancy(P'_r, P) = min(Discrepancy(\forall P^i_r, P))$, where P^i_r is the distribution of any substream in $(W_r + W_t)$ with size $|W_r|$. However, finding such a substream can be computationally expensive. For stream applications that require high efficiency (e.g., fraud detection and stock ticker monitoring), an approximate algorithm is required.

The goal for the approximation is to find a substream W'_r with size

[1]In the rest of this section, W_r and W'_r are used to denote the reference windows before and after merge-and-select process, respectively.



t_1 — Timestamp last distribution change is detected
At time t_2, $W_r' = W_r$, since $W_r + W_t = W_r$
t
t_1
t_2
S
(a)
W_r
W_t
t_3
(b)
W_r
W_t
...
...
W_r'
$|W_r| = |W_r'|$
t_4
(c)
W_r
W_t
...
...
W_r'

Figure 3.1: Reference window generation

$|W_r|$ from $W_r + W_t$, such that:

$$Discrepancy(P'_r, P) \leq min(Discrepancy(P_t, P), Discrepancy(P_r, P)) \tag{3.1}$$

To achieve this goal, a two-step sampling approach is proposed. First the density function of the distribution of data set in (W_r+W_t) is estimated using the popular kernel density estimation [133]:

$$\hat{f}_h(s) = \frac{1}{(|W_r + W_t|)h} \sum_{i=1}^{|W_r+W_t|} K(\frac{s - s_i}{h}) \tag{3.2}$$

where $K()$ is the kernel function, h is the smoothing parameter called bandwidth, and s_i is a data element in $(W_r + W_t)$. $K()$ is set to be the standard Gaussian function with mean zero and variance 1. This setting is usually chosen when no pre-knowledge of the distribution is available [146]. Thus,

$$K(s) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}s^2} \tag{3.3}$$

Hence, equation 3.2 can be rewritten as

$$\hat{f}_h(s) = \frac{1}{(|W_r + W_t|)\sqrt{2\pi h^2}} \sum_{i=1}^{|W_r+W_t|} e^{-(s-s_i)^2/2h^2} \tag{3.4}$$

The bandwidth h is selected using Silverman's rule of thumb [146], i.e.,

$$h = 0.9min(\hat{\sigma}, \mathcal{Q}/1.34)(|W_r + W_t|)^{-1/5} \tag{3.5}$$

where $\hat{\sigma}$ is the standard deviation of $K()$ and $\mathcal{Q}$ is the interquartile range of $K()$. This is a classical setting that is used by many real-world applications.

Although the kernel density estimation approach can generate an estimated density function of the current distribution, as discussed in Section

3.2, the accuracy of this estimation heavily dependents on the bandwidth selection. The distribution of a data set can be over-simplified or over-complicated with a fixed bandwidth. However, it is difficult to obtain optimal bandwidth for each distribution in a dynamic stream. Hence, kernel density estimation is used only as a preliminary guidance for the sampling strategy.

Figure 3.2 illustrates the idea of the proposed two-step sampling approach, where x-axis is the value of data s in $(W_r + W_t)$, and y-axis is the density. The probability of s falling in the range $[v_i, v_j]$ is the area under the section of curve in $[v_i, v_j]$. In the first step, the whole area under the density function $\hat{f}_h(s)$ is partitioned into k disjoint groups $G_1, G_2, ..., G_k$, where k is a constant (k value selection is discussed shortly). The start and end value for each partition G_i is denoted as g_i^S and g_i^E, respectively. Each partition has the same area, i.e., $area(G_i) = \frac{1}{k} area(\hat{f}_h(s))$ $(i = 1, .., k)$.

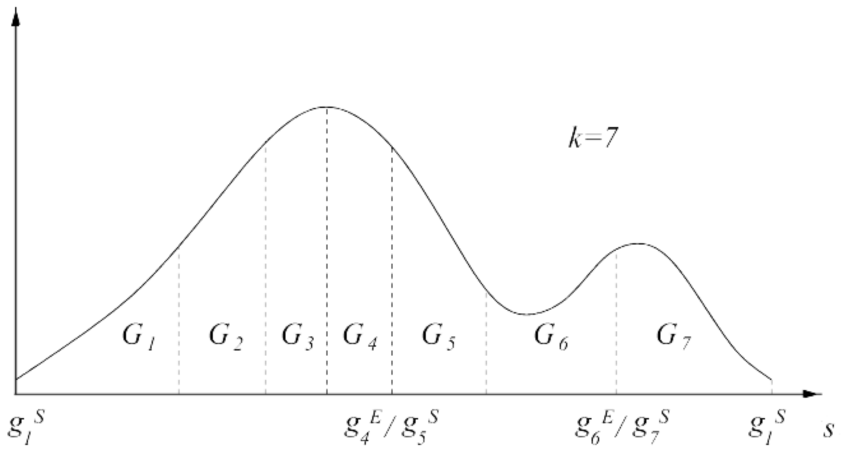


Figure 3.2: Example of two-step sampling

The second step is sampling from each partition. The same number of data elements are sampled from each partition. Since the resulting representative set is stored in W_r, in each partition G_i a total of $S_{Gi} = \frac{1}{k}|W_r|$ data are selected, where S_{Gi} represents the data elements selected from G_i.

To partition $(W_r + W_t)$ into a number of groups G_i $(i = 1, ..., k)$, the empirical cumulative distribution function of the data set in $(W_r + W_t)$ is calculated as:

$$\hat{F}(x) = \frac{1}{|W_r + W_t|} \sum_{i=1}^{|W_r+W_t|} 1_{(-\infty,x]}(s_i), s_i \in (W_r + W_t) \tag{3.6}$$

where $1_{(-\infty,x]}(s_i)$ is the indicator function defined as:

$$1_{(-\infty,x]}(s_i) = \begin{cases} 1 & \text{if } s_i \leq x; \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

The start and end values g_i^S and g_i^E for each partition G_i $(i = 1, ..., k)$ is then calculated as follows:

$$g_i^S = \begin{cases} min(s_j), \forall \langle s_j, t_j \rangle \in (W_r + W_t) & i = 1; \\ x \text{ where } \hat{F}(x) = (i-1)/k & i = 2, ..., k \end{cases} \tag{3.8}$$

$$g_i^E = \begin{cases} x \text{ where } \hat{F}(x) = i/k & i = 1, ..., k-1; \\ max(s_j), \forall \langle s_j, t_j \rangle \in (W_r + W_t) & i = k \end{cases} \tag{3.9}$$

On the second step, a number of representative data elements are selected from each partition. The number of representative data selected from partition G_i is determined by $k_i = round(|W_r|/k)$. If $\sum_1^k k_i \neq |W_r|$, then several data elements are added to or removed from random groups to ensure $|W_r'| = |W_r|$.

Within each partition G_i, k_i data elements are randomly selected with the indicator function $1_{G_i(s)}$ as:

$$1_{G_i(s)} = \begin{cases} 1 & \text{if } g_i^S \leq s \leq g_i^E; \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

The random selection strategy is done to eliminate biases introduced through the kernel density estimation and partition process. The final substream S'_r in the updated reference window W'_r is the union of all data elements selected from all groups, i.e., $S'_r = S_{G_1} \cup S_{G_2} \cup ... \cup S_{G_k}$.

The number of total partitions k is a predefined constant value for each distribution. A higher k value implies a better quality of the representative set selection. However, note that computation cost and memory consumption will also increase with higher k. Therefore, for a "smooth" distribution, i.e., fewer high density areas or less "bumpy" shape, k can be kept smaller, which means that $(W_r + W_t)$ can be partitioned into fewer groups.

3.3.4 Change detection

The change detection procedure is triggered every time W_t tumbles. Let S_r and S_t be the substreams in W_r and W_t, respectively. Let P_r and P_t be the distributions that S_r and S_t represent, respectively. If $Discrepancy(P_r, P_t) > 1 - \tau$, where τ is a predefined threshold referred to as *significance level*, then a distribution change is detected. In other words, S_r and S_t will be considered as being generated by the same distribution only when P_r and P_t have a similarity same or higher than the significance level τ.

To measure the discrepancy between P_r and P_t, some statistical test can be applied. Examples of such tests include kernel density comparison [3] and various distance function based tests [142, 155, 28, 78].

Therefore, a complete window-based change detection technique includes a window selection method and a statistical test for calculating discrepancy. The choice of the window selection method and the choice of the statistical test are independent. Hence, the merged window method proposed in Section 3.3.3 can be associated with existing statistical tests to form new change detection techniques. The performances of some of these techniques are evaluated in Section 3.3.6.

3.3.5 Experimental framework

To evaluate the performance of different change detection approaches over streams with various distributions, change durations, and parameter settings, an experiment framework is designed and implemented. This framework includes synthetic stream generation, change detection, and experimental results illustration.

Data stream generation

To generate a stream S with distribution changes, three parameters need to be set: stream size n, number of distribution changes in the stream $mChg$, and the significance level τ. The stream size n determines the number of data elements in the synthetic stream. Although, theoretically, a data stream is unbounded and sometimes even considered infinite, to gain control over the testing streams, in the experiment framework the unbounded stream size is replaced by a sufficiently large number n. The performance of a change detection technique on unbounded streams can be estimated by its performance on streams with a large number of data elements. The other two parameters $mChg$ and τ are used to generate distribution changes in the stream.

The change duration of each distribution in a stream is the total number of data elements that are generated by this distribution. Let $S_1, S_2, ..., S_{mChg+1}$ be the substreams that contain data elements generated from different distribution. For stream S with total of n data elements and $mChg$ changes, we generate $mChg$ random numbers $chg_1, chg_2, ..., chg_{mChg}$ within the range $(1, n)$. These $mChg$ numbers indicate the "location" where the new distribution change occurs. Therefore, stream S contains $mChg + 1$ different distributions with change durations $S_1[s_1, s_{chg_1})$, $S_2[s_{chg_1}, s_{chg_2}), ..., S_{mChg}[s_{chg_{mChg-1}}, s_{chg_{mChg}}), S_{mChg+1}[s_{chg_{mChg}}, s_n]$, where $s_1, ..., s_n$ are the n data elements in stream S. Because the locations of the distribution changes are randomly generated, in stream S there could be distributions that last for a long time and distributions that only contain a few data elements.

The data in each substream $S_i[s_{chg_{i-1}}, s_{chg_i})$ is generated by one distribution type P_i. The type of changes among $P_1, P_2, ..., P_{mChg+1}$ of substreams $S_1, S_2, ..., S_{mChg+1}$ can be either location change (i.e., mean values change) or scale change (i.e., standard deviation change).

The location or scale (depends on the change type) of the distribution for each substream is randomly generated. However, it is possible that the randomly generated location or scale of the distributions of two consecutive substreams S_i and S_{i+1} are very close, so that $Discrepancy(P_i, P_{i+1}) \leq 1-\tau$. For this case, there is actually no distribution change between S_i and S_{i+1}. To avoid this case, each time the location or scale of a distribution P_i is generated, it is compared with its immediate preceding distribution P_{i-1}. If the two distributions are too similar according to τ, the location or scale of P_i is regenerated until $Discrepancy(P_i, P_{i-1}) > 1 - \tau$. This guarantees that there are $mChg$ numbers of true changes in S.

The speed of a distribution change can be either abrupt or gradual. The abrupt changes are called distribution *shifts* and the gradual changes are called distribution *drifts*. Distribution drifts are usually more difficult to detect and may not be detected as fast as distribution shifts. A change detection technique may perform differently between detecting shifts and detecting drifts. To evaluate the performances of change detection techniques over different change speeds, streams with either shifts or drifts are generated.

For a stream with only distribution shifts, the stream is a direct concatenation of all the substreams, i.e., $S = S_1 + S_2 + ... + S_{mChg+1}$. To generate streams with slow and gradual distribution drifts, the parameter of drift duration $driftDur$ is defined. For substreams $S_i[s_{chg_{i-1}}, s_{chg_i})$ and $S_{i+1}[s_{chg_i}, s_{chg_{i+1}})$, the distribution drift starts from data $s_{chg_i - \lfloor driftDur/2 \rfloor}$ and ends at $s_{chg_i + \lfloor driftDur/2 \rfloor}$. The values of the data in the drifting period gradually change from the value of $s_{chg_i - \lfloor driftDur/2 \rfloor}$ to the value of $s_{chg_i + \lfloor driftDur/2 \rfloor}$ linearly, i.e., $s_{chg_i - \lfloor driftDur/2 \rfloor + j} = s_{chg_i - \lfloor driftDur/2 \rfloor}$ $+ j \times (s_{chg_i + \lfloor driftDur/2 \rfloor} - s_{chg_i - \lfloor driftDur/2 \rfloor})/driftDur$.

Based on the above discussion, 18 stream types are generated using the proposed experimental framework. These stream types are described in Table 3.1

Change detection techniques

For each change detection technique implemented within the experimental framework, a reference window and an observation window are used

Table 3.1: Stream types generated

Stream type	Distribution type	Chg type	Chg speed
$Stream1$	Normal distribution	Location	Shifts
$Stream2$	Uniform distribution	Location	Shifts
$Stream3$	Exponential distribution	Location	Shifts
$Stream4$	Binomial distribution	Location	Shifts
$Stream5$	Half-half mix of two Normals with different mean	Location	Shifts
$Stream6$	Half-half mix of one Normal and one Uniform with different mean	Location	Shifts
$Stream7$	Normal distribution	Scale	Shifts
$Stream8$	Uniform distribution	Scale	Shifts
$Stream9$	Exponential distribution	Scale	Shifts
$Stream10$	Binomial distribution	Scale	Shifts
$Stream11$	Half-half mix of two Normals with different mean	Scale	Shifts
$Stream12$	Half-half mix of one Normal and one Uniform with different mean	Location	Shifts
$Stream13$	Normal distribution	Location	Drifts
$Stream14$	Exponential distribution	Location	Drifts
$Stream15$	Binomial distribution	Location	Drifts
$Stream16$	Half-half mix of two Normals with different mean	Location	Drifts
$Stream17$	Uniform distribution	Scale	Drifts
$Stream18$	Half-half mix of one Normal and one Uniform with different mean	Scale	Drifts

to store the representative data set and the newly arrived data. Three reference window moving methods are implemented and compared:

- Fixed window method.

  The reference window is fixed at the beginning of the current distribution. It is the first window after the last distribution change has been detected. This is the window moving method adopted in [78].

- Moving window method.

  The reference window is always immediately before the observation window, i.e., reference window contains data arrived during $(t_1, t_2]$ and observation window contains data arrived in $(t_2, t]$, where t is the current time. This is the window moving method used in kernel density based approach [3].

- Merged window method.

  This is the window moving method that we propose, where the previous reference window is merged with observation window and a subset of the merged window is selected to be the current reference window.

For calculating the discrepancy between two distributions, two statistical tests are implemented using these three window moving method: the kernel density comparison (KD) [3] and the distance function-based approach (XI) [78]. The KD test uses kernel density estimation to generate the densities of these two data sets and calculate the difference between them. Among different choices of kernel functions, the gaussian kernel function is recommended by the author; hence, in the experiments, we apply gaussian function to KD test. In XI test, distribution change is detected by calculating the distance between the data sets in the two windows. Several distance functions are proposed in [78]. According to their experiments, the XI distance has the overall best performance. Thus, XI distance testing is implemented to compare with the KD test.

By combining the three window moving methods with the two tests, there are a total of six change detection techniques implemented and compared in the experimental framework. These six techniques are summarized in Table 3.2. The "Legend" column in the table illustrates the line colors and styles that are used in the result figures in Section 3.3.6.

Table 3.2: Window-based change detection techniques

Detection technique	Window moving method	Statistical test	Legend
XI-fixed	Fixed window method	XI distance testing	—
KD-fixed	Fixed window method	Kernel density comparison	━
XI-moving	Moving window method	XI distance testing	- - -
KD-moving	Moving window method	Kernel density comparison	━ ━ ━
XI-merged	Merged window method	XI distance testing	⋯
KD-merged	Merged window method	Kernel density comparison	⋯

Each stream type in Table 3.1 is repeatedly generated and tested using these six change detection techniques. The total number of streams generated for each stream type and tested over each change detection techniques is controlled by parameter *numRun*.

Results analysis

For each change detection technique applied on each stream type, there are a total of *numRun* sets of results generated. These result sets record five important criteria for evaluating the performance of the change detection technique:

- Number of true changes detected.

  For one stream S generated using one of the 18 stream types in Table 3.1, let $chg_1, chg_2, ..., chg_{mChg}$ be the locations where true changes occur. Let $det_1, det_2, ...,$
  det_{kDet} be the locations where distribution changes are reported by the change detection technique. Note that the number of true

changes $mChg$ may not be the same as the number of changes detected $kDet$, since there may be true changes missed and false changes reported.

For streams with distribution shifts, if $det_{i-1} < chg_j \leq det_i$, then det_i is a true change detection point. Because there is a true change occurred within $(s_{det_{i-1}}, s_{det_i}]$ and this change is detected at location det_i.

For streams with distribution drifts with drifting period $[chg_j - \lfloor driftDur/2 \rfloor,\ chg_j + \lfloor driftDur/2 \rfloor]$, any change detection point that is within this drifting period is considered neither a true change nor a false change. Since a distribution does change during the drifting period, the changes reported during this period are not false. However, these changes are not important as the new distribution is not stabilized and, thus, they are not counted in the results. Hence, a true change detection point det_i for streams with drifts must be at a location where $det_{i-1} < chg_j + \lfloor driftDur/2 \rfloor \leq det_i$.

- Number of false changes detected.

  For streams with distribution shifts, if $chg_{j-1} \leq det_{i-1} < det_i < chg_j$, then det_i is regarded as a false change detection, because no true change has occurred within $(s_{det_{i-1}}, s_{det_i}]$. For streams with distribution drifts, det_i is regarded as a false change detection if $chg_{j-1} + \lfloor driftDur/2 \rfloor \leq det_{i-1} < det_i < chg_j - \lfloor driftDur/2 \rfloor$.

- Mean duration for detecting true changes.

  For streams with distribution shifts, for each true change detection point det_i that detects the distribution change at chg_j, the change detection duration is calculated as $Dur_j = det_i - chg_j$. For streams with distribution drifts, the change detection duration is calculated as $Dur_j = det_i - (chg_j + \lfloor driftDur/2 \rfloor)$. A smaller duration indicates that the distribution change is detected quickly, whereas a larger duration indicates a long delay in detecting the change.

  The durations of false detection and missed true changes are not recorded. The mean duration is the mean value of the durations for all true change detections.

- Standard deviation of durations for detecting true changes.

  The standard deviation of duration is the standard deviation of the durations for all true change detections.

- Maximal duration for detecting true changes.

  The maximal duration is the maximal value of the durations for all true change detections.

To summarize the *numRun* result sets for each stream type, density figures are generated for the five criteria discussed above. In each figure, the x-axis is the value of each criteria and y-axis is the density. The densities of the six change detection techniques are compared and analyzed for each stream type.

3.3.6 Experiments

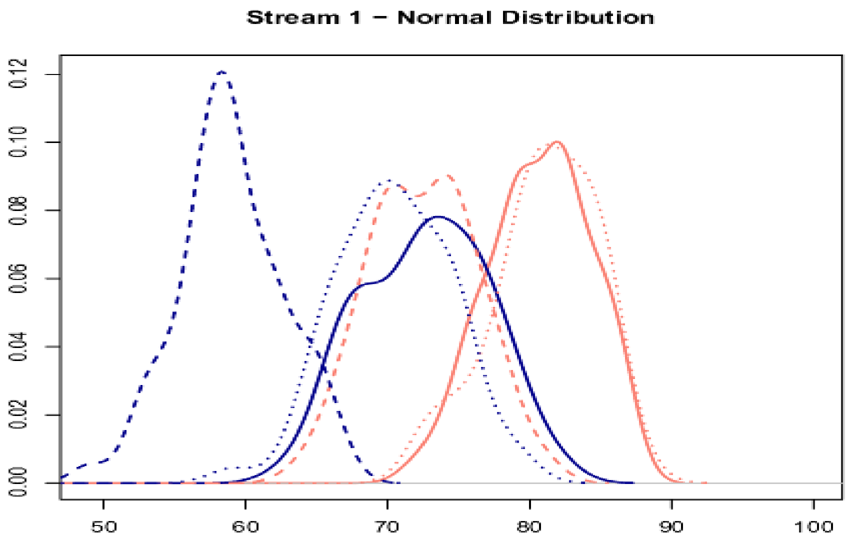
In this section, a series of experiments are presented using synthetic data streams generated by the proposed experimental framework to compare the performance of the three window moving method and two statistical tests discussed above. These experiments are carried out on a PC with 3GHz Pentium 4 processor and 1GB of RAM, running Windows XP. All algorithms are implemented in R.

Change detection evaluation

The parameters discussed in the experimental framework are set as follows. The size n of each generated data stream is set to 100,000 data elements. The number of true distribution changes $mChg$ in each stream is 100. Significance level τ is set to 80%. The impact of τ values over the change detection performance is discussed shortly. The total number of streams generated and tested for each stream type is $numRun = 100$.

The arrival speed of each stream is stable, with one tuple per unit time. This is for the purpose of gaining control over the window's length, since a time-based sliding window will be equal to a count-based one when the stream speed is stable. However, note that all the implemented change detection techniques do not require the stream to have an even speed. The sizes of both W_r and W_t are set to 100 data elements. The impact of window size is studied in later section. The number of partitions k used in the merged window method is set to 10.

To study the performance of the six testing change detection techniques over streams with different distribution types, the first set of experiments is conducted using stream types $Stream1, ..., Stream6$. The experimental results are presented by five sets of figures, where the x-axis represents the values of the five criteria discussed in Section 3.3.5 and y-axis is their density. The legend of these figures is described in Table 3.2. Figures 3.3, 3.4 and 3.5 present the numbers of true changes detected in this set of streams. The numbers of false changes detected are shown in Figures 3.6, 3.7 and 3.8. Figures 3.9, 3.10 and 3.11 show the mean values of the durations for each type of stream. Figures 3.12, 3.13 and 3.14 demonstrate the standard deviations of the durations. Distribution on the maximal durations are illustrated in Figures 3.15, 3.16 and 3.17. The line colors and styles in the figures are consistent with the ones shown in Table 3.2.



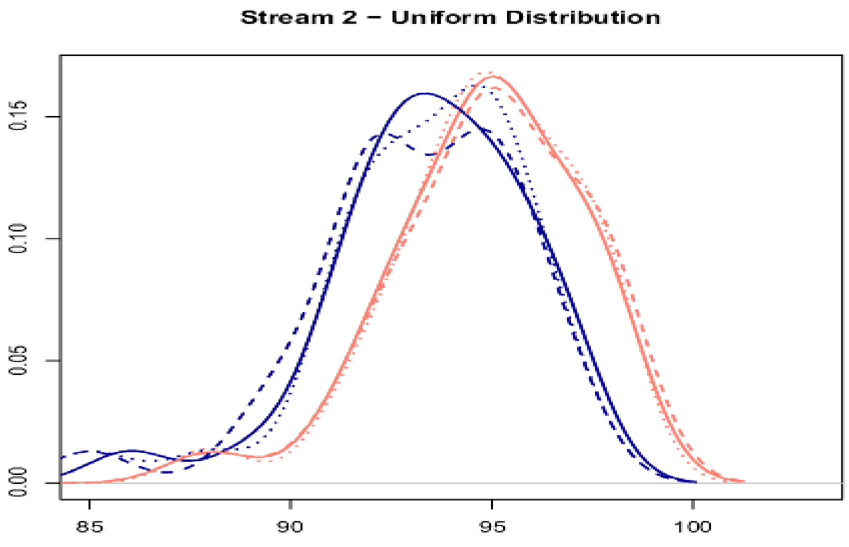
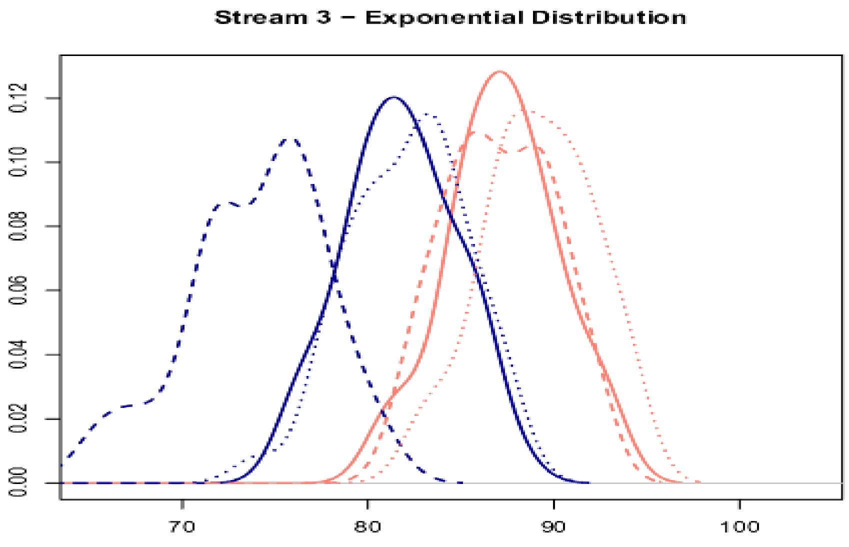


Figure 3.3: Number of true changes detected for $Stream1$ and $Stream2$



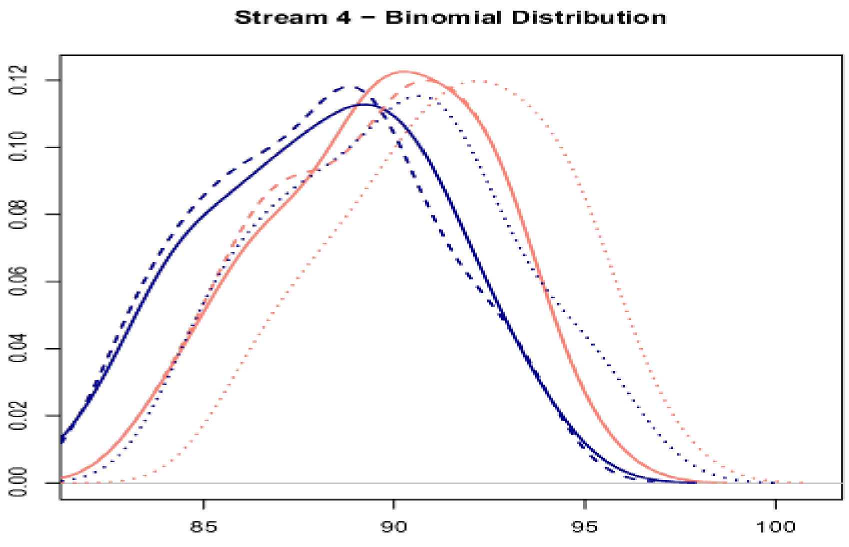
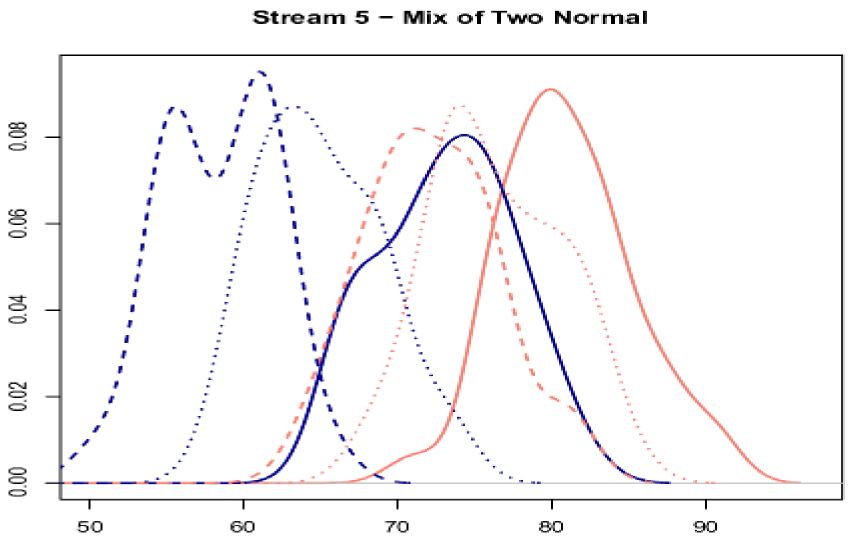


Figure 3.4: Number of true changes detected for *Stream*3 and *Stream*4



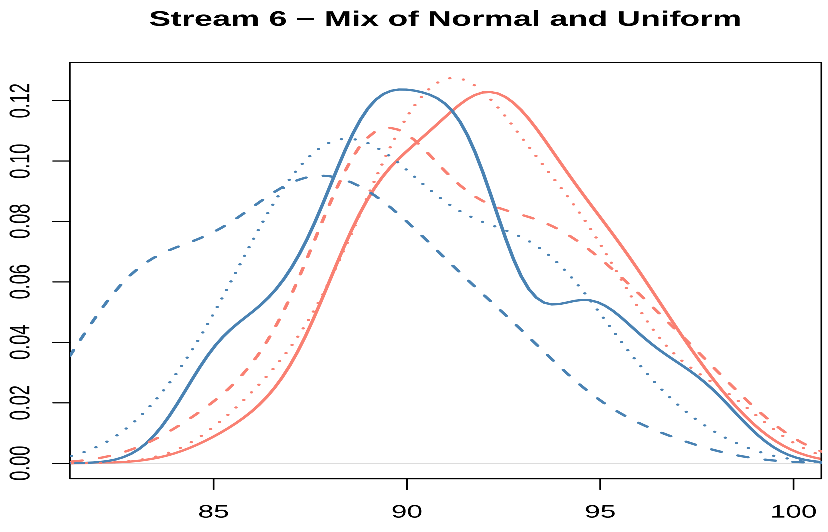
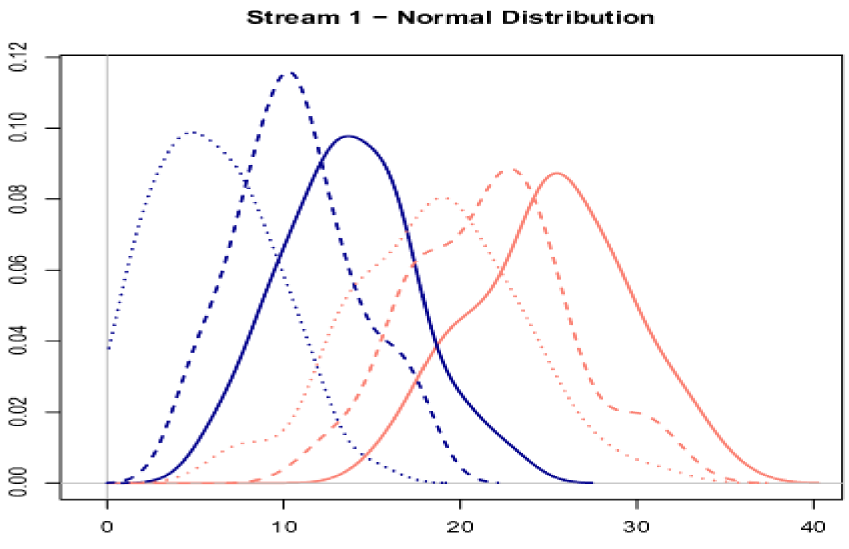


Figure 3.5: Number of true changes detected for *Stream*5 and *Stream*6



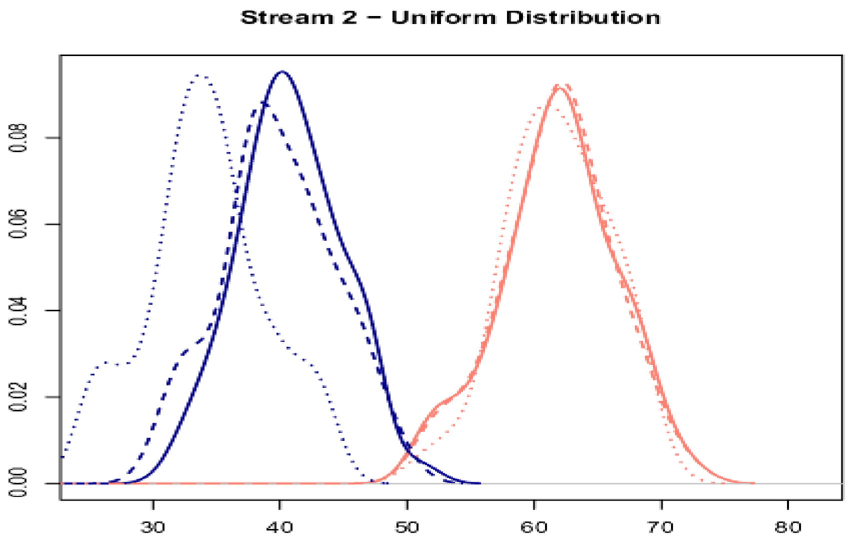
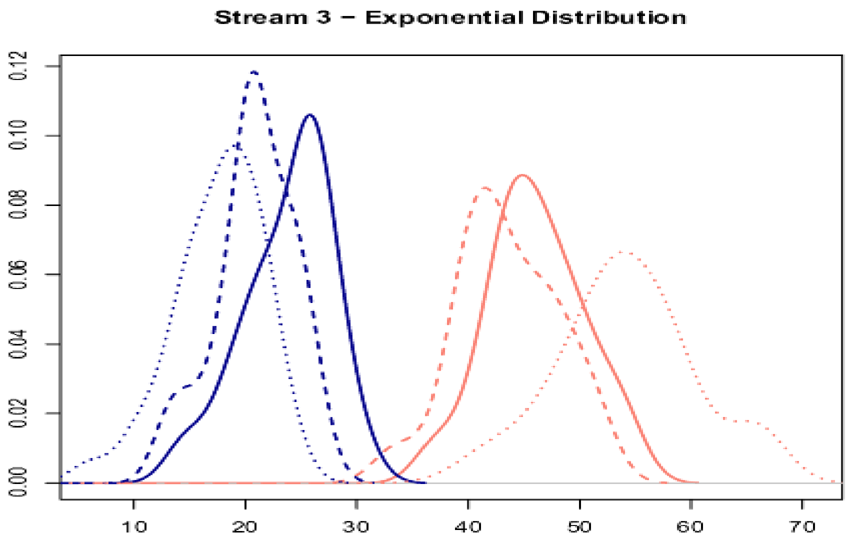


Figure 3.6: Number of false changes detected for $Stream1$ and $Stream2$



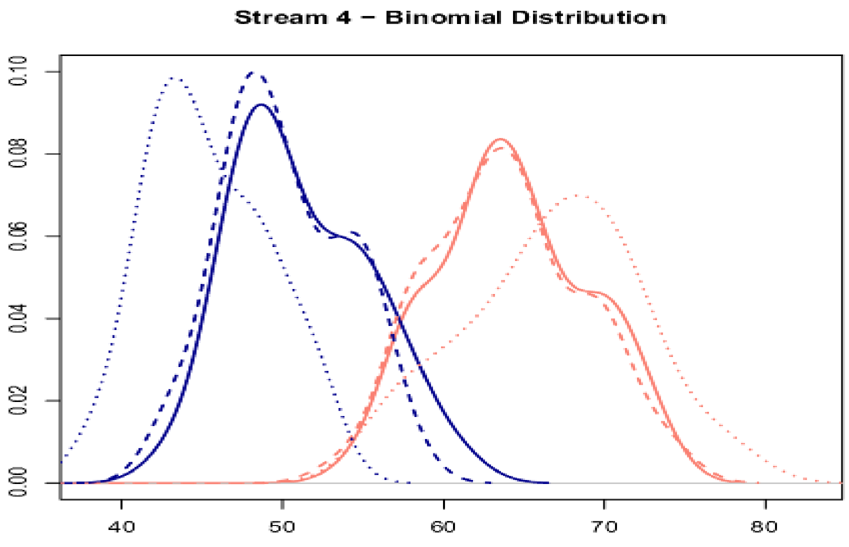
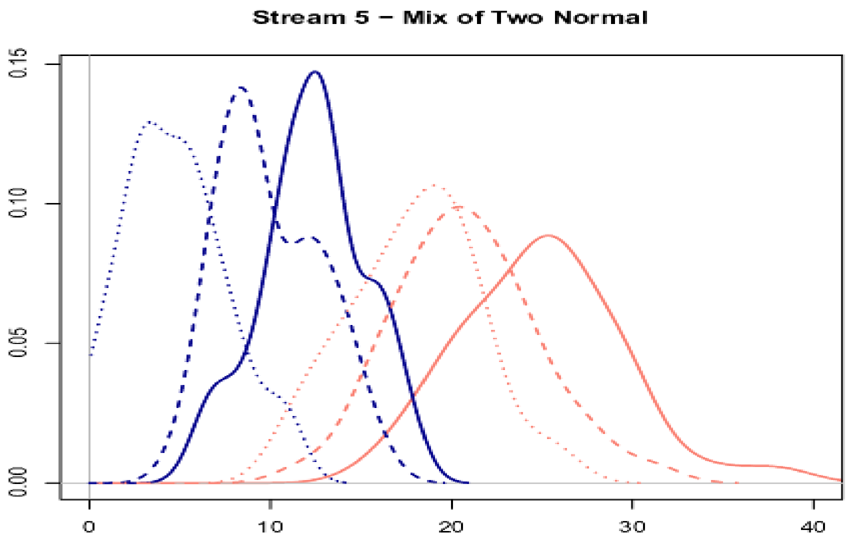


Figure 3.7: Number of false changes detected for *Stream*3 and *Stream*4



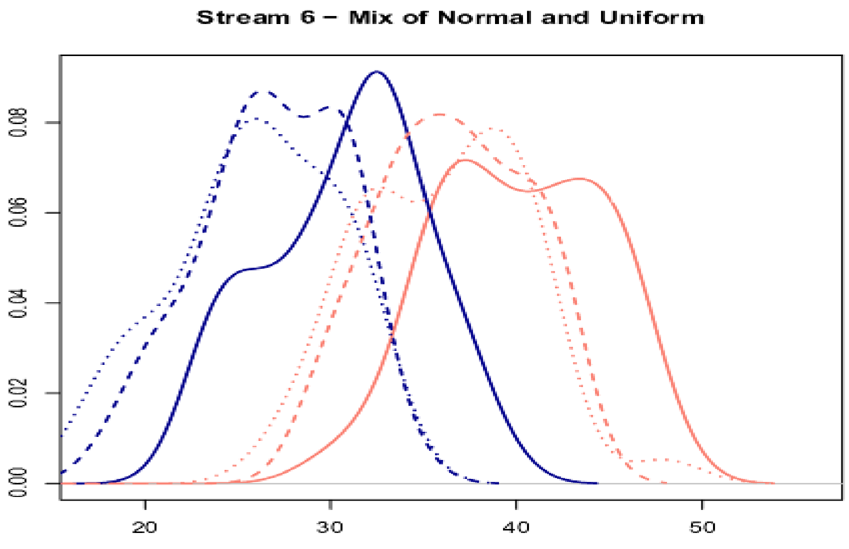
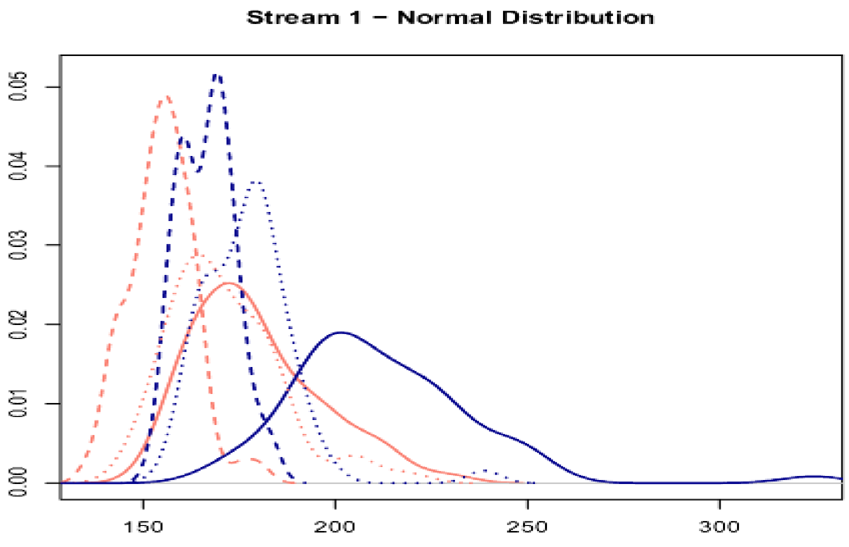


Figure 3.8: Number of false changes detected for $Stream5$ and $Stream6$



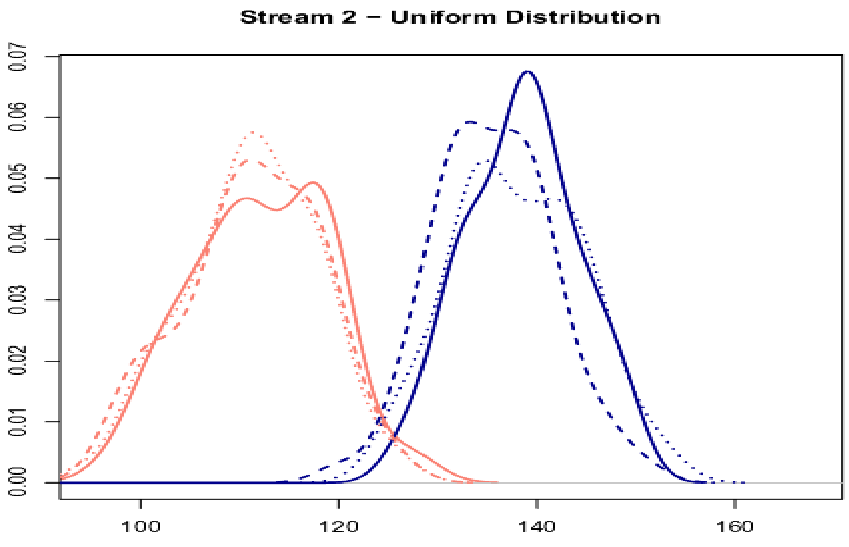
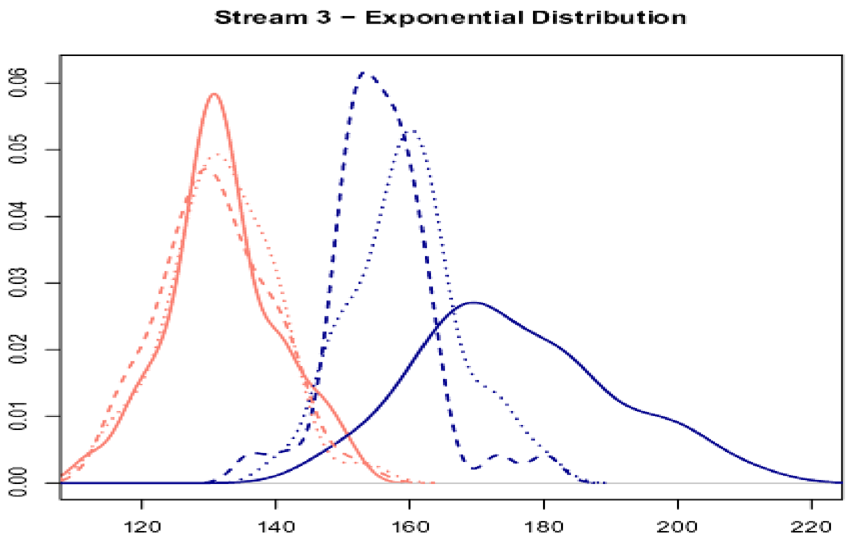


Figure 3.9: Mean duration for detecting true changes in $Stream1$ and $Stream2$



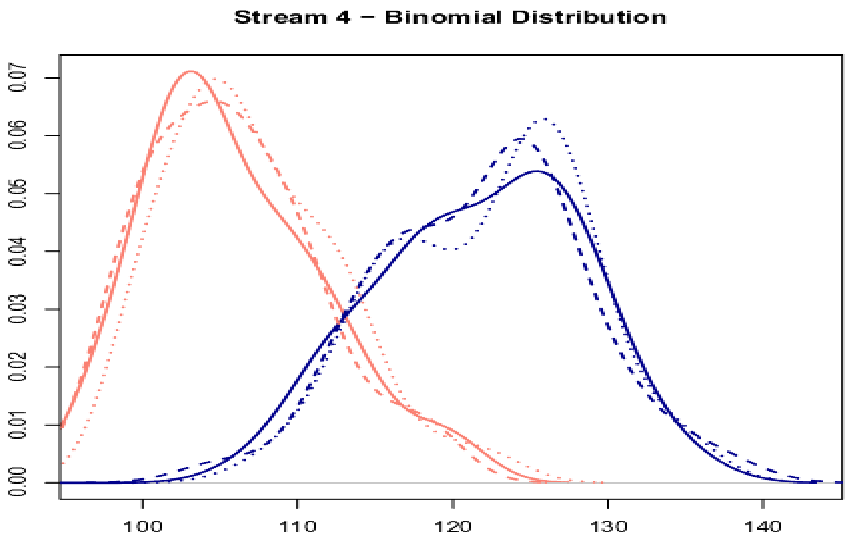
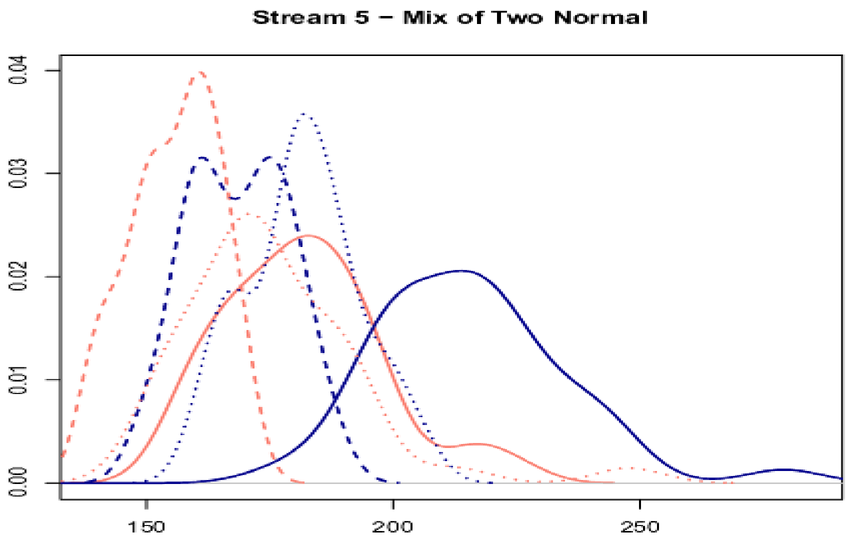


Figure 3.10: Mean duration for detecting true changes in *Stream*3 and *Stream*4



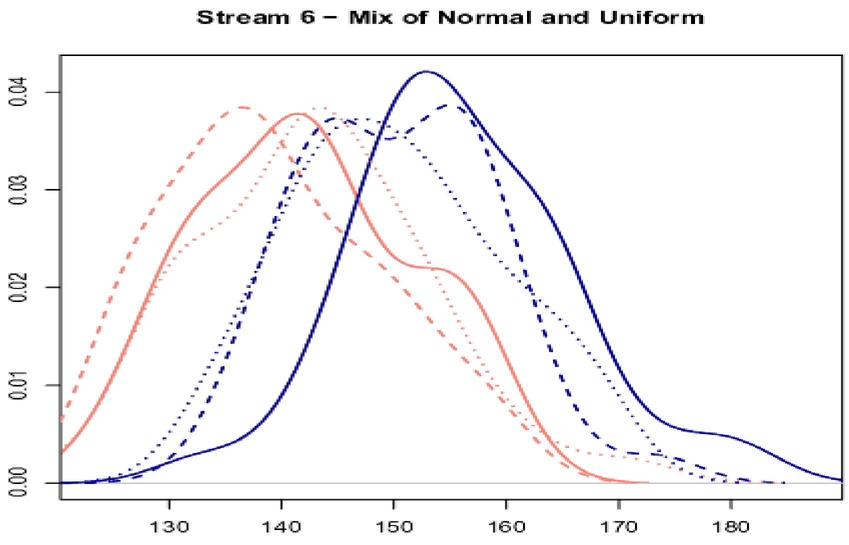
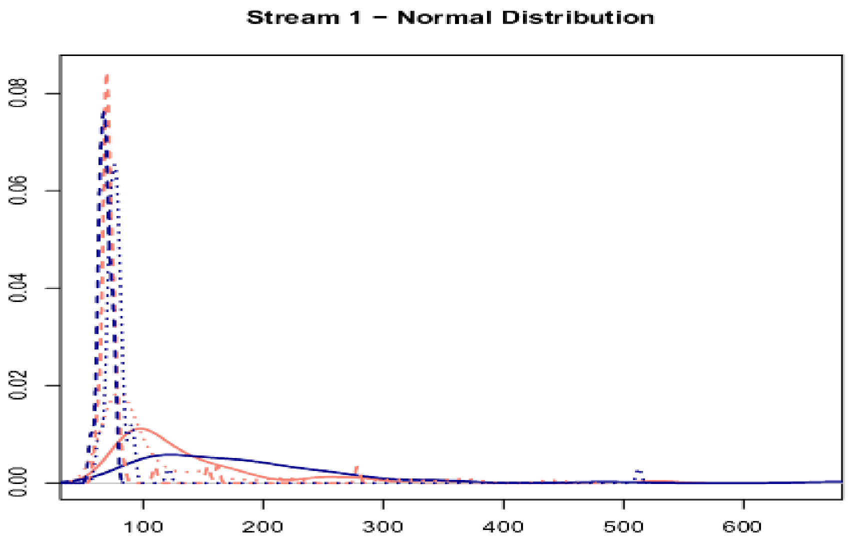


Figure 3.11: Mean duration for detecting true changes in $Stream5$ and $Stream6$



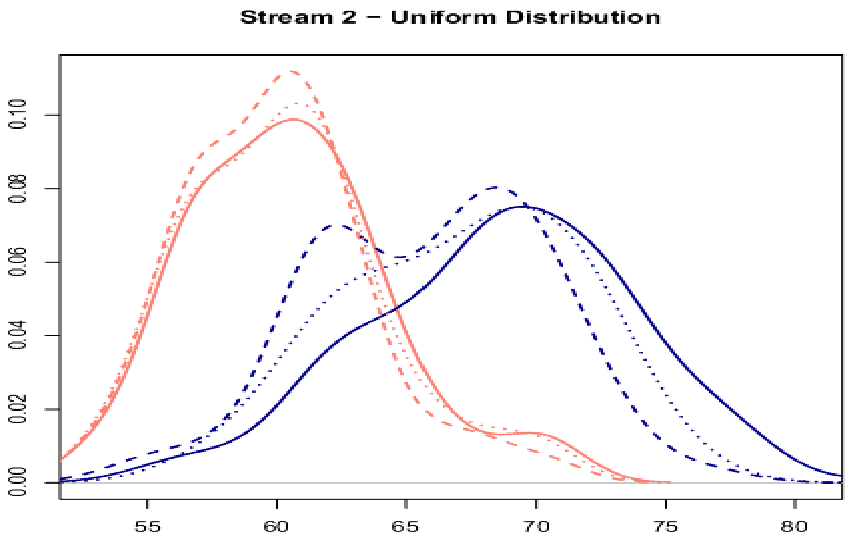
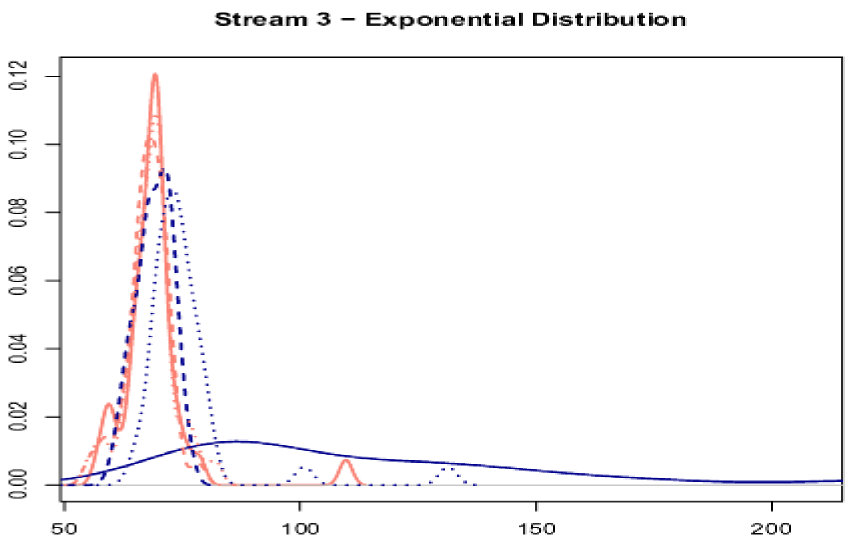


Figure 3.12: Standard deviation of the duration for detecting true changes in $Stream1$ and $Stream2$



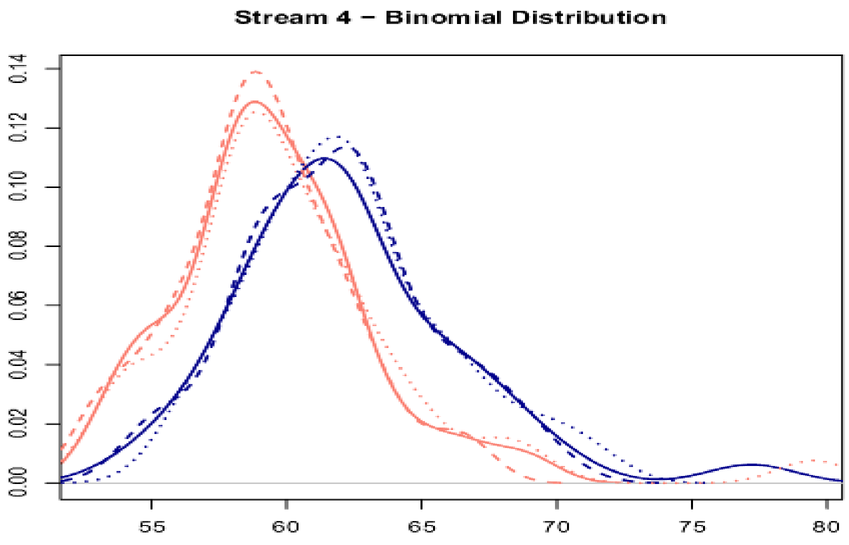
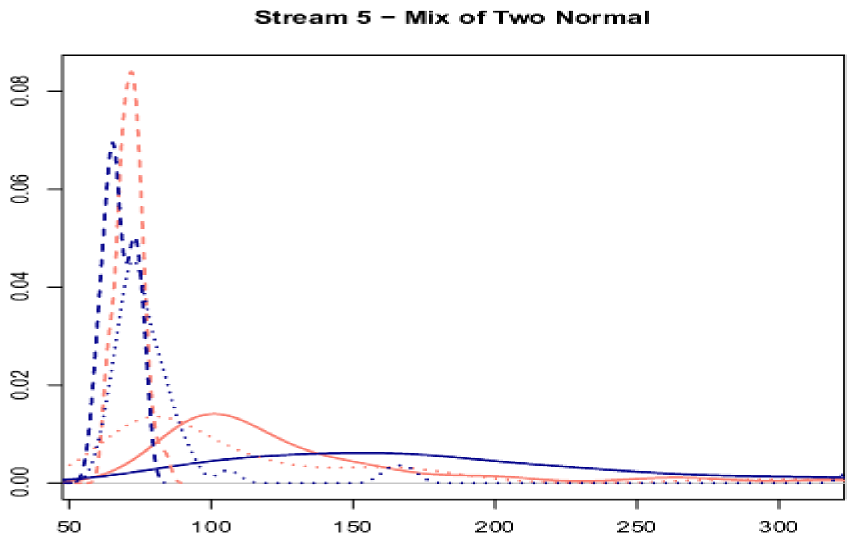


Figure 3.13: Standard deviation of the duration for detecting true changes in *Stream*3 and *Stream*4



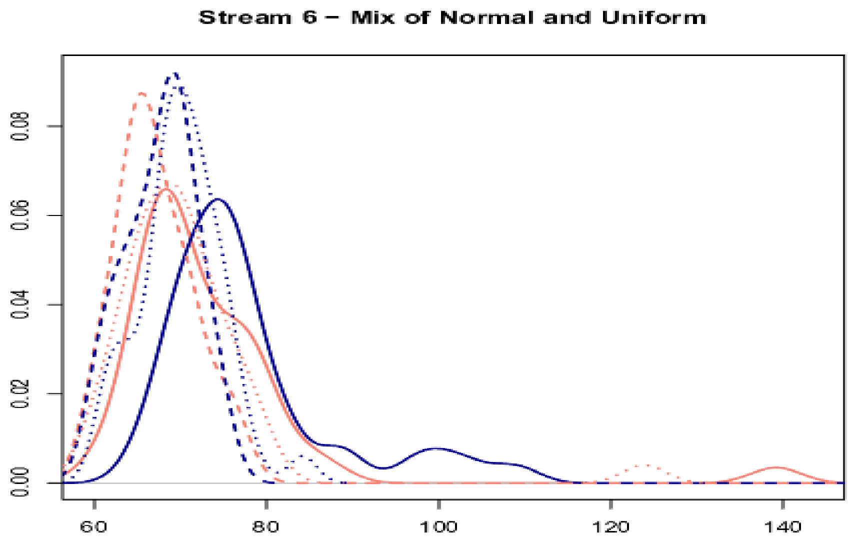


Figure 3.14: Standard deviation of the duration for detecting true changes in *Stream*5 and *Stream*6

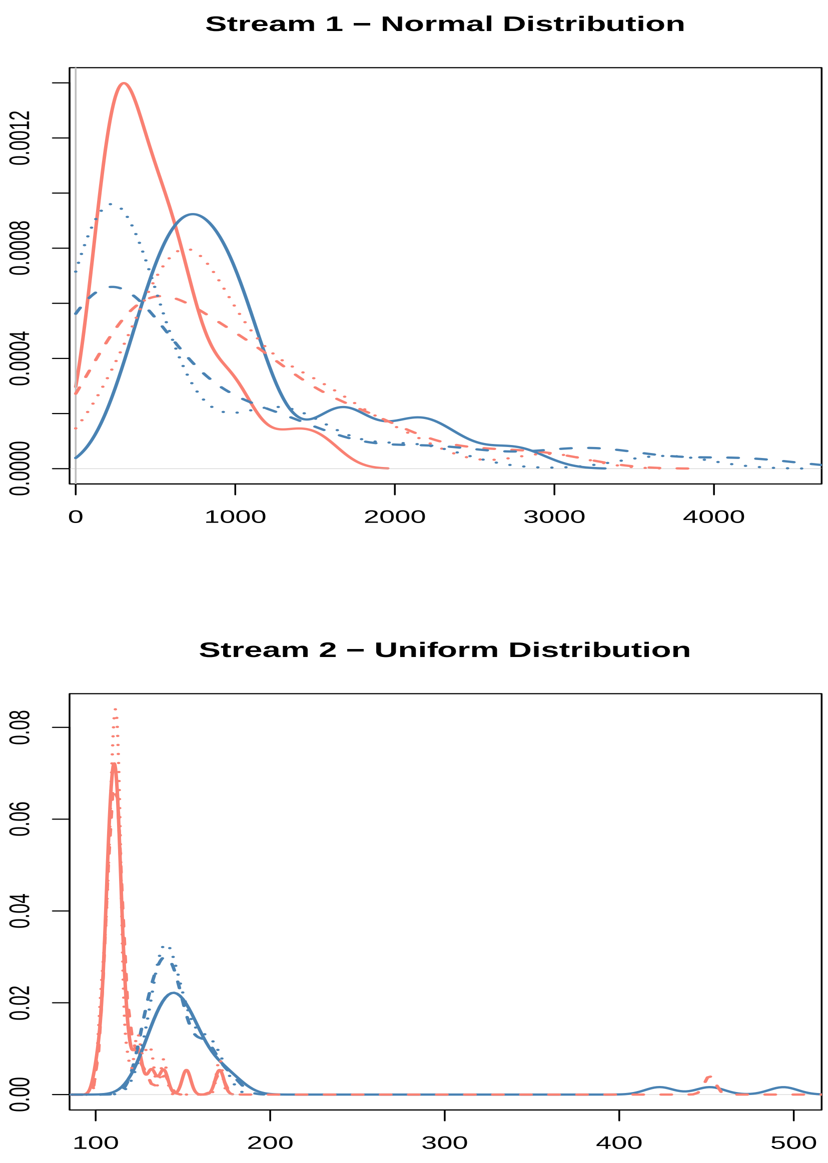
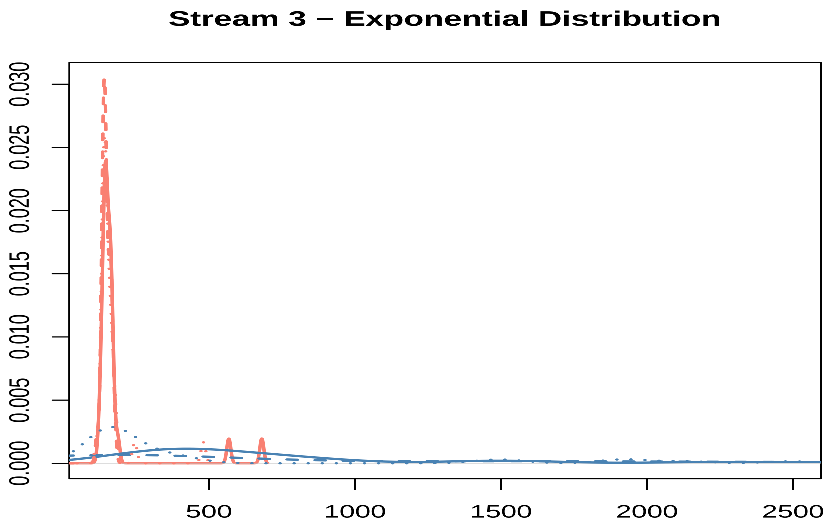


Figure 3.15: Max duration for detecting true changes in $Stream1$ and $Stream2$



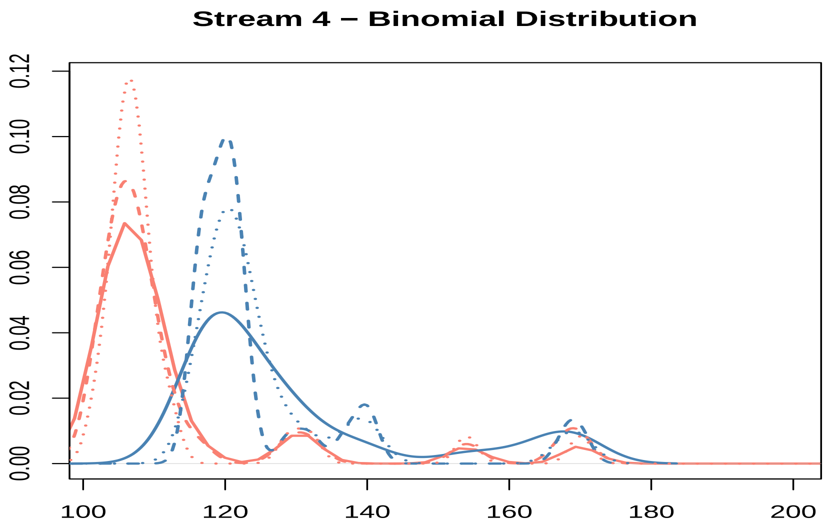
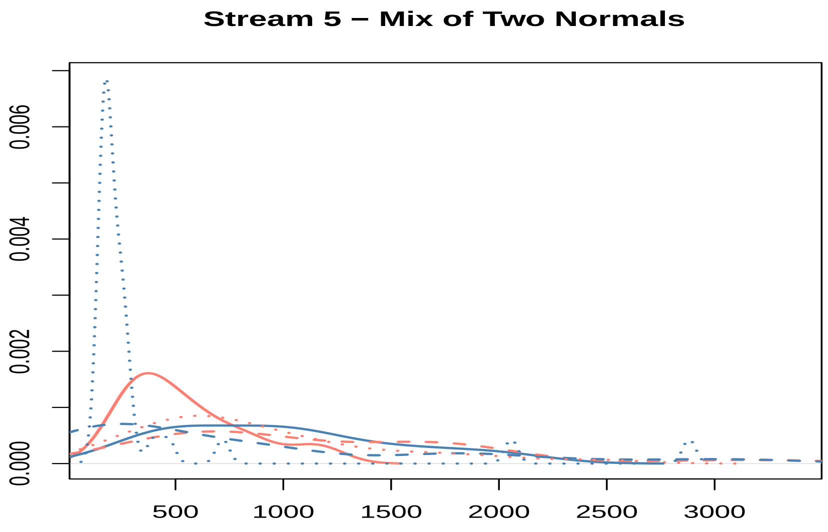


Figure 3.16: Max duration for detecting true changes in *Stream*3 and *Stream*4



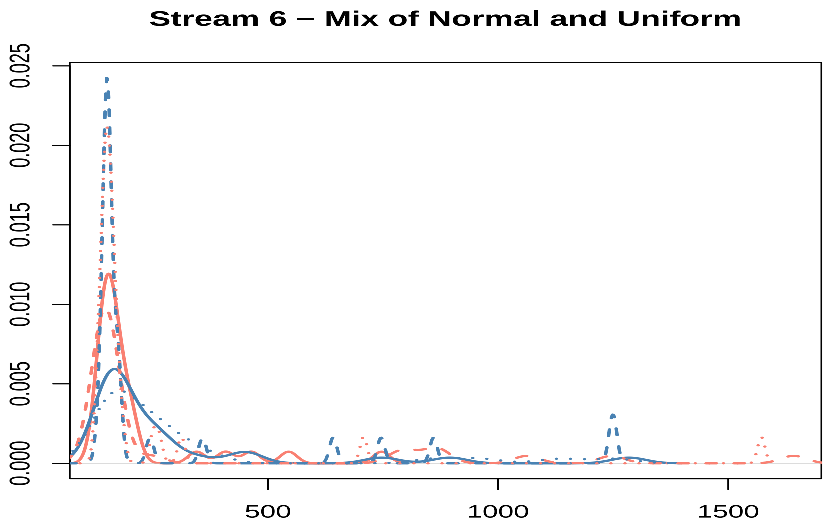


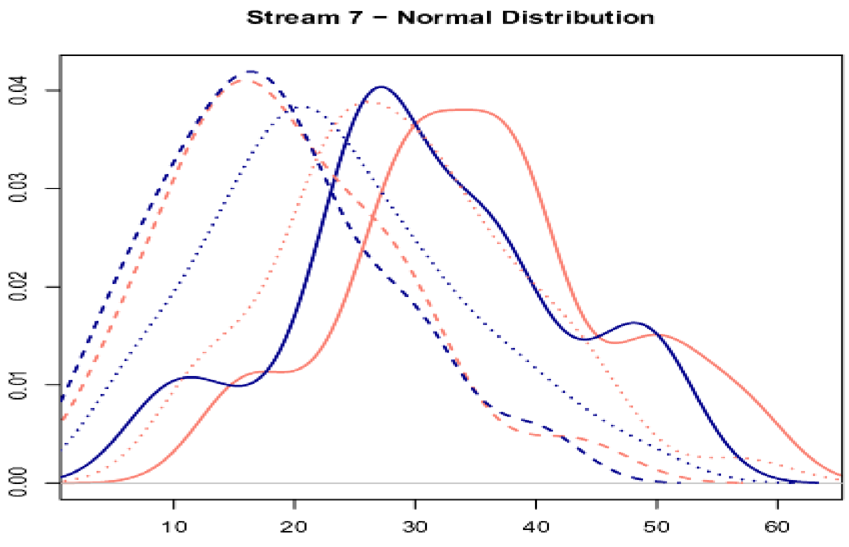
Figure 3.17: Max duration for detecting true changes in $Stream5$ and $Stream6$

The experimental results demonstrate that XI test can detect more true changes at the cost of a higher false rate (Figures 3.3-3.8). KD test detects fewer true changes than XI test for all six types of streams. However, the number of false changes detected is also lower than XI test.

Different window moving methods also affect the performance of the change detection techniques. The moving window method detects less true changes and less false changes than the fixed window method. The proposed merged window method has comparative performance with the fixed window method in detecting true changes. For $Stream1$, $Stream2$ and $Stream6$, the number of true changes detected using both window moving methods are very close. Merged window method outperforms fixed window approach in the test cases of $Stream3$ and $Stream4$ (Figure 3.4 and 3.7), whereas fixed window approach performs better for stream $Stream5$ (Figure 3.5 and 3.8). However, the proposed merged window method usually generates less false alarms than the fixed window method. The exceptions are the test cases in stream $Stream3$ and $Stream4$ when merged window method is applied on XI test (Figure 3.7).

The change duration analysis (Figures 3.9-3.17) shows that KD test takes longer time to detect changes than XI test. Fixed window method has the longest overall change detection duration than moving window method and the proposed merged window approach. Fixed window method may detect some distribution changes very late, e.g., the change may be detected after 1000 data since the change occurs. The moving window method and merged window method can both detect most true changes quickly, with moving window approach slightly better.

To evaluate the impact of change type over the performance of all tested change detection techniques, the second set of experiments is conducted using $Stream7, ..., Stream12$ using the same parameters as the first set. Same as the previous set of experiments, the results are demonstrated by five sets of figures, with the x-axis being the values of the five criteria and y-axis being their density. The number of true changes detected after 100 run for each stream type is shown in Figures 3.18, 3.19 and 3.20. The number of false detections is shown in Figures 3.21, 3.22 and 3.23. Figures 3.24 - 3.32 illustrate the analysis on the durations of true change detections.



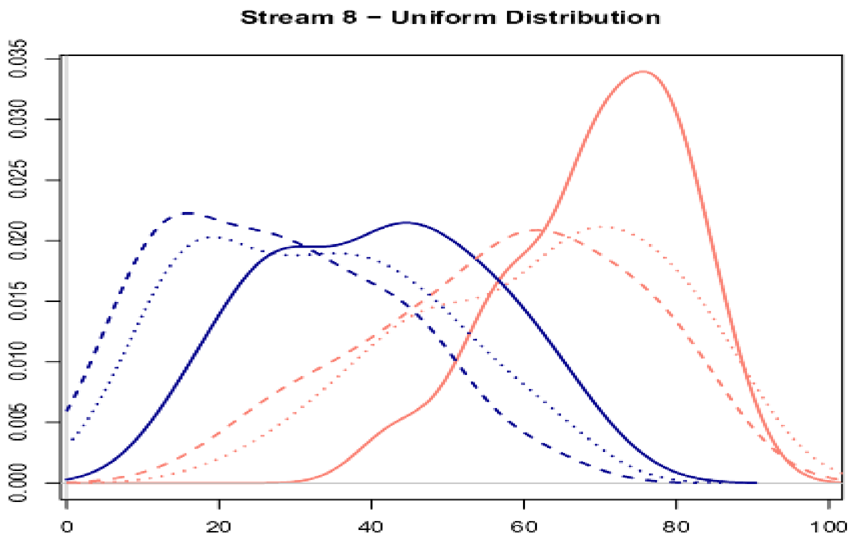
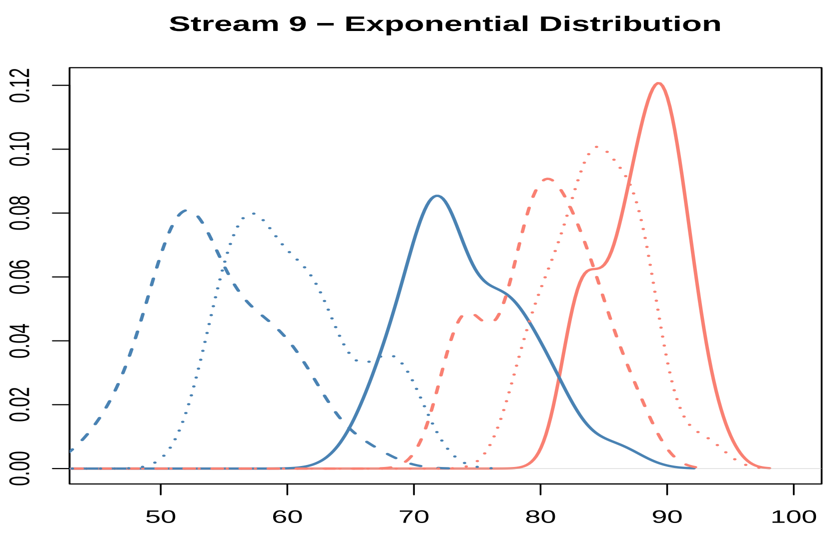


Figure 3.18: Number of true changes detected for $Stream7$ and $Stream8$



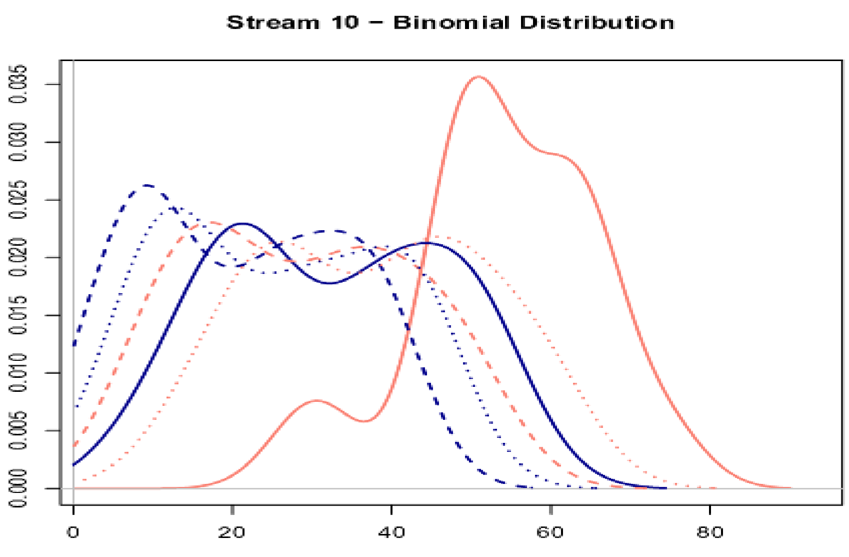
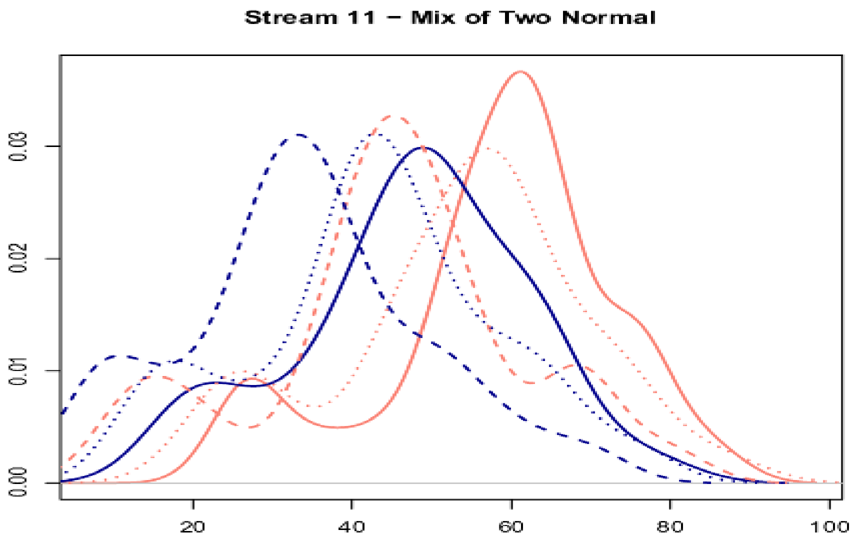


Figure 3.19: Number of true changes detected for $Stream9$ and $Stream10$



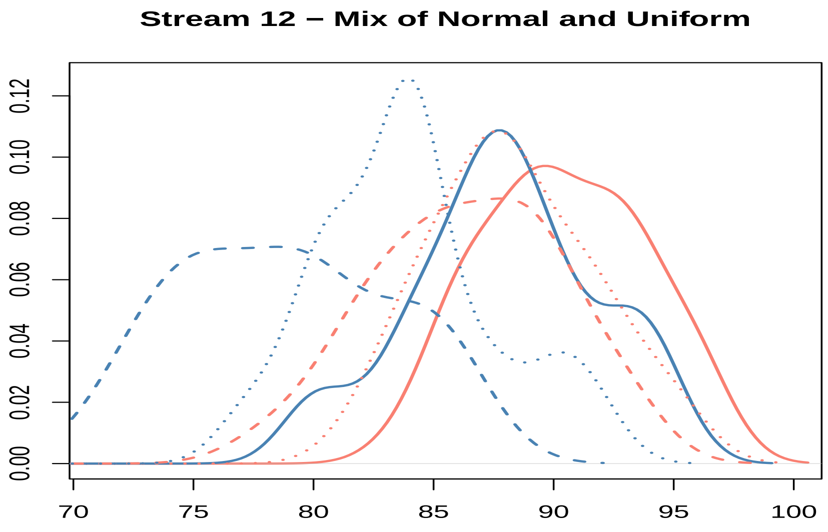
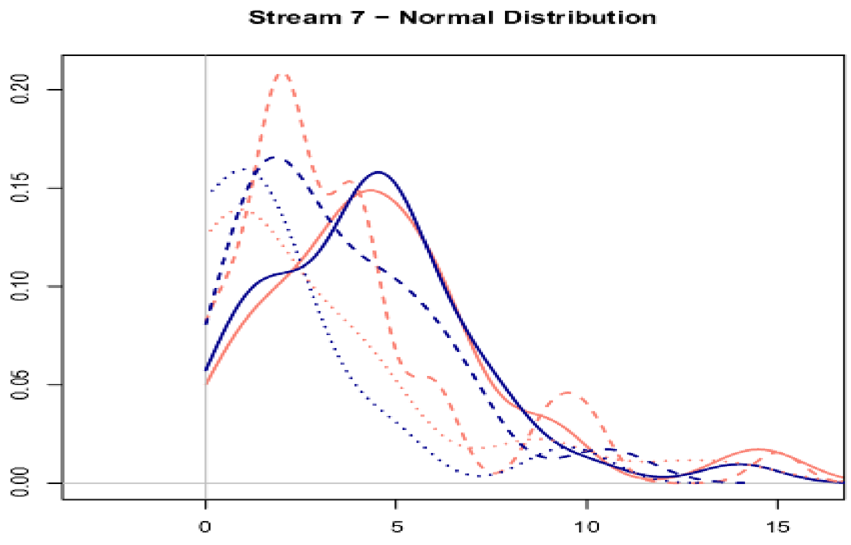


Figure 3.20: Number of true changes detected for $Stream11$ and $Stream12$



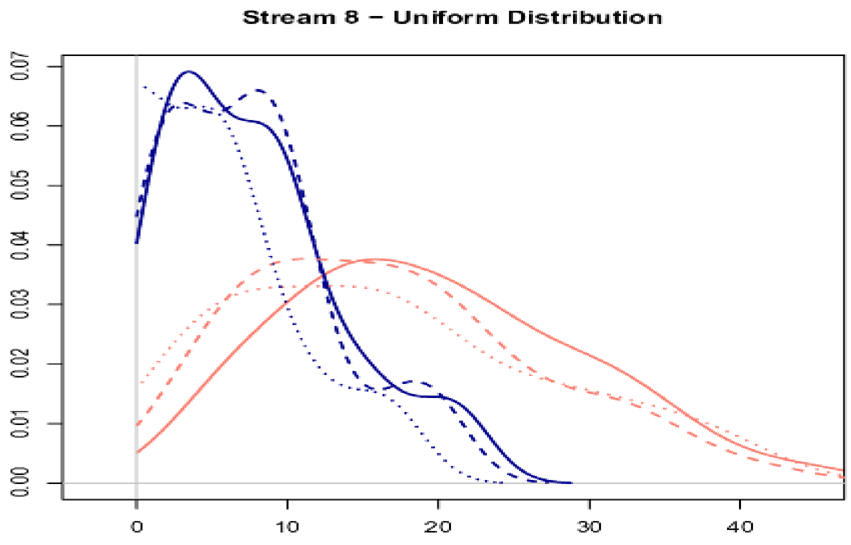
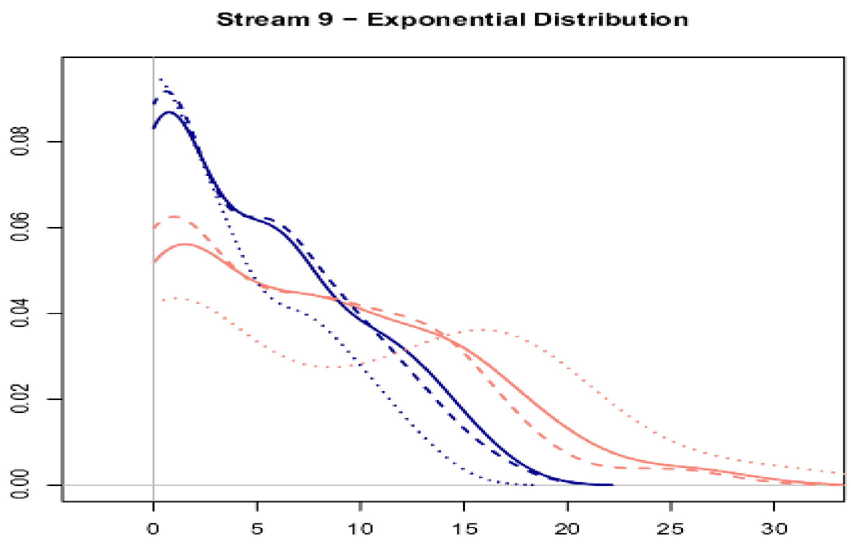


Figure 3.21: Number of false changes detected for *Stream*7 and *Stream*8



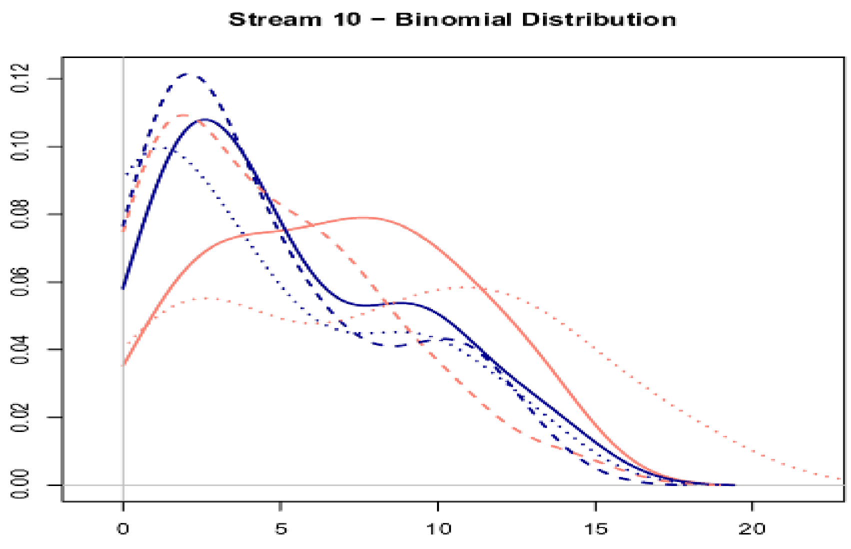
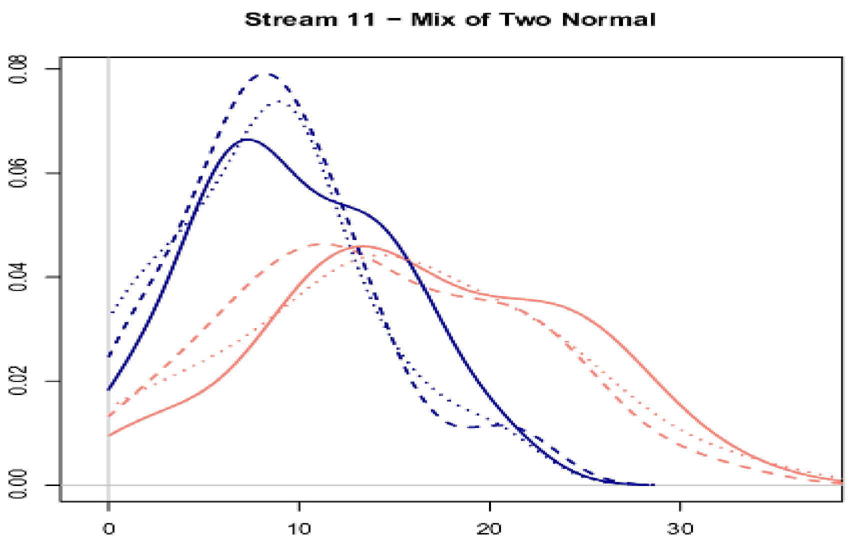


Figure 3.22: Number of false changes detected for $Stream9$ and $Stream10$



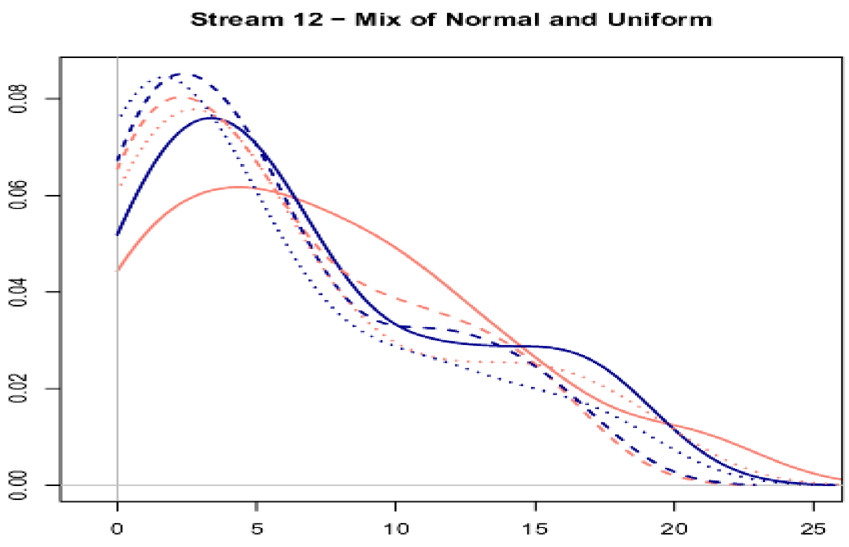
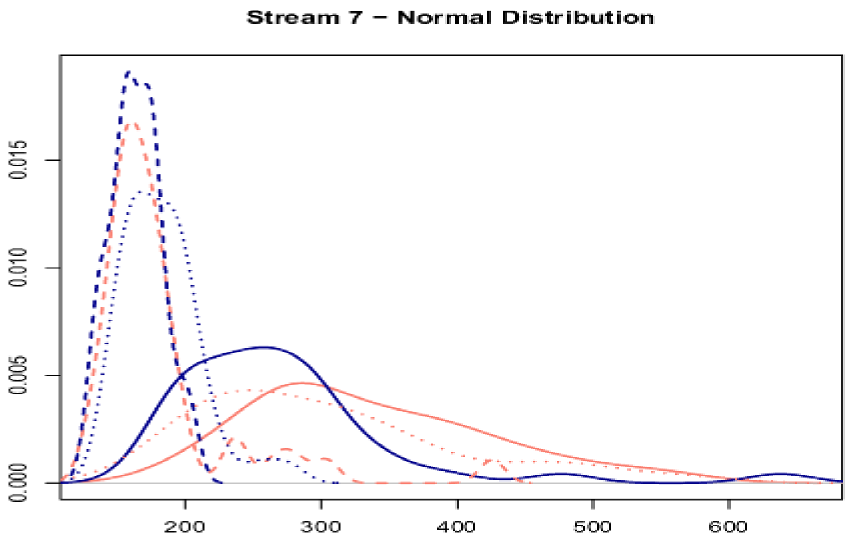


Figure 3.23: Number of false changes detected for $Stream11$ and $Stream12$



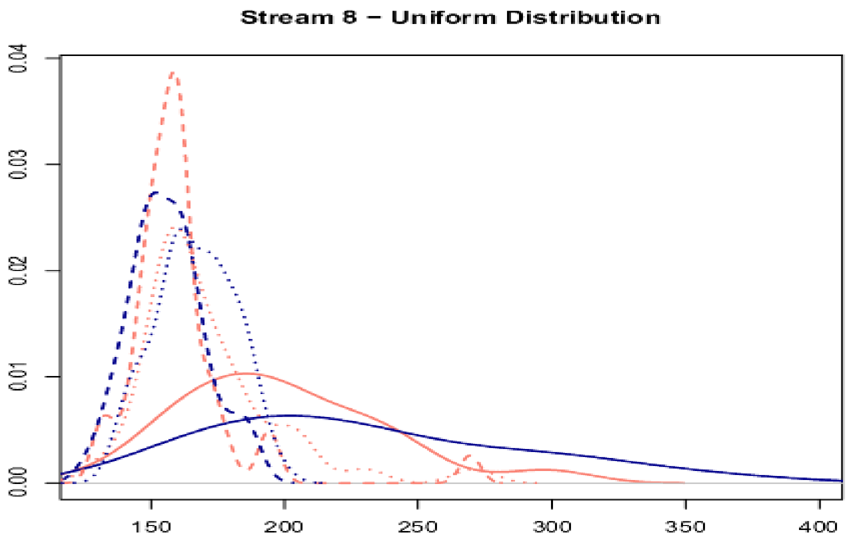
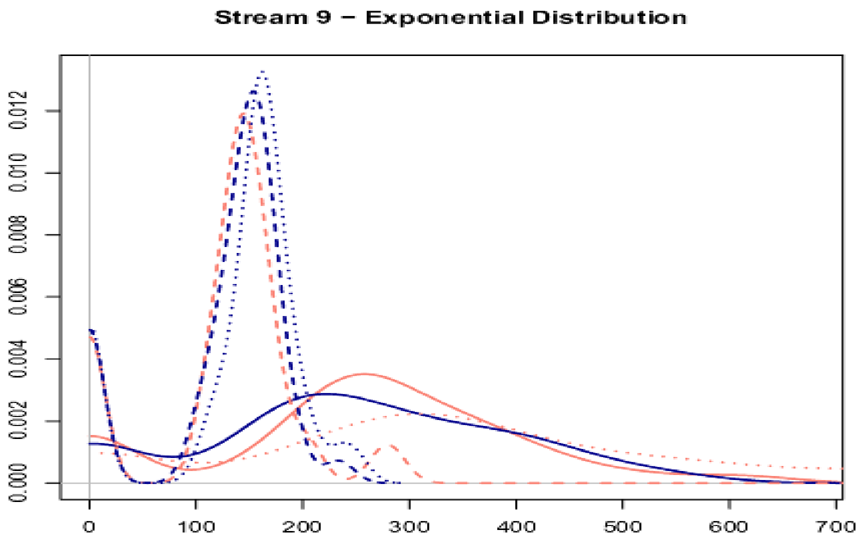


Figure 3.24: Mean duration for detecting true changes in $Stream7$ and $Stream8$



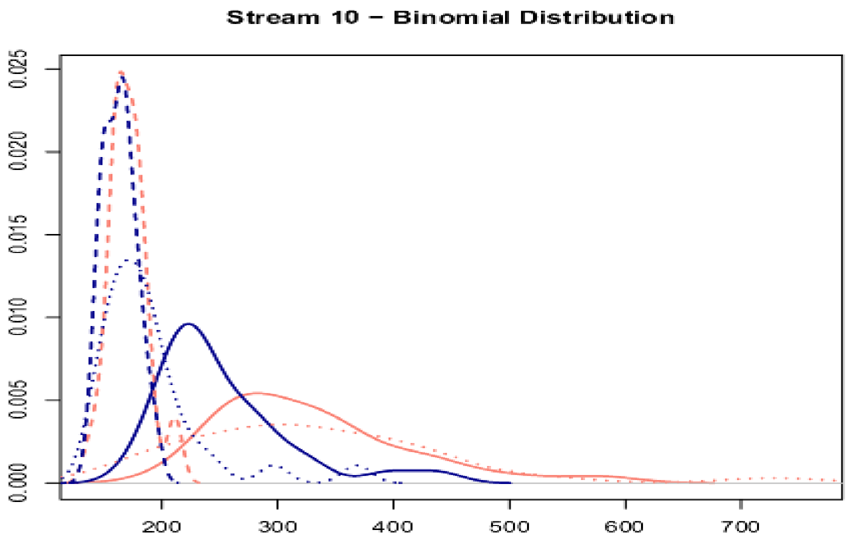
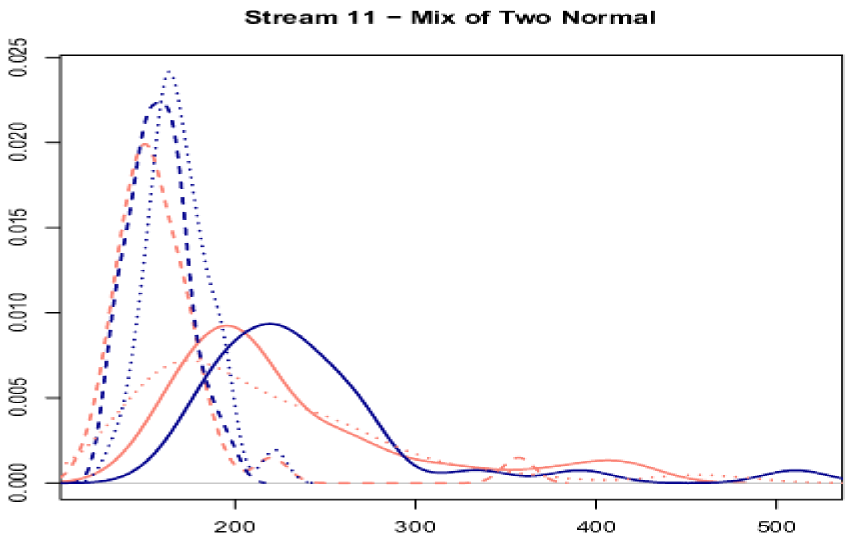


Figure 3.25: Mean duration for detecting true changes in *Stream*9 and *Stream*10



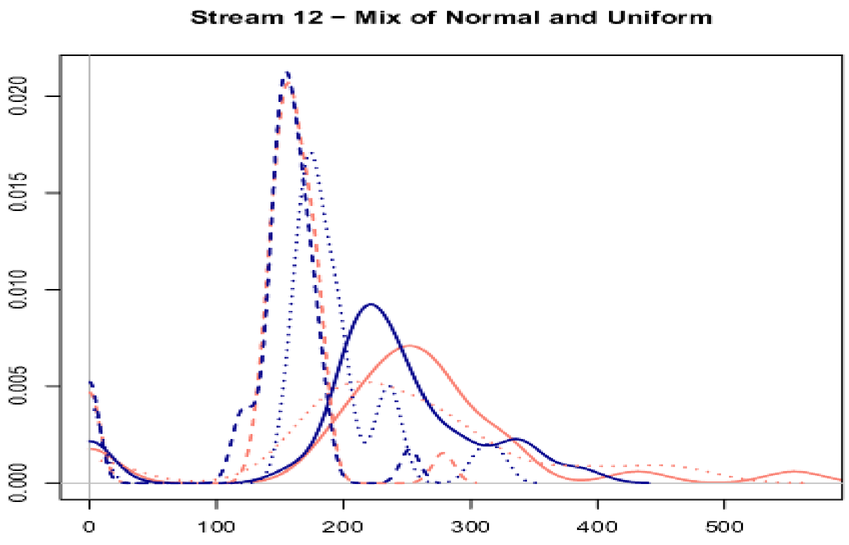


Figure 3.26: Mean duration for detecting true changes in *Stream*11 and *Stream*12

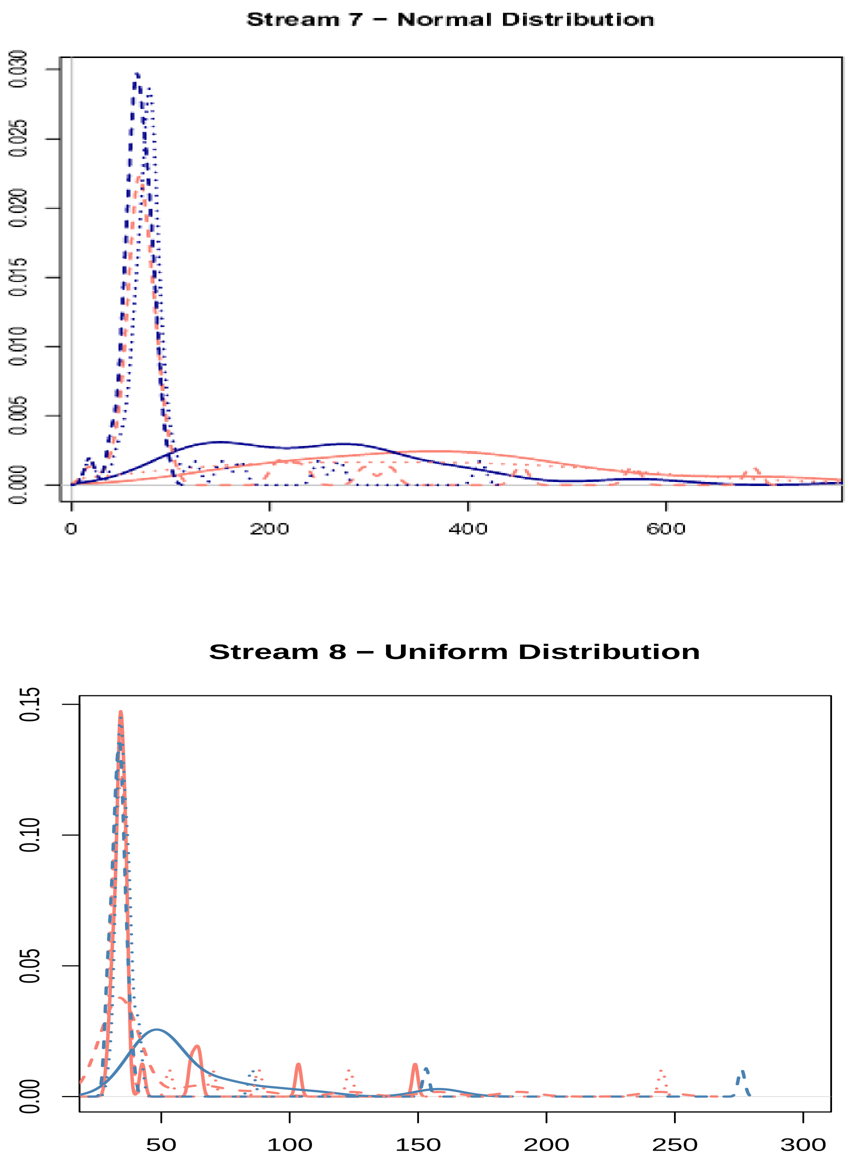
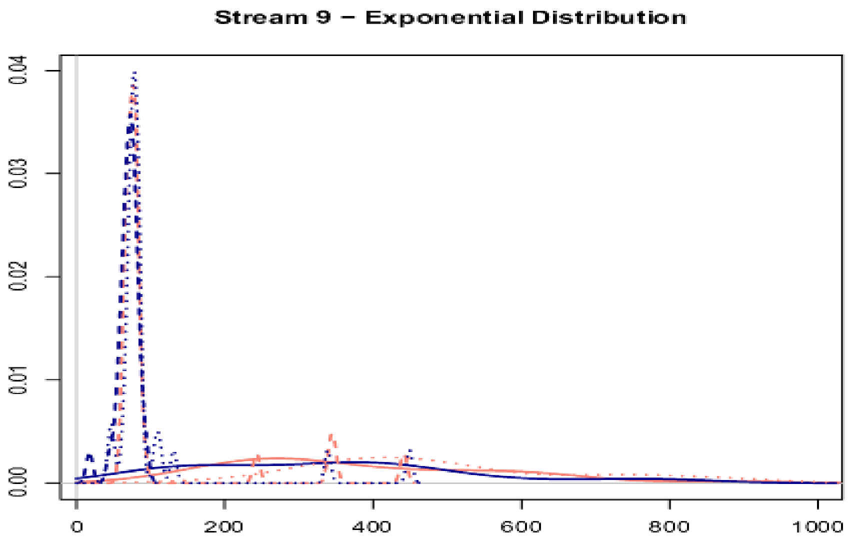


Figure 3.27: Standard deviation of the duration for detecting true changes in *Stream*7 and *Stream*8



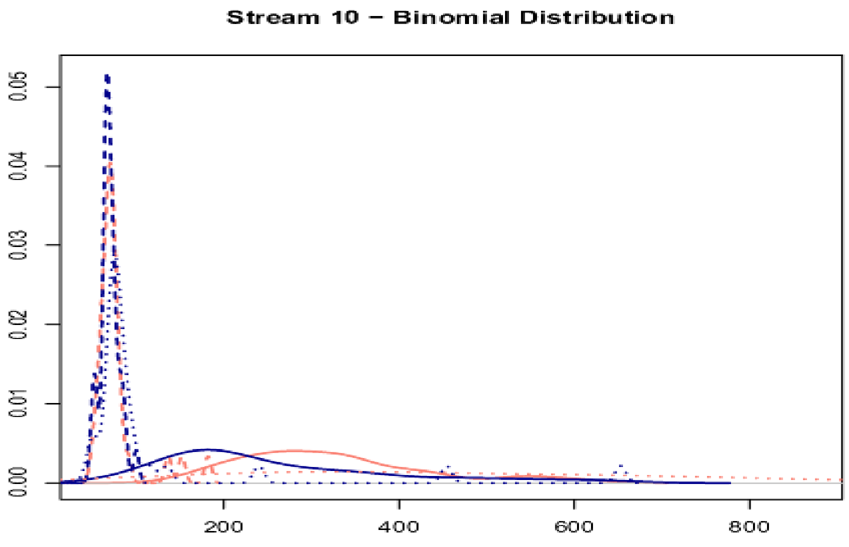
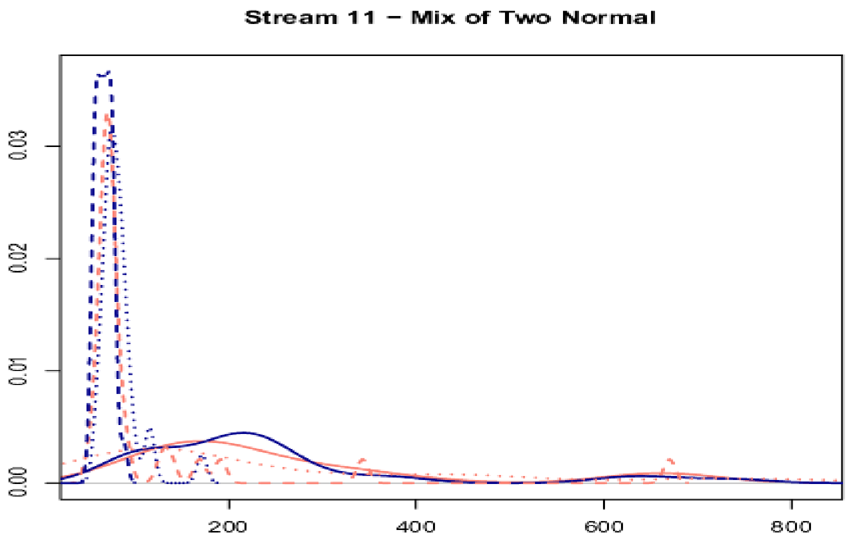


Figure 3.28: Standard deviation of the duration for detecting true changes in $Stream9$ and $Stream10$



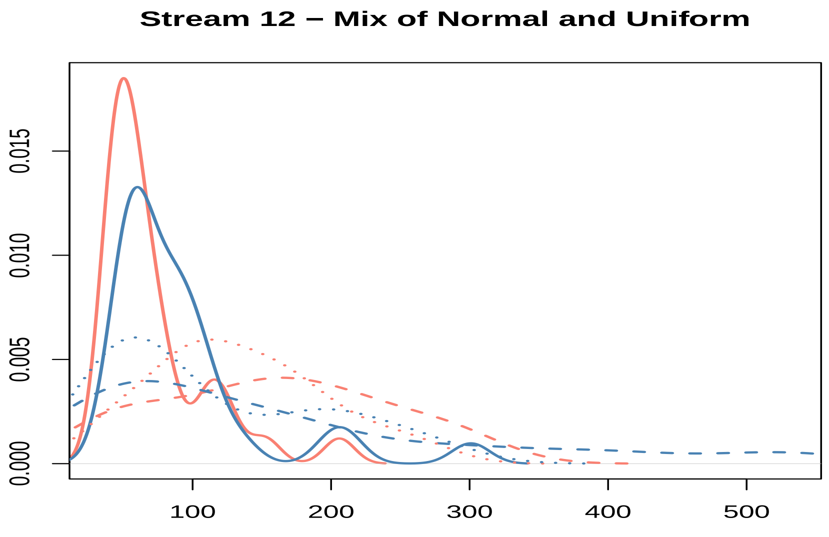
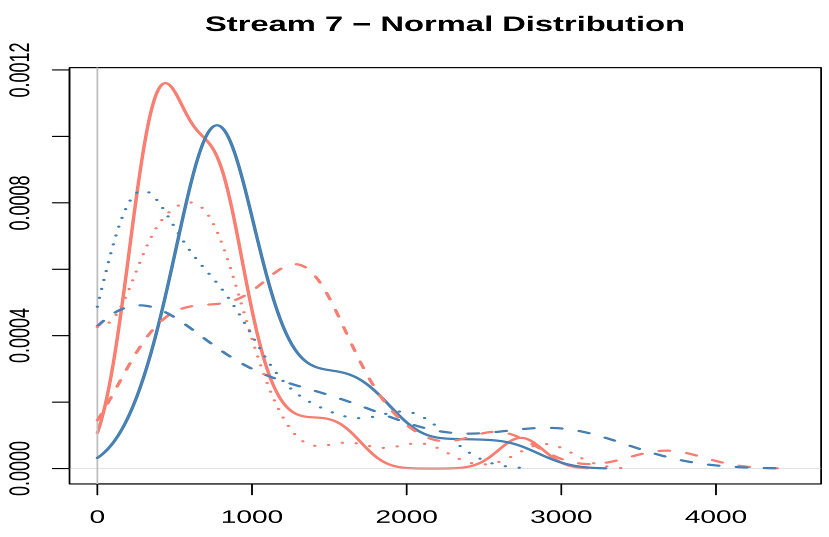


Figure 3.29: Standard deviation of the duration for detecting true changes in $Stream11$ and $Stream12$



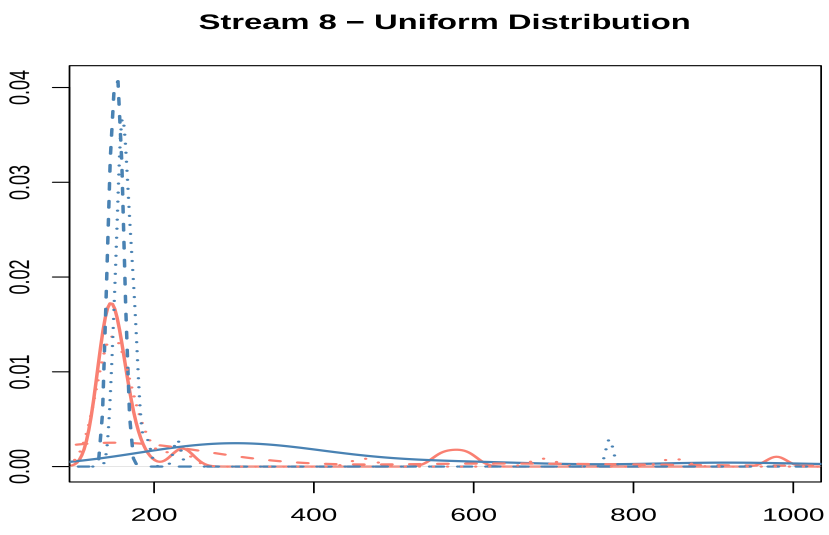
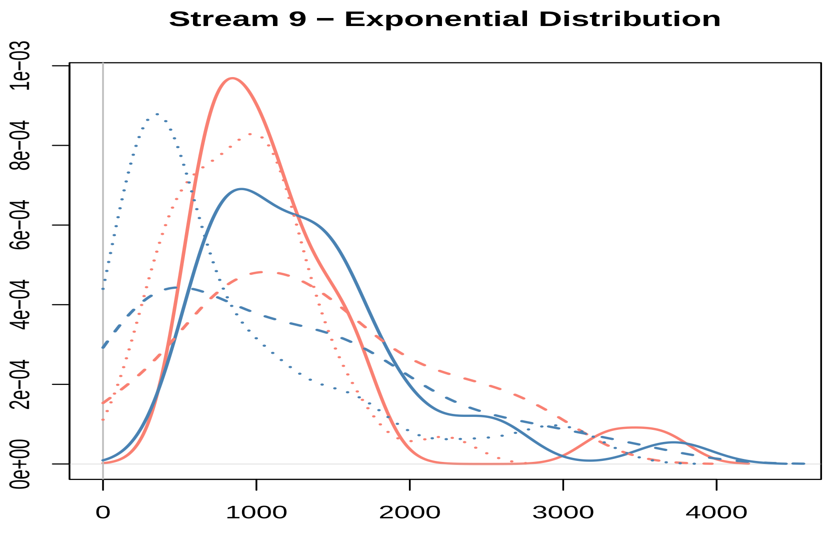


Figure 3.30: Max duration for detecting true changes in *Stream*7 and *Stream*8



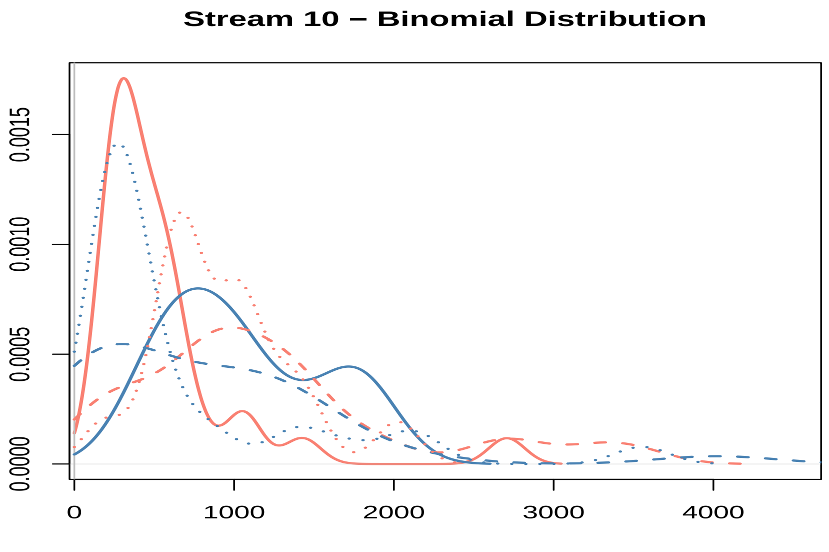
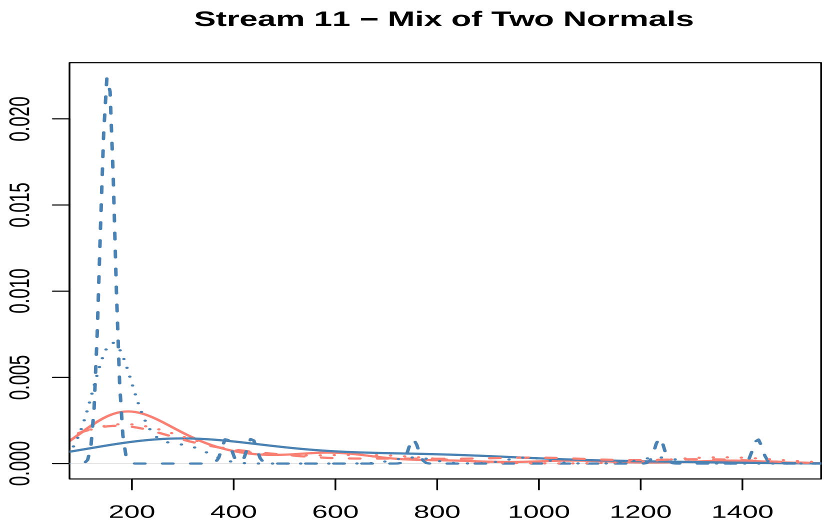


Figure 3.31: Max duration for detecting true changes in $Stream9$ and $Stream10$



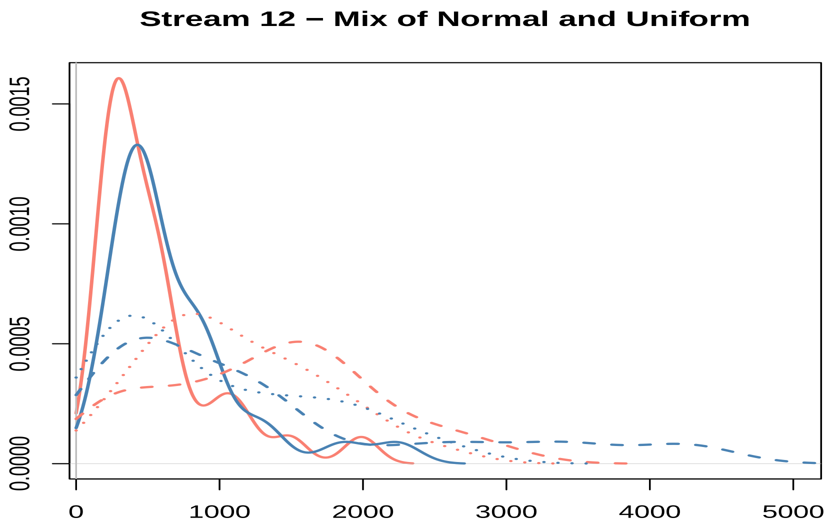


Figure 3.32: Max duration for detecting true changes in *Stream*11 and *Stream*12

The results of the second set of experiments do not show significant difference than the results from the first set of experiments. The observations and conclusions made based on the first set of results are still valid. The second set of results demonstrate that the performance of two statistical tests and three window moving methods tested in the experiments are not greatly impacted by the change type of the stream. This is because all the methods in the experiments are generic approaches that do not make any assumption on the type of distribution or the type of changes in the underlying stream.

From the results of change detection in $Stream1, ..., Stream12$, it can be noticed that XI test is more "aggressive" than KD test. Its performance has higher recall with shorter response time to the changes but also has lower precision. KD test is more "conservative" that generates less errors by sacrificing the number of true detections and takes longer time to confirm a distribution change truly occurs.

One possible explanation of the performance difference between KD and XI tests is that, KD test "smoothes" the distribution of a data set by replacing each data element with the kernel. Therefore, for two data sets generated by different but similar distributions, the difference in density estimation may not be significant after smoothing, especially if the two distributions are of the same type, e.g., both are normal distributions. By reducing the bandwidth, KD may detect more true changes but the number of false changes may also increase. In contrast, even for two data sets generated by the same distribution, it is possible that the XI distance between the two sets is not small. Hence, XI test is very sensitive to the changes in data sets. However, if we compare the rate of true detections over the total number of detections that represent the accuracy of the change detection technique, XI test may not be superior than KD test. XI test simply reports more changes and, hence, even with the same accuracy as KD test, XI test will detect more true changes.

The proposed merged window approach outperforms the moving window approach with more true changes detected, less number of false detections and comparable true change detection durations. The performance of merged window method is slightly worse than fixed window method in terms of true detections. However, it also generates fewer false alarms than fixed window method. Fixed window method is slower in detecting true changes and in some cases may take very long time to report a distribution change. Hence, it may not be suitable for some applications that fast responses to distribution changes are required.

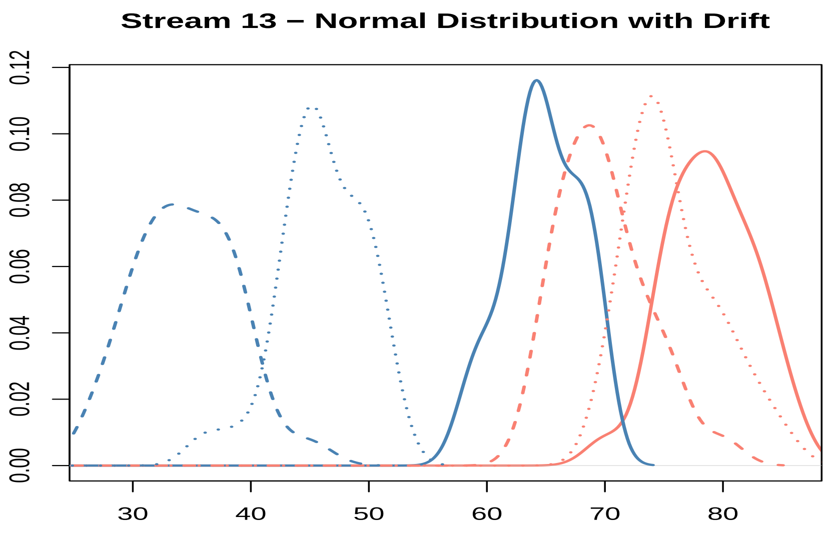
Fixed window method chooses the first data set of the distribution to be its representative set. However, as discussed in Section 3.3.1, this representative set may not truly represent the entire distribution. It is also possible that some data in the representative set may belong to the previous distribution. This could be the reason why fixed window method generate the most number of false alarms among all three window moving methods. Although it also produces the most number of true detections, the accuracy, i.e., the total number of true detections over the total number of changes detected, may be similar or even worse than the other two windows moving methods.

Moving reference window method compares two data sets that arrive subsequently, and, hence, the distribution changes must be abrupt and significant to be detected. Therefore, many true changes are missed when using moving reference window method. However, such significant and abrupt can be detected without long delay.

The proposed merged window approach is balanced among the three criteria: the number of true change detections, the accuracy, and the change detection durations. It may not outperform the other two window moving methods in one of the three criteria, but the overall performance is the best among the three window moving methods.

Detecting distribution drifts

The streams $Stream1, ..., Stream12$ tested in previous experiments only contain abrupt changes. As mentioned previously, distribution drifts are usually more difficult to detect because the changes are not significant. To study the performance difference between detecting distribution shifts versus drifts, we conduct experiments using stream types $Stream13, ...,$ $Stream18$ with the same parameter settings. The drift duration $driftDur$ is set to four times of the windows size, i.e., 400 data elements. The experimental results are demonstrated in Figures 3.33 – 3.47.



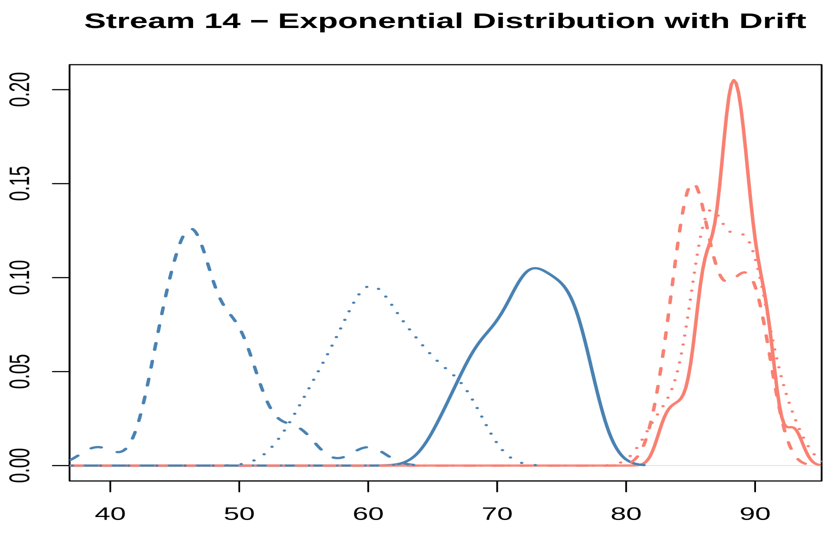
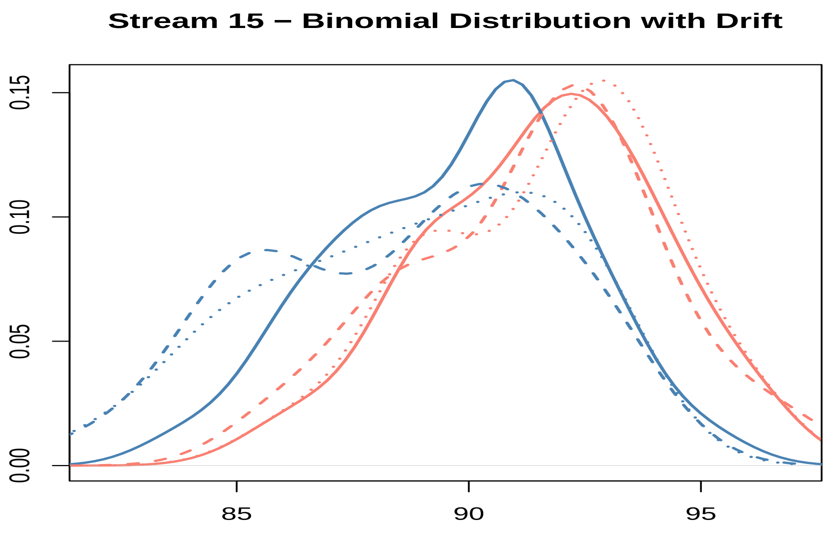


Figure 3.33: Number of true changes detected for $Stream13$ and $Stream14$



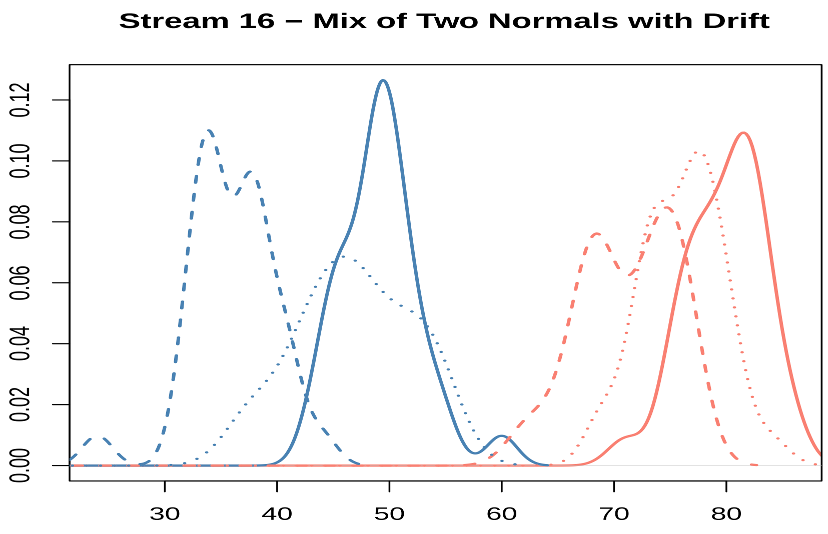
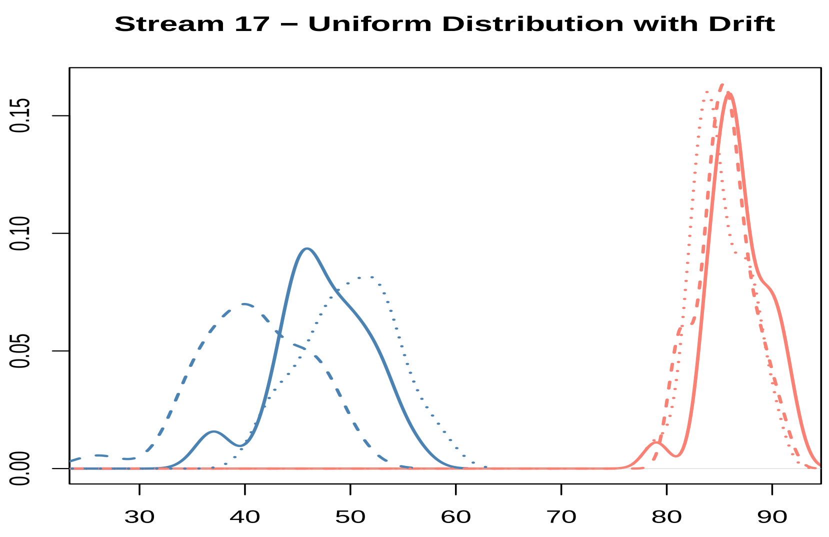


Figure 3.34: Number of true changes detected for $Stream15$ and $Stream16$



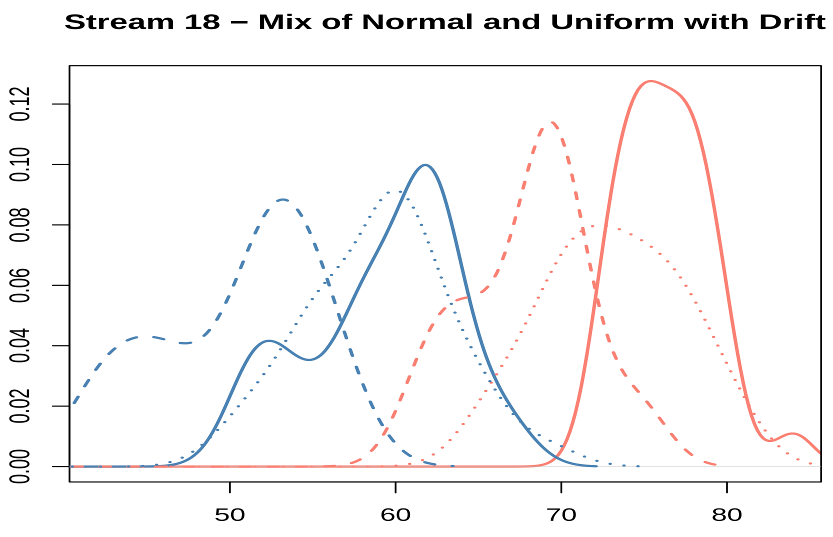
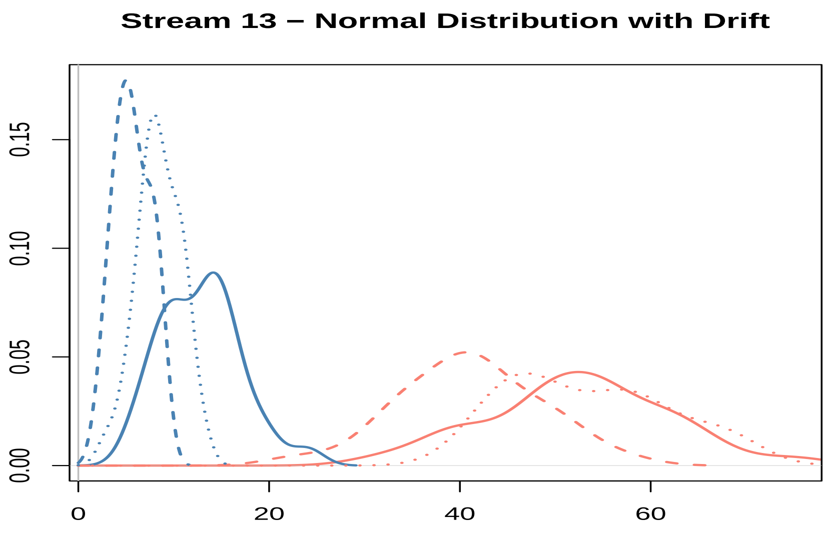


Figure 3.35: Number of true changes detected for $Stream17$ and $Stream18$



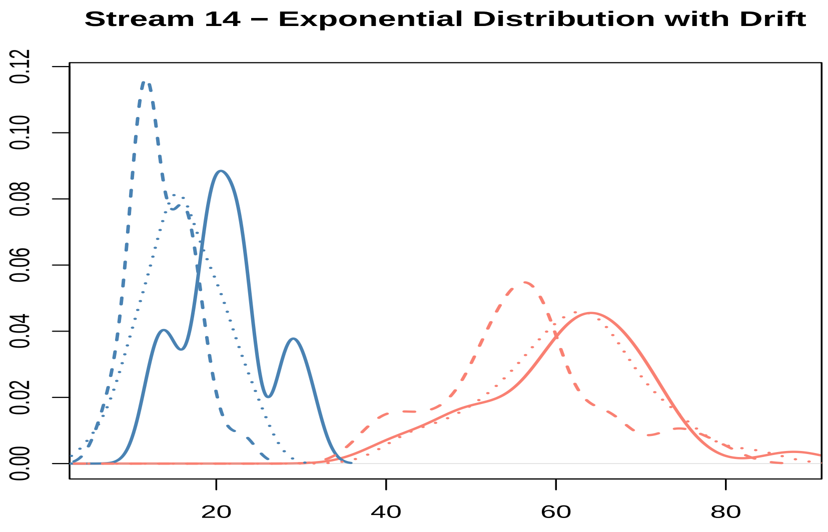
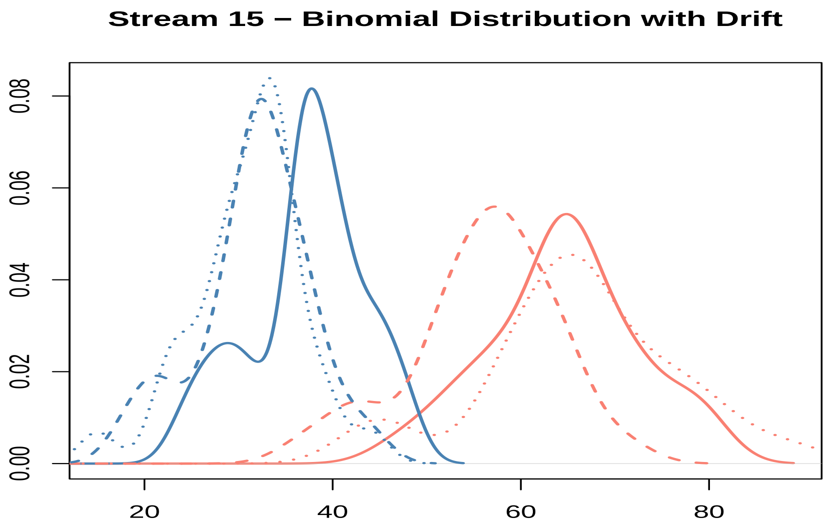


Figure 3.36: Number of false changes detected for $Stream13$ and $Stream14$



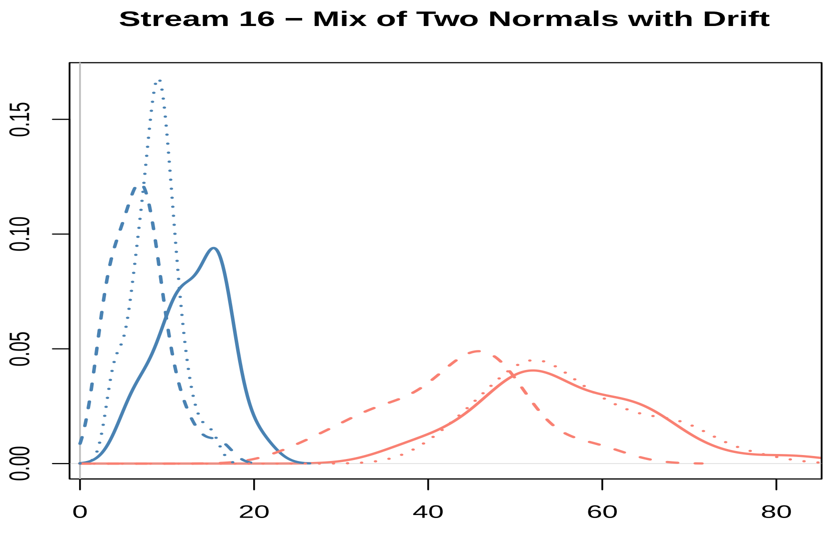
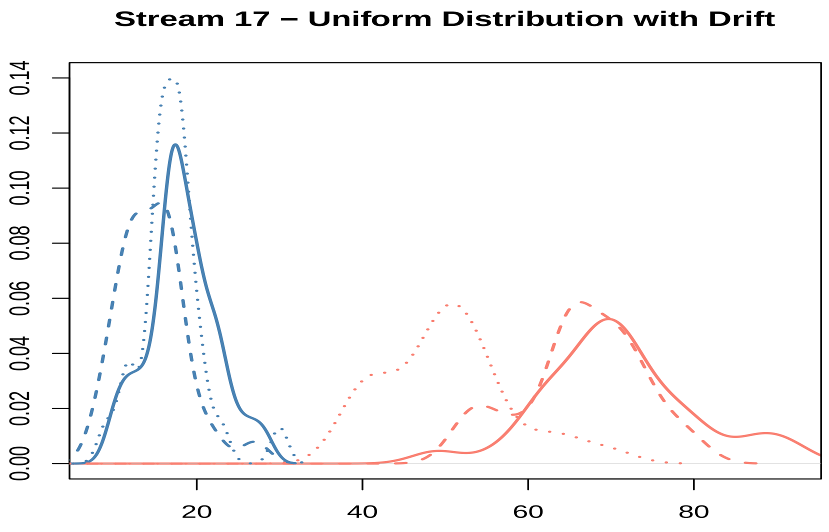


Figure 3.37: Number of false changes detected for $Stream15$ and $Stream16$



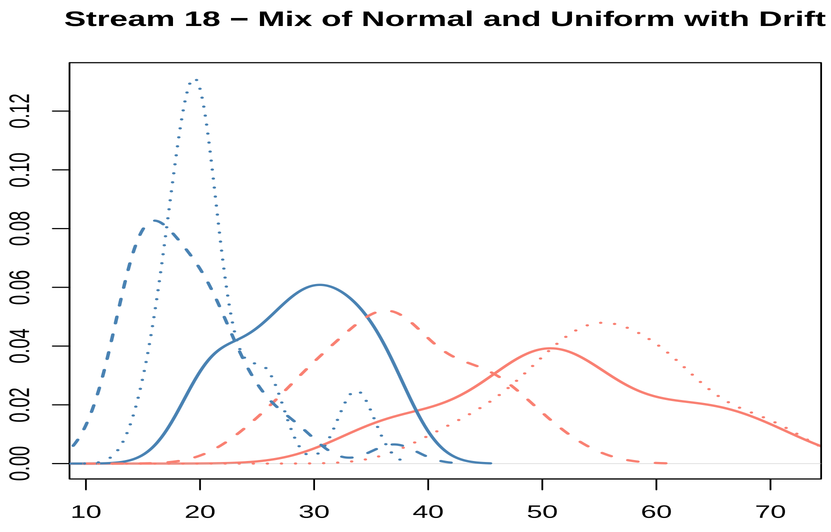
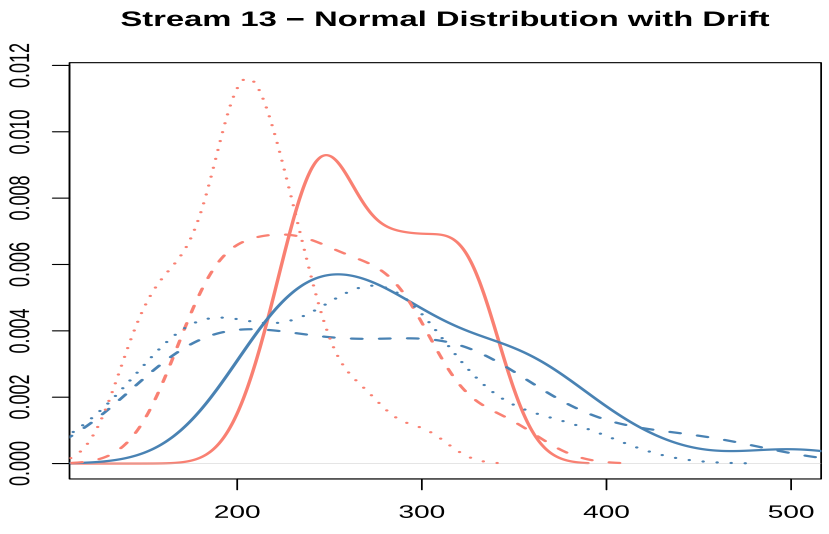


Figure 3.38: Number of false changes detected for $Stream17$ and $Stream18$



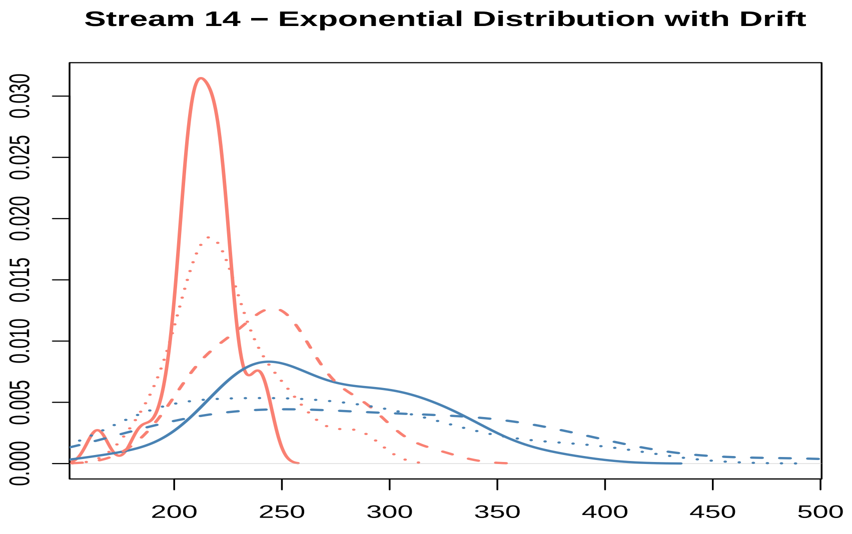


Figure 3.39: Mean duration for detecting true changes in $Stream13$ and $Stream14$

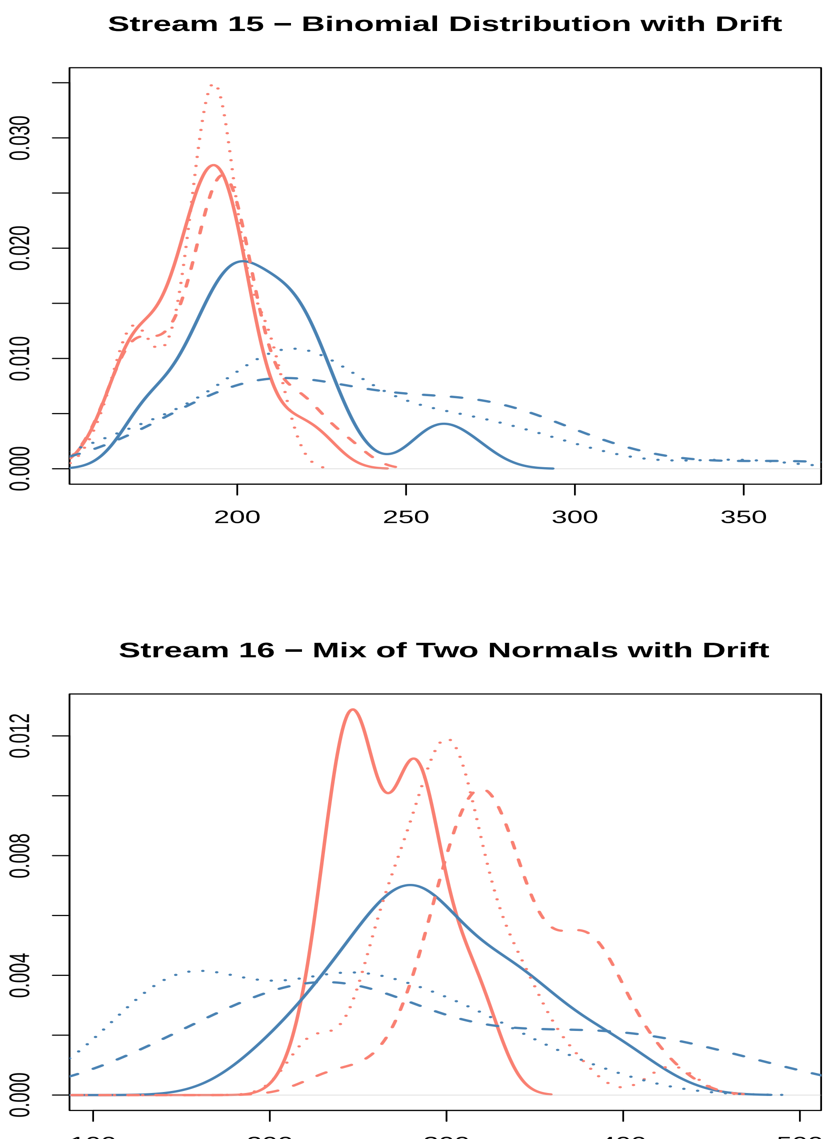
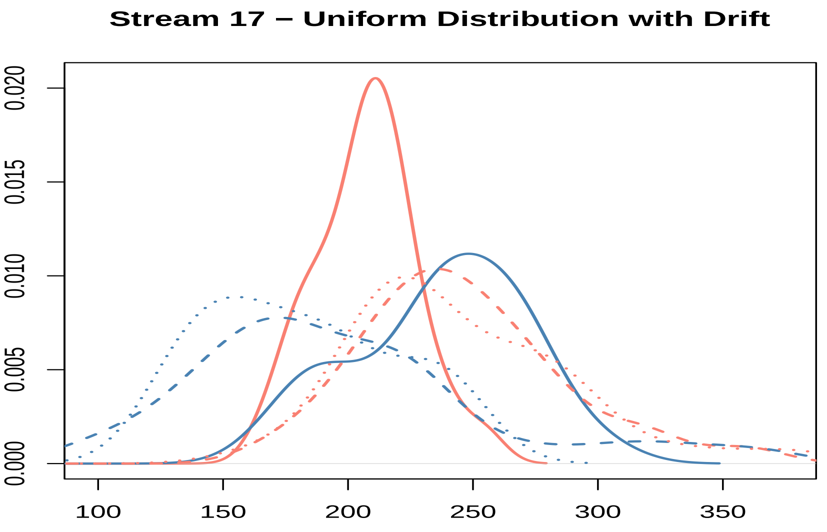


Figure 3.40: Mean duration for detecting true changes in $Stream15$ and $Stream16$



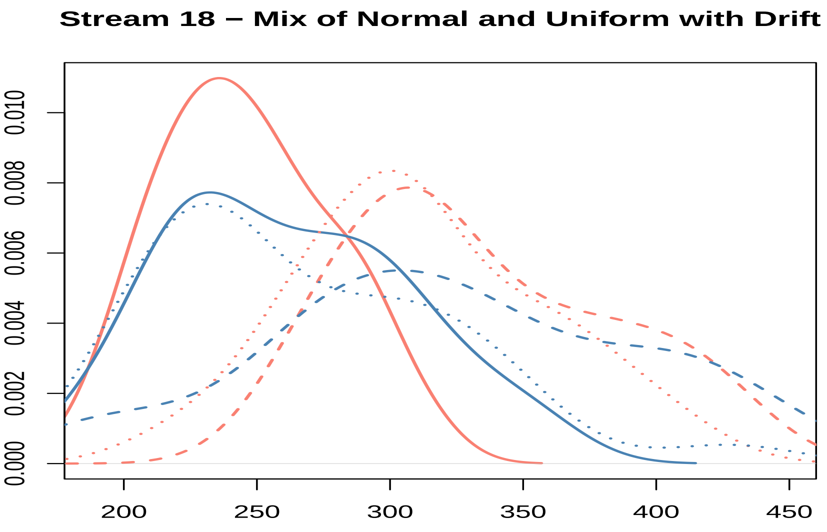
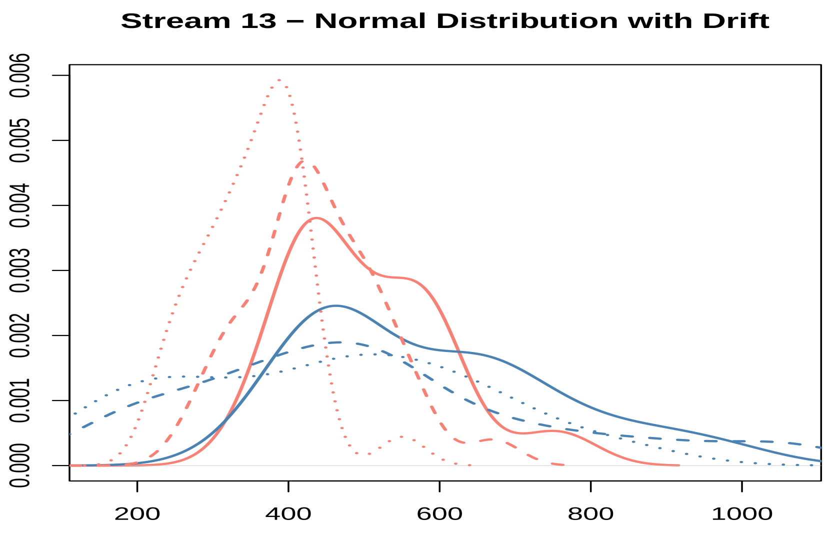


Figure 3.41: Mean duration for detecting true changes in $Stream17$ and $Stream18$



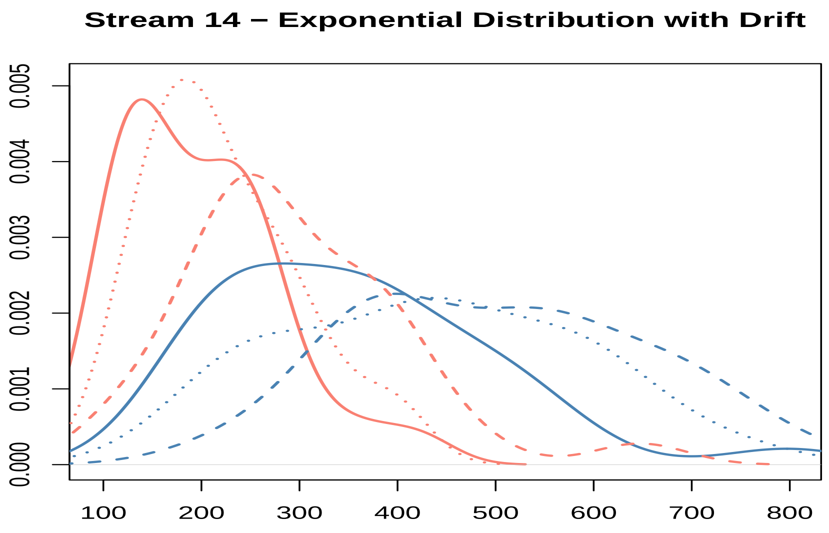
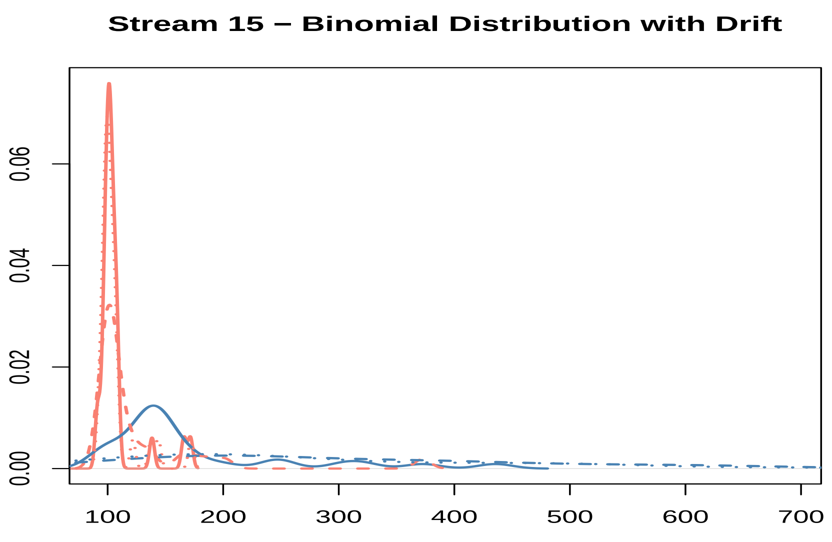


Figure 3.42: Standard deviation of the duration for detecting true changes in $Stream13$ and $Stream14$



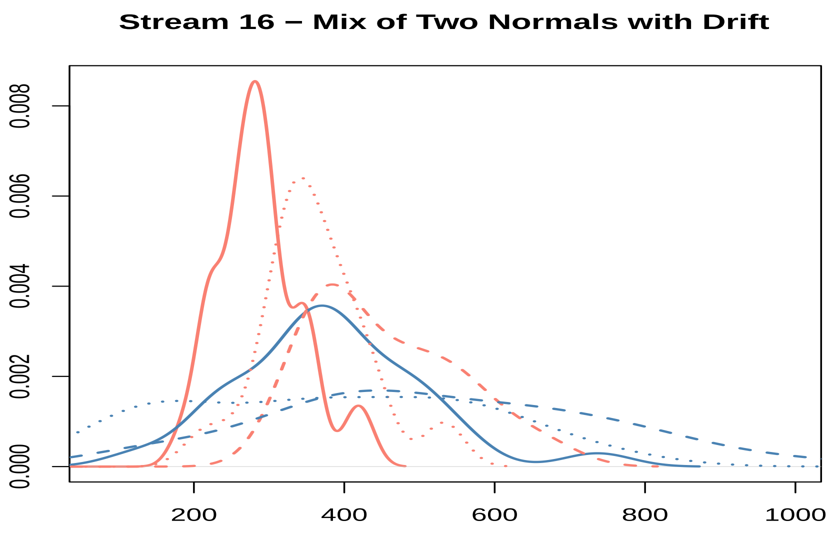
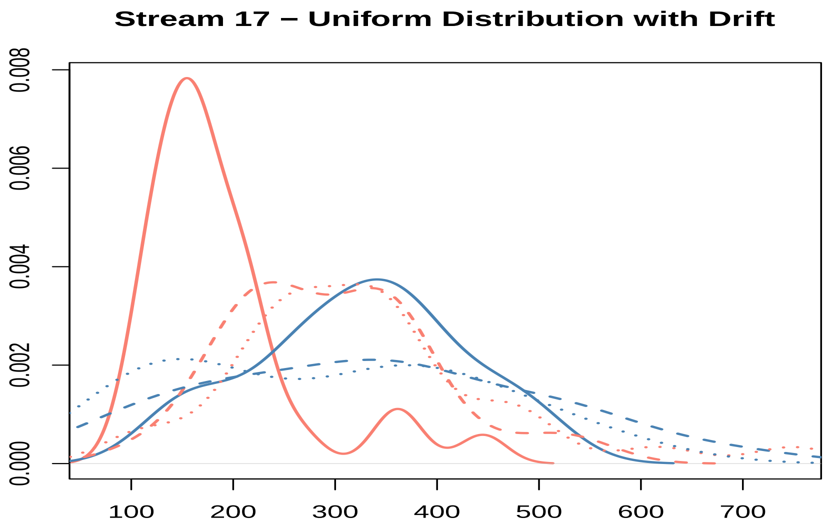


Figure 3.43: Standard deviation of the duration for detecting true changes in $Stream15$ and $Stream16$



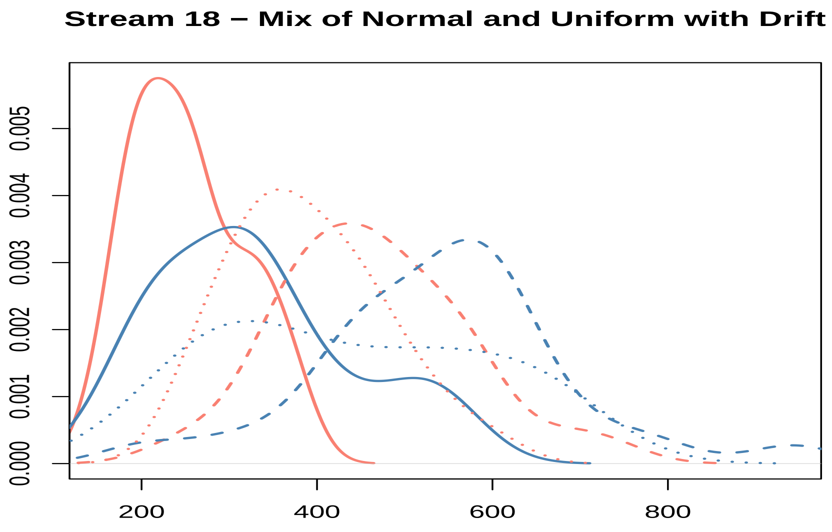
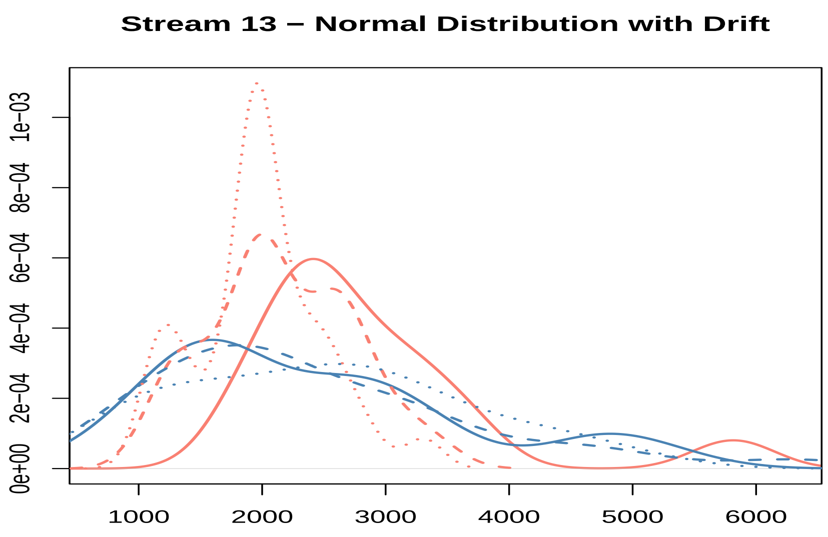


Figure 3.44: Standard deviation of the duration for detecting true changes in $Stream17$ and $Stream18$



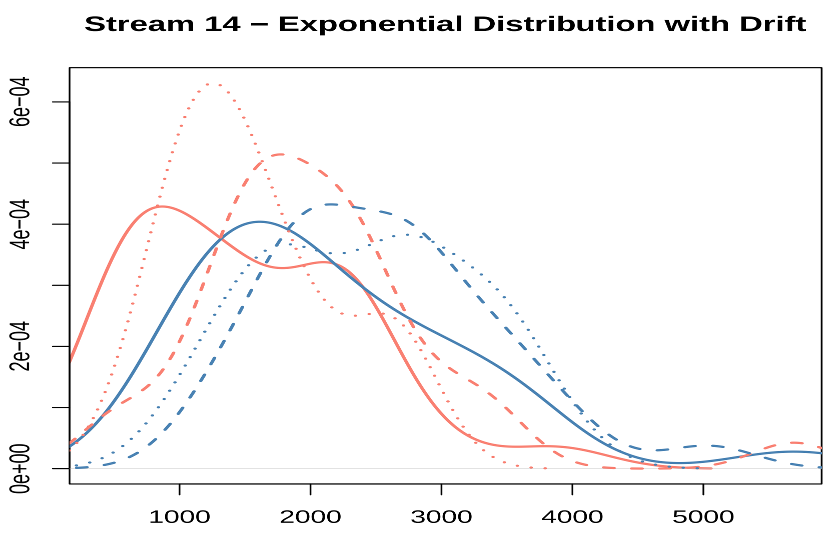
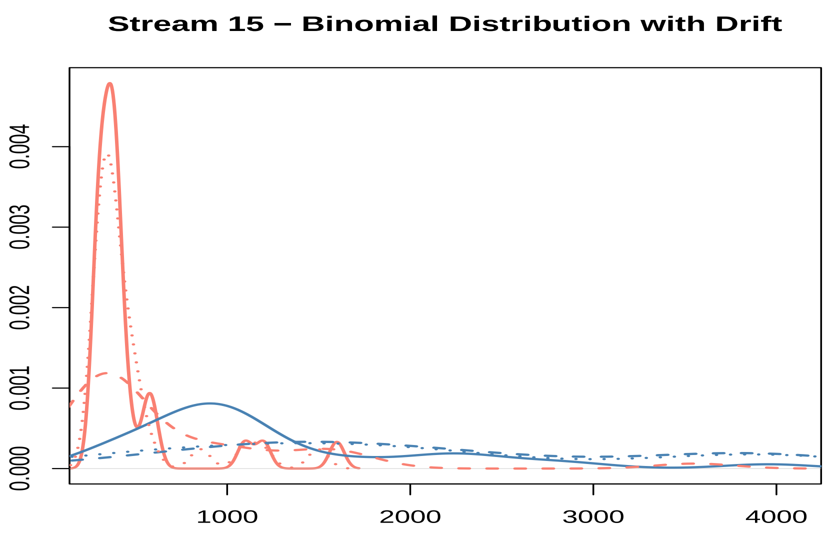


Figure 3.45: Max duration for detecting true changes in $Stream13$ and $Stream14$



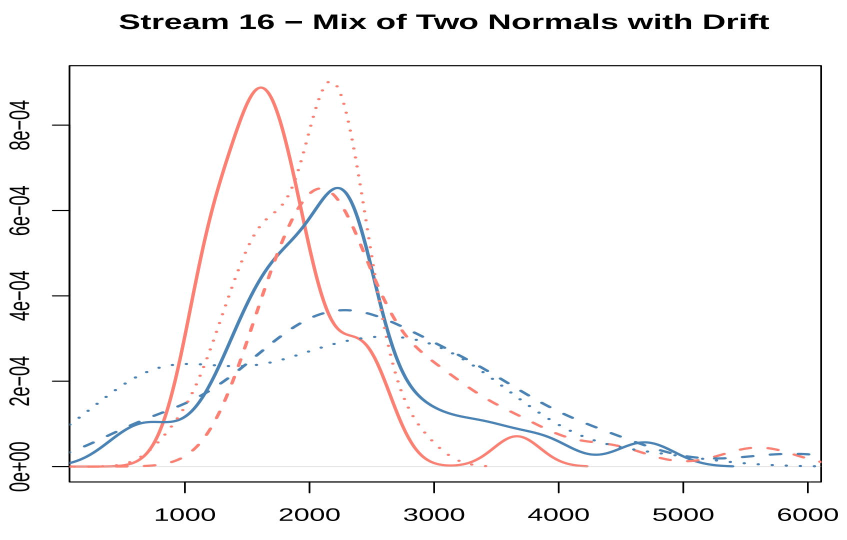
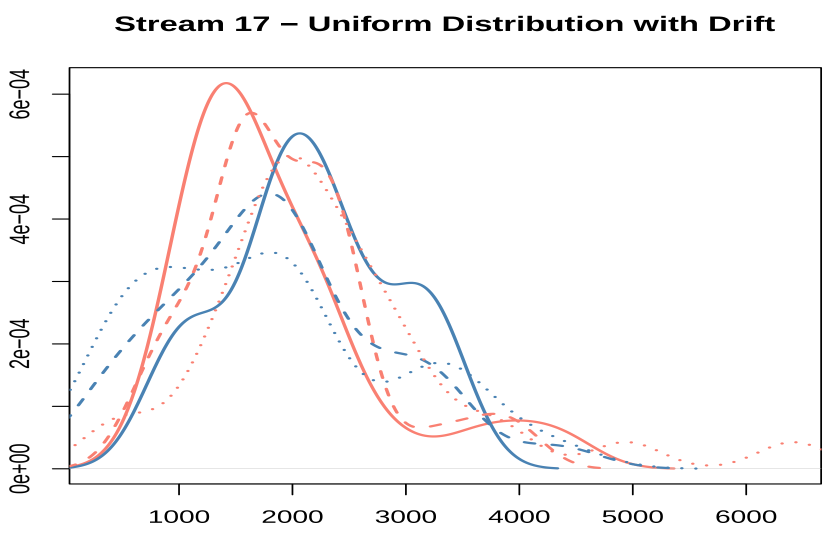


Figure 3.46: Max duration for detecting true changes in $Stream15$ and $Stream16$



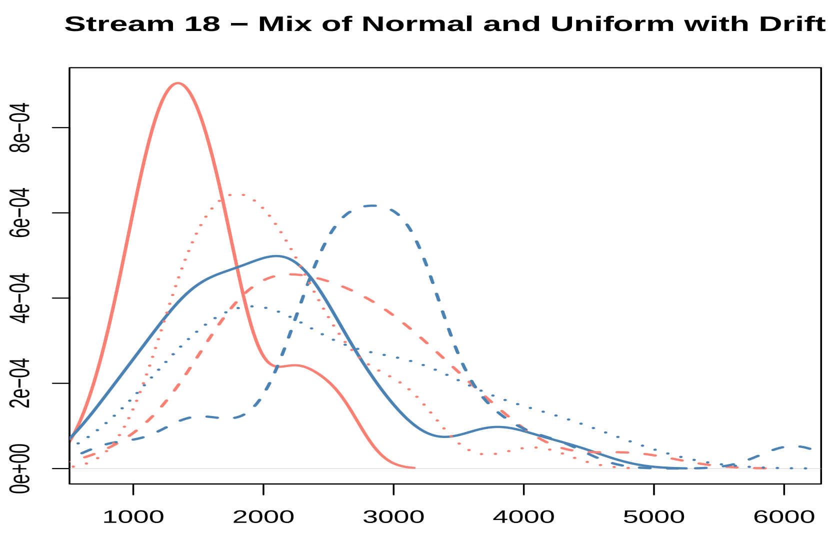


Figure 3.47: Max duration for detecting true changes in *Stream*17 and *Stream*18

The following observations can be made from the experimental results.

- The performances of KD test and XI test are consistent with the previous experiments. KD test still detects less true changes and less false changes than XI test. The durations of true change detections of KD test are still slightly better than XI test with the same window moving method.

- Compared to the stream types with the same distribution type and the same change type but different change speed (e.g., $Stream1$ and $Stream13$), XI test seems to detect slightly more number of true changes but also report more false changes for distribution drifts. This indicates that the XI test tends to be more aggressive for streams with distribution drifts. In contrast, KD test becomes more conservative in reporting true changes and false changes for distribution drifts.

  The explanation to this observation is similar to the one made in previous sets of experiments. That is, the distance function based test is more sensitive to changes than kernel estimation. Therefore, if the observation window contains some data that arrive within drifting period and the rest of the data from the stabilized new distribution, XI test is more likely to report a distribution change. If the data generated by the stabilized new distribution is dominant in the observation window, KD test may not notice the change.

- The window moving methods have more significant impact on the overall change detection performance than pervious experiments on distribution shifts. This is especially noticeable when using KD test. Moving window method generates a lot fewer true change detections than fixed window method and merged window method. Merged window method generates fewer true changes than fixed window method on $Stream13, Stream14$ and $Stream16$. For $Stream15$, $Stream17, Stream18$, the performances of merged window method and fixed window method on number of true changes are comparative.

Since moving window method continuously moves the reference window to the end of the current distribution, when a distribution drift occur, both reference window and observation window contains data set in the drifting period. The discrepancy between S_r and S_t is usually small, and, thus, many distribution drifts cannot be detected when using moving window method.

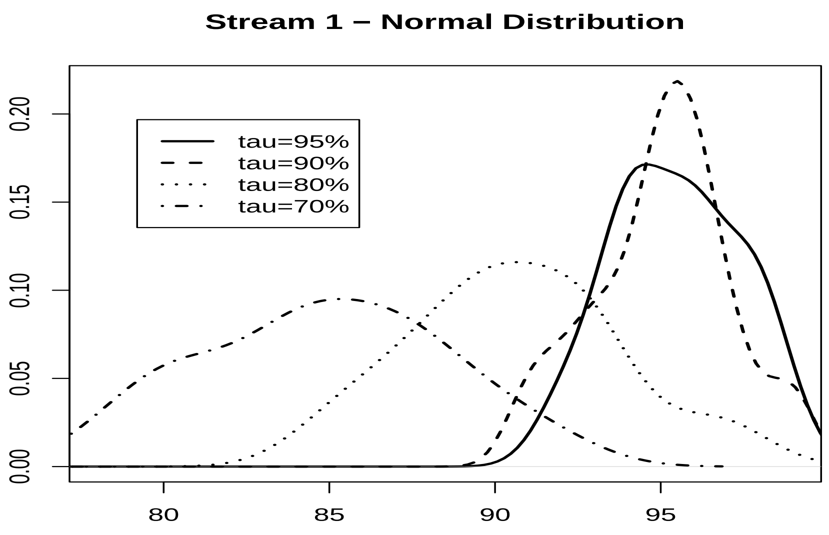
Fixed window method sets the reference window at the beginning of current distribution, and, hence, S_r usually does not contain any data arriving within the drifting period. Therefore, the discrepancy between S_r and S_t is the most significant among three window moving methods. However, note that although fixed window method performs best on detecting true changes, it also returns the most number of false changes.

- All six change detection techniques take a lot longer to detect drifts than detecting shifts. This observation confirms our conclusions that distribution drifts usually cannot be detected as fast as distribution shifts because the change is gradual.
- Fixed window method takes the longest time among three window moving methods to detect shifts. However, in detecting drifts it is usually the fastest. Moving window method performs worst on true change detection durations. The reason is the same as discussed above. $Discrepancy(S_r, S_t)$ is the smallest for moving window method and the largest for fixed window method. Therefore, fixed window method can detect changes fast, whereas moving window method may take very long time until the discrepancy is significant.

Effect of significance level τ

Significance level τ defines the concept of "distribution change". Two distributions P_A and P_B are considered same only when their similarity is greater or equal to τ. Hence, a larger τ value indicates that a small discrepancy between the distributions P_r and P_t of S_r and S_t may be considered as a distribution change. In contrast, a change detection technique with smaller τ setting only report a distribution change when $Discrepancy(P_r, P_t)$ is large.

To study the effect of τ, a set of experiments is conducted using the proposed experimental framework with τ value set at $70\%, 80\%, 90\%$ and 95%. The rest of parameter settings are the same as previous experiments. According to the experimental results, the impact of τ is consistent for different change detection techniques on various stream types. Hence, we only demonstrate the results for XI-fixed, KD-moving and XI-merged change detection techniques described in Table 3.2 on $Stream1$ and $Stream17$ described in Table 3.1. These empirical results are shown in Figures 3.48 – 3.62.



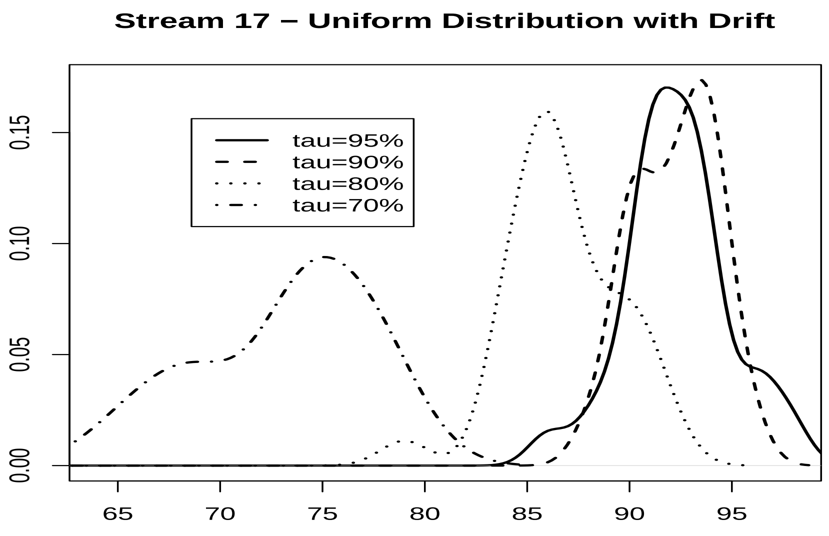
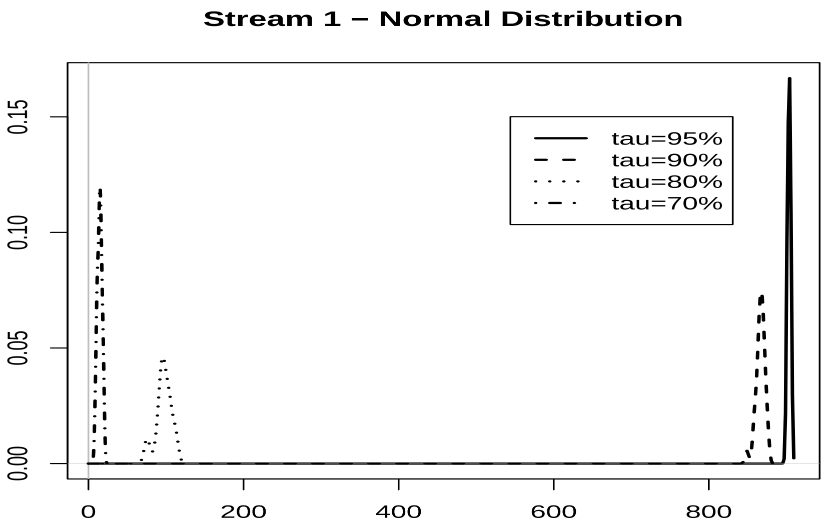


Figure 3.48: Number of true changes detected for $Stream1$ and $Stream17$ using XI-fixed with different τ values



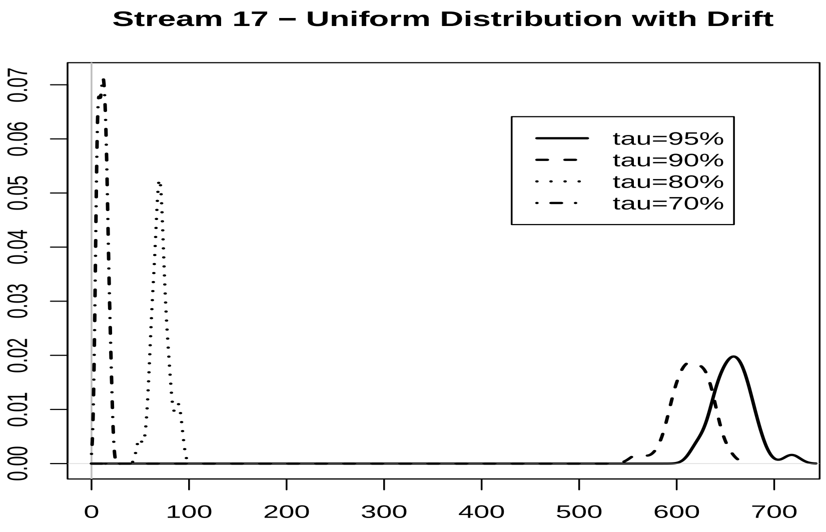
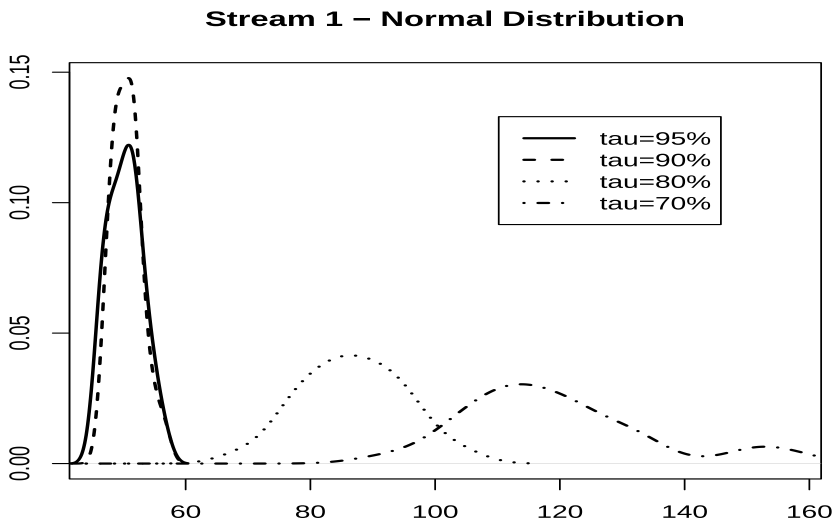


Figure 3.49: Number of false changes detected for $Stream1$ and $Stream17$ using XI-fixed with different τ values



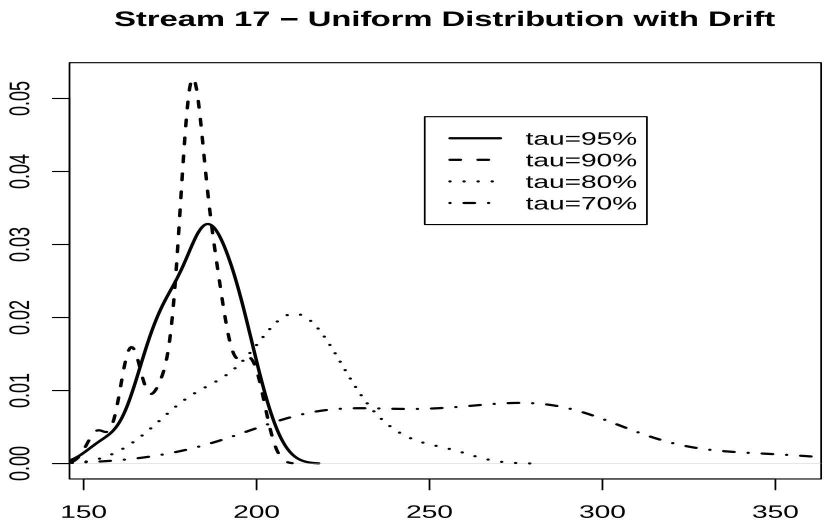
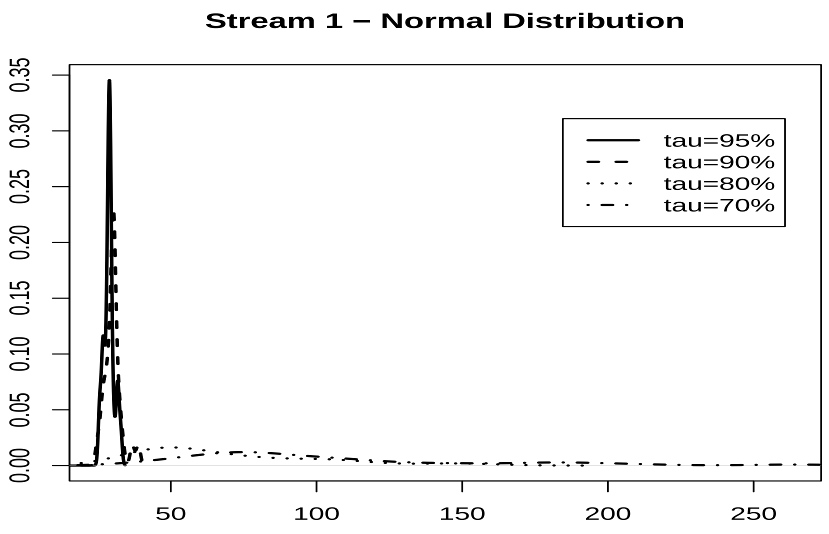


Figure 3.50: Mean duration for detecting true changes in $Stream1$ and $Stream17$ using XI-fixed with different τ values



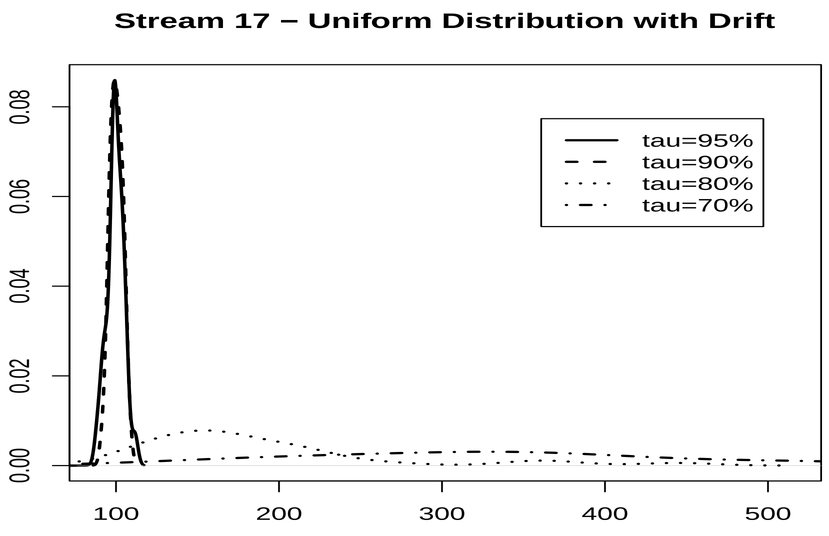
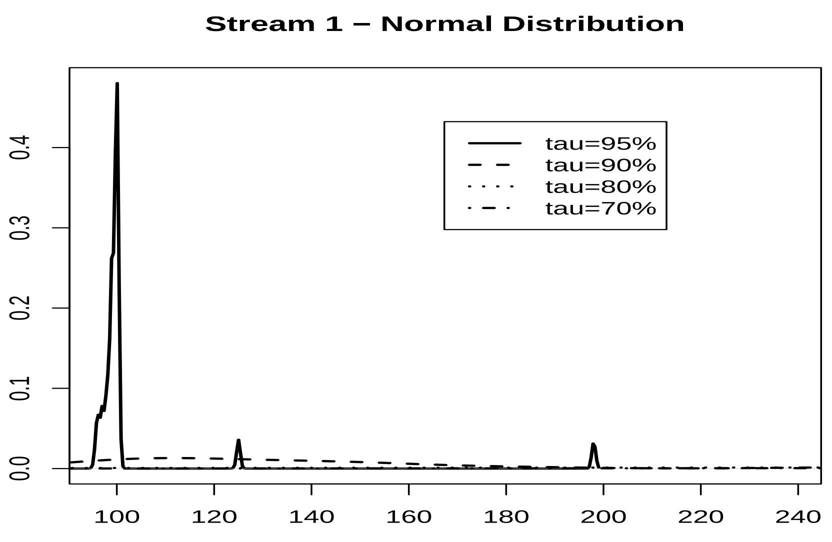


Figure 3.51: Standard deviation of the duration for detecting true changes in *Stream*1 and *Stream*17 using XI-fixed with different τ values



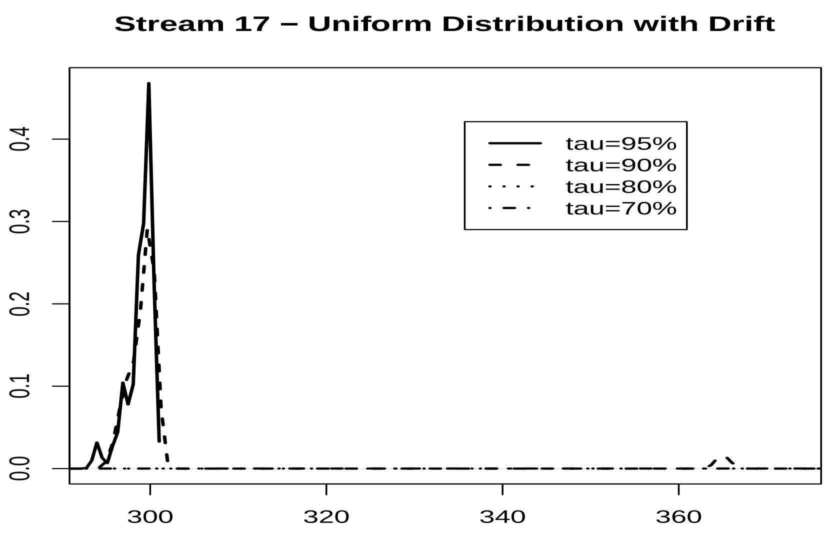
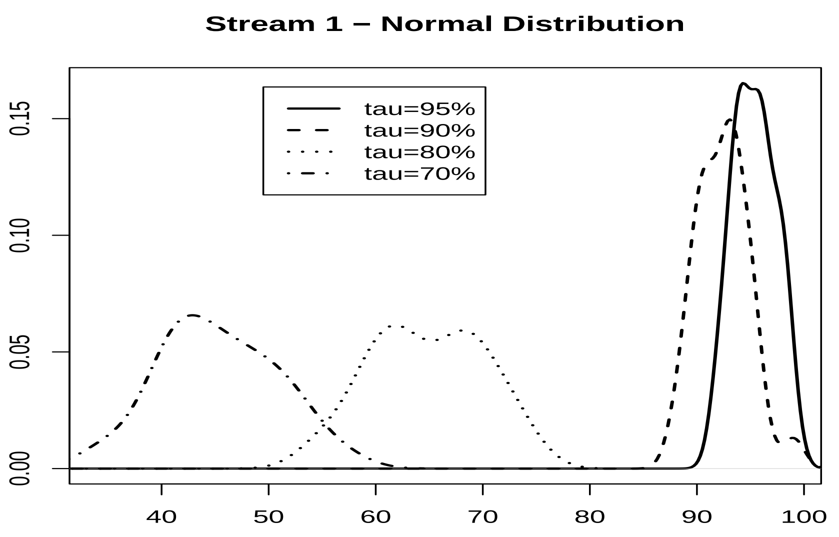


Figure 3.52: Max duration for detecting true changes in $Stream1$ and $Stream17$ using XI-fixed with different τ values



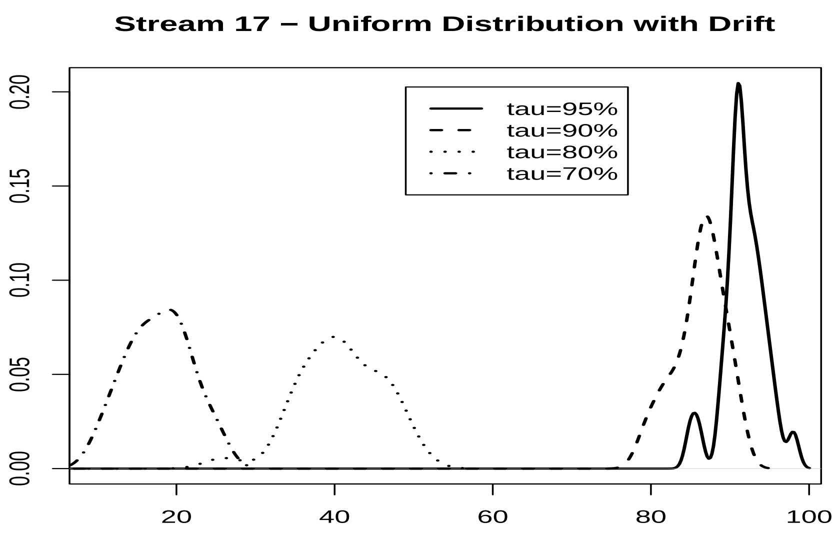
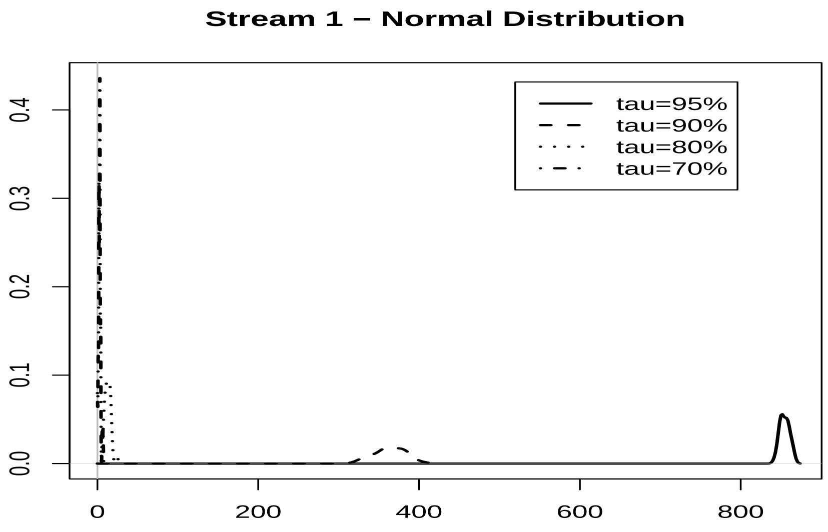


Figure 3.53: Number of true changes detected for $Stream1$ and $Stream17$ using KD-moving with different τ values



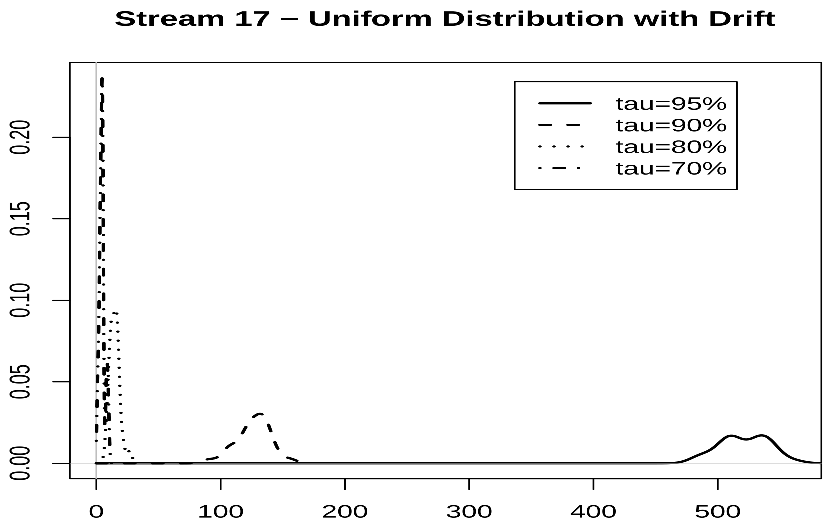
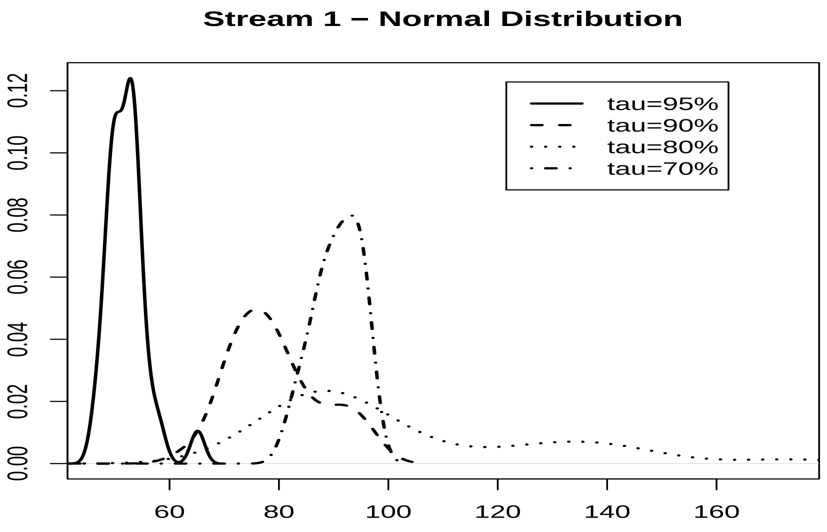


Figure 3.54: Number of false changes detected for $Stream1$ and $Stream17$ using KD-moving with different τ values



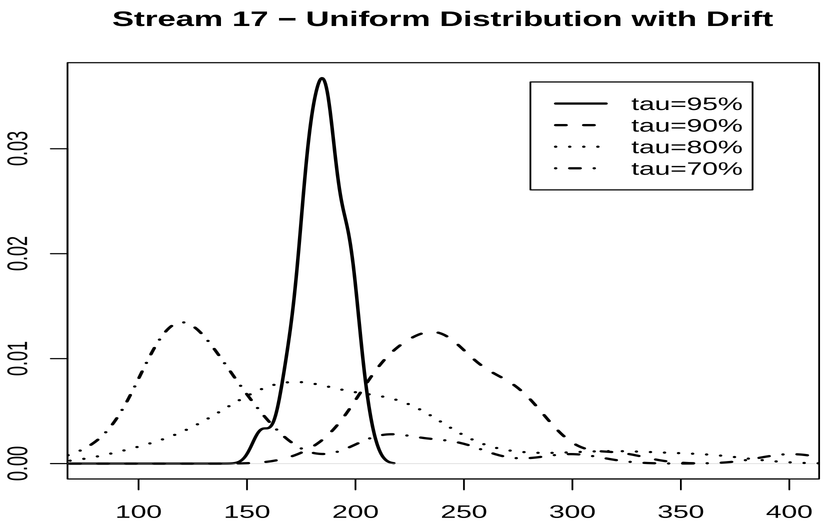
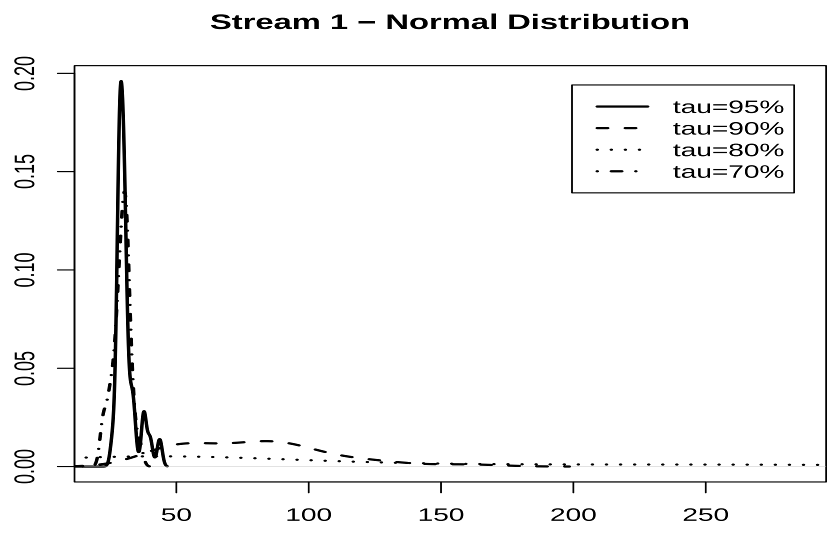


Figure 3.55: Mean duration for detecting true changes in $Stream1$ and $Stream17$ using KD-moving with different τ values



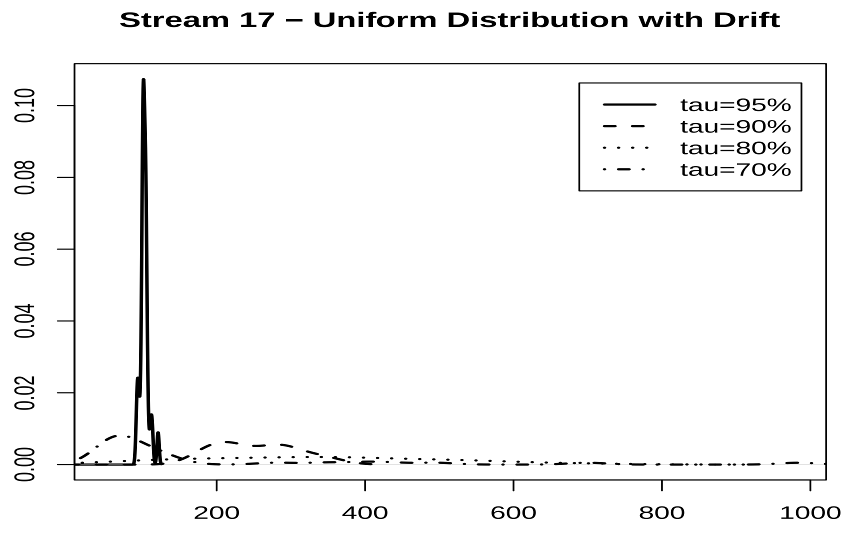
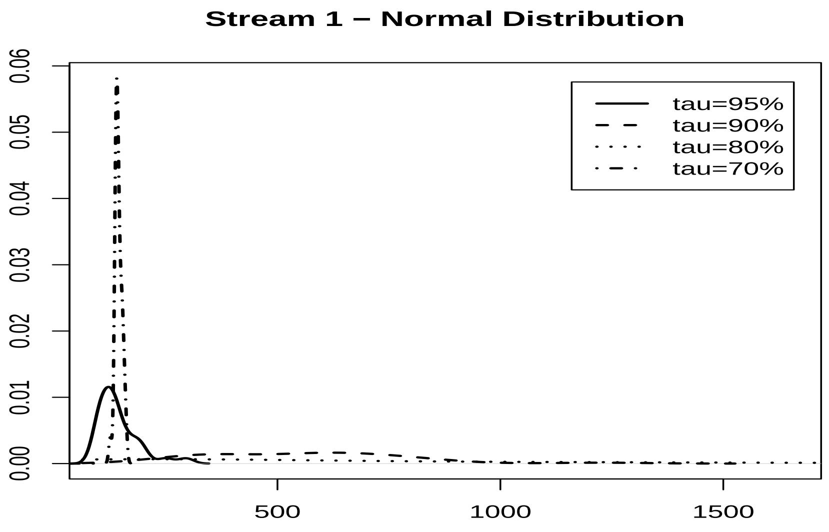


Figure 3.56: Standard deviation of the duration for detecting true changes in *Stream*1 and *Stream*17 using KD-moving with different τ values



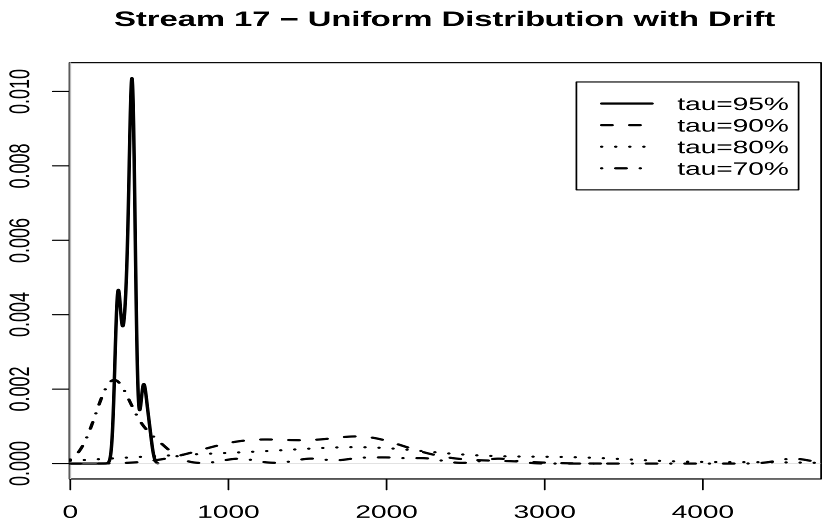
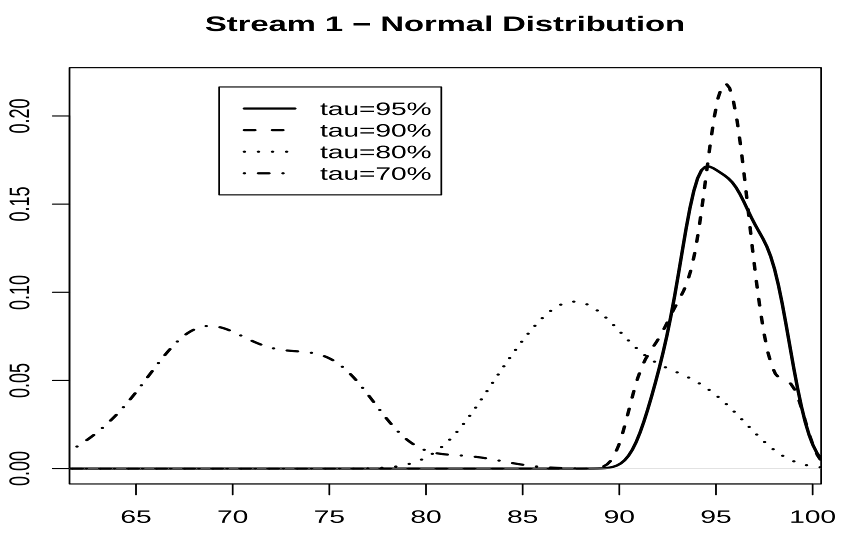


Figure 3.57: Max duration for detecting true changes in $Stream1$ and $Stream17$ using KD-moving with different τ values



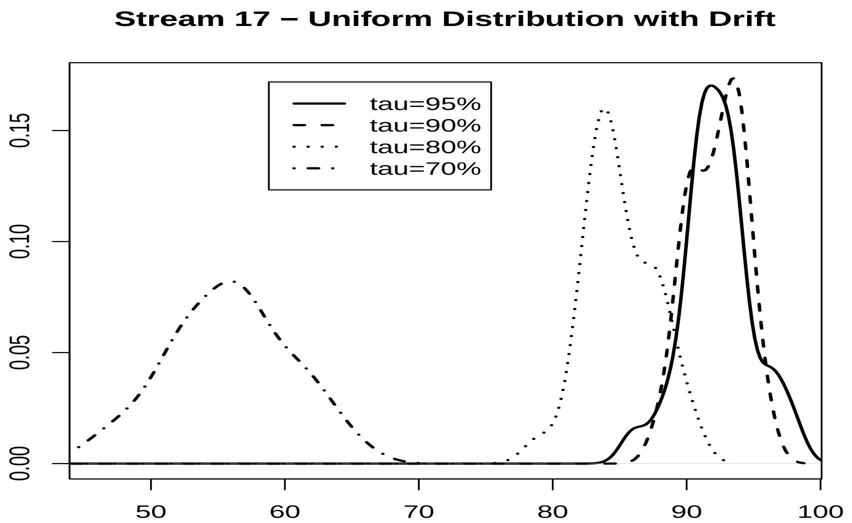
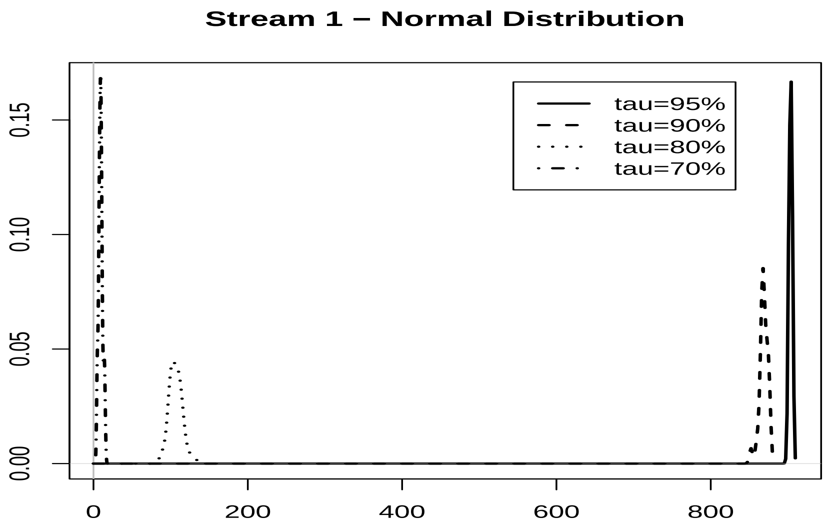


Figure 3.58: Number of true changes detected for $Stream1$ and $Stream17$ using XI-merged with different τ values



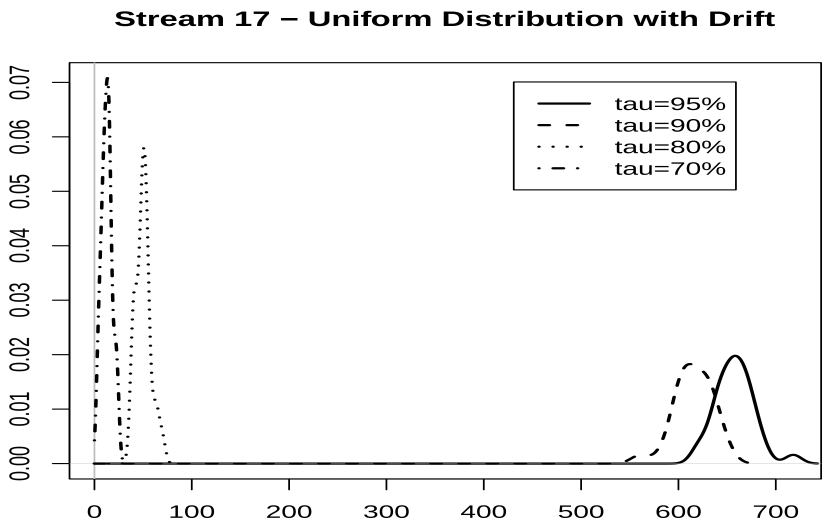
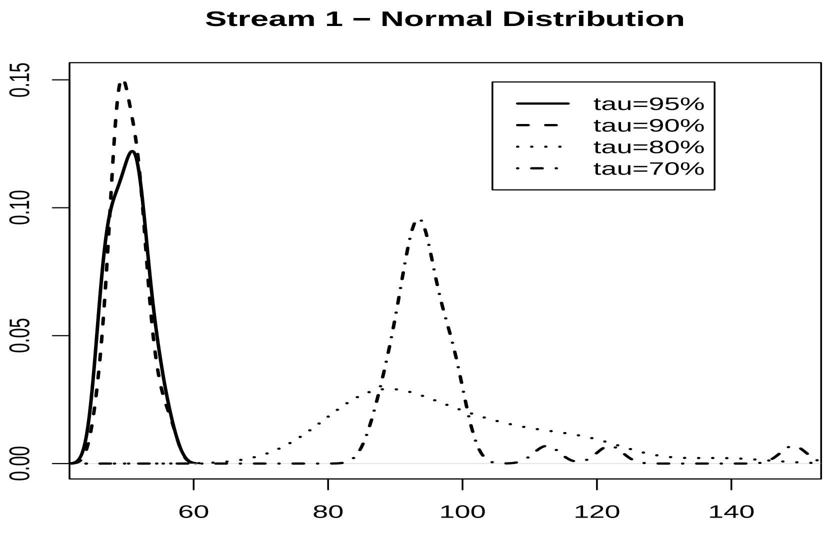


Figure 3.59: Number of false changes detected for $Stream1$ and $Stream17$ using XI-merged with different τ values



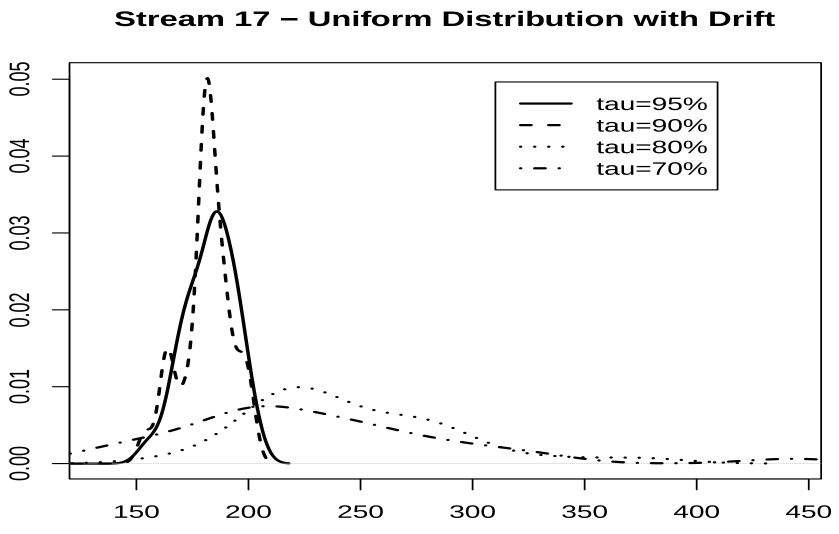
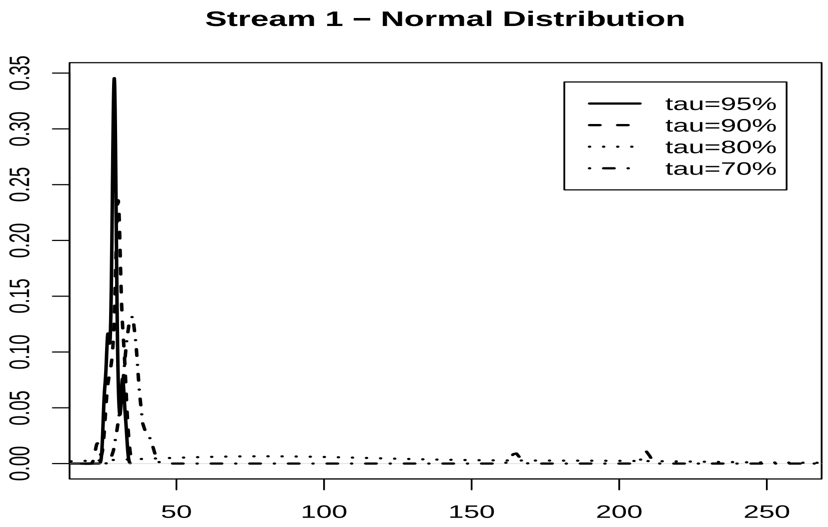


Figure 3.60: Mean duration for detecting true changes in $Stream1$ and $Stream17$ using XI-merged with different τ values



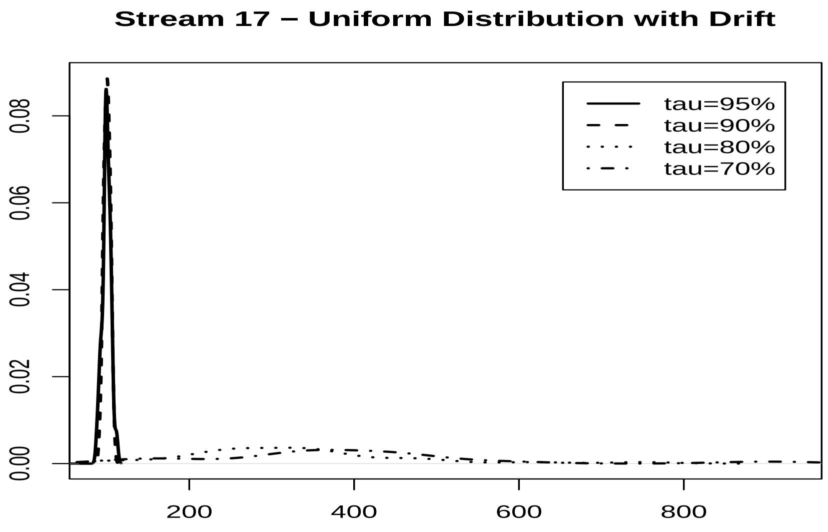
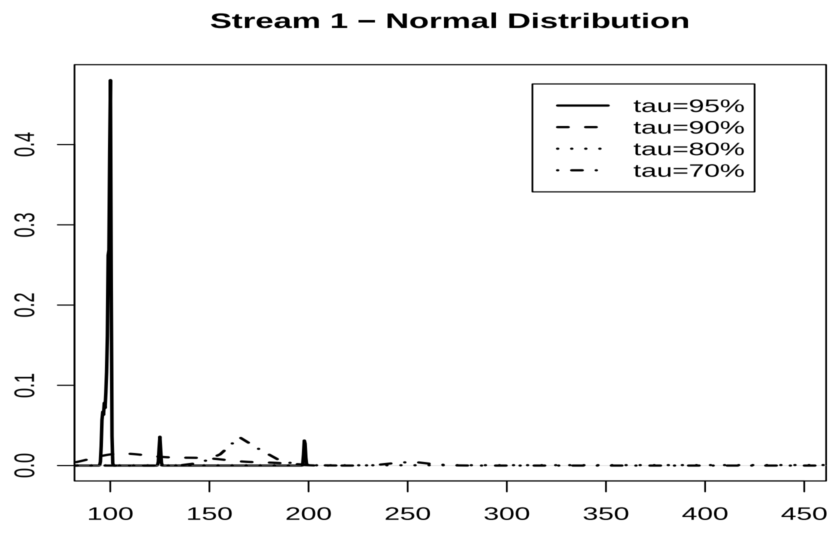


Figure 3.61: Standard deviation of the duration for detecting true changes in $Stream1$ and $Stream17$ using XI-merged with different τ values



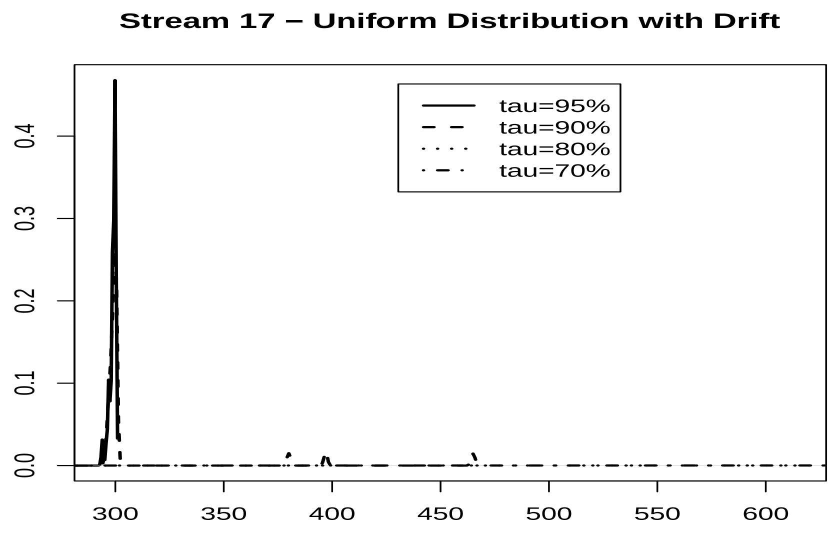


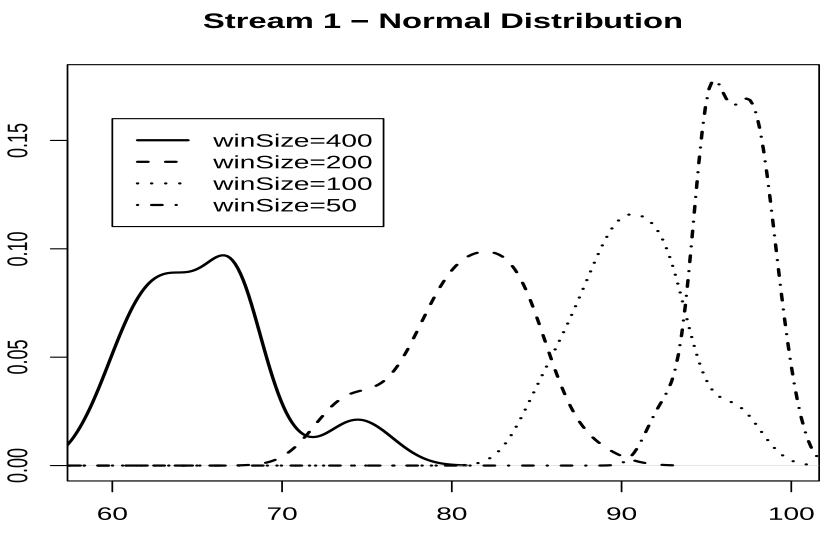
Figure 3.62: Max duration for detecting true changes in *Stream*1 and *Stream*17 using XI-merged with different τ values

These results show that larger τ values lead to a larger number of true detections. However, the number of false changes that are detected increases dramatically when τ value increases. These observations confirm the previous discussion. Furthermore, the true change detection durations with larger τ settings are significantly shorter than the durations with smaller τ values. This is because a larger τ value indicates more significant distribution changes, which are easier to detect.

Effect of window size

As discussed in Section 3.3.2, the sizes of windows W_r and W_t affect both the accuracy and efficiency of the change detection technique. To study the effect of window size, we conduct a set of experiments using the proposed experimental framework with sizes of W_r and W_t set as 50, 100, 200, and 400 data elements. The rest of the parameter settings are the same as previous experiments. Similar to the experiments on different τ values, only the results for XI-fixed, KD-moving and XI-merged change detection techniques on $Stream1$ and $Stream17$ are demonstrated in Figures 3.63 – 3.77.

According to these results, a very small windows size will result in a large number of false detections, although the number of true changes detected is also larger. This may be because the representative set is too small to represent the true distribution that generates it. Therefore, the accuracy of the change detection technique is greatly impacted. However, true distribution changes can be detected quickly with a small windows size, because the discrepancy between P_r and P_t is calculated frequently. We also note that the experiments with same *numRun* take noticeable longer time when using a small windows size. Hence, the efficiency of the change detection techniques may be affected with small windows size setting.



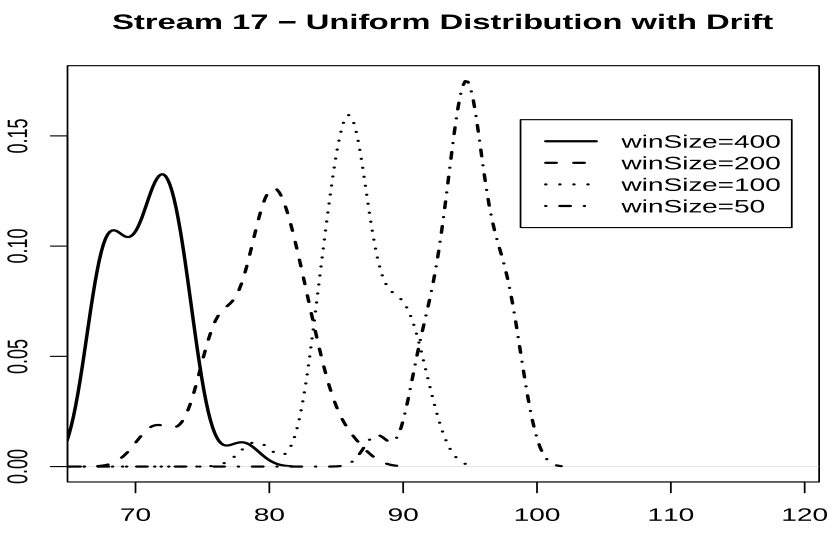
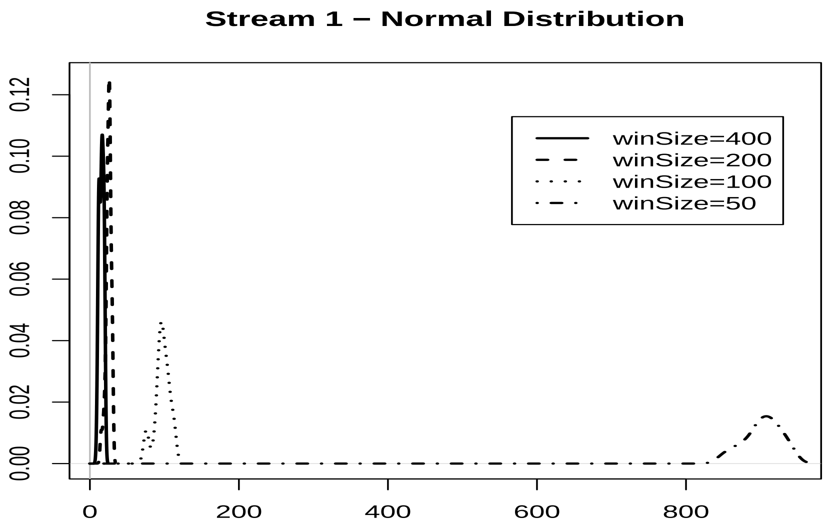


Figure 3.63: Number of true changes detected for $Stream1$ and $Stream17$ using XI-fixed with different windows size



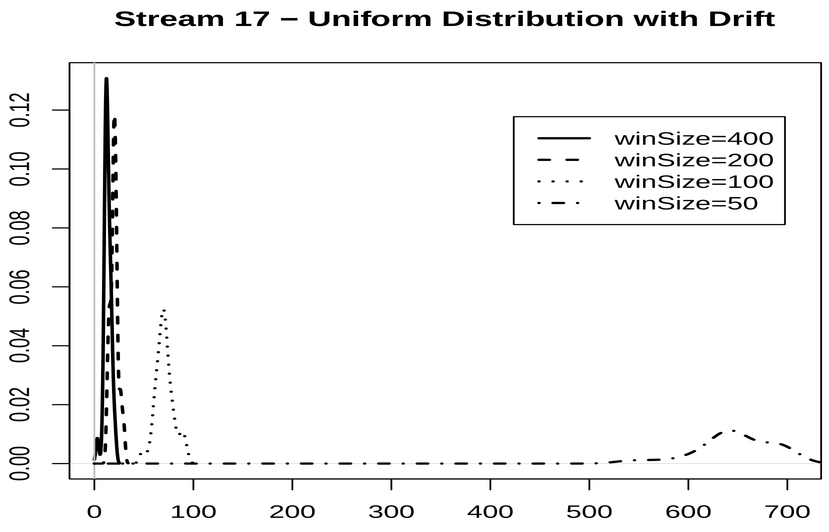
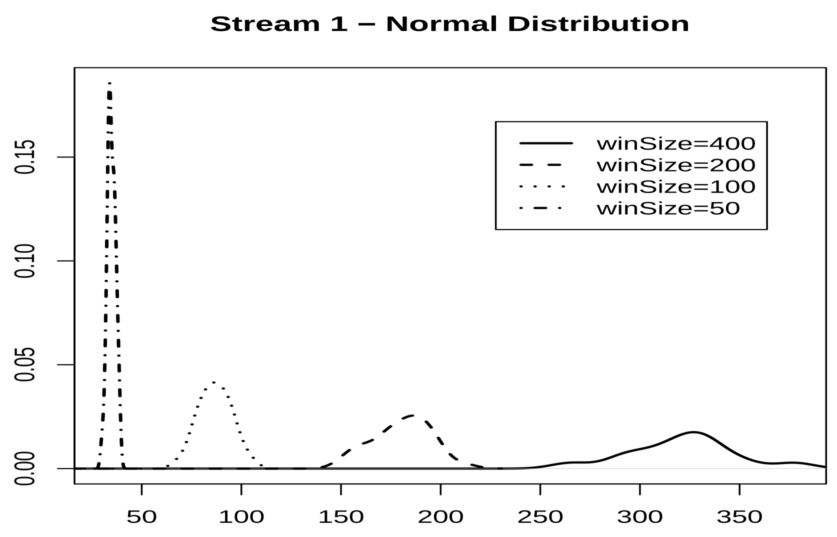


Figure 3.64: Number of false changes detected for $Stream1$ and $Stream17$ using XI-fixed with different windows size



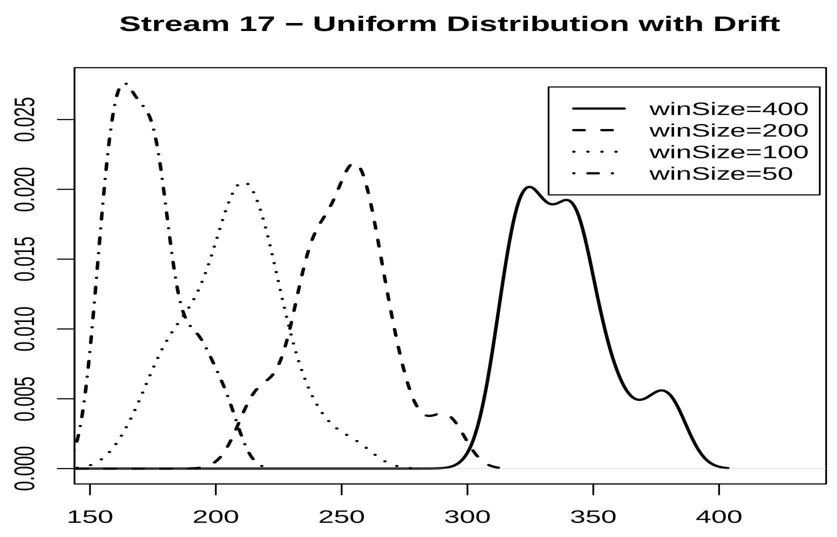
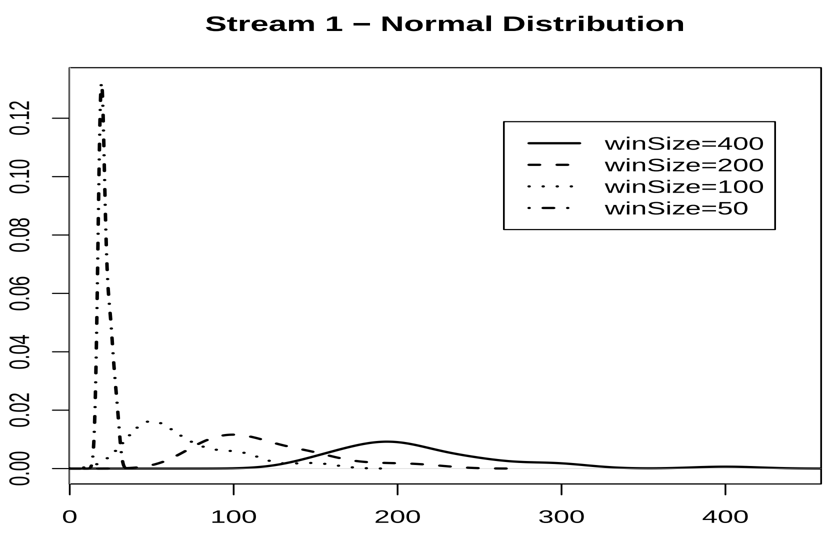


Figure 3.65: Mean duration for detecting true changes in $Stream1$ and $Stream17$ using XI-fixed with different windows size



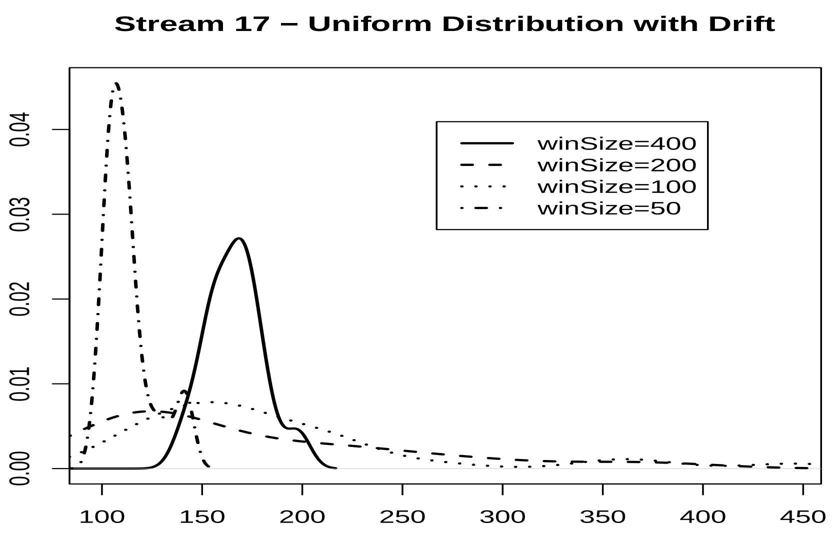
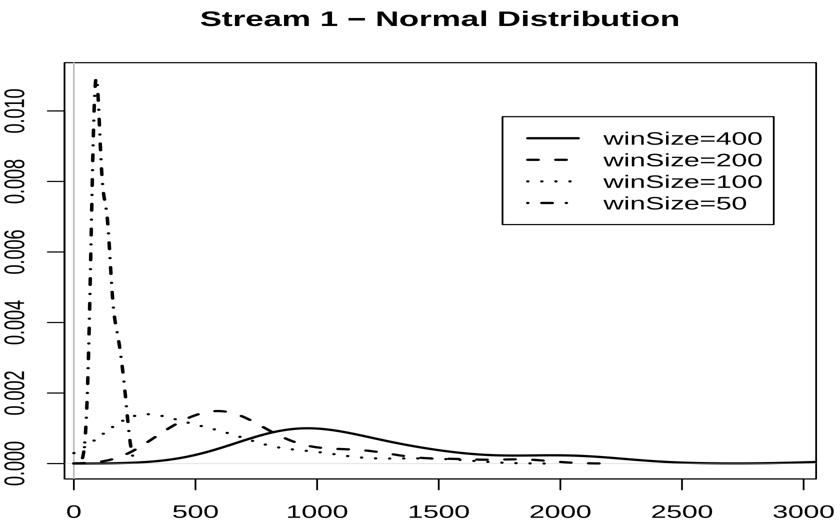


Figure 3.66: Standard deviation of the duration for detecting true changes in *Stream*1 and *Stream*17 using XI-fixed with different windows size



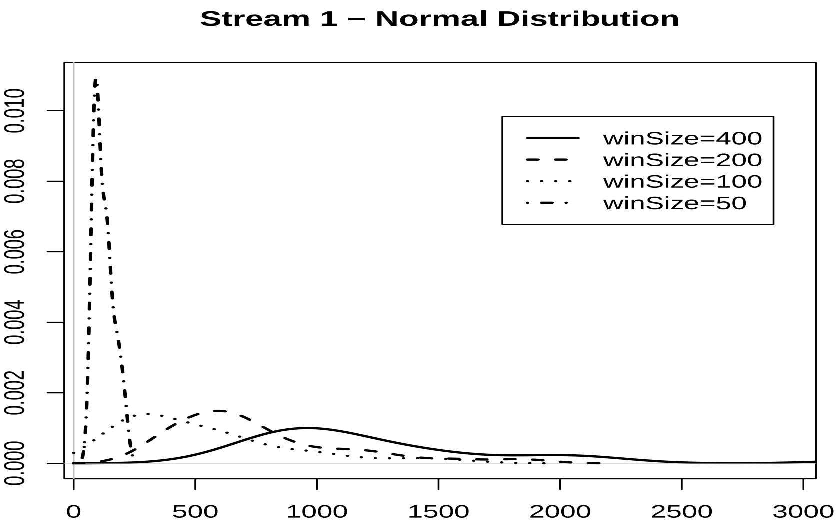
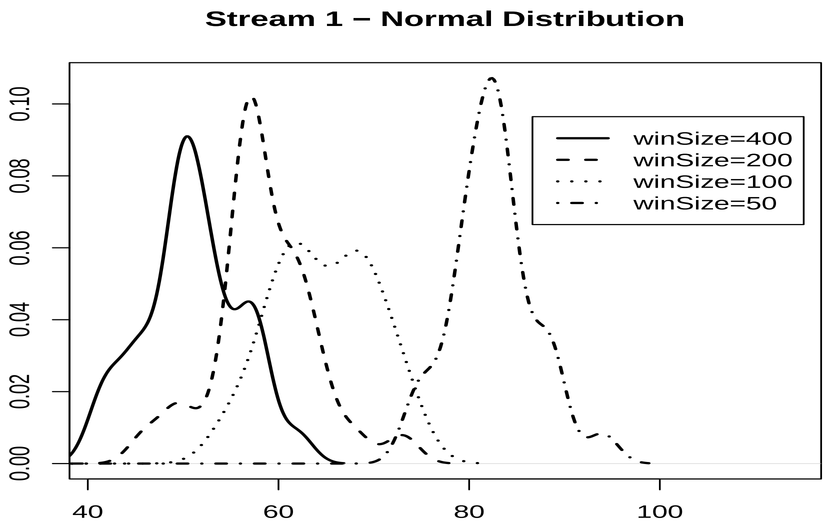


Figure 3.67: Max duration for detecting true changes in $Stream1$ and $Stream17$ using XI-fixed with different windows size



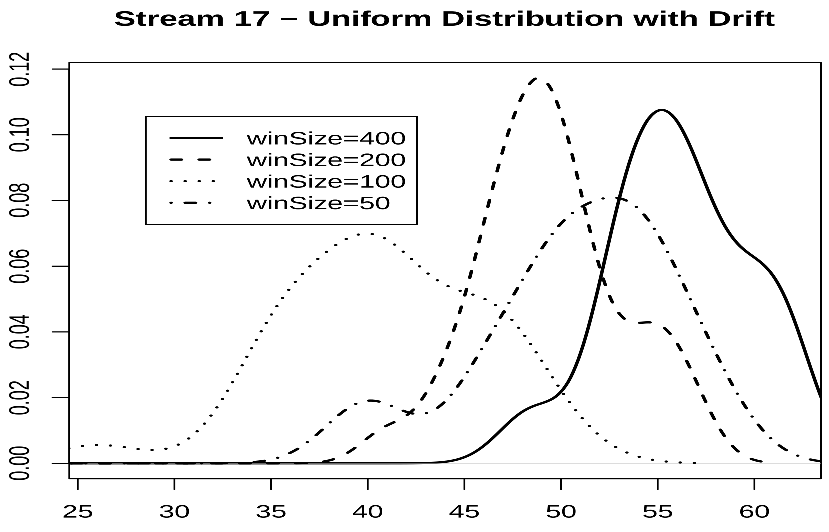
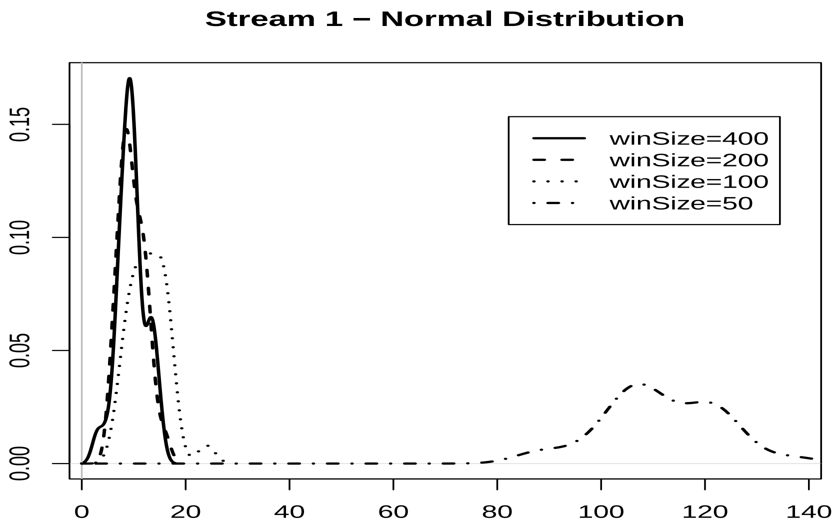


Figure 3.68: Number of true changes detected for $Stream1$ and $Stream17$ using KD-moving with different windows size



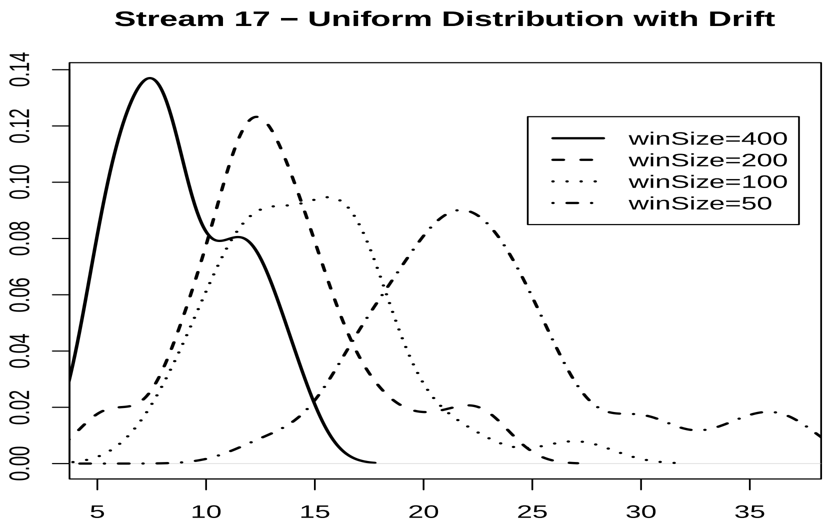
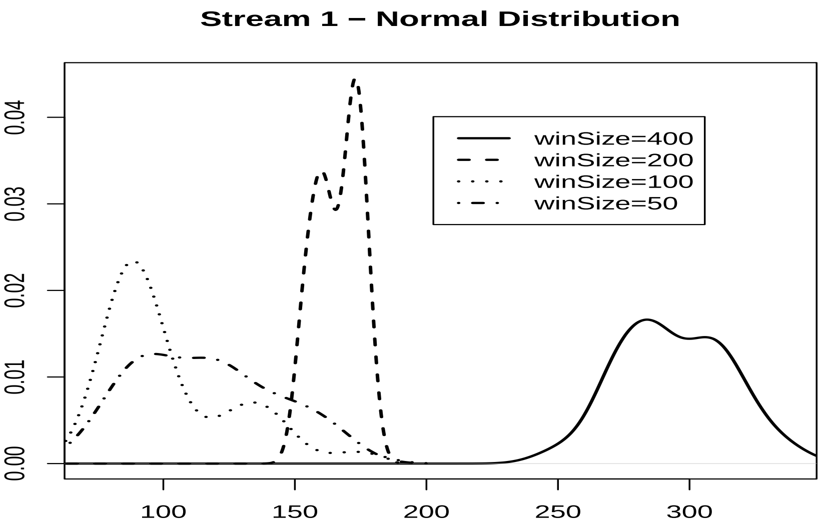


Figure 3.69: Number of false changes detected for $Stream1$ and $Stream17$ using KD-moving with different windows size



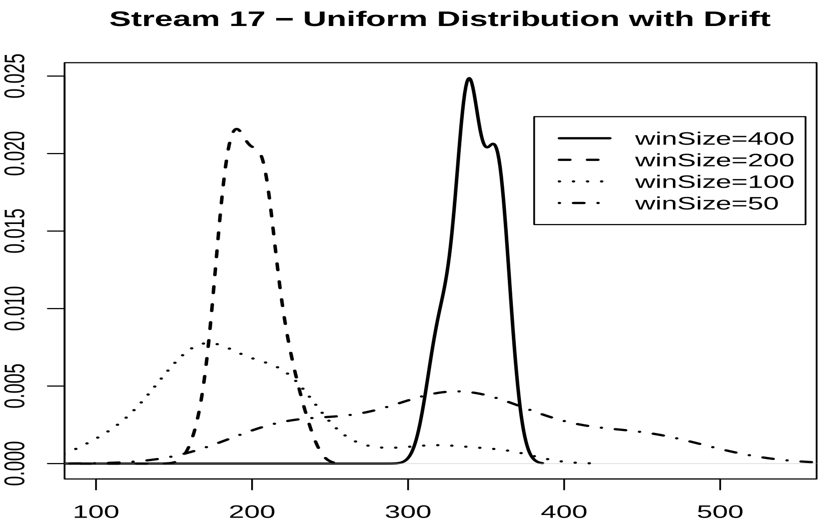
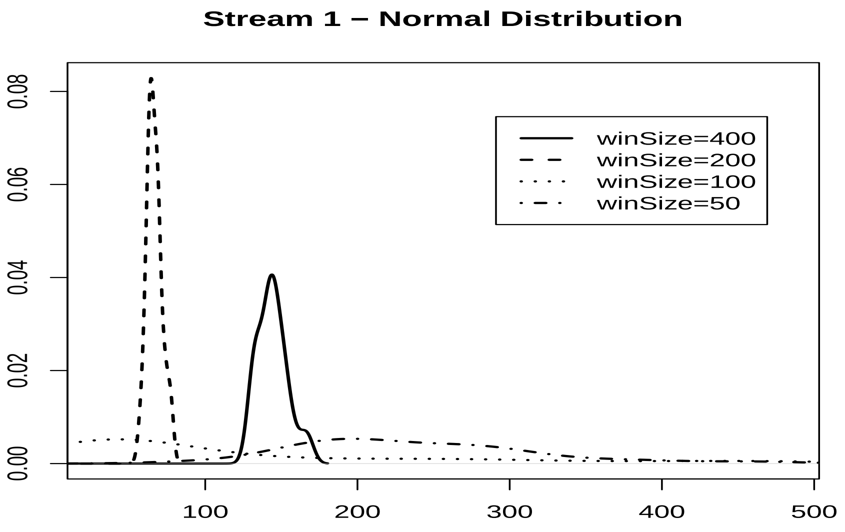


Figure 3.70: Mean duration for detecting true changes in $Stream1$ and $Stream17$ using KD-moving with different windows size



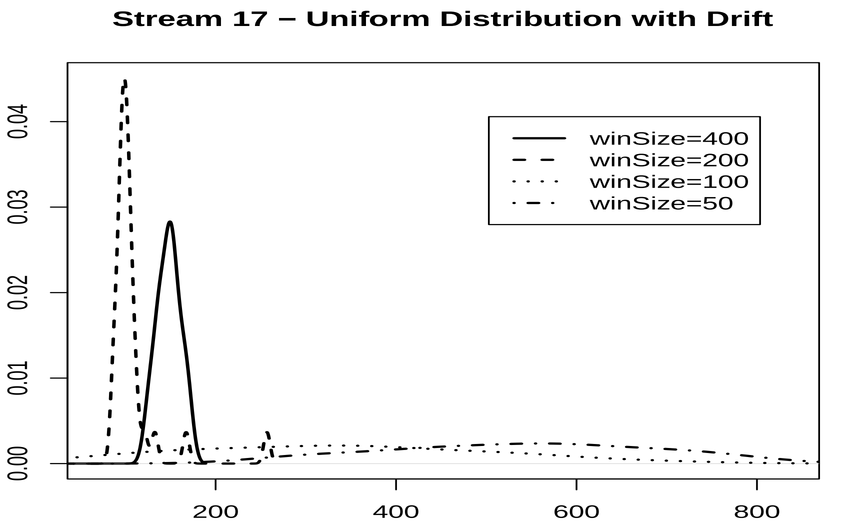
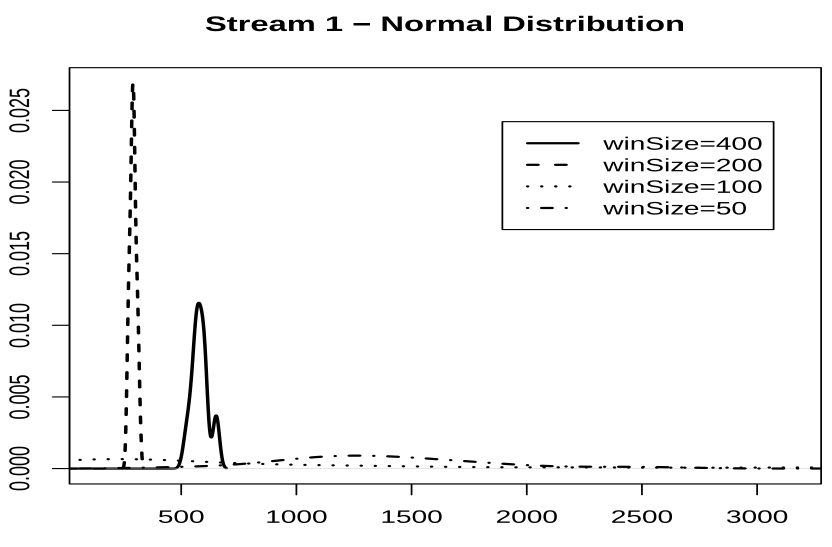


Figure 3.71: Standard deviation of the duration for detecting true changes in $Stream1$ and $Stream17$ using KD-moving with different windows size



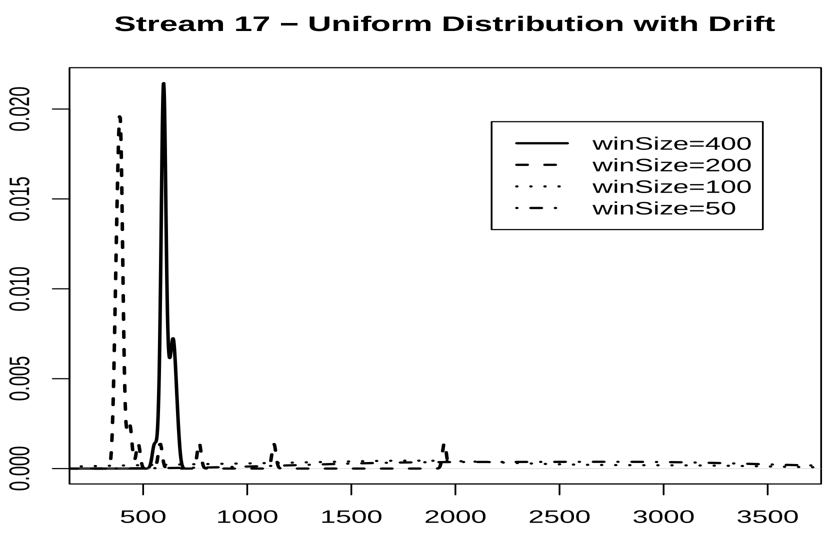
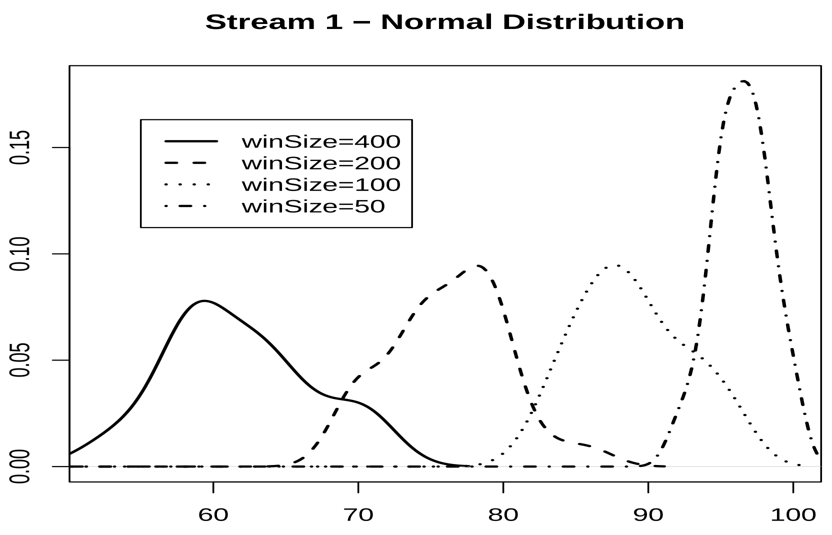


Figure 3.72: Max duration for detecting true changes in $Stream1$ and $Stream17$ using KD-moving with different windows size



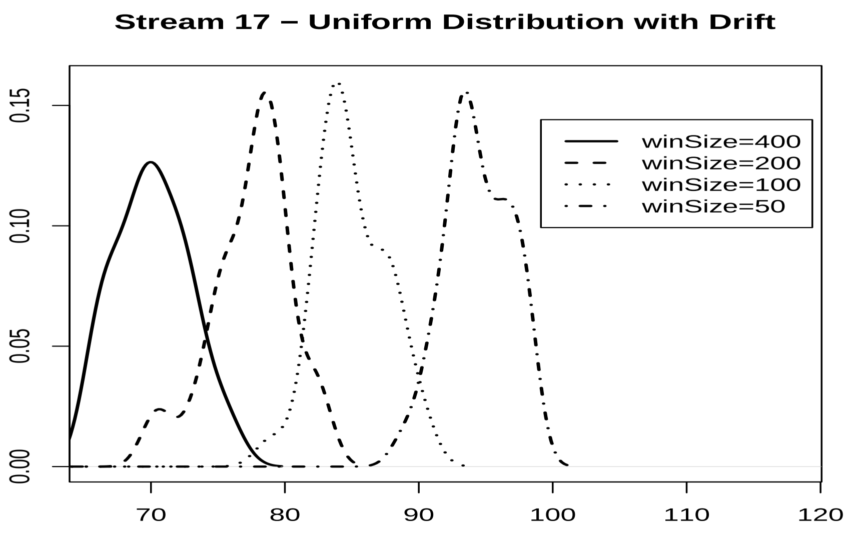
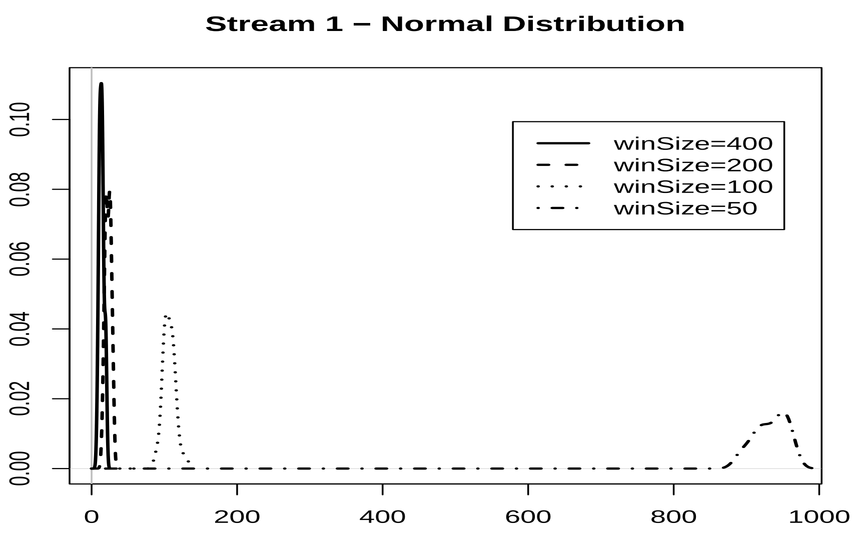


Figure 3.73: Number of true changes detected for $Stream1$ and $Stream17$ using XI-merged with different windows size



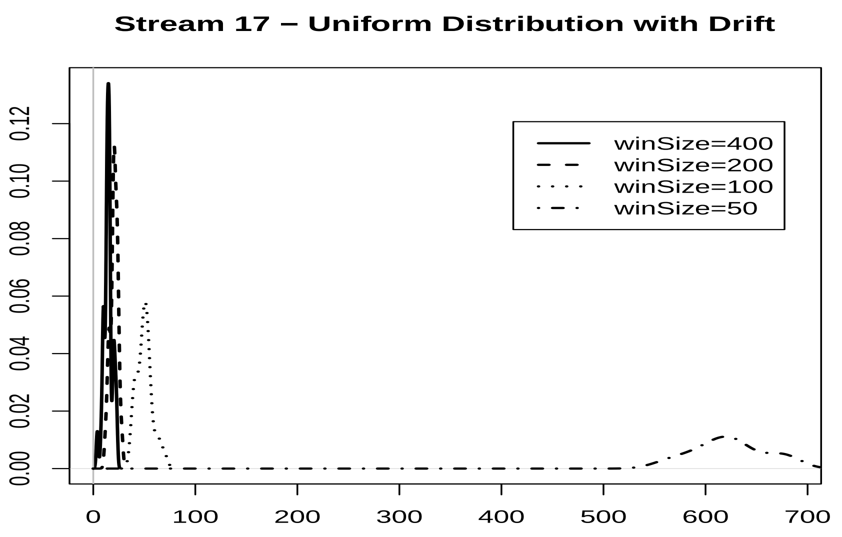
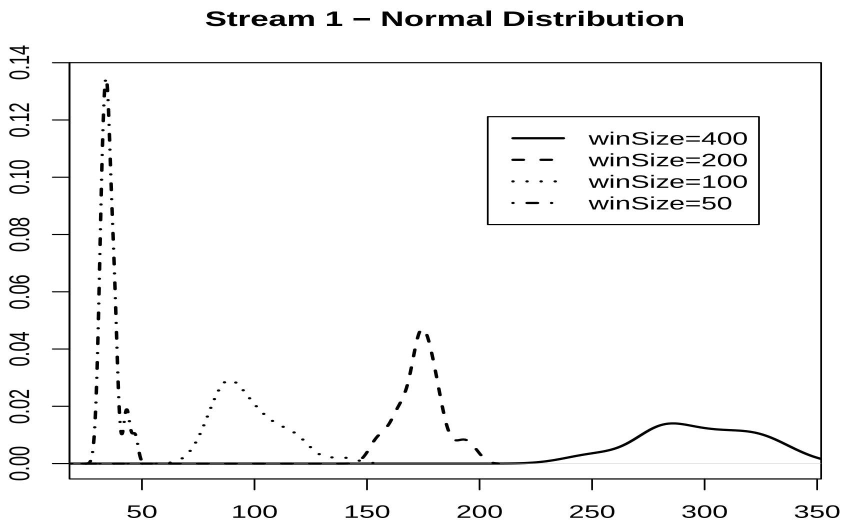


Figure 3.74: Number of false changes detected for $Stream1$ and $Stream17$ using XI-merged with different windows size



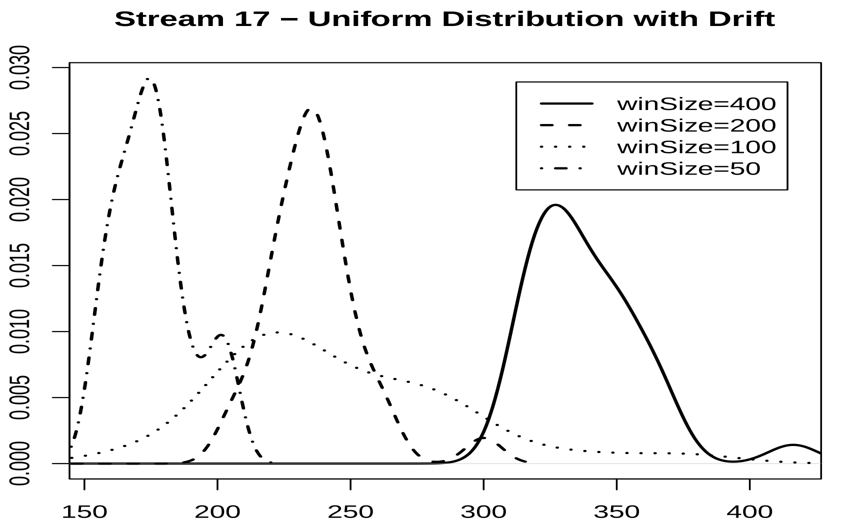
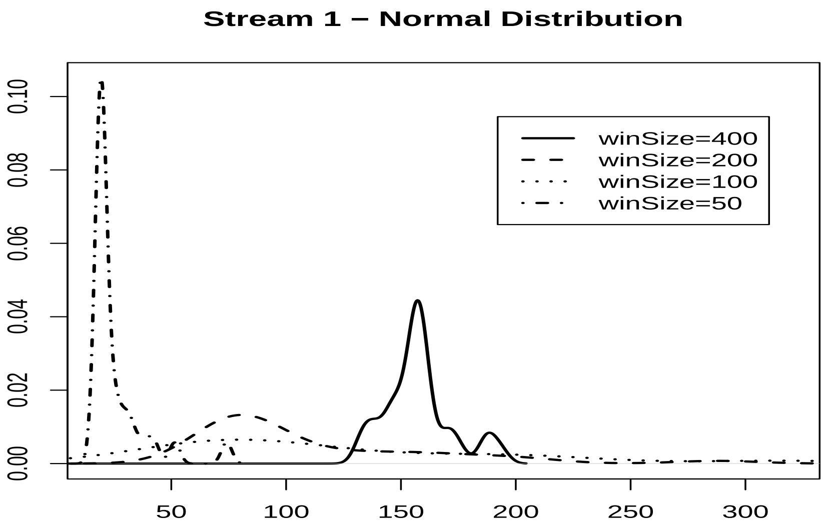


Figure 3.75: Mean duration for detecting true changes in $Stream1$ and $Stream17$ using XI-merged with different windows size



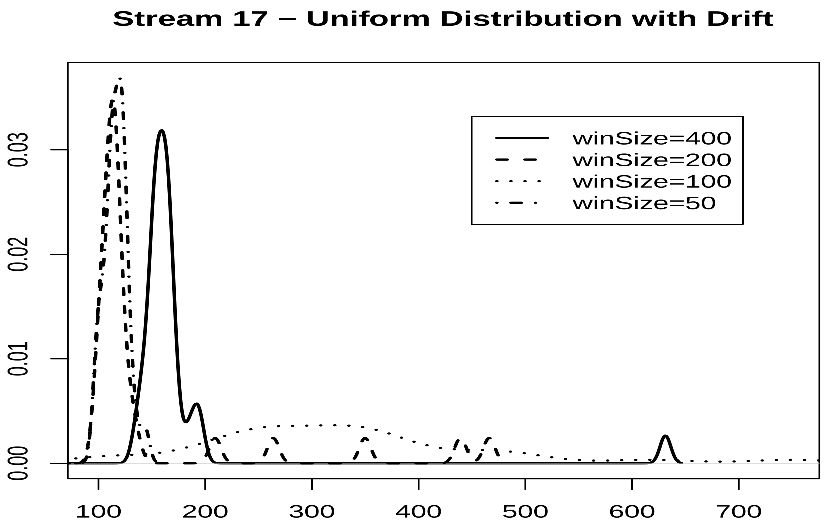
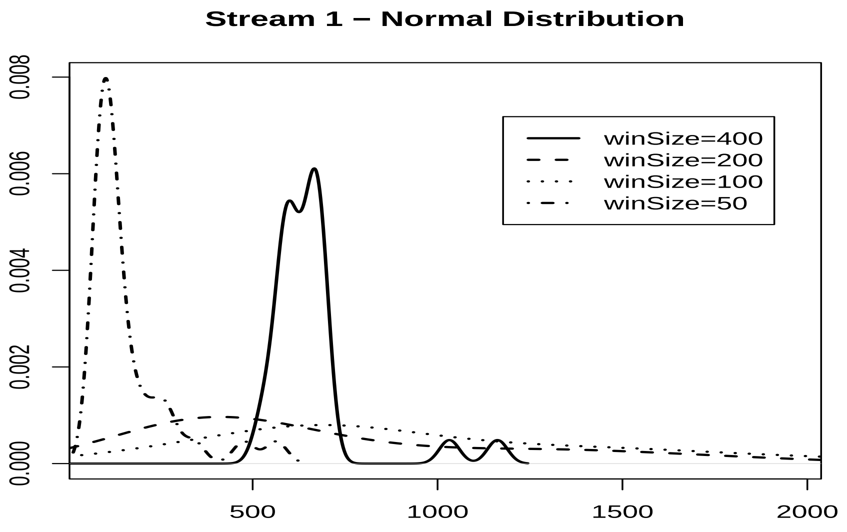


Figure 3.76: Standard deviation of the duration for detecting true changes in *Stream*1 and *Stream*17 using XI-merged with different windows size



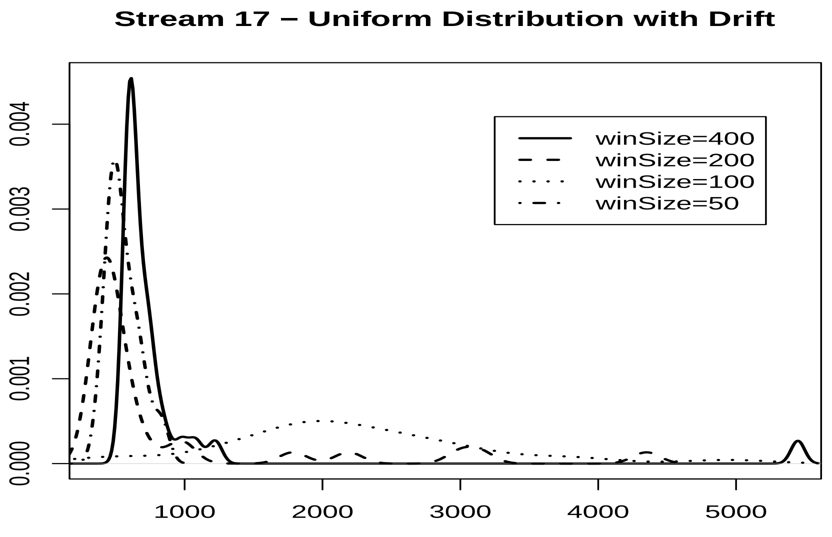


Figure 3.77: Max duration for detecting true changes in $Stream1$ and $Stream17$ using XI-merged with different windows size

3.3.7 Mining streams with periodical changes

For dynamic data streams, an interesting question is whether the distribution changes are entirely random and unpredictable, or whether it is possible for the distribution changes to follow certain patterns. If some regularity can be detected, this can be exploited in mining. An analysis of different applications and data streams reveals that there is a large class of data streams that exhibit periodic distribution changes; that is, a distribution that has occurred in the past generally reappears after a certain time period. Consider, for example, a scientific stream that records sea surface temperature. There is periodicity to the distribution of collected data that may not be visible within one year, but is apparent over multiple years.

For two substreams in a data stream generated by the same (or highly similar) distributions, mining results should be the same (or highly similar). Examples of such mining results include a list of all frequent items/itemsets for frequent pattern discovery, and set of clusters/classes for clustering and classifications. Therefore, if a historical distribution reoccurs and if its mining results have been archived previously, then it is possible to skip the re-mining process and directly output the archived mining results as the new results. This match-and-reuse strategy is faster for periodically changing streams compared with the traditional detect-and-mine stream mining approaches, since pattern matching is usually considerably less time consuming than mining.

Based on this insight, we propose a approach called DMM (Detect, Match, and Mine) for mining data streams with periodically changing distributions. Change detection procedure runs throughout the lifetime of a stream to monitor its distribution changes. Once a new distribution is detected, this new distribution is matched against those that have occurred and archived earlier. If a match is found between the new distribution and an archived one, DMM skips the re-mining process and outputs the mining results of the archived distribution. Consequently, processing time is greatly reduced. If pattern matching fails (i.e., no similar distribution has been seen previously), then the mining process is activated to generate new results for this new distribution.

Note that in DMM the procedures of change detection, pattern matching, and data mining are independent of each other. This provides considerable flexibility as it is possible to plug in any technique for each procedure at any point if the application requirements are altered.

The key problems that need to be resolved in the proposed DMM approach are pattern representation, pattern selection, and matching. Solutions for each of these problems are discussed in the following sections.

Pattern representation

During the lifespan of a dynamic stream that has continued for a long time, there may be a large number of different distributions occurred. This number will keep increasing over time. Therefore, due to memory concern, each archived distribution (pattern) needs to be stored as succinctly as possible. One of the most popular approach in data stream mining is to represent the current distribution using a representative data set.

The performance of DMM approach greatly depends on the method of selecting representative set. If the discrepancy between the true distribution and the distribution of its representative set is large, the accuracy of pattern matching results may be low. When two distinct distributions are mistakenly considered similar, archived mining results that are not similar to the real mining results of the new distribution will be output. If two highly similar distributions are not recognized as a match because of the poorly selected representative sets, time will be wasted on the unnecessary re-mining.

The DMM approach adopts the merged window method proposed in Section 3.3.3 for choosing representative set of each newly detected distribution. The representative set is continuously updated until a distribution change occur. The performance of merged window method have been extensively studied in Section 3.3.6.

Choosing important distributions

For a dynamic data stream, there could be a large number of different distributions that are observed during the lifespan of the stream. Due to limited memory, it is infeasible to record all of these distributions along with their mining results. Furthermore, maintaining a large number of distributions could increase the time it takes to match a newly detected distribution. Hence, only important distributions, i.e., the ones that have a high probability to be observed again in the future, should be archived. We use $\mathcal{P} = \{P_1, ..., P_m\}$ to denote the set of important distributions.

The importance of distributions is determined by the following heuristics:

1. Distributions that have occurred in the stream for more times are more important than the ones that have been observed fewer times. For each archived distribution P_i, a counter occ_i is used to indicate the number of times P_i has occurred. Hence, for two distributions P_i and P_j, if $occ_i > occ_j$, then P_i is more important than P_j.

2. The longer a distribution lasts in the stream's lifespan, the more important it is. If a distribution P_i is detected at t_1 and the subsequent change is detected at t_2, P_i's lifespan is $T_i = t_2 - t_1$. Hence, if $T_i > T_j$, then P_i is more important than P_j.

3. The more distinctive a distribution is, the higher is the chance that it will be archived. A distribution that is similar to an existing distribution in $\mathcal{P}$ (but not enough to be recognized as a match) is regarded as less important. When searching for a match of a new distribution P_i, the smallest discrepancy $min_i(Discrepancy)$ between P_i and any of the archived distributions in $\mathcal{P}$ is recorded. Hence, for two distributions P_i and P_j, if $min_i(Discrepancy) > min_j(Discrepancy)$, then P_i is more important than P_j.

4. A distribution P_i that has mining results R_i with higher accuracy is more important than a distribution with less accurate mining results.

Let $Acc(R_i)$ be the accuracy of the mining results for P_i; hence, if $Acc(R_i) > Acc(R_j)$, then P_i is more important than P_j[2].

When a distribution change is detected, whether or not a match is found, the important distribution set $\mathcal{P}$ is updated. The distribution P_r that was in effect when a change was detected is evaluated to determine whether it should be included in $\mathcal{P}$. If P_r had been matched with pattern $P_i \in \mathcal{P}$, then P_r replaces P_i in $\mathcal{P}$ if its lifespan is longer than P_i's lifespan (rule 2). If P_r has no matching distribution in $\mathcal{P}$ and $\mathcal{P}$ has not reached its maximum memory allowance, P_r is added to $\mathcal{P}$. Otherwise, the distribution that is the least important is pruned from $\mathcal{P}$ according to rules 1-4.

Distribution matching

When a new distribution is detected at time t_1, a set of data elements is chosen as the sample data set representing this new distribution. For the applications that require high efficiency, it is possible to use the substream in observation window W_t from timestamp t_1 to $t_1+\Delta_t$, so that the matching can start as soon as possible. However, as discussed in Section 3.3.1, the first set of data that arrive at the beginning of a new distribution may not capture the true distribution, especially when the distribution is complicated or the distribution change is slow. Therefore, if the accuracy is more important than efficiency, the new distribution is observed for a longer time (i.e., wait until W_t tumbles several times) and the representative set is refined by using the merged window method discussed previously.

This new distribution is then matched with a set of important historical distributions that have been preserved. Let S_A and S_B be two substreams

[2]The mining results of a supervised learning task is used as the ground truth. $Acc(R_i)$ refers to the accuracy of R_i compared with the ground truth. If R_i is later reused for another distribution P_k that matches P_i, the accuracy of R_i over P_k is still calculated w.r.t to the ground truth. Therefore, the error introduced by the adopted mining technique and the error caused by the match are both taken into account.

containing the representative sets generated by the new distribution P_A and a historical distribution P_B, respectively.

Definition 3.2 Given S_A, S_B, P_A and P_B as defined above, if $Discrepancy(P_A,\ P_B) \leq 1-\tau$, where τ is the predefined significance level, then the distributions of substreams S_A and S_B match each other, denoted as $P_A \rightleftharpoons_\delta P_B$.

The discrepancy between P_A and P_B is calculated using a chosen statistical test. The performance of two statistical tests, kernel density test and XI-distance test, were demonstrated in Section 3.3.6. The choice of the test depends on the application requirement.

Let R_A and R_B be the mining results obtained during the period that distributions P_A and P_B are in effect, respectively. If a match is found, i.e., $P_A \rightleftharpoons_\delta P_B$, then the preserved mining results R_B for the stream is output as the new mining result R_A, i.e., $R_A = R_B$. The justification is that for two highly similar distributions, their mining results should present high similarity as well. This way, the data mining time is dramatically reduced without reducing the quality of mining results.

The significance level τ is important: larger τ implies a higher accuracy of the two matching distributions, while a smaller τ increases the possibility of a new distribution to match a pattern in the preserved set. A smaller τ leads to higher efficiency, since the time for matching distributions and reusing mining results are far less than the time for re-mining the new distribution. Therefore, the question of finding a balance between accuracy and efficiency for setting τ value arises. The solution depends on the nature of the application and the nature of streams generated by it. The impact of τ is empirically studied in Section 3.3.6.

3.4 Detecting mean and standard deviation changes

3.4.1 Motivation

As discussed in Section 3.1, it is difficult to achieve a high accuracy in estimating the distribution using a small data set generated by it. However, some key features of this distribution, such as mean, range, variance, and cardinality (i.e., the number of distinct values) can be obtained with high accuracy using a small data set. For many stream mining applications, these key features are the sole interest and are sufficient for generating mining results. Examples of such applications include fraud detection, temperature monitoring, production quality control, and trend analysis. Therefore,a control chart [129] based approach is proposed that detects mean and standard deviation changes in any dynamic data stream.

There are two types of control charts that are widely used in real-world applications: Shewhart control charts (SCC) [144] and Exponentially Weighted Moving Average (EWMA) control charts [9]. In SCC, the decision regarding the in/out-of-control state of a data point depends solely on that data and, hence, no historical data are used in its estimations. In contrast, EWMA's decision depends on the *EWMA statistic*, which is an exponentially weighted average of all prior data points and, thus, its decisions are affected by historical data. EWMA can be sensitive to small but gradual distribution drifts and are suitable for streaming applications such as temperature monitoring and financial marketing, whereas SCC is ideal for streaming applications, such as fraud detection and machine monitoring, that expect sudden distribution shifts. Since many data stream mining applications use past data and previous mining results as a guidance to mine the newly arrived data, the EWMA control charts are used in the proposed technique so that historical data are taken into consideration when detecting distribution changes.

Unlike many of the statistical techniques for change detection that require a large number of samples to get promising results, control chart method can achieve high accuracy with only a small sample set. This is an important feature for detecting changes in the streaming environment, since stream processing techniques are single pass and memory is always limited. As the experiments demonstrate, the proposed approach has high efficiency so that fast distribution changes in the stream can be captured.

3.4.2 Introduction to control charts

Statistical control charts are widely used for controlling and monitoring manufacturing processes. They are powerful tools that can detect distribution changes in sensor readings. Control chart techniques are not capable of providing the underlying distribution function for the given data sets. Instead, they only focus on monitoring the important characteristics of the data set (i.e., the ones in which the user is most interested). In many practical cases, the mean and standard deviation of the observed data are monitored by the charts. Although control charts cannot identify the distribution function of the data set, they are effective, efficient, and their interpretations can assist in the identification of the nature of the data.

A control chart is a graphical display of (usually) one feature that has been measured or computed from a set of sample observations. A control chart contains three values: the mean value of the reading that is obtained by learning a small sample set at the beginning of the process (center line in Figure 3.78), the Upper Control Limit (UCL), and the Lower Control Limit (LCL) (the two horizontal lines above and below the mean in Figure 3.78). The control limits are probability limits that indicate the probability that a data point falls outside these values. If a point falls outside the limits, then a possible "out-of-control" state is reported; otherwise, the process is said to be "in-control". Figure 3.78 is an example of a control chart with one out-of-control point.

The major issue in control chart design is to determine the UCL and LCL, and the time intervals used to update them. Many control chart design algorithms have been developed [167, 67, 164, 169]. However, none of them are specifically designed for dynamic data stream change detection. The user has to manually readjust UCL and LCL at each point a distribution change is detected.

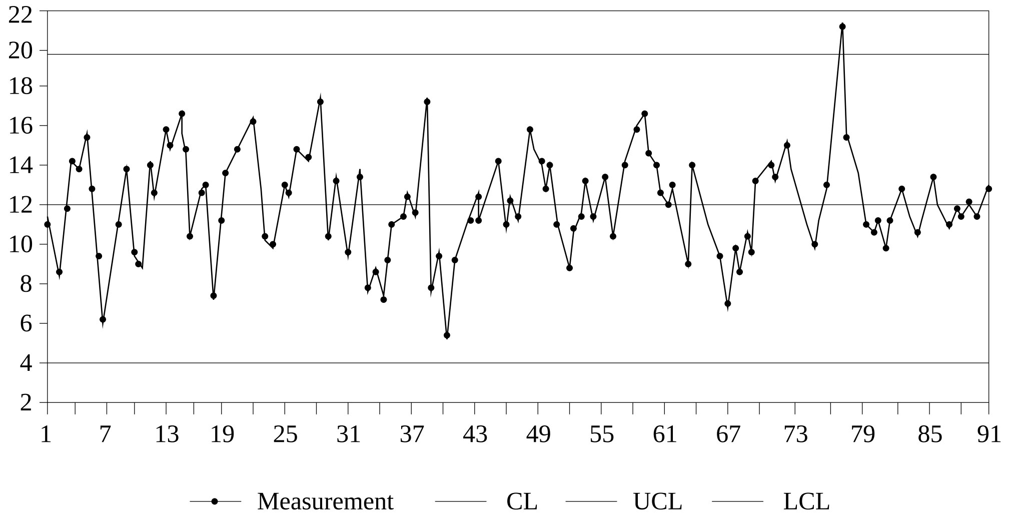


Figure 3.78: Example of a control chart

3.4.3 Control chart-based approach

In this section, a EWMA control chart based approach is proposed for detecting mean and standard deviation changes in dynamic data streams. This approach uses a tumbling window W to store the most up-to-date substream from which the test samples are obtained. Control chart C_μ monitors the mean changes and control chart C_σ monitors the standard deviation changes of the stream. Every time W moves, new samples are processed and then fed into C_μ and C_σ. If the new samples fall outside LCL and UCL of one of the control charts, then a distribution change is reported. Charts C_μ and C_σ will be automatically adjusted to reflect the new distribution.

Stopping rule design

Let S_r be a substream generated by the current distribution of S, and substream S_o contain newly arrived data. A tumbling window W with tumbling interval Δ is introduced for capturing the set of new elements that have arrived in S in the last Δ time units (i.e., substream S_o). Every time the window tumbles (i.e., every Δ time units), change detection process is triggered. Note that the commonly used sliding window model is not adopted, because sliding window causes frequent update of the statistics every time a new element arrives, and, thus, is not suitable for streams with high speed.

Let t' be the timestamp of the last distribution change detection in S, and t be the current timestamp. Let $S_r = S(t', t - \Delta]$ be the set of data elements generated by the current distribution. $S_o = S(t - \Delta, t]$ is the set of new elements that have arrived in S in the last Δ time unit. Distribution change detection can be formulated as a hypothesis testing problem. For substreams S_r and S_o with probability distributions P_r and P_o, respectively, assuming that data sets in S_r and S_o are independent and identically-distributed (i.i.d), the null hypothesis H_0 asserts P_r and P_o are identical. In other words, H_0 is in favor if the data sets in S_r and S_o belong to the same distribution. The problem of distribution change detection for dynamic data streams is to find a proper test so that H_0 will be refuted if it is no longer true. The "alarm time" when H_0 is refuted is denoted as t_a. At time $t = t_a$, the alternative hypothesis H_1 that asserts P_r is different than P_o is in favor.

For the proposed control-chart based approach, the stopping rule [14] is defined as the test to refute H_0:

$$StR = inf\{(|\mu(S_o) - \mu(S_r)| > \epsilon_1) \vee (|\sigma(S_o) - \sigma(S_r)| > \epsilon_2)\} \qquad (3.11)$$

where inf indicates infimum (i.e., greatest lower bound), and ϵ_1 and ϵ_2 are user defined thresholds. Every time the stopping rule is triggered, a distribution change is reported.

Thresholds ϵ_1 and ϵ_2 indicate the sensitivity of the stopping rule to changes. Smaller thresholds make the change detection technique more sensitive to minor changes, but may introduce a higher false-alarm rate. Whereas a change detection technique with higher thresholds can only detect significant changes. In control chart based change detection techniques, ϵ_1 and ϵ_2 are represented by the control limits UCL and LCL.

Significance level τ is the minimum probability of refuting H_0 when stopping rule StR is satisfied, i.e.,

$$Pr(H_0|StR) < 1 - \tau \tag{3.12}$$

Building control chart C_μ for detecting mean changes

Let $\mu(S_t)$ denote the mean value of S from the last distribution change t_a to current time t. Let UCL_μ and LCL_μ denote UCL and LCL of control chart C_μ, respectively. The setting of $\mu(S_t)$ for dynamic streams is difficult, since S continuously grows over time, causing the mean value of the stream to continuously change. Furthermore, since historical data (i.e., the data that fall out of W) are lost, $\mu(S_t)$ is, in fact, unknown. Thus, a weighted moving mean formula is used to estimate the mean value of S up to time t:

$$\mu(S_t) = \omega * \mu(S(t', t]) + (1 - \omega) * \mu(S'_t) \tag{3.13}$$

$S(t', t]$ is the substream that contains data that have arrived within range $(t', t]$, where $t' < t$. $\mu(S'_t)$ is the weighted moving mean calculated at time t', and $0 < \omega \leq 1$ is the weight that defines the importance of historical data. The higher the ω value, the more important is the recent data. In the extreme case when $\omega = 1$, $\mu(S_t)$ is determined solely by the small sample set received between t' and t. The value of ω for a particular application is determined by the significance value τ. Tables provided by Lucas and Saccucci [91] are used to select appropriate ω value.

The control limit formulas for EWMA control chart on detecting mean changes are designed as:

$$\begin{aligned} UCL^t_\mu &= \mu(S_t) + \kappa * \sigma(S_t) \\ LCL^t_\mu &= \mu(S_t) - \kappa * \sigma(S_t) \end{aligned} \tag{3.14}$$

where κ is the distance of the control limits from the center line, expressed in terms of *limits*. The value of κ is determined by the weight ω and significance level τ. Tables in [91] can be used to select the value of κ for a particular application.

Building control chart C_σ for detecting standard deviation changes

The control chart designed in the previous section monitor the mean value changes in data streams. However, for some applications the standard deviation, or the "scale", of the data may change while its mean remains the same. One typical example is mechanic parts manufacturing, where standard deviation represents the precision of the instrument and anomalies in the precision must be address in time. In weather analysis, extreme temperatures may greatly affect the standard deviation of the annual temperature, while the changes on mean temperature over the year may be insignificant. Therefore, a control chart C_σ for monitoring standard deviation changes in the stream is designed in the proposed technique.

Similar to the mean value $\mu(S)$, the standard deviation $\sigma(S)$ is also unknown due to the fact that S is continuously growing and historical data are lost. Hence, the weighted moving standard deviation $\sigma(S_t)$ of stream S at time t has to be estimated using the following formula:

$$\sigma(S_t) = \sqrt{\frac{\omega}{(2-\omega)*n} * \sigma(S'_t)} \tag{3.15}$$

where ω is the weight used in Equation 3.13, $\sigma(S'_t)$ is the weighted moving standard deviation calculated at time t', and n is the the number of data elements in $S(t', t]$.

Since a F-test is commonly used for testing if standard deviations from two data set differ significantly, the control limits for detecting standard deviation changes are designed as following:

$$\begin{aligned} UCL_\sigma^t &= \sigma(S_t) * \sqrt{F_\tau(freedom1; freedom2)} \\ LCL_\sigma^t &= \sigma(S_t) * \sqrt{F_{1-\tau}(freedom1; freedom2)} \end{aligned} \tag{3.16}$$

where τ is the significance level, and $freedom1$ and $freedom2$ are the user-specified degree of freedoms in F-distribution.

Detecting changes

When data from stream S first arrive, a set of data are selected for use as learning samples and control charts C_μ and C_σ are constructed using the method discussed in the previous section. Once C_μ and C_σ are built and all parameters are tuned, a tumbling window W is used, with tumbling interval Δ, to capture the most current substream and feed it to C_μ and C_σ as a set of new samples.

Let t be the current timestamp. W contains data that have arrived in $(t - \Delta, t]$ time interval. Each time W moves, the new sample set will be evaluated and the evaluation results will indicate whether S is in control (i.e., distribution is stable) or it is out-of-control (i.e., distribution has changed). After the evaluation, all data inside W are removed, making room for new data.

Let t_a be the last time distribution change was detected. If the distribution has not changed since the beginning of S, then $t_a = t_0$. Null hypothesis H_0 that asserts distribution of S does not change is refuted iff the distance between previous and present mean values or standard deviation is above the threshold. When H_0 is refuted, a distribution change is noted and t_a is set to t.

One of the keys in the proposed change detection technique is to find a proper window tumbling interval Δ. A smaller interval implies frequent evaluation and, hence, the distribution change can be quickly reported. In the extreme case when W moves every time a new element arrives, the tumbling window model is equal to a sliding window model. However, frequent evaluation increases computation cost significantly, since $\mu(S_t)$, $\sigma(S_t)$ and all the control limits need to be recalculated upon every update. A large Δ may reduce the number of true changes detected, because some rapid distribution changes could be missed; even if a change will be detected eventually, there will be a long delay. The effect of Δ is empirically studied in Section 3.4.4.

Full algorithm

The full algorithm of the proposed approach is given in Algorithm 1.

The time complexity of Algorithm 1 is linear $O(n)$, where n is the number of data received within each Δ time period, hence, the proposed approach is very efficient.

3.4.4 Experiments

To evaluate the performance of proposed control chart based change detection technique, a series of experiments are conducted. The data streams used in the experiments are generated using the experimental framework discussed in Section 3.3.5. The criteria for evaluating experimental results is the same as in Section 3.3.5. All experiments are conducted on a PC with 3GHz Pentium 4 processor and 1GB of RAM, running Windows XP system. All programs are implemented in R.

Change detection evaluation

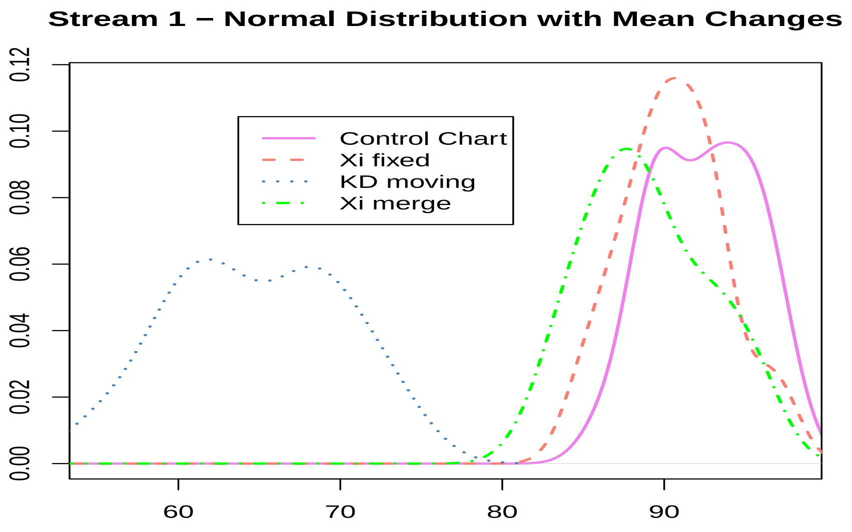
The control chart based approach is compared with three change detection techniques discussed in Section 3.3.5: XI-fixed, KD-moving, and XI-merged. Each synthetic stream contains 100,000 data that arrive at an even speed. There are total 100 changes in each stream. Significance level τ is set to 80%. Sizes of all windows, W used in the control chart based technique and W_r and W_t used in the three comparing techniques, are set to 100 data. The number of partitions k used in the merged window method is set to 10. The total number of streams generated and tested for each stream type is $numRun = 100$.

A set of experiments is conducted using stream types $Stream1, ...,$ $Stream12$ described in Table 3.1. The results are presented in sets of figures with x-axis being values of the five criteria discussed in Section 3.3.5 and y-axis being the density of these criteria. Figures 3.79 – 3.84 illustrate the number of true changes detected in each type of stream. Figures 3.85 – 3.90 show the number of false changed detected. The mean

Algorithm 1 Mean and Standard Deviation Detection using Control Charts

```
1: INPUT: Data stream $S$
2:          Significance level $\tau$
            Degree of freedom $freedom1$ and $freedom2$
3: Determine $\omega$ and $\kappa$ based on $\tau$;
4: Record timestamp $t_a = t_0$;
5: while $t - t_0 < \Delta$ do
6:     Collect data into $W$;
7:     Record timestamp $t$;
8: end while
9: $\mu(S'_t) = \mu(S[t_0, t])$;
10: $\sigma(S'_t) = \sigma(S[t_0, t])$;
11: while $S$ does not terminate do
12:     $t' = t$;
13:     while $t - t' < \Delta$ do
14:         Collect data into $W$;
15:         Record timestamp $t$;
16:     end while
17:     $\mu(S_t) = \mu(S(t', t])$;
18:     $\sigma(S_t) = \sigma(S(t', t])$;
19:     Calculate $UCL^t_\mu$ and $LCL^t_\mu$ using Equation 3.14;
20:     Calculate $UCL^t_\sigma$ and $LCL^t_\sigma$ using Equation 3.16;
21:     if $\mu(S_t) > UCL^t_\mu$ and $\mu(S_t) < LCL^t_\mu$ and $\sigma(S_t) > UCL^t_\sigma$ and
        $\sigma(S_t) > UCL^t_\sigma$ then
22:         Report distribution change;
23:         //Reset initial mean and standard deviation
24:         $t_a = t'$;
25:         $\mu(S'_t) = \mu(S(t', t])$;
26:         $\sigma(S'_t) = \sigma(S(t', t])$;
27:     else
28:         //Update EWMA mean and standard deviation
29:         Calculate $\mu(S_t)$ using Equation 3.13;
30:         Calculate $\sigma(S_t)$ using Equation 3.15;
31:     end if
32: end while
```

values of the change detection durations for each stream type is shown in Figures 3.91 – 3.96. The standard deviations of the change detection durations are demonstrated in Figures 3.97 – 3.102. Figures 3.103 – 3.108 present the maximal durations for detecting changes in each stream type.



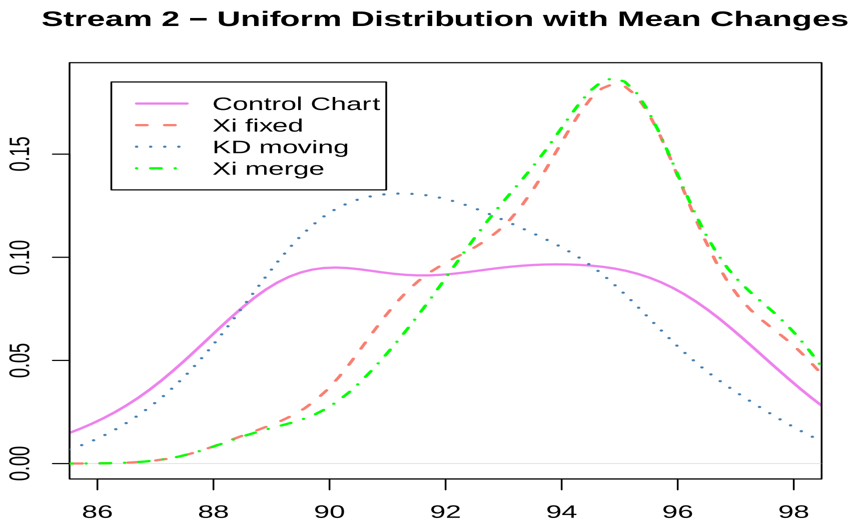
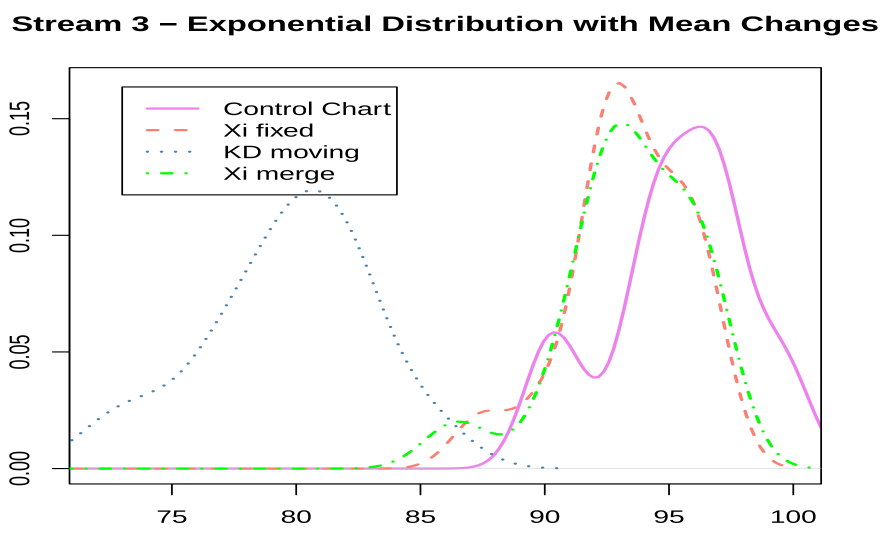


Figure 3.79: Number of true changes detected for $Stream1$ and $Stream2$



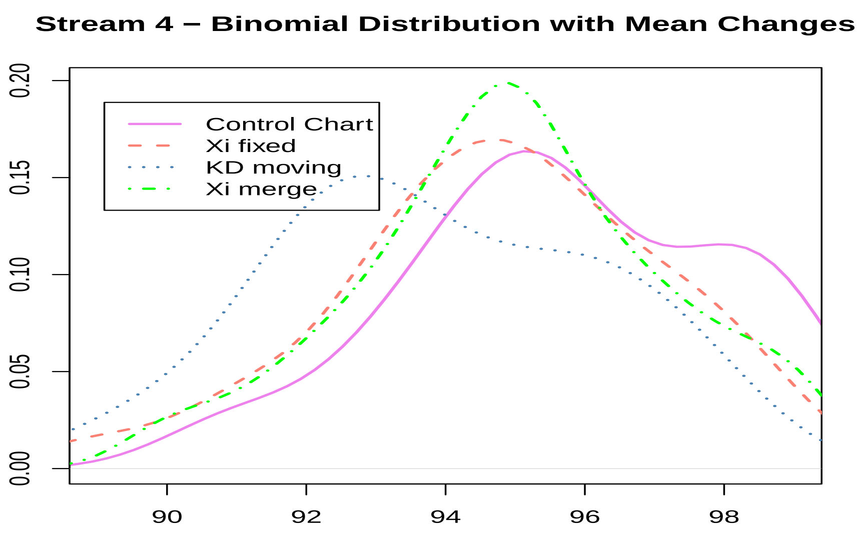
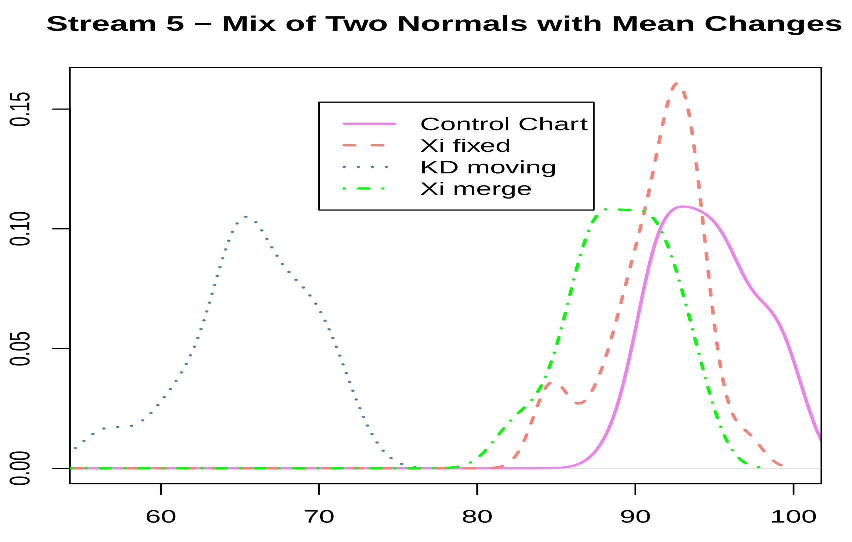


Figure 3.80: Number of true changes detected for *Stream*3 and *Stream*4



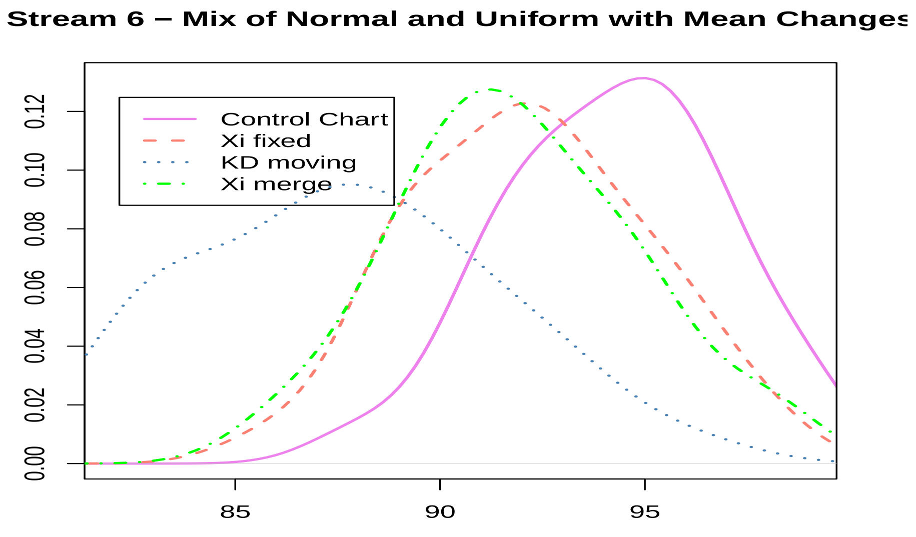
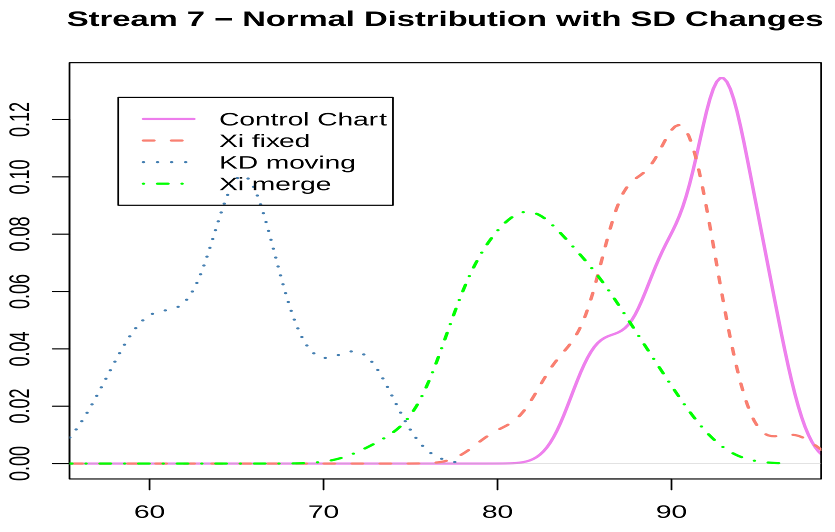


Figure 3.81: Number of true changes detected for *Stream*5 and *Stream*6



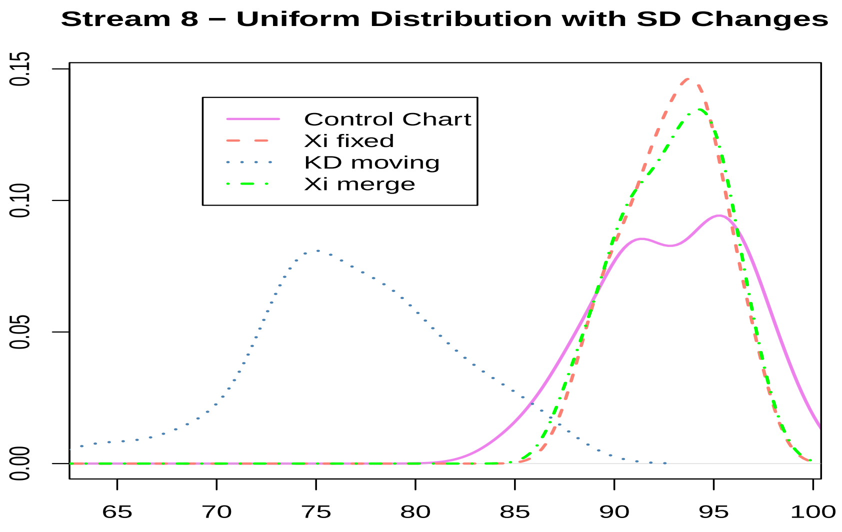
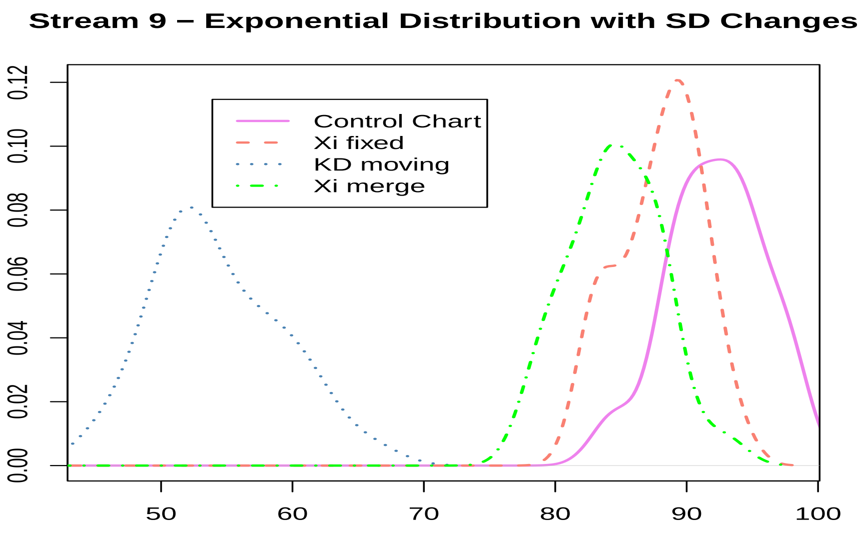


Figure 3.82: Number of true changes detected for *Stream*7 and *Stream*8



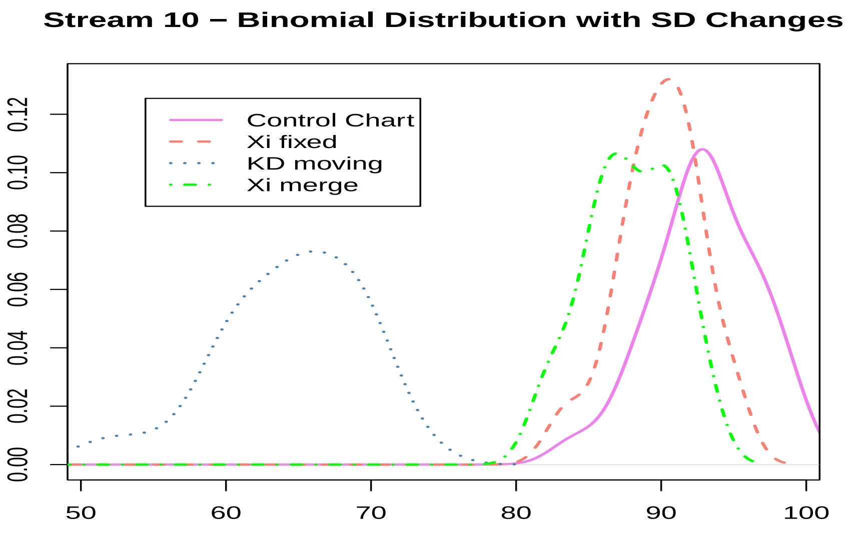
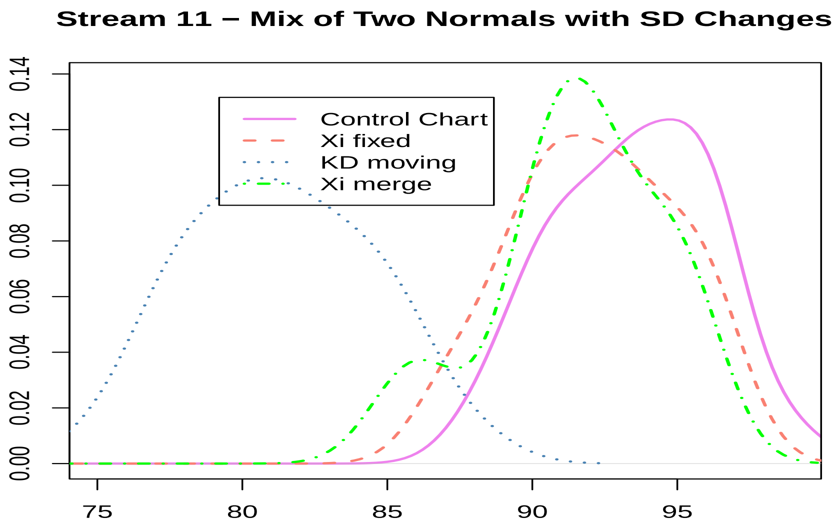


Figure 3.83: Number of true changes detected for $Stream9$ and $Stream10$



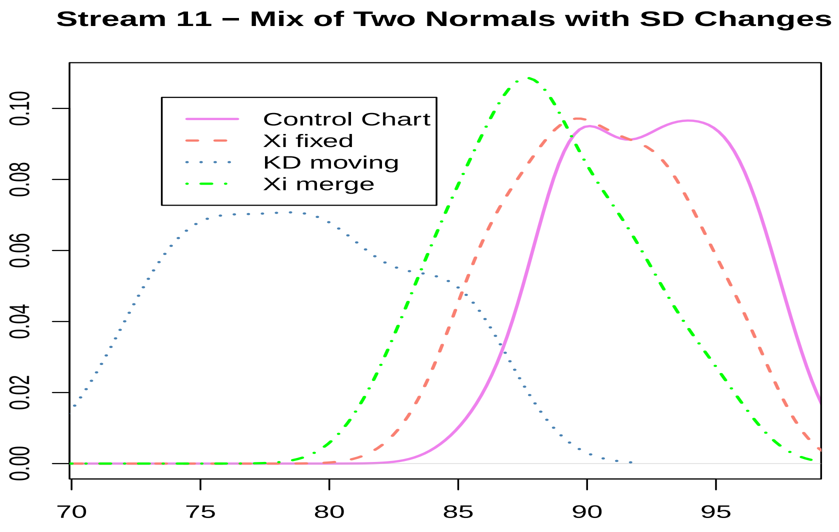
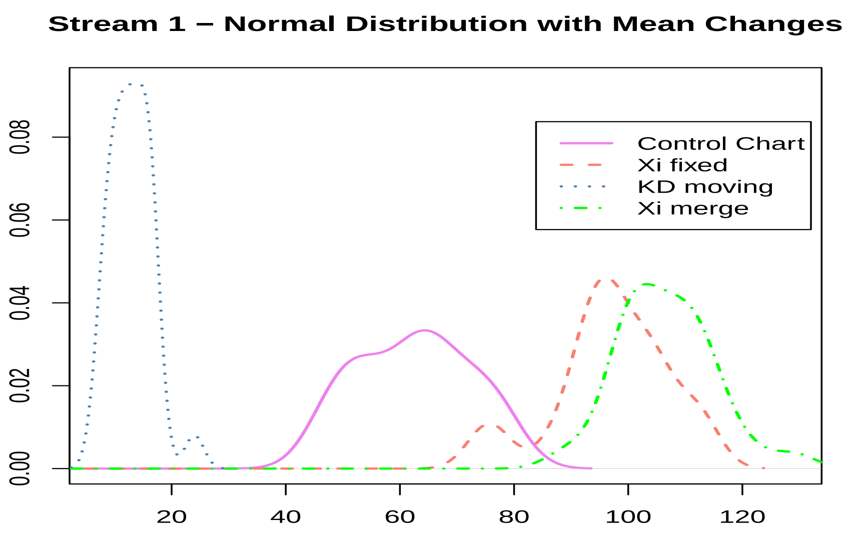


Figure 3.84: Number of true changes detected for $Stream11$ and $Stream12$



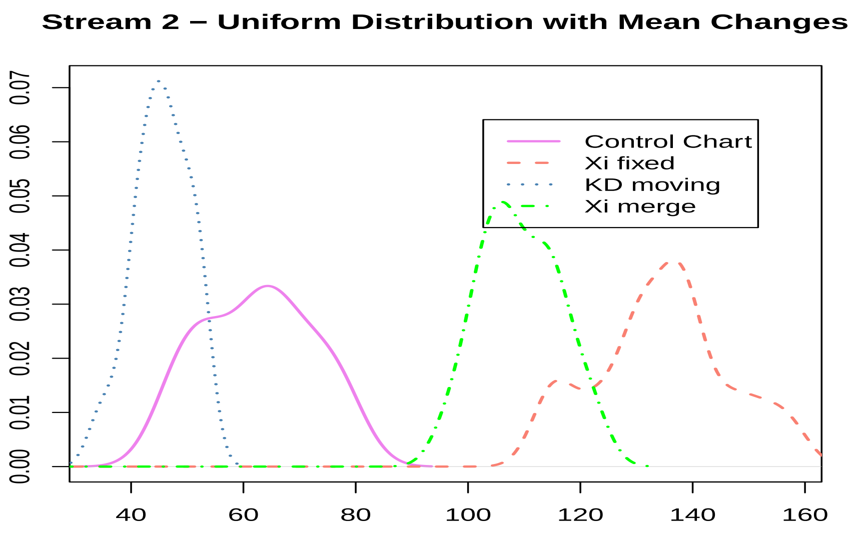
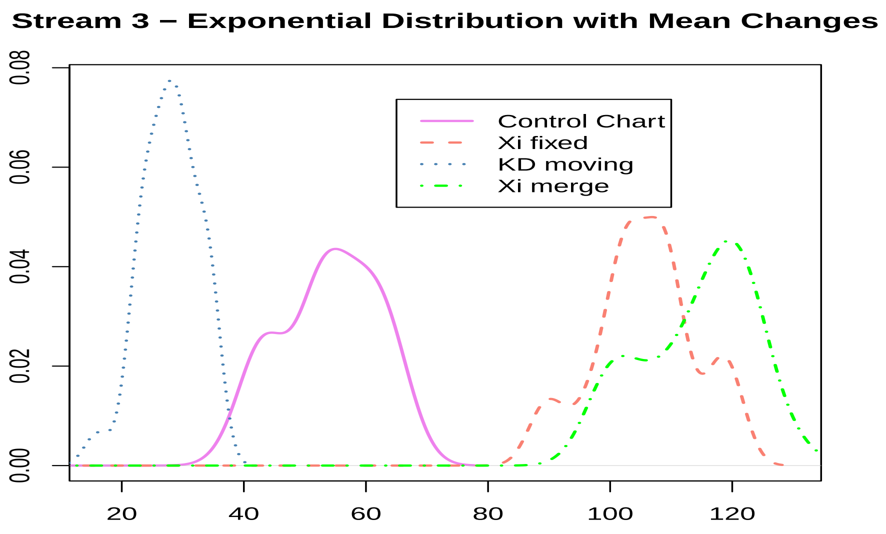


Figure 3.85: Number of false changes detected for $Stream1$ and $Stream2$



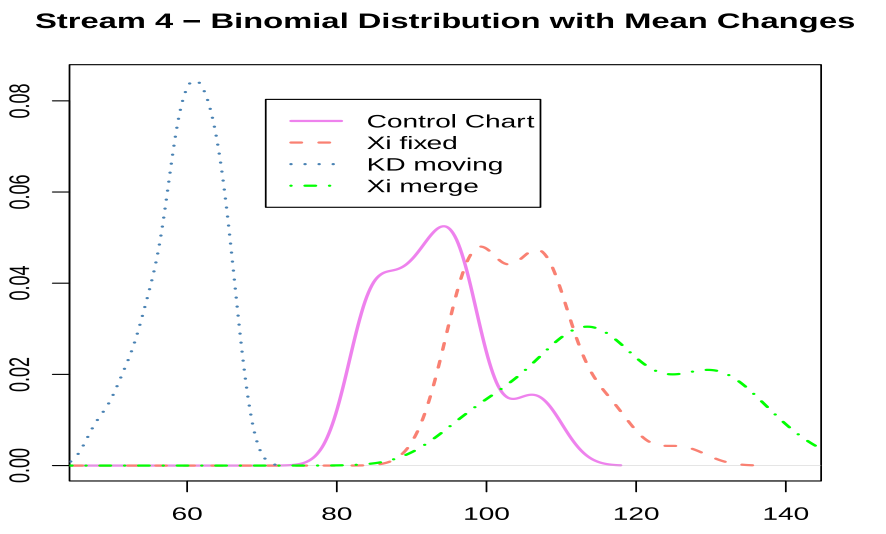
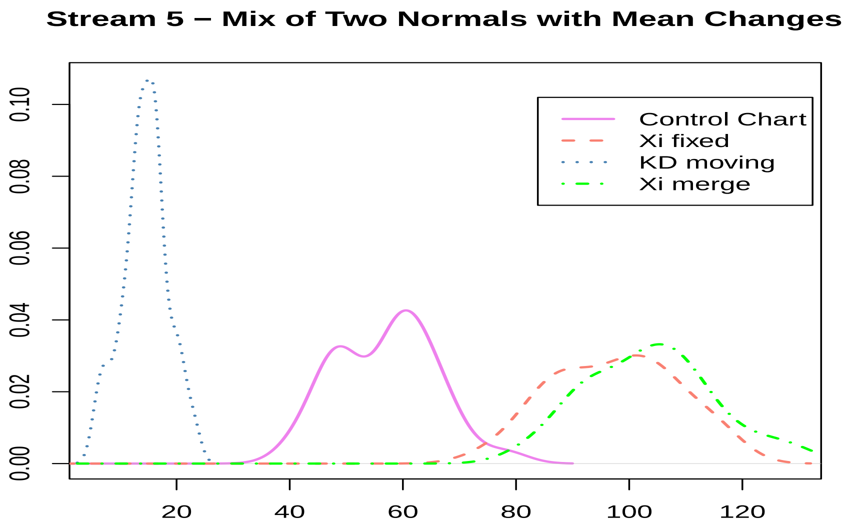


Figure 3.86: Number of false changes detected for *Stream*3 and *Stream*4



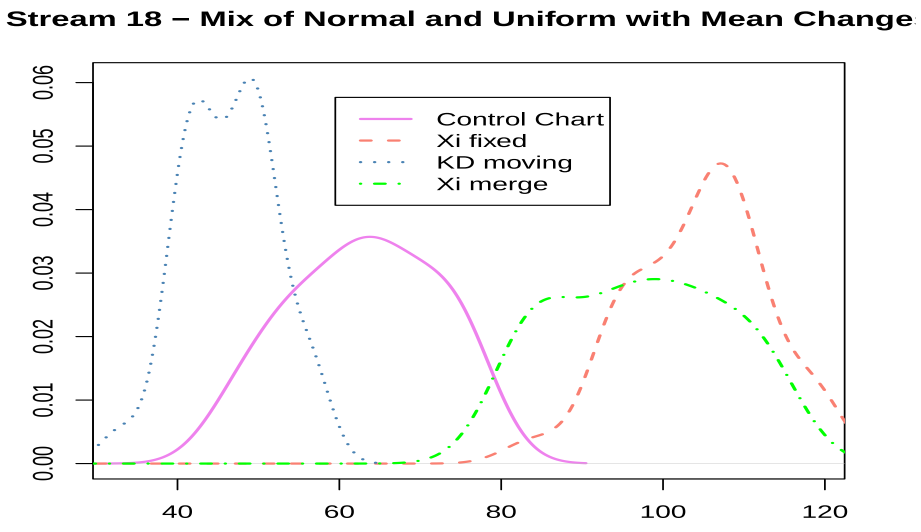
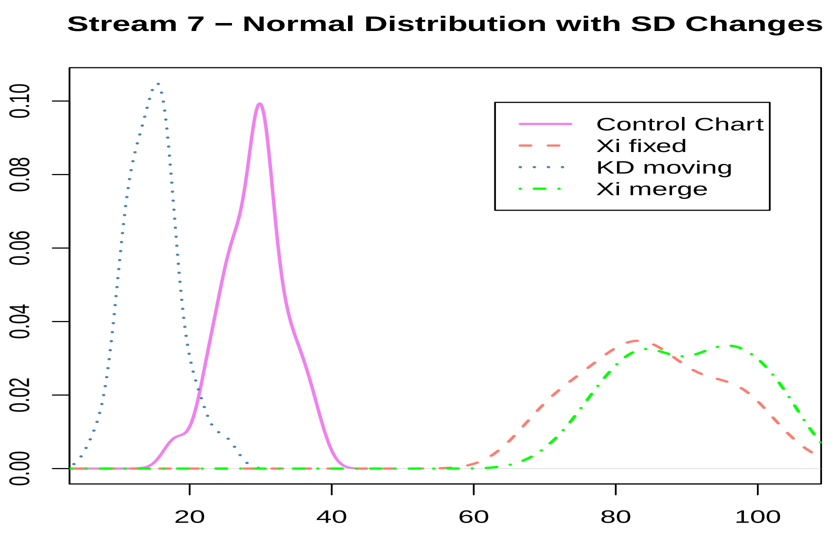


Figure 3.87: Number of false changes detected for $Stream5$ and $Stream6$



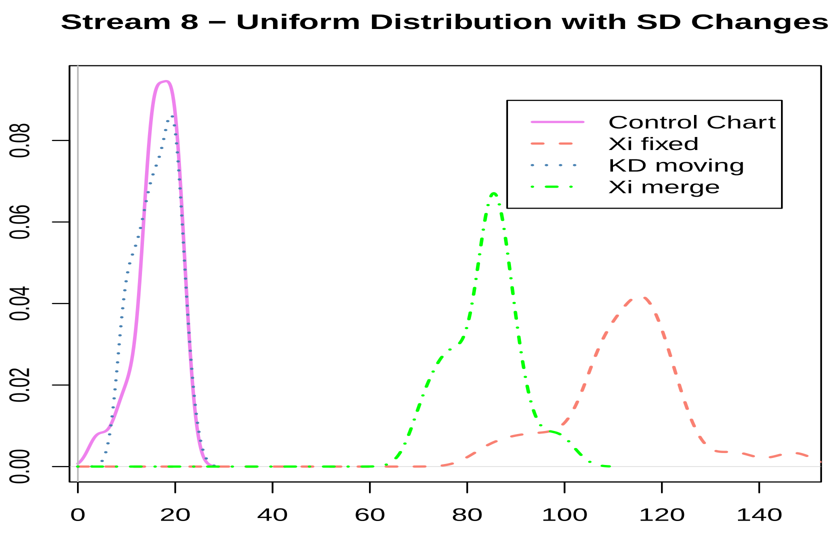
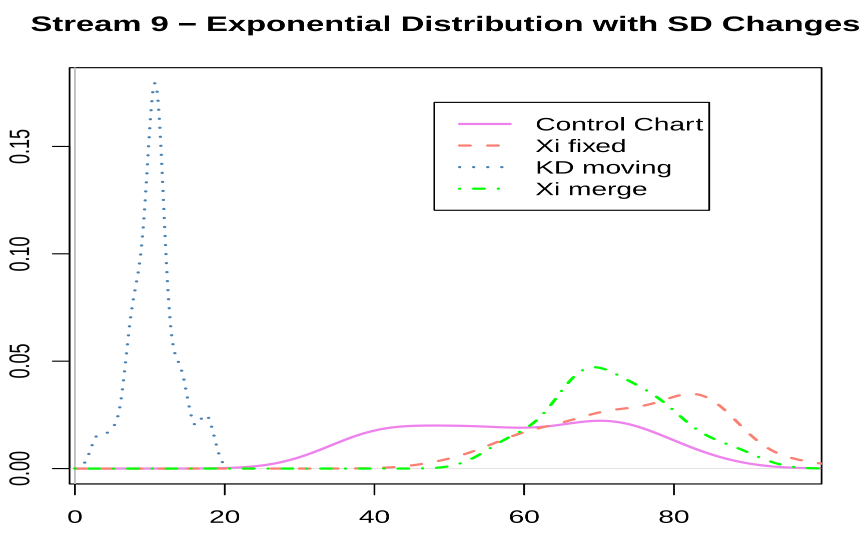


Figure 3.88: Number of false changes detected for *Stream*7 and *Stream*8



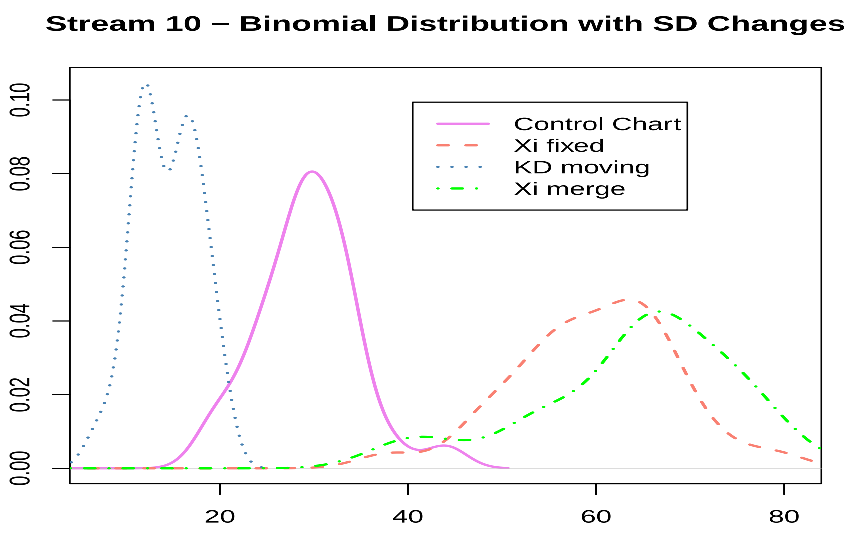
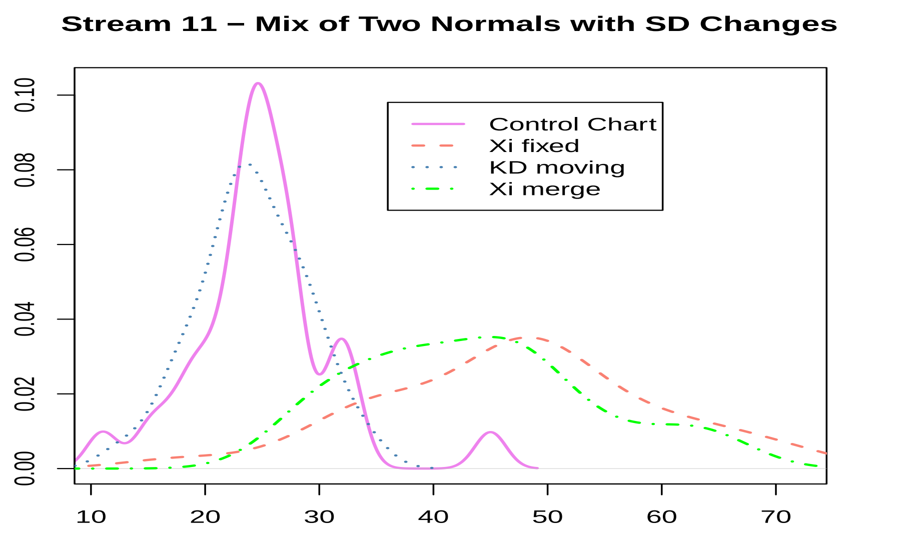


Figure 3.89: Number of false changes detected for $Stream9$ and $Stream10$



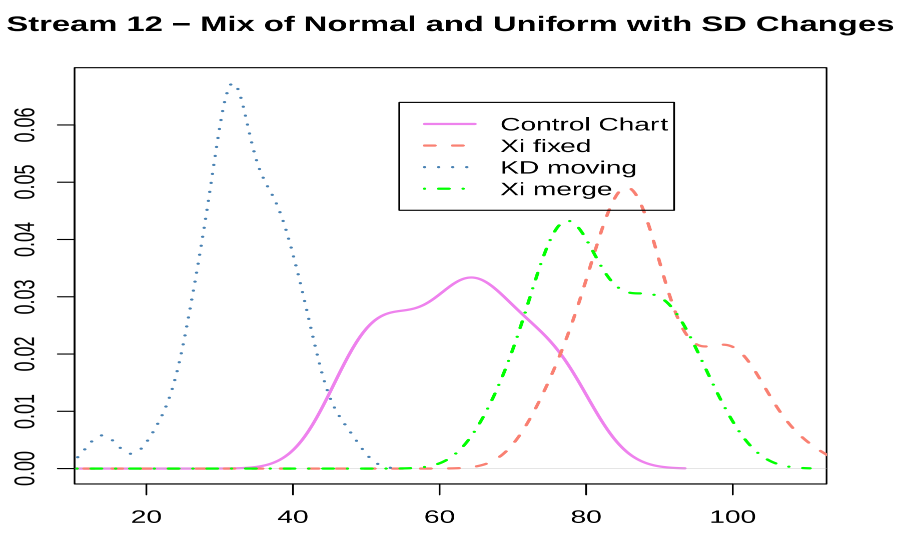
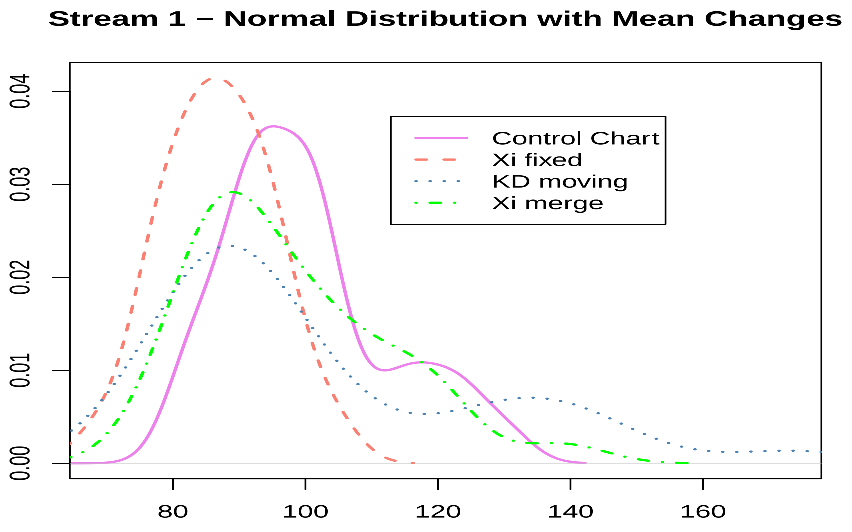


Figure 3.90: Number of false changes detected for *Stream*11 and *Stream*12



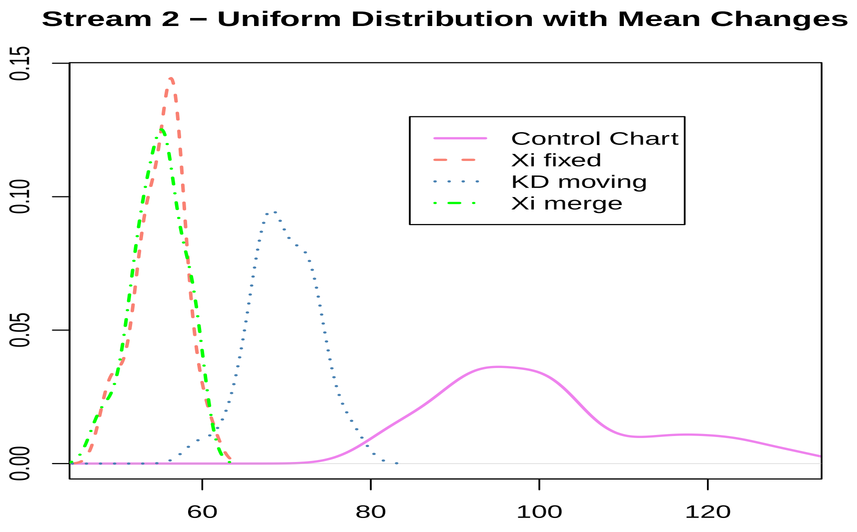
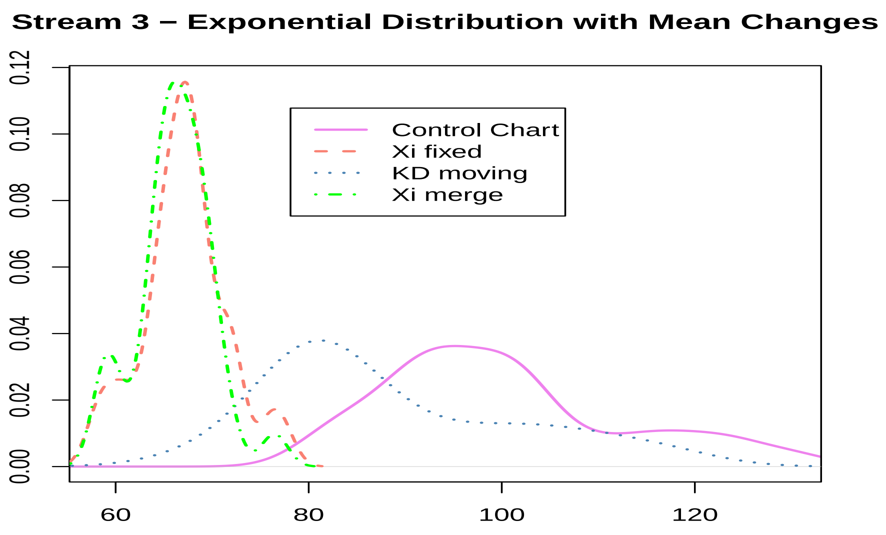


Figure 3.91: Mean duration for detecting true changes in *Stream*1 and *Stream*2



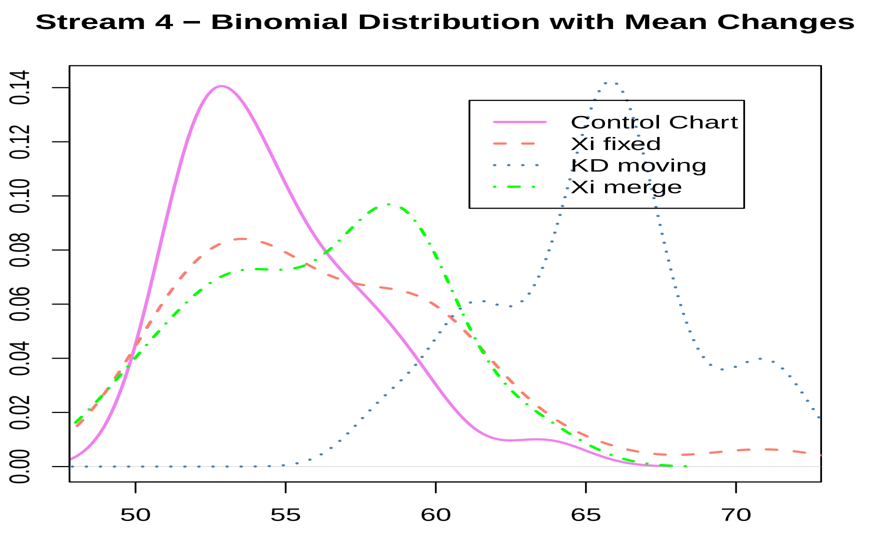
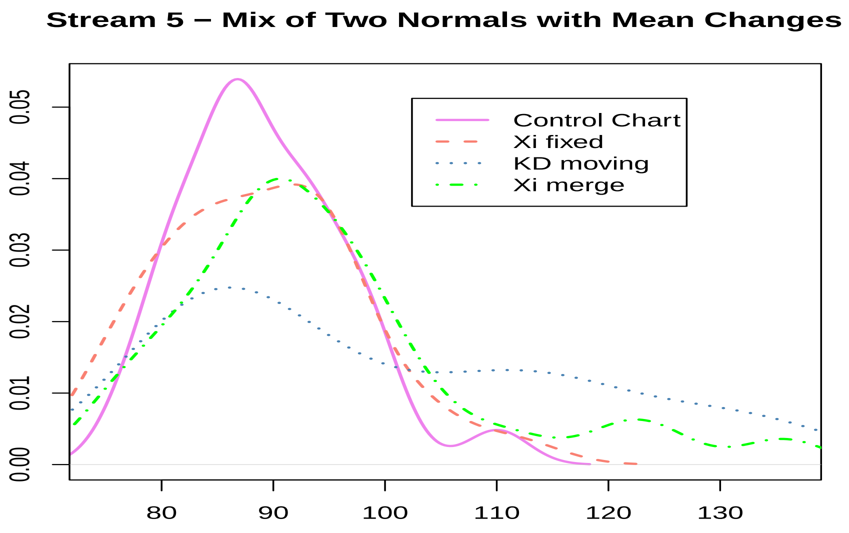


Figure 3.92: Mean duration for detecting true changes in *Stream*3 and *Stream*4



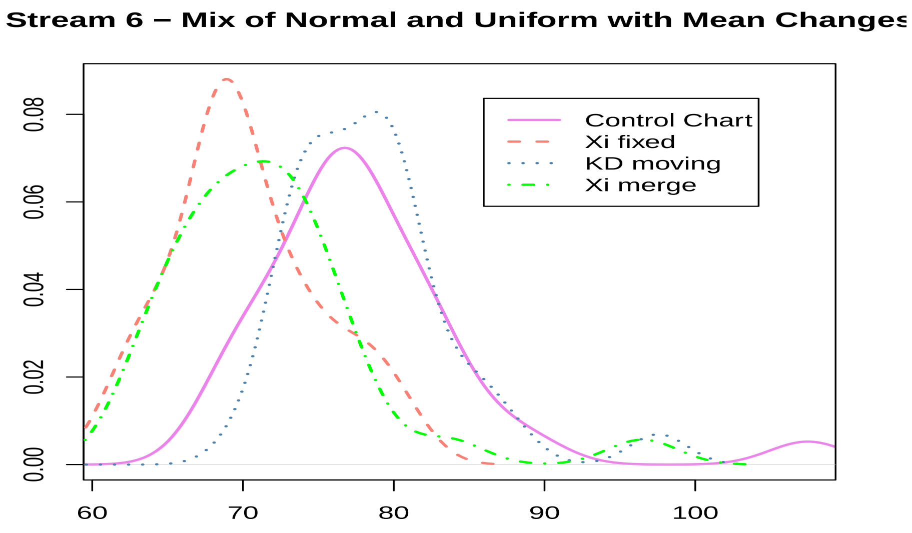
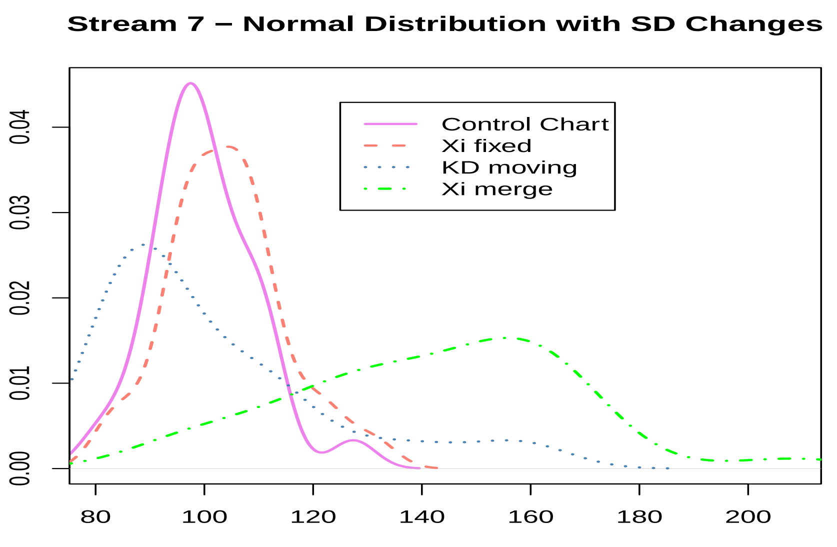


Figure 3.93: Mean duration for detecting true changes in *Stream*5 and *Stream*6



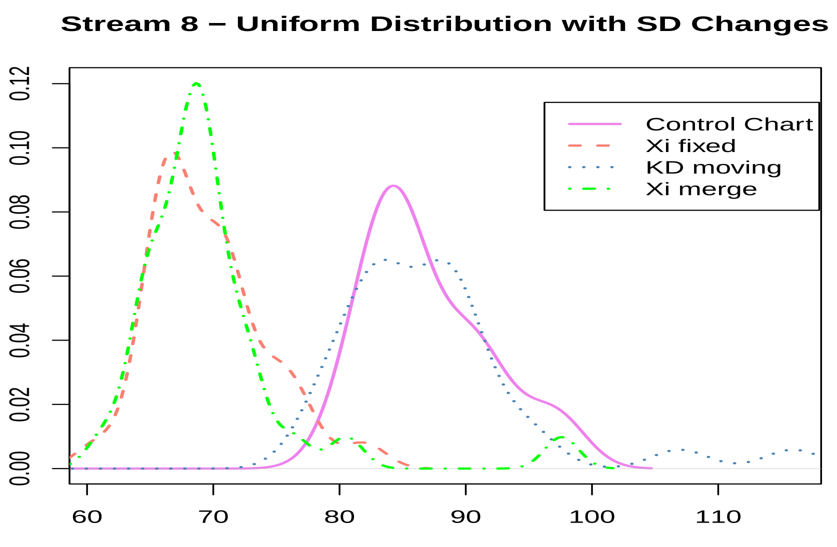
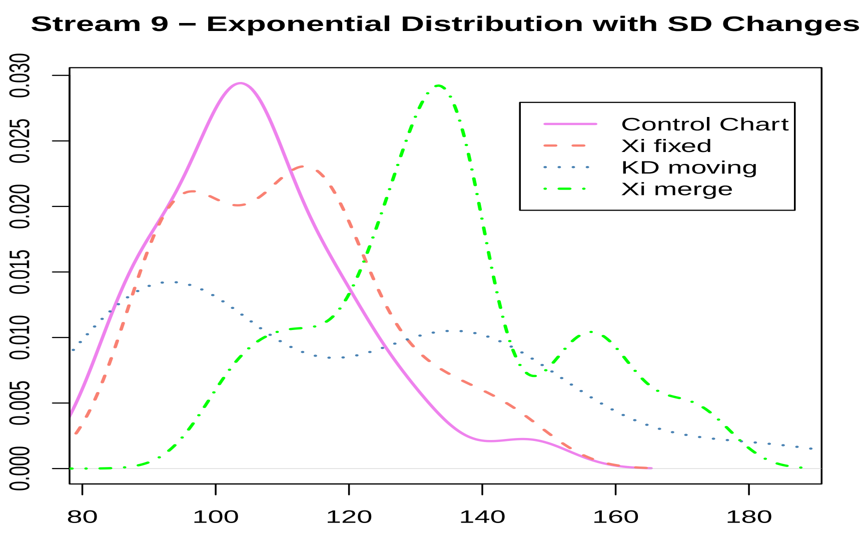


Figure 3.94: Mean duration for detecting true changes in $Stream7$ and $Stream8$



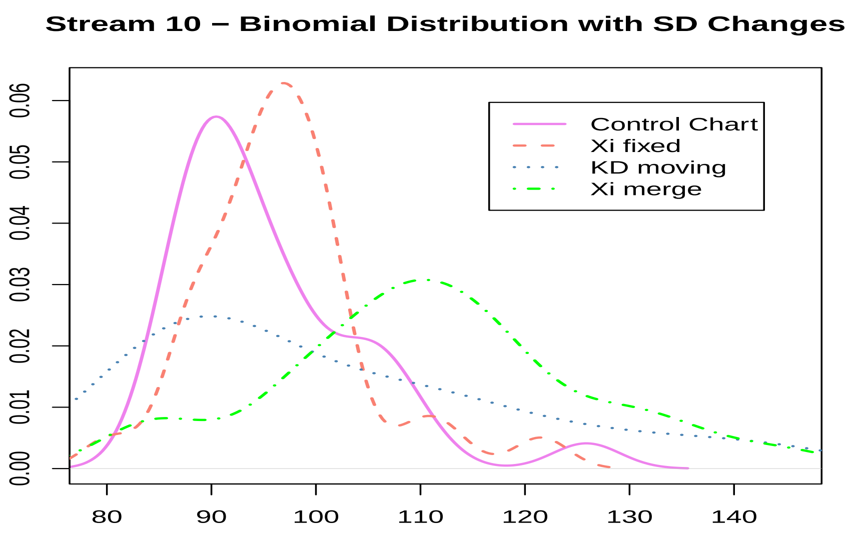
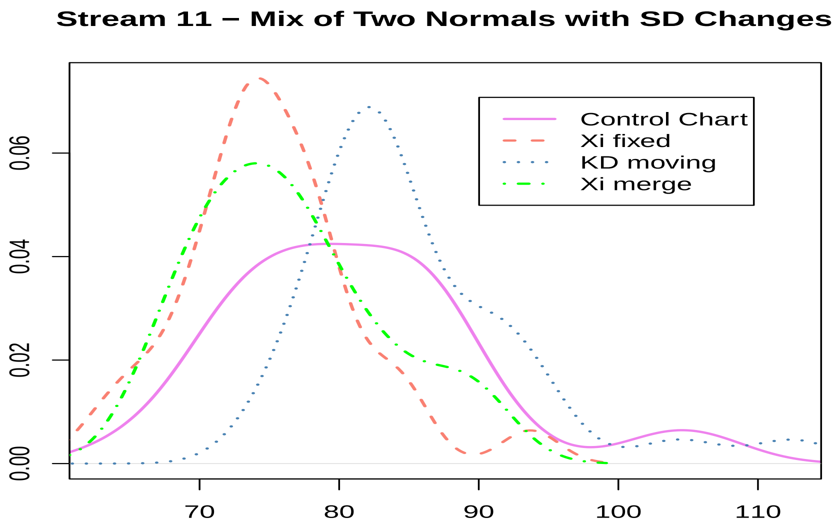


Figure 3.95: Mean duration for detecting true changes in $Stream9$ and $Stream10$



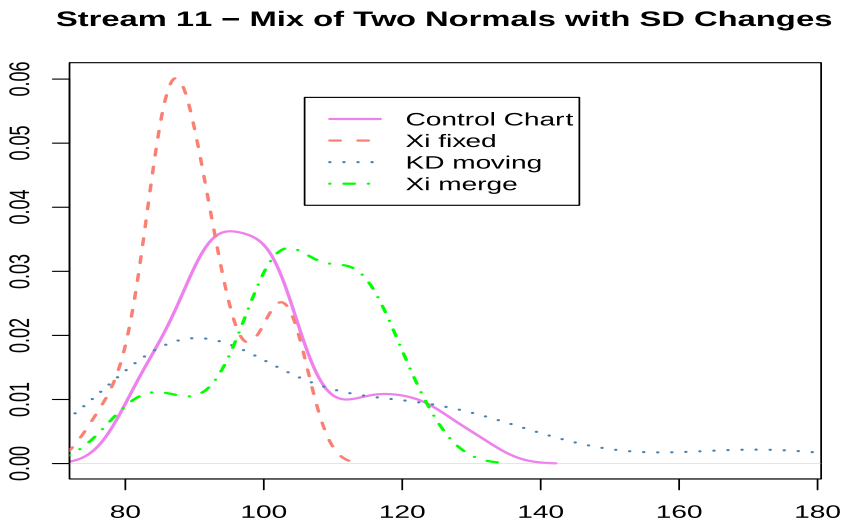
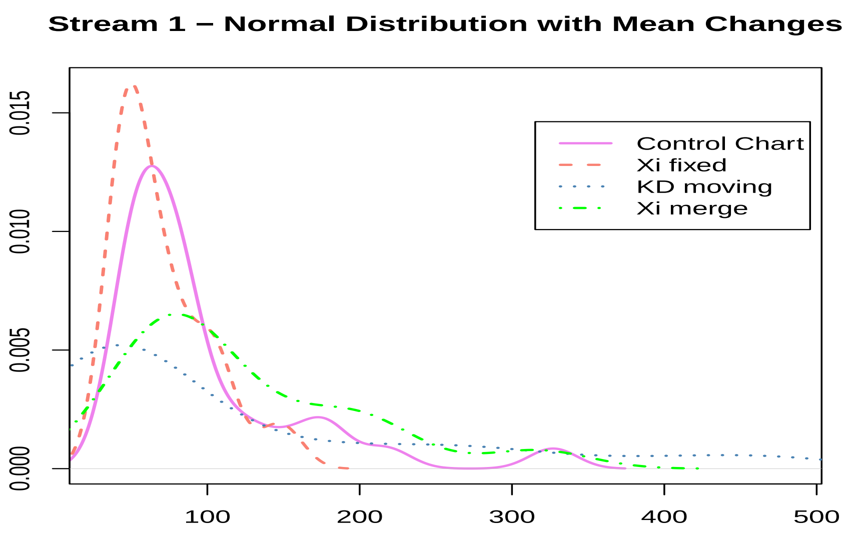


Figure 3.96: Mean duration for detecting true changes in $Stream11$ and $Stream12$



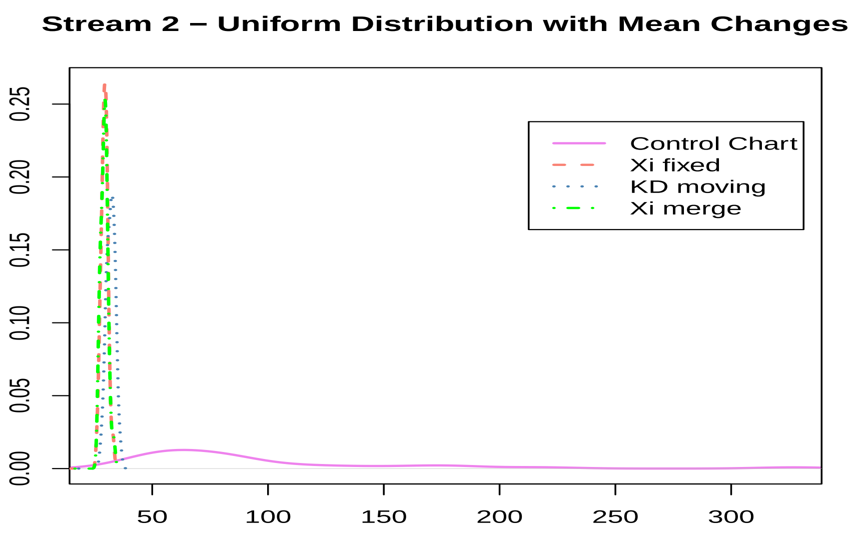
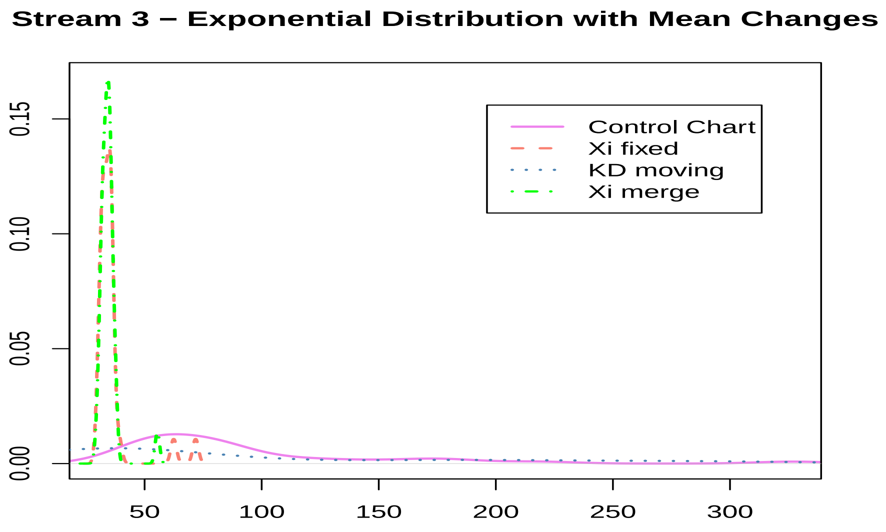


Figure 3.97: Standard deviation of the duration for detecting true changes in $Stream1$ and $Stream2$



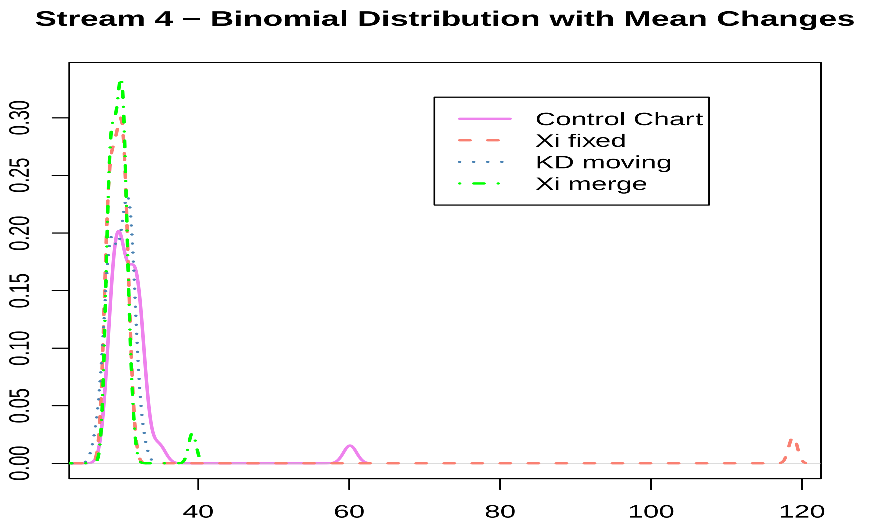
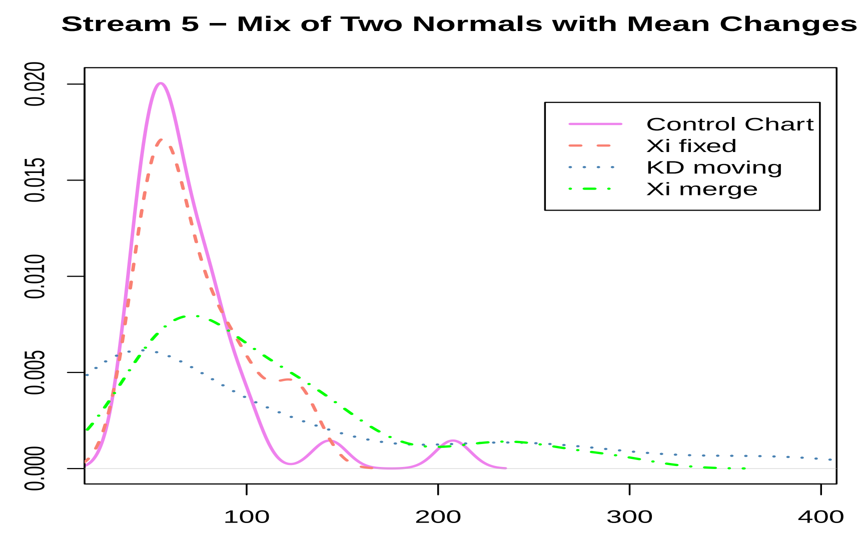


Figure 3.98: Standard deviation of the duration for detecting true changes in *Stream*3 and *Stream*4



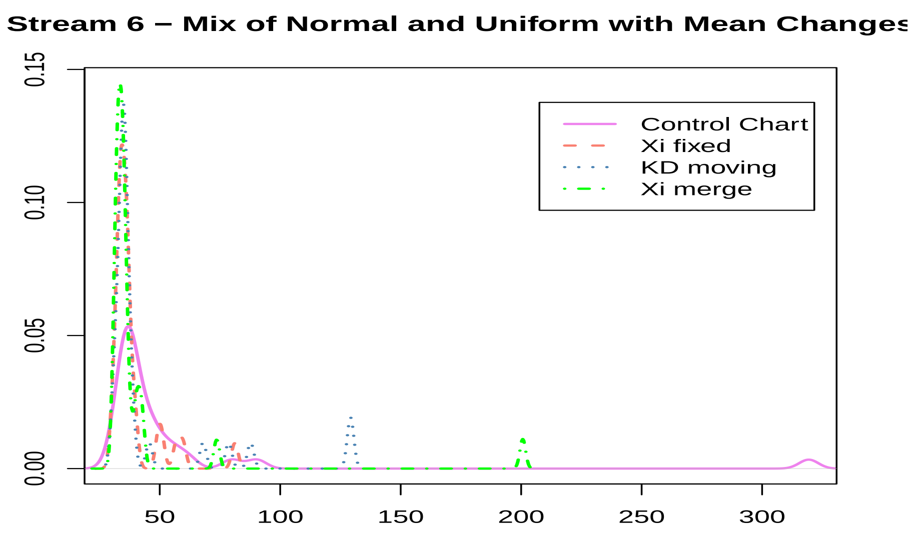
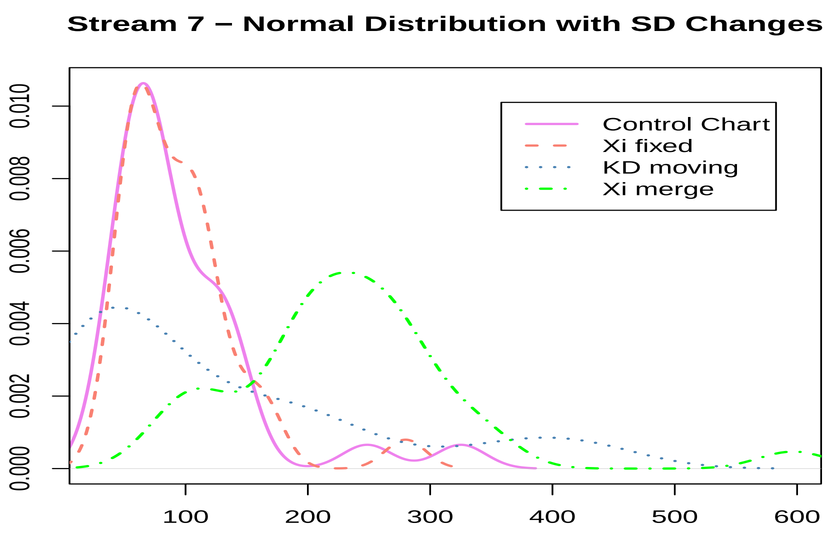


Figure 3.99: Standard deviation of the duration for detecting true changes in *Stream*5 and *Stream*6



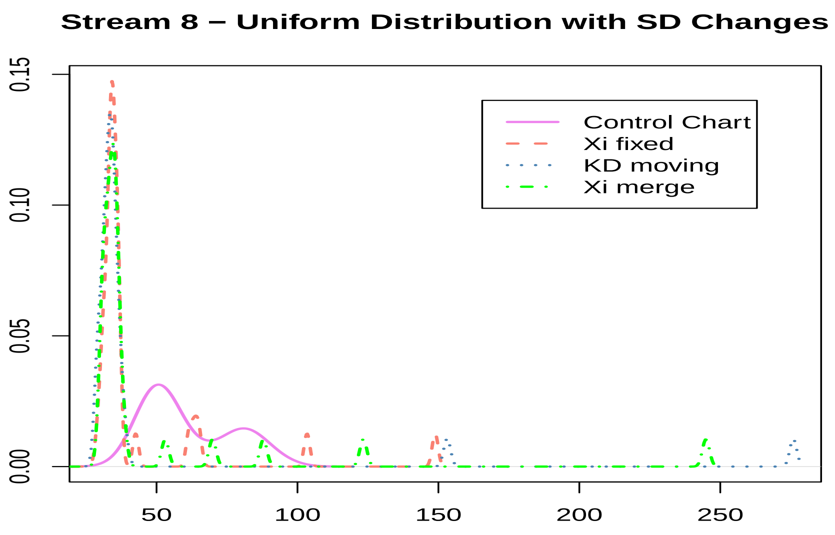
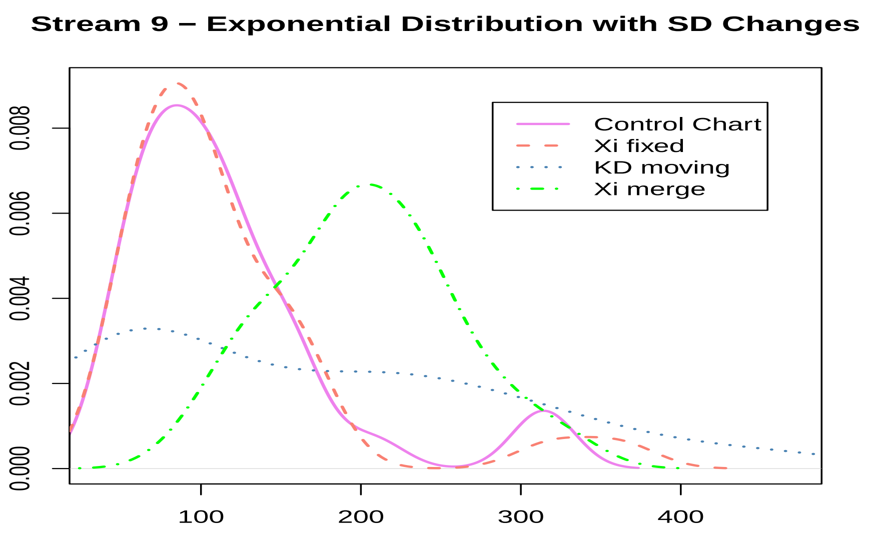


Figure 3.100: Standard deviation of the duration for detecting true changes in *Stream*7 and *Stream*8



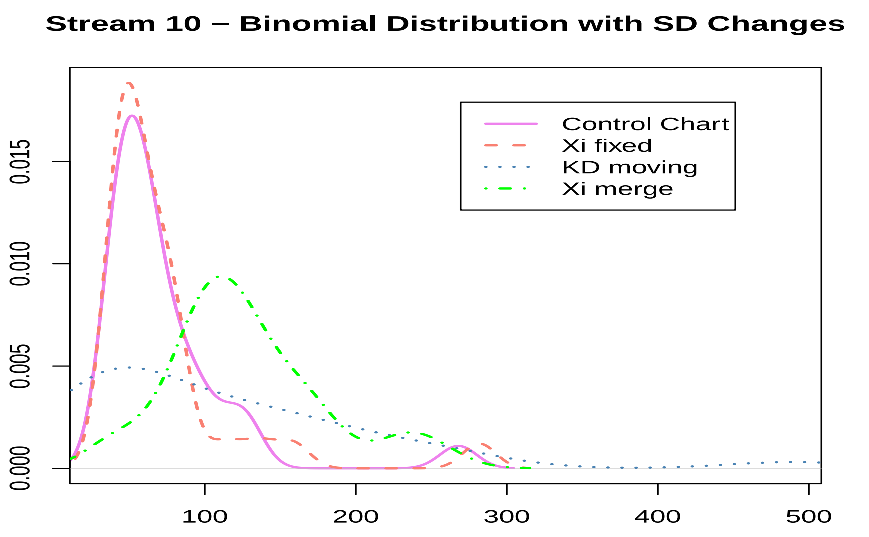
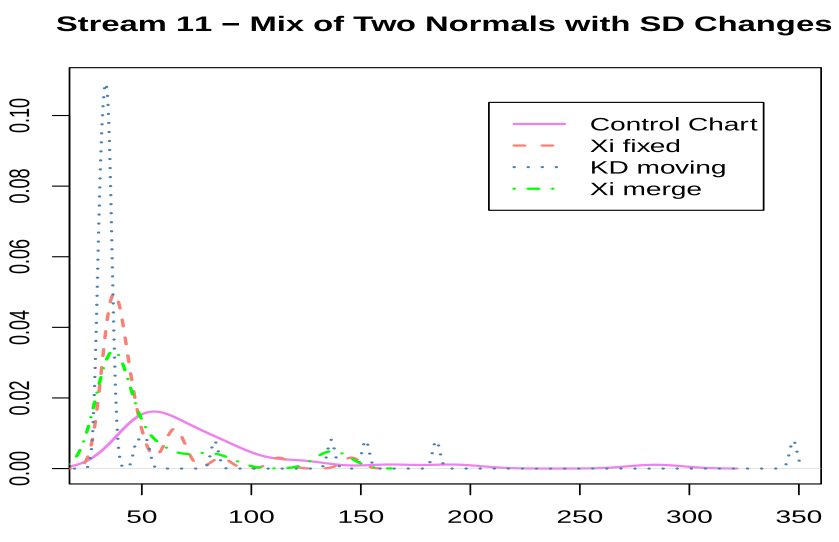


Figure 3.101: Standard deviation of the duration for detecting true changes in $Stream9$ and $Stream10$



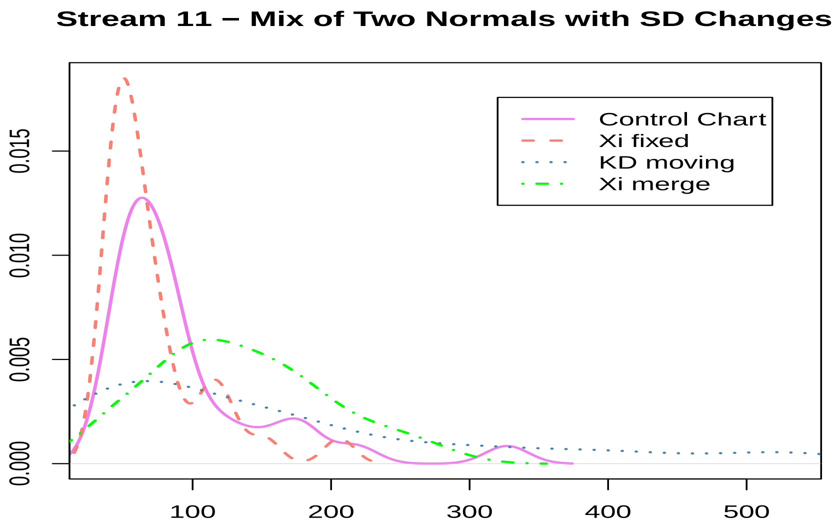
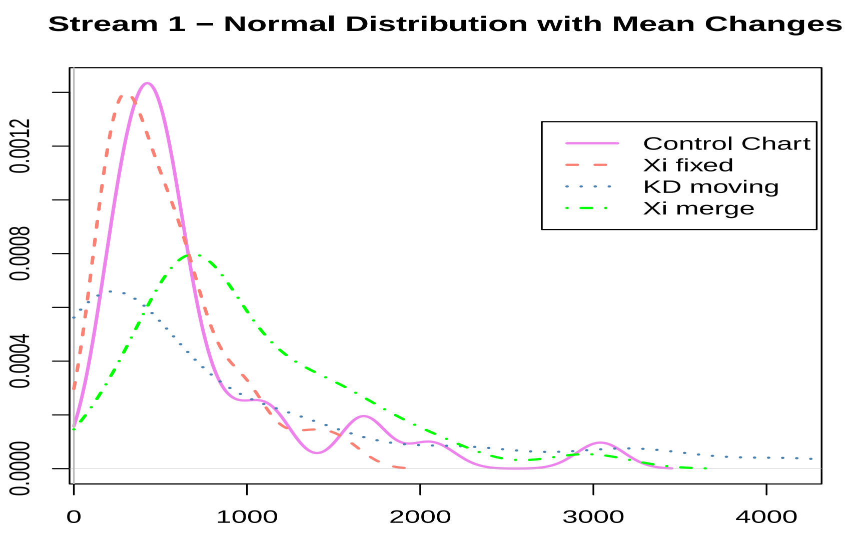


Figure 3.102: Standard deviation of the duration for detecting true changes in *Stream*11 and *Stream*12



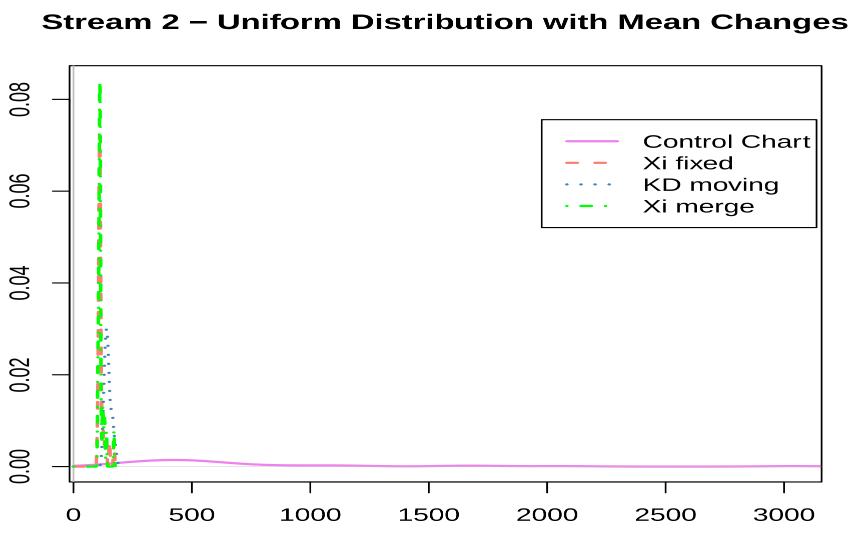
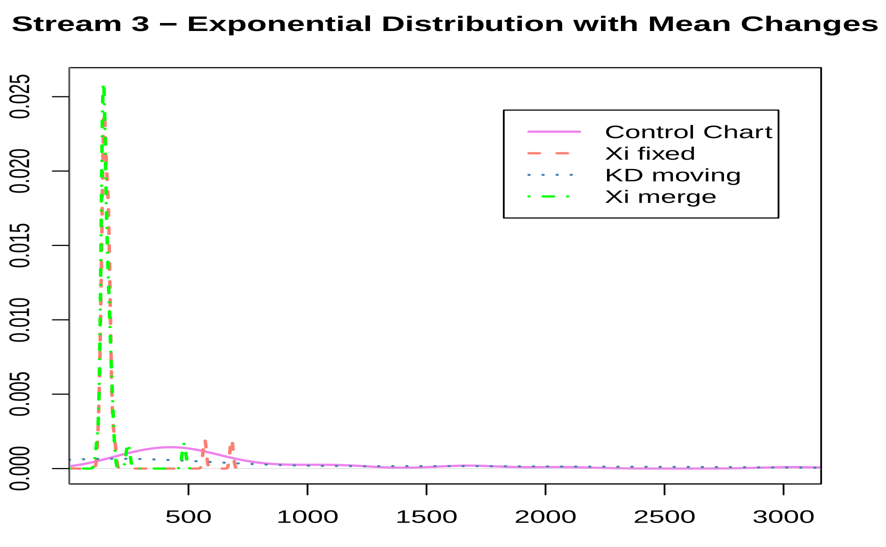


Figure 3.103: Max duration for detecting true changes in $Stream1$ and $Stream2$



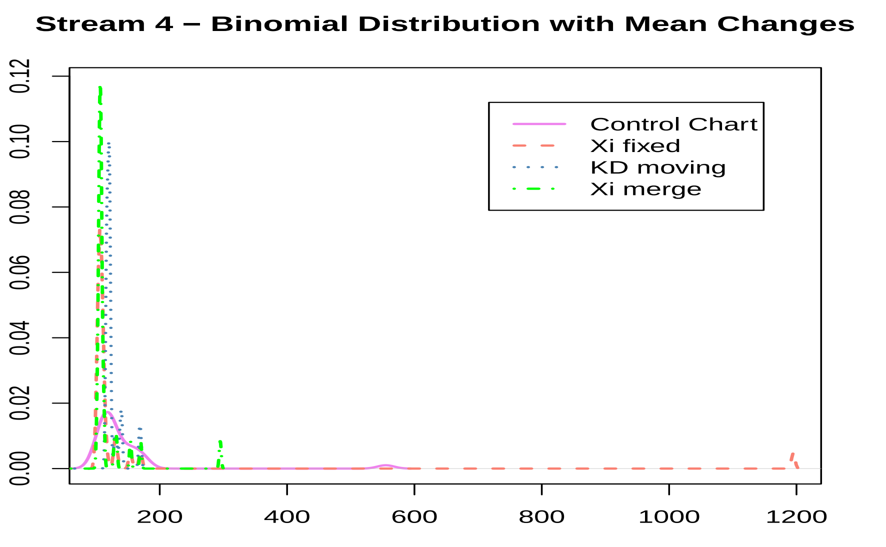
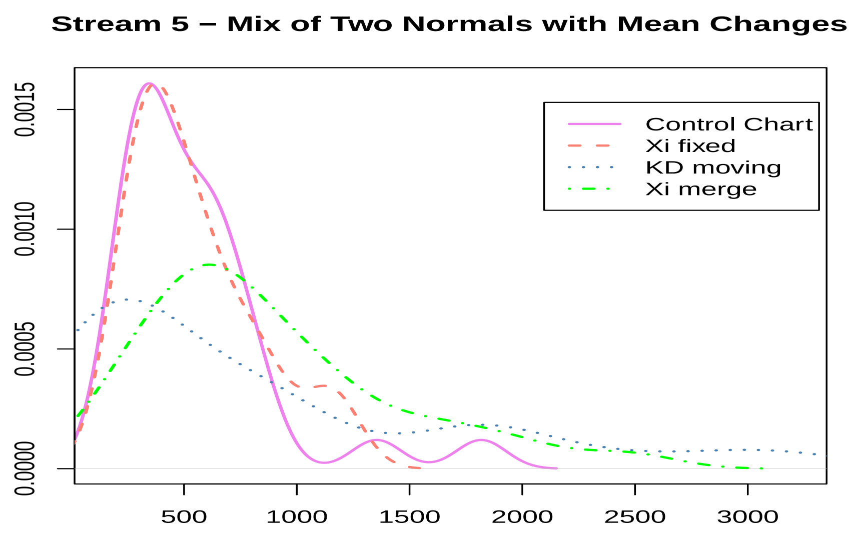


Figure 3.104: Max duration for detecting true changes in *Stream*3 and *Stream*4



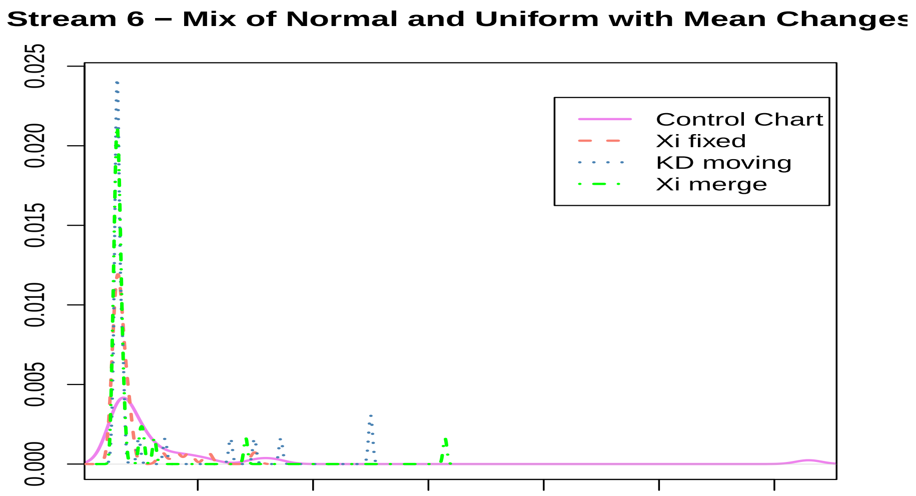
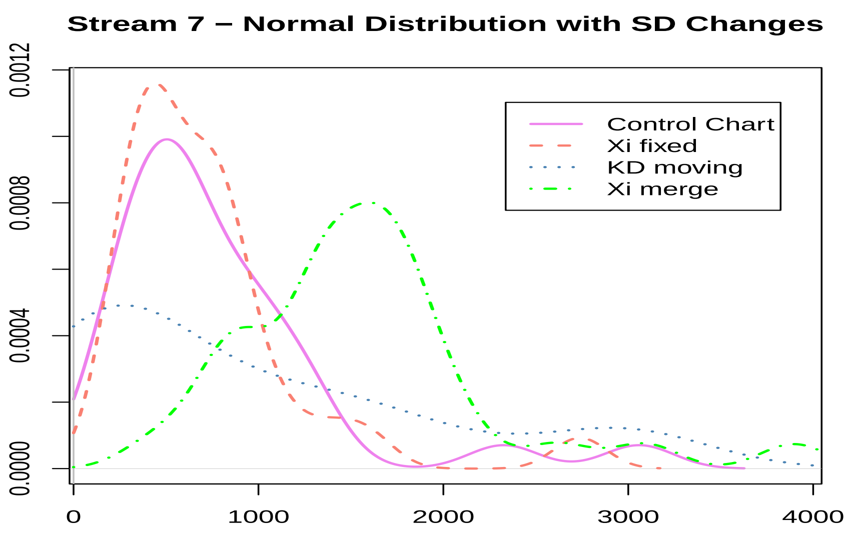


Figure 3.105: Max duration for detecting true changes in $Stream5$ and $Stream6$



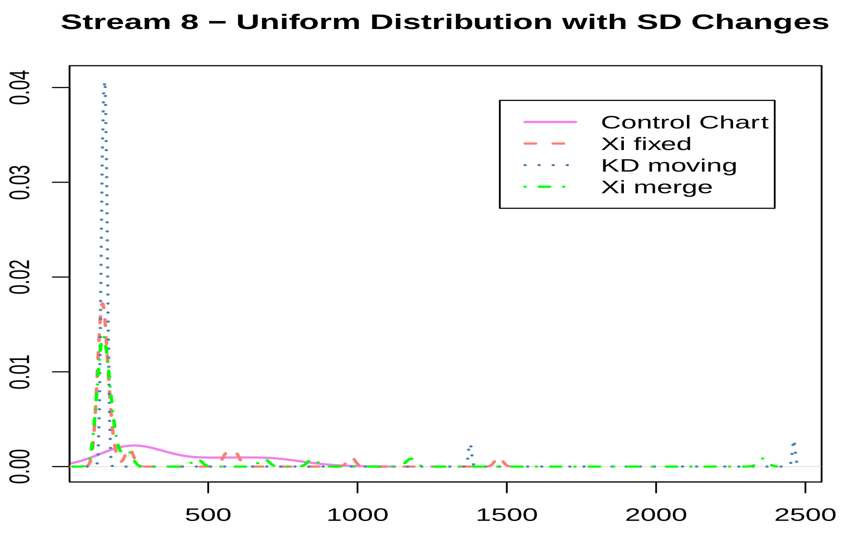
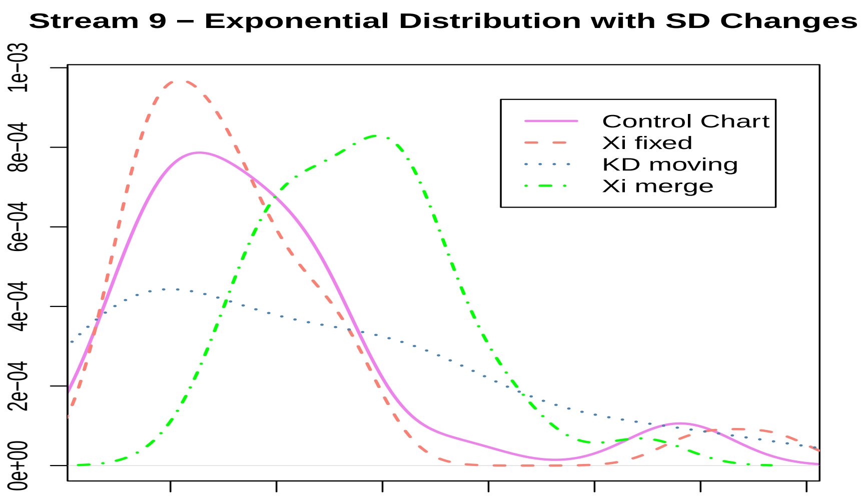


Figure 3.106: Max duration for detecting true changes in $Stream7$ and $Stream8$



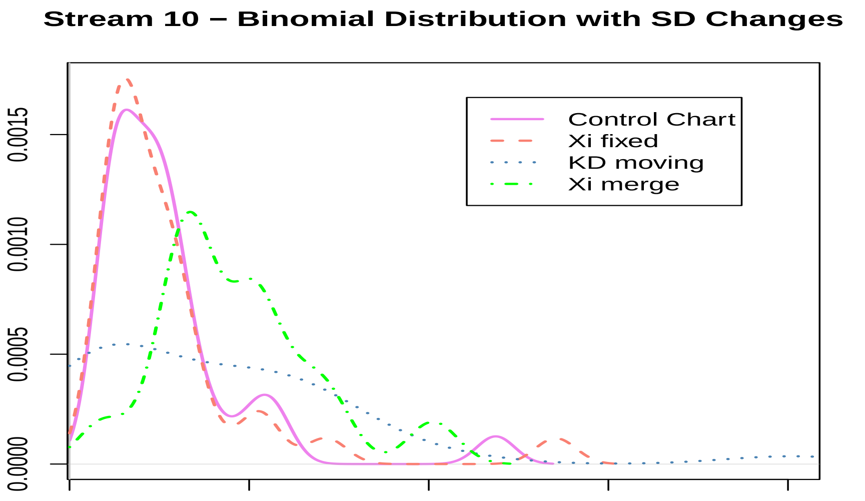
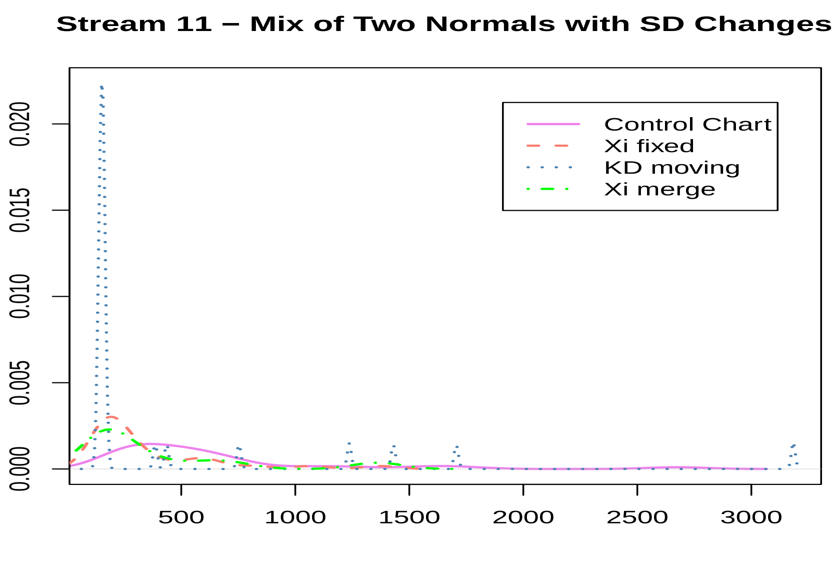


Figure 3.107: Max duration for detecting true changes in $Stream9$ and $Stream10$



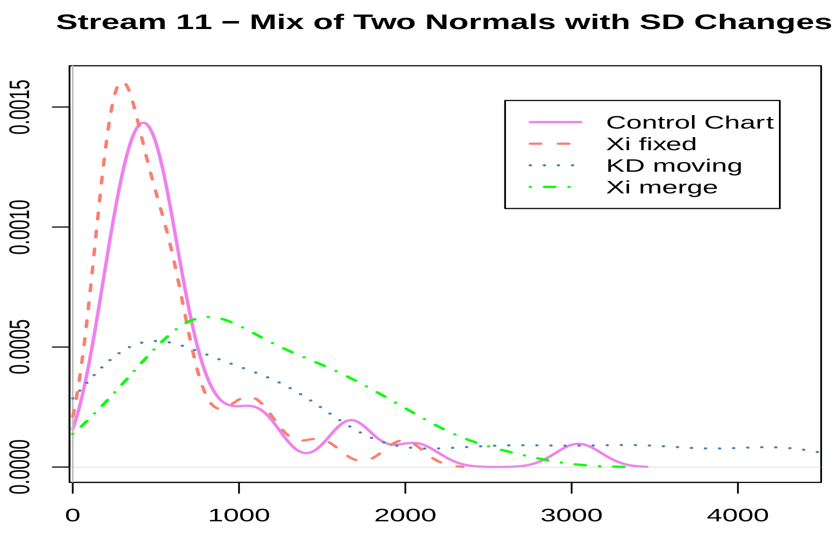


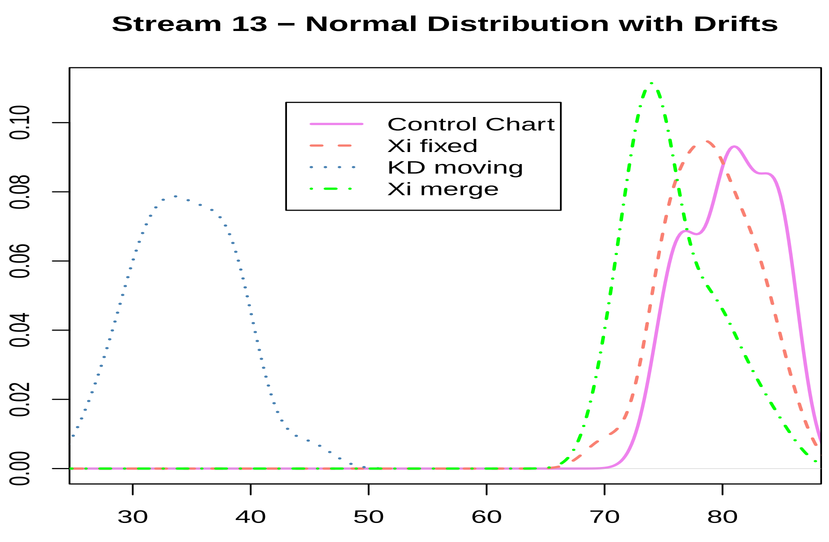
Figure 3.108: Max duration for detecting true changes in *Stream*11 and *Stream*12

These results reveal that the proposed control chart based technique can detect the most number of true changes among the four techniques. It detects more false changes than KD-moving, however, the number of false detection is significantly less than XI-fixed and XI-merged. The durations for detecting true changes using the proposed method are comparable to KD-moving, XI-fixed, and XI-merged methods. Therefore, the control chart based approach has the best overall performance among all four techniques compared in the experiments. The run time of the control chart based approach is also noticeably shorter than the other three methods.

The reason why the proposed technique shows the best overall performance in the experiments is because the control charts used in the technique are "specialized" in monitoring mean and standard deviation over the stream and, hence, are very sensitive to the changes in these two characteristics. In contrast, the other three approaches are generalized techniques that are designed to detect any type of changes in a stream and, thus, may not be as "focused" to monitoring changes in mean values and standard deviations. Therefore, for a stream mining application where mean values and/or standard deviations are the sole interest, the control chart based change detection technique is more suitable.

Detecting distribution drifts

To study the performance of the proposed technique on detecting distribution drifts, we conduct a set of experiments using stream types $Stream13$, ..., $Stream18$ with the same parameter settings. The drift duration $driftDur$ is set to four times the windows size, i.e., 400 data elements. The experimental results are demonstrated in Figures 3.109 – 3.123.



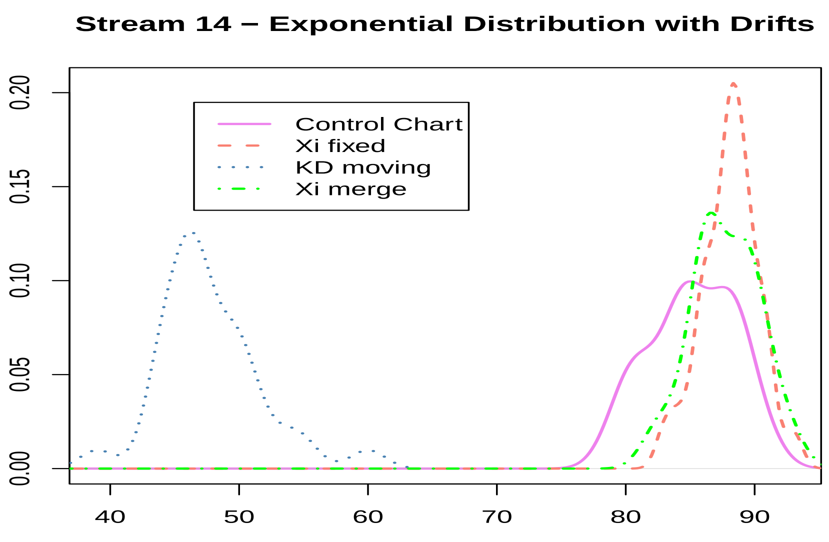
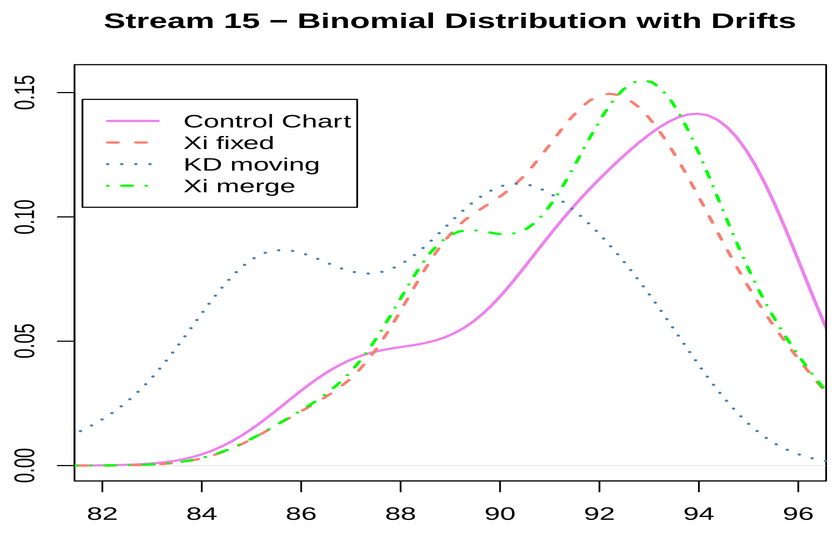


Figure 3.109: Number of true changes detected for *Stream*13 and *Stream*14



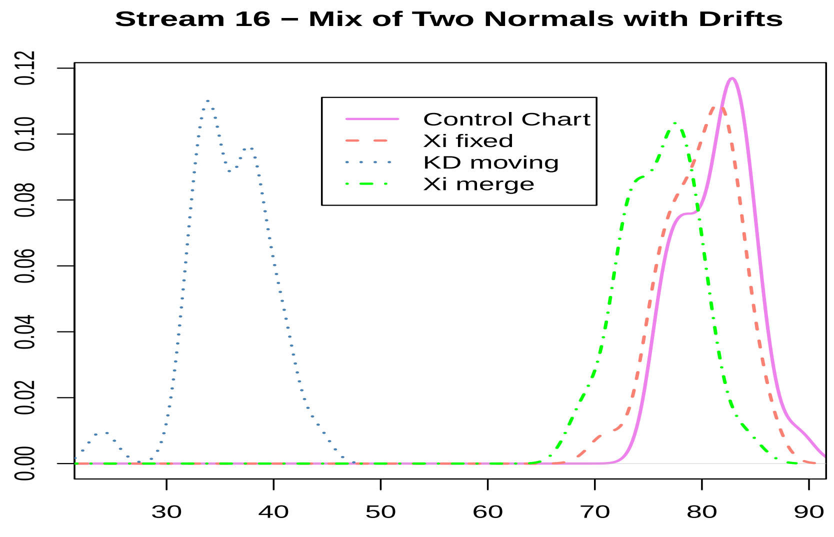
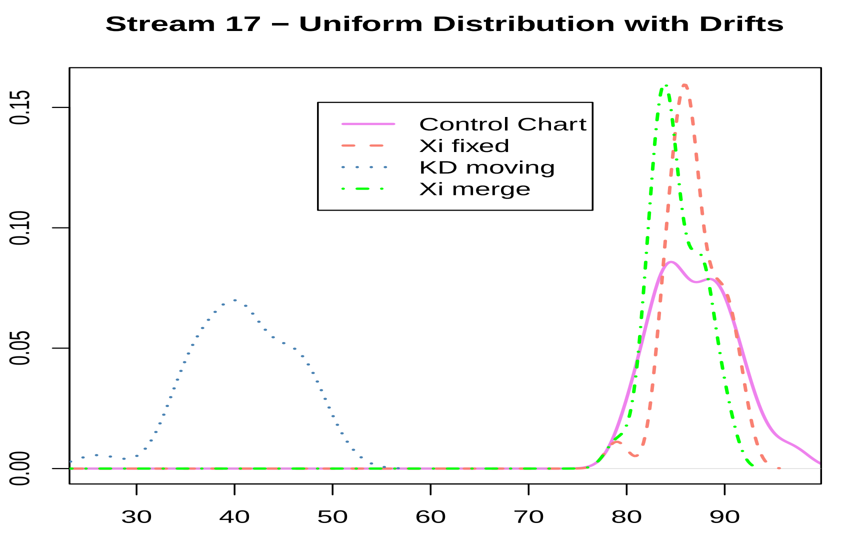


Figure 3.110: Number of true changes detected for *Stream*15 and *Stream*16



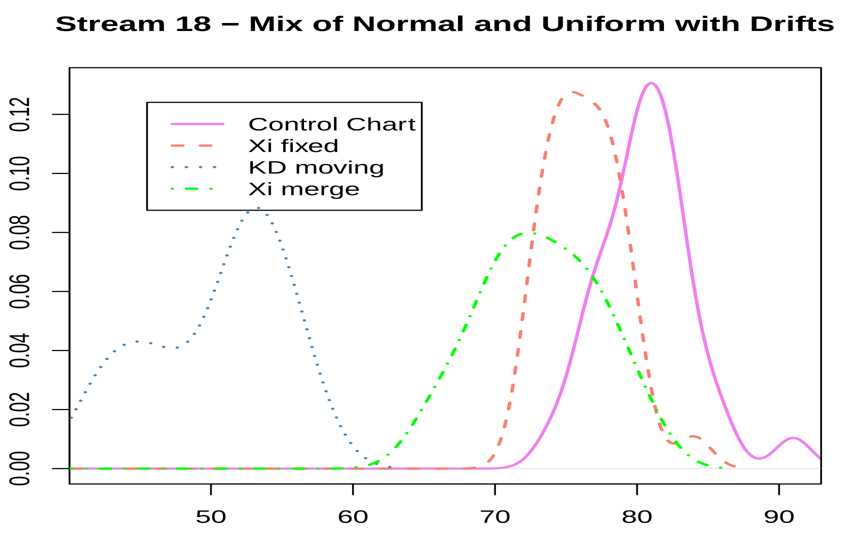
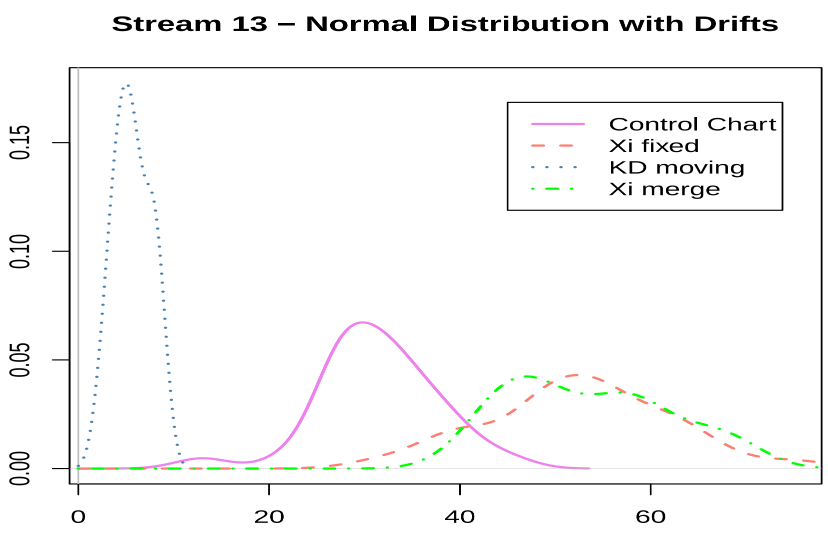


Figure 3.111: Number of true changes detected for *Stream*17 and *Stream*18



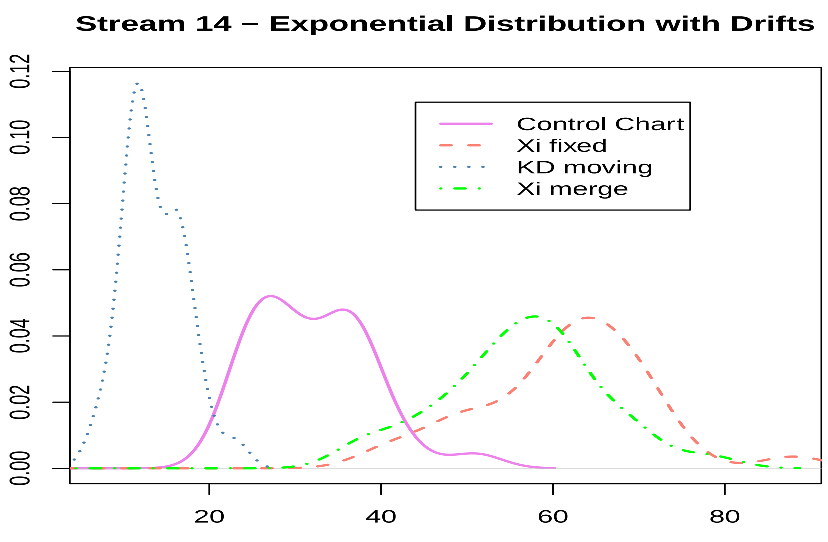
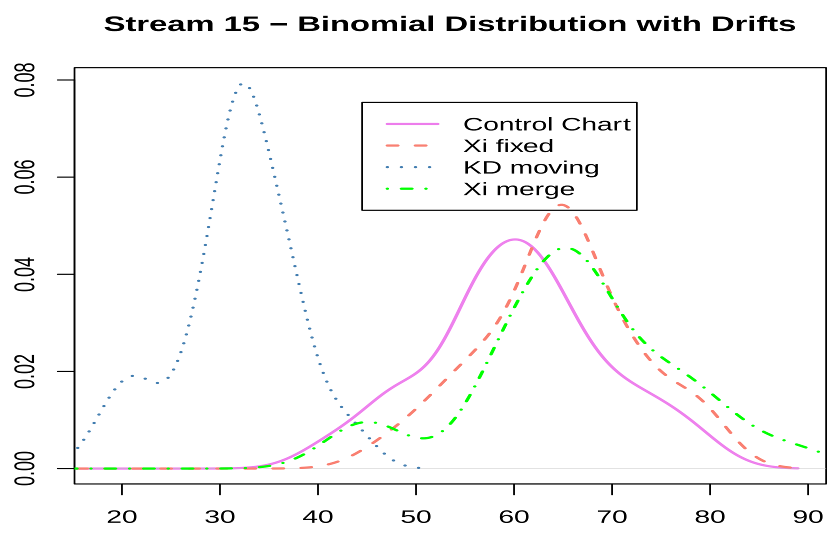


Figure 3.112: Number of false changes detected for $Stream13$ and $Stream14$



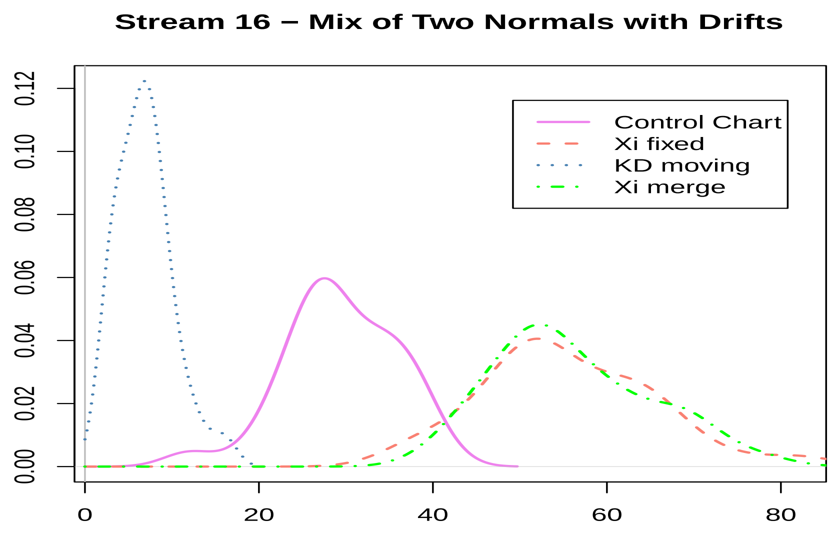
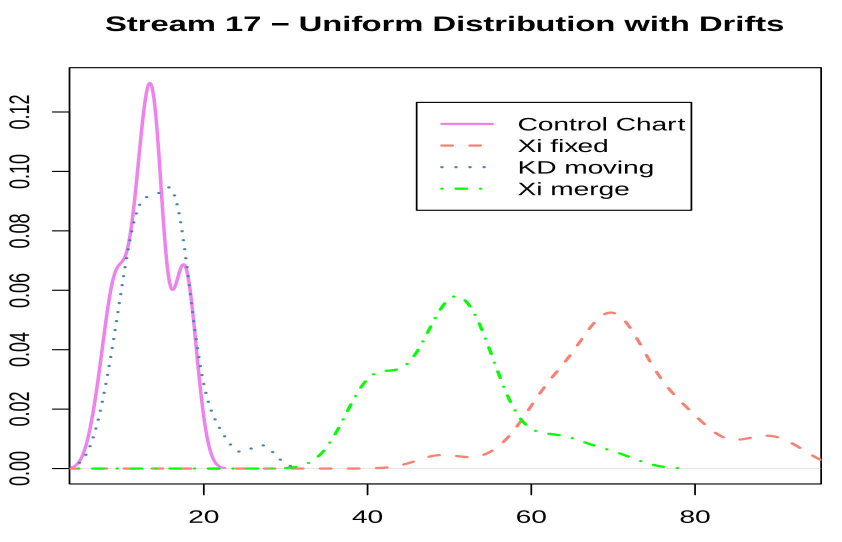


Figure 3.113: Number of false changes detected for $Stream15$ and $Stream16$



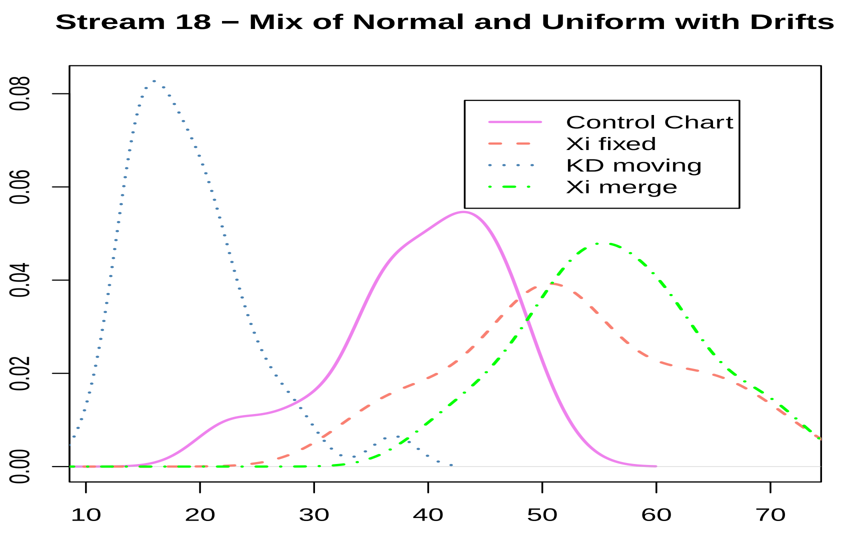
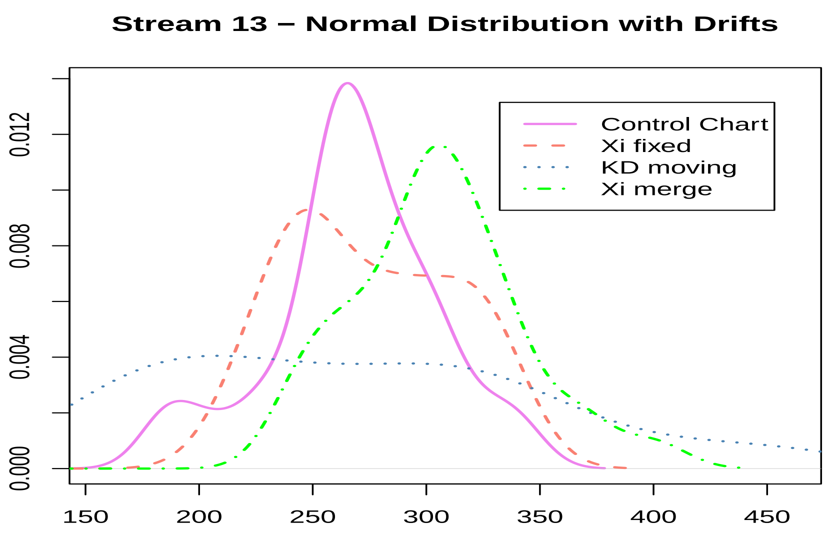


Figure 3.114: Number of false changes detected for $Stream17$ and $Stream18$



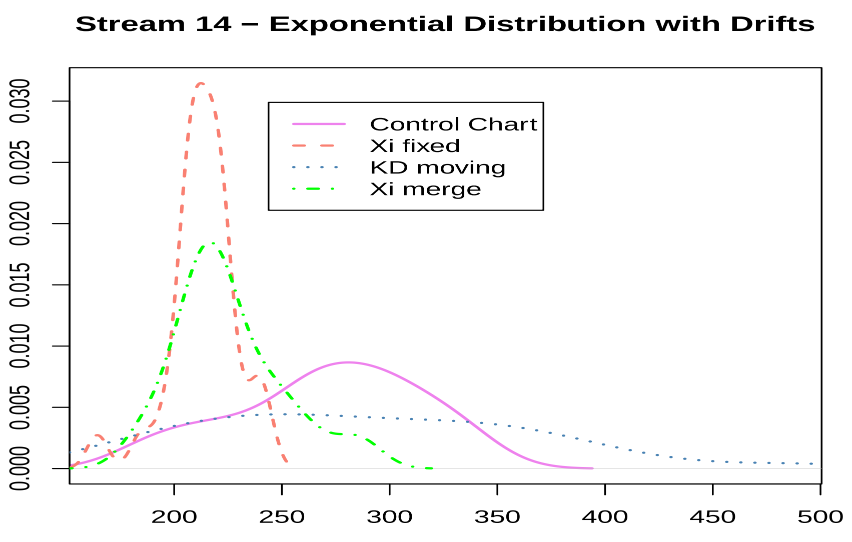
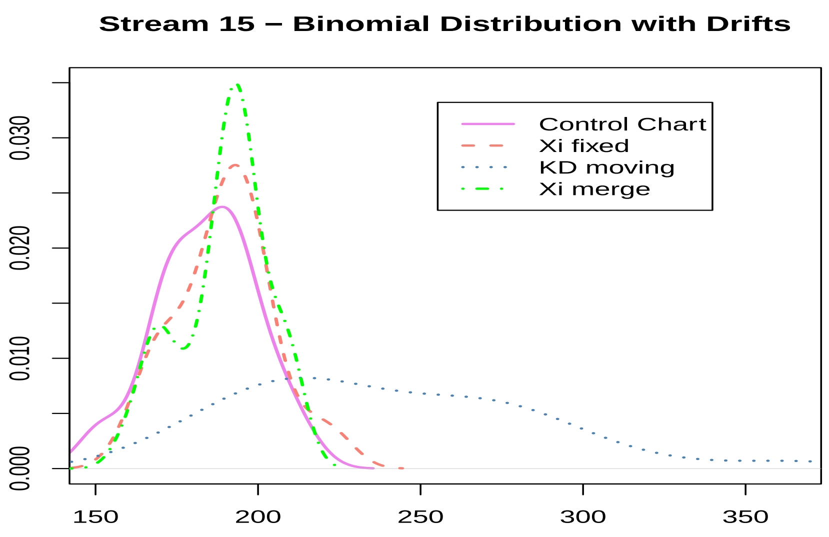


Figure 3.115: Mean duration for detecting true changes in $Stream13$ and $Stream14$



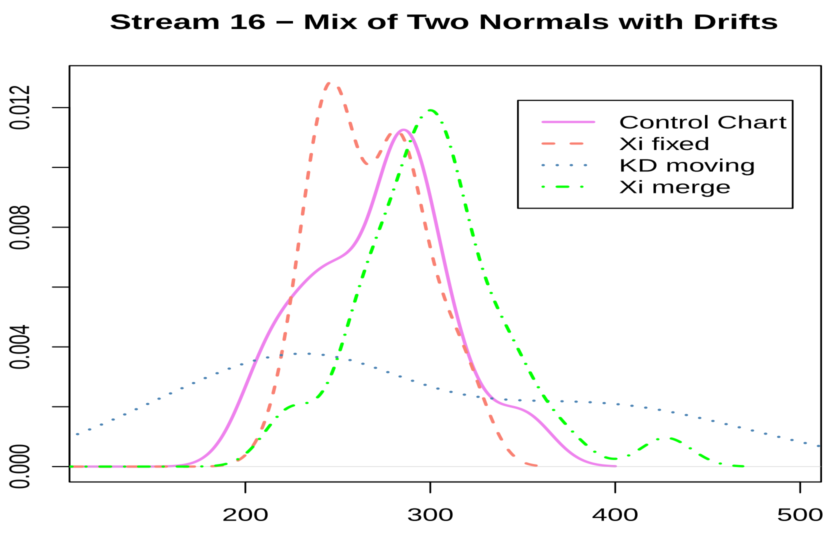
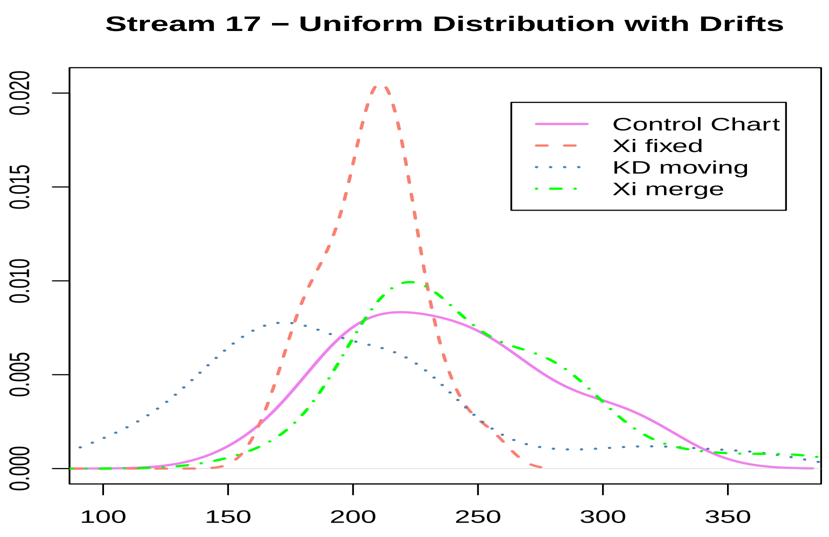


Figure 3.116: Mean duration for detecting true changes in $Stream15$ and $Stream16$



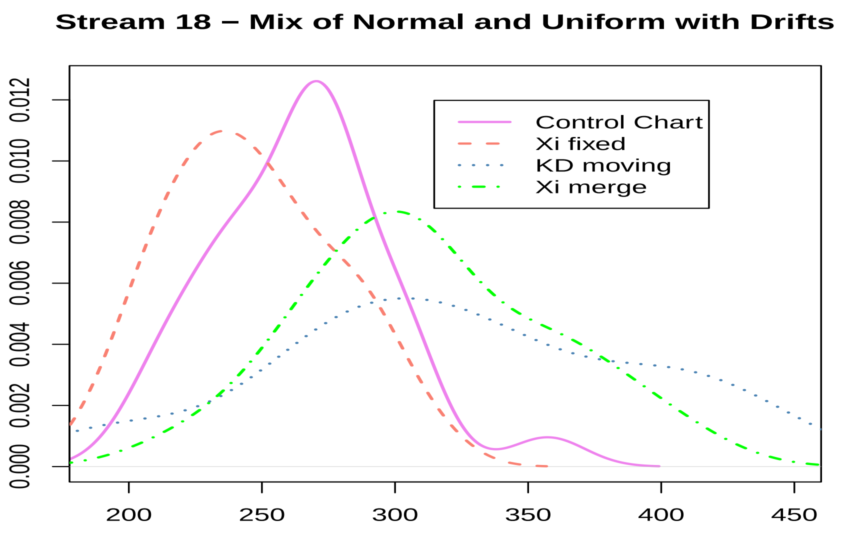
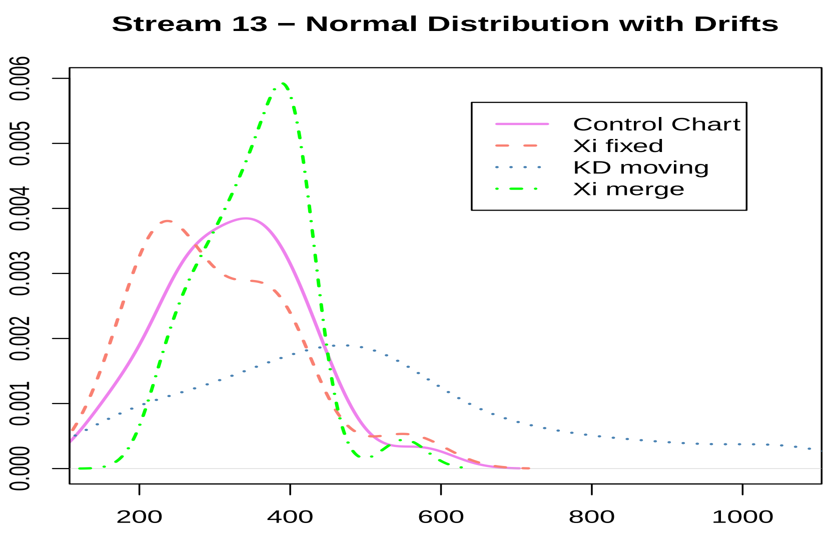


Figure 3.117: Mean duration for detecting true changes in $Stream17$ and $Stream18$



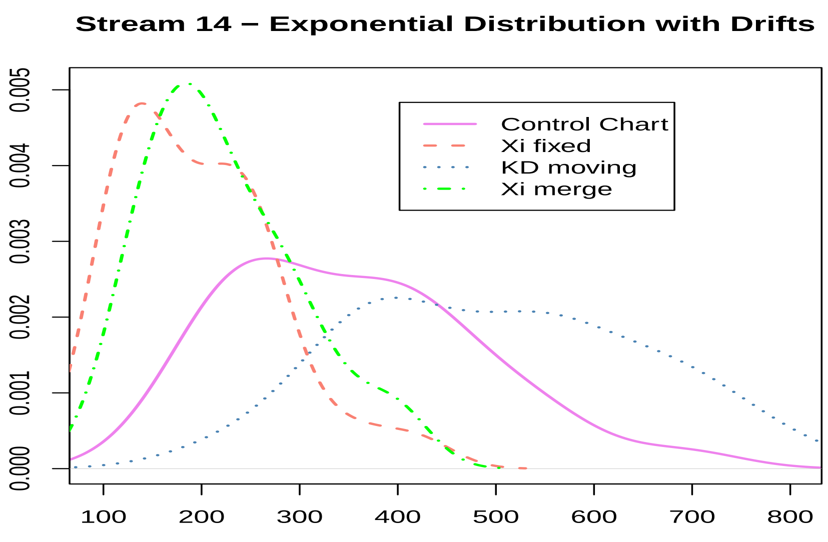
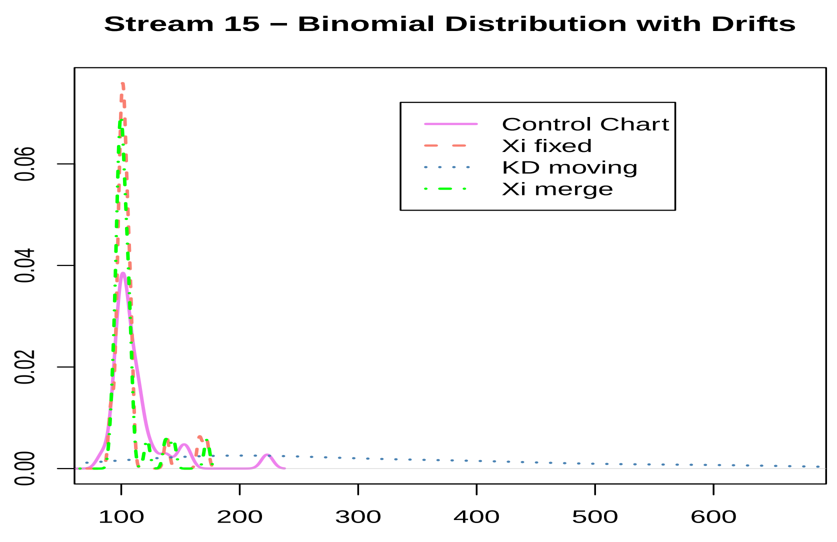


Figure 3.118: Standard deviation of the duration for detecting true changes in $Stream13$ and $Stream14$



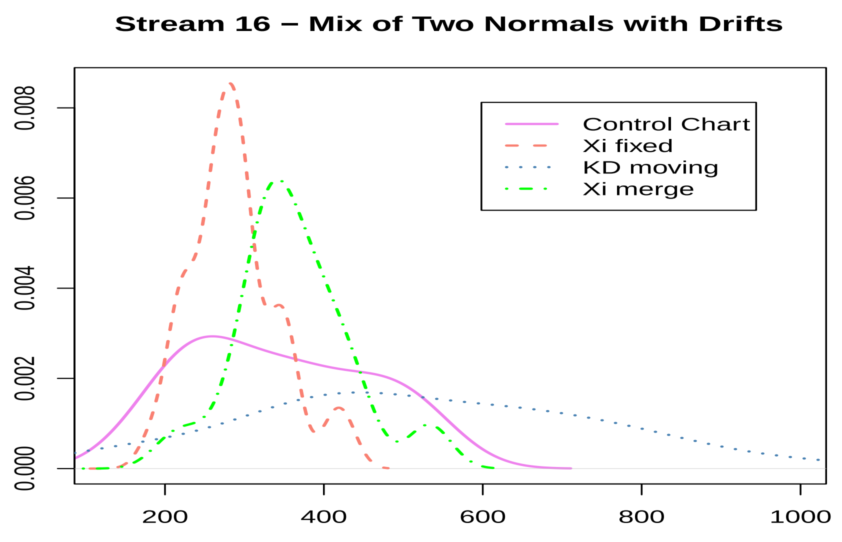
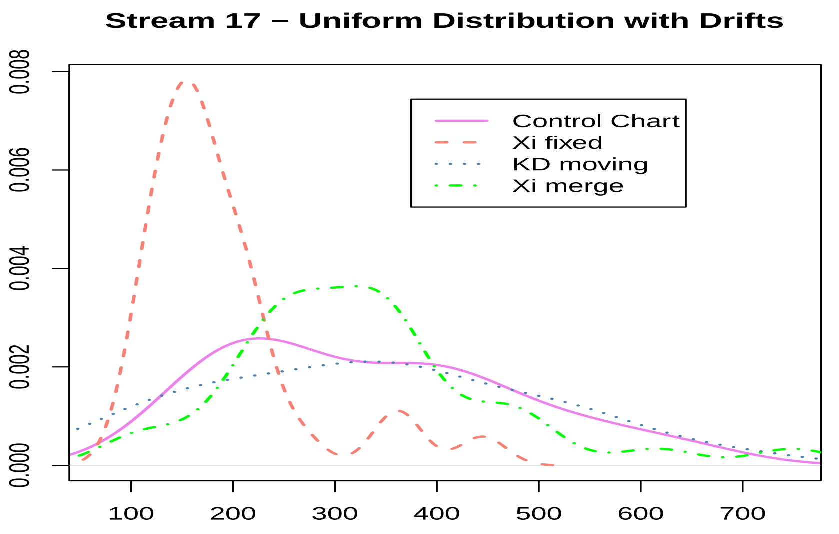


Figure 3.119: Standard deviation of the duration for detecting true changes in $Stream15$ and $Stream16$



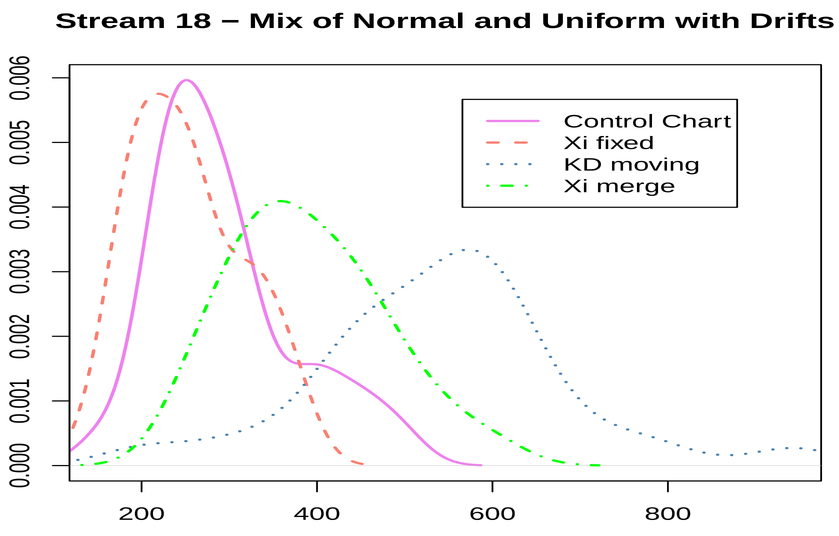
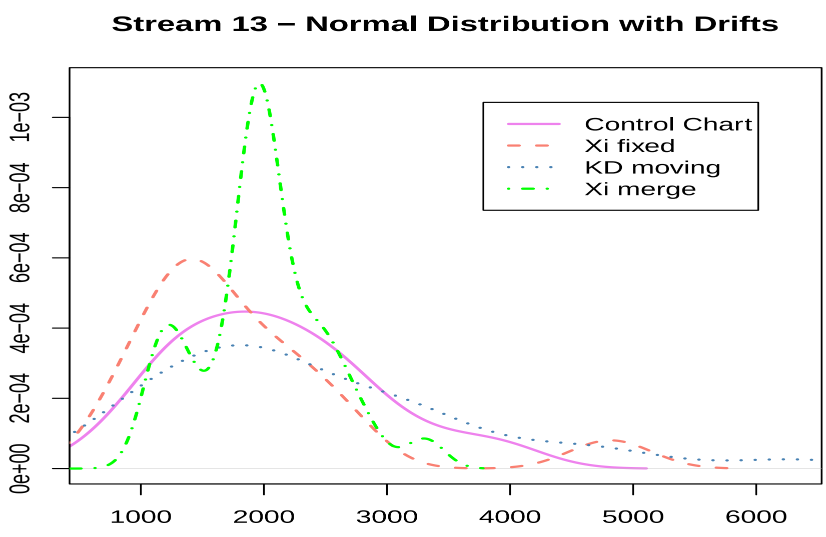


Figure 3.120: Standard deviation of the duration for detecting true changes in $Stream17$ and $Stream18$



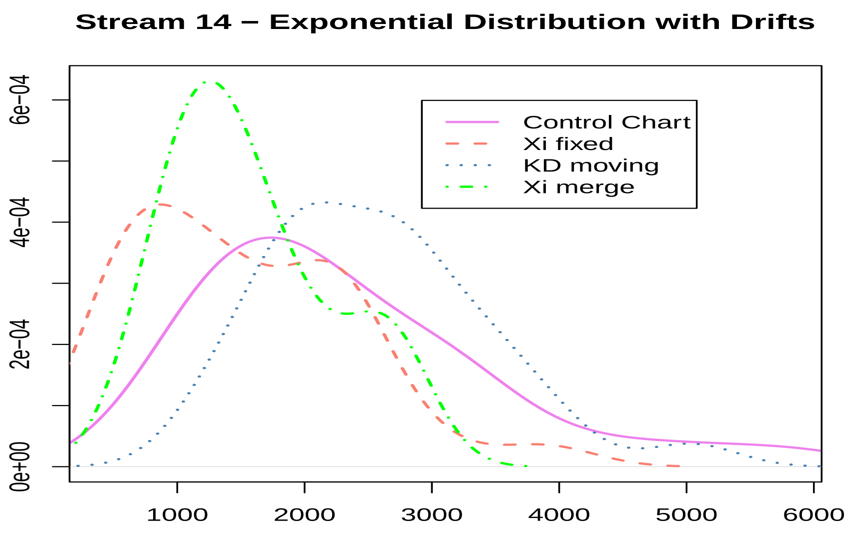
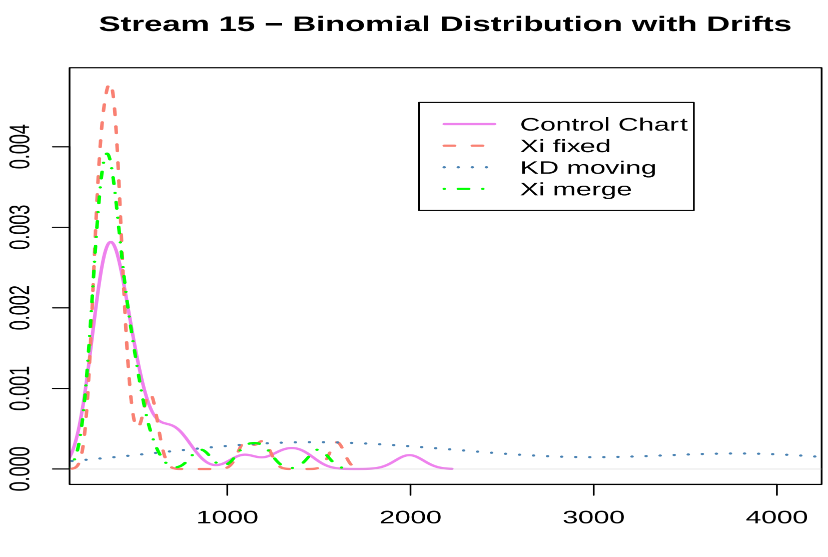


Figure 3.121: Max duration for detecting true changes in $Stream13$ and $Stream14$



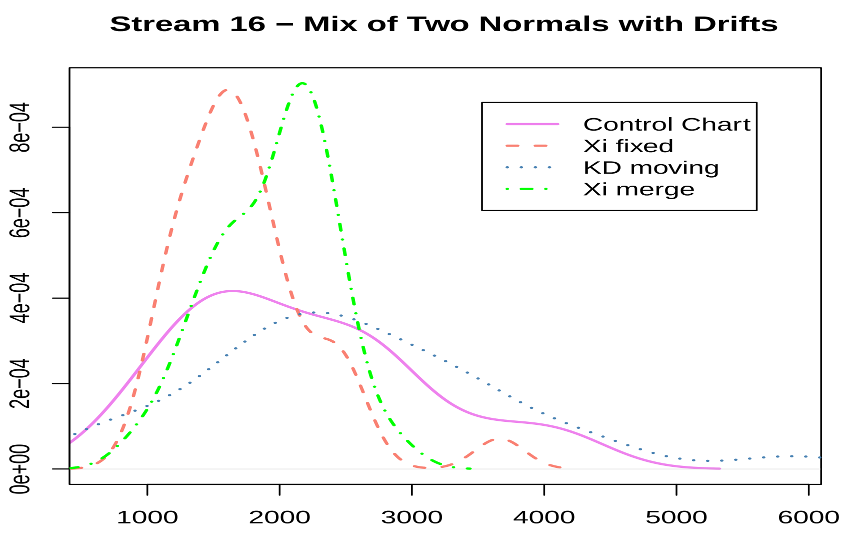
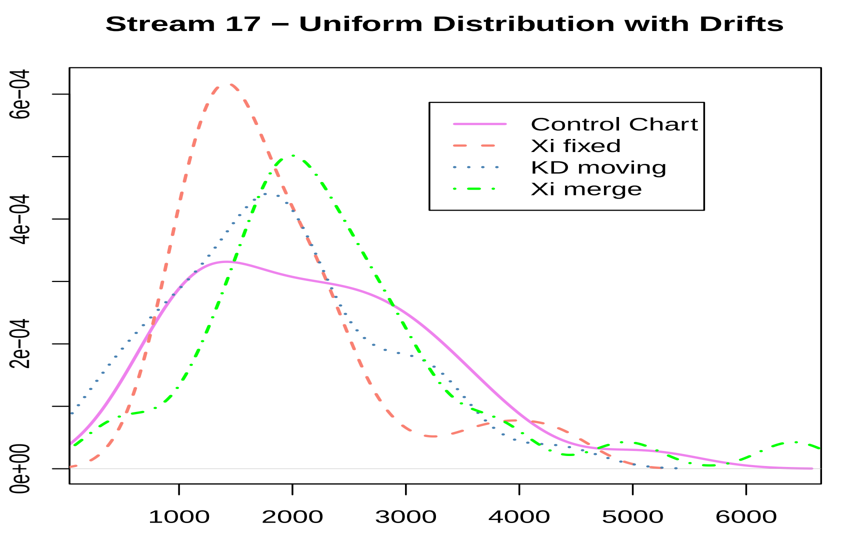


Figure 3.122: Max duration for detecting true changes in $Stream15$ and $Stream16$



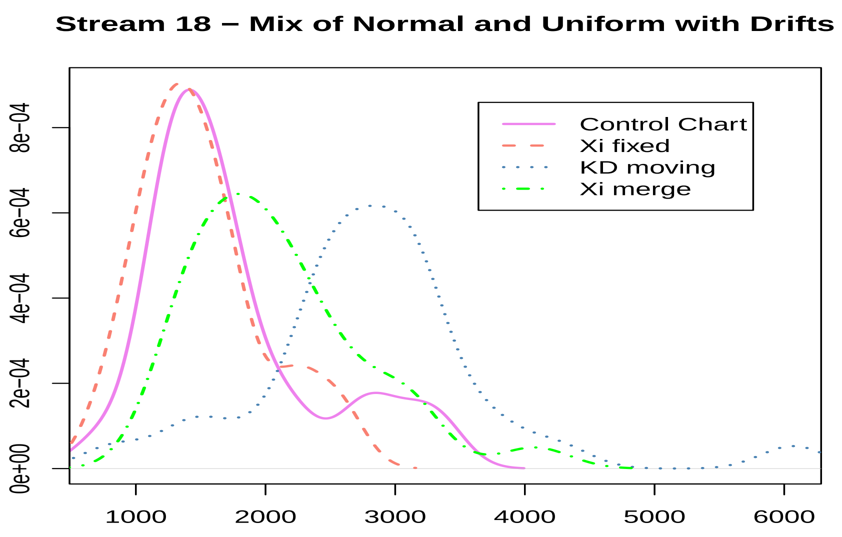


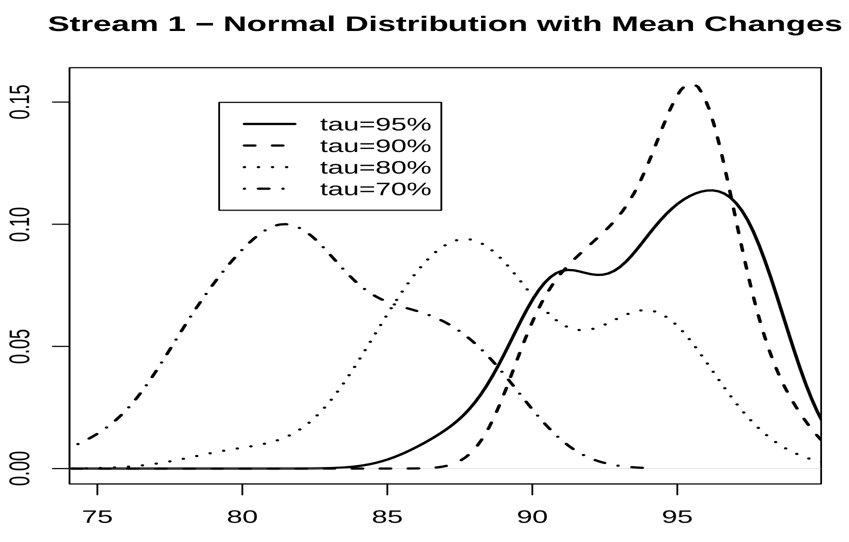
Figure 3.123: Max duration for detecting true changes in $Stream17$ and $Stream18$

Similar conclusions as the previous set of experiments can be made for detecting distribution drifts. The control chart based approach generates the most number of true changes with number of false detections less than XI-fixed and XI-merged, but more than KD-moving. The true detection durations are comparable to the other three methods. These results demonstrate that the proposed technique can achieve good performance in detecting distribution drifts. This is because the EWMA control charts adopted in the technique are designed for detecting small and gradual changes.

Effect of significance level τ

Significance level τ is an important parameter in the control chart that can be used to determine parameters ω and κ. τ is a user defined parameter that should be determined by analyzing application requirements. A high significance setting indicates that only severe distribution changes are of interest, while minor distribution changes will be reported with a low τ value.

The effect of τ values over the change detection results are studied by repeating the previous experiments with τ value set at $70\%, 80\%, 90\%$ and 95%. From the results, it is noticeable that the impact of τ is consistent for different stream types, and, hence, only the empirical results of detecting changes in stream types $Stream1$ and $Stream17$ is demonstrated. These results are shown in Figures 3.124 – 3.128.



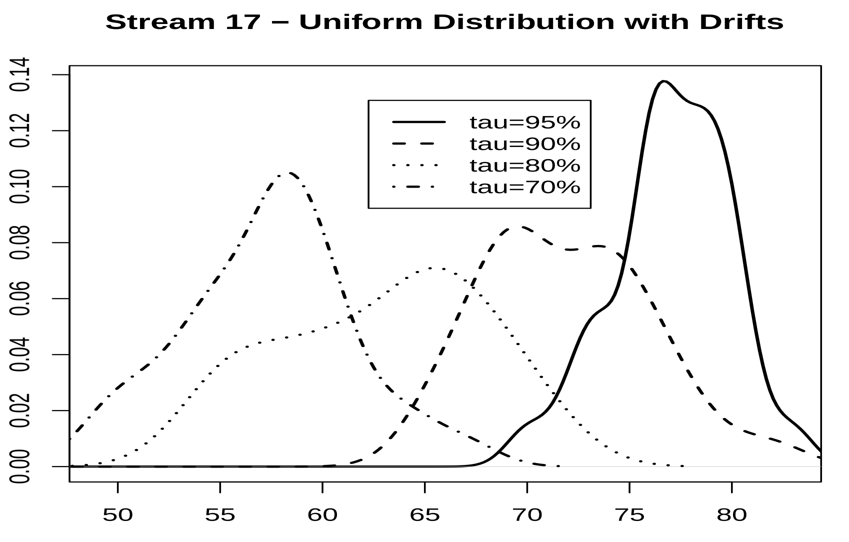
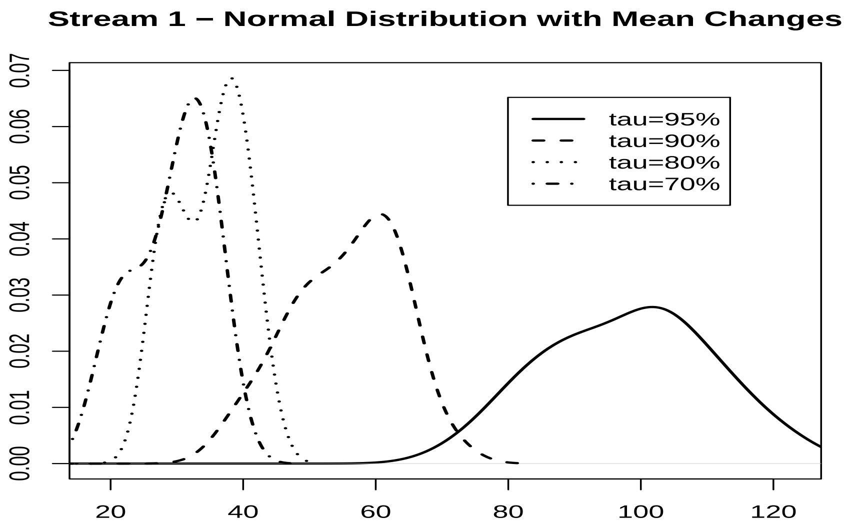


Figure 3.124: Number of true changes detected for $Stream1$ and $Stream17$ with different τ values



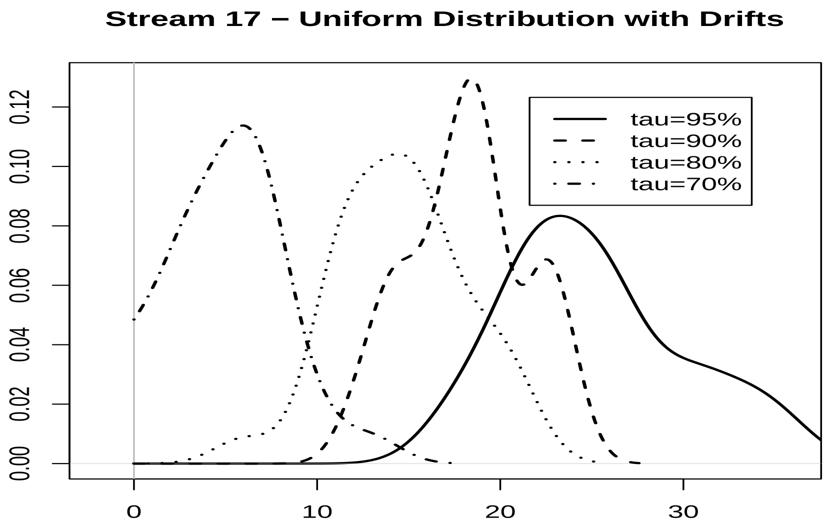
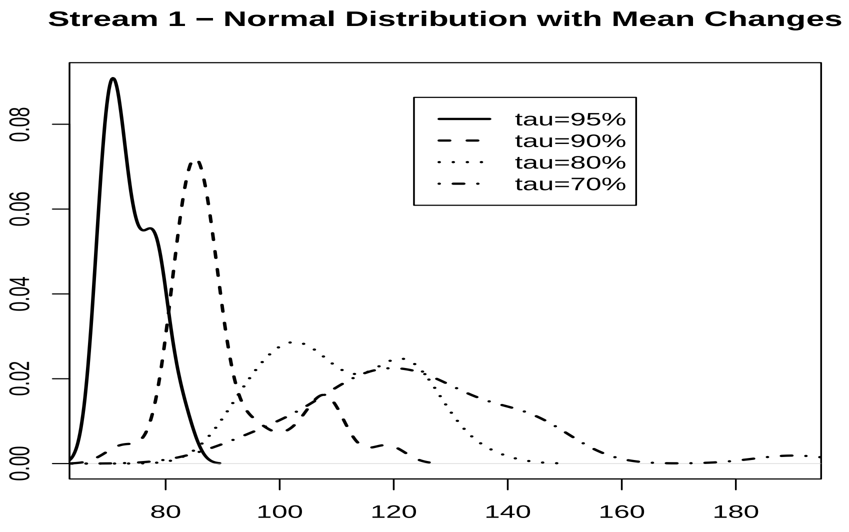


Figure 3.125: Number of false changes detected for $Stream1$ and $Stream17$ with different τ values



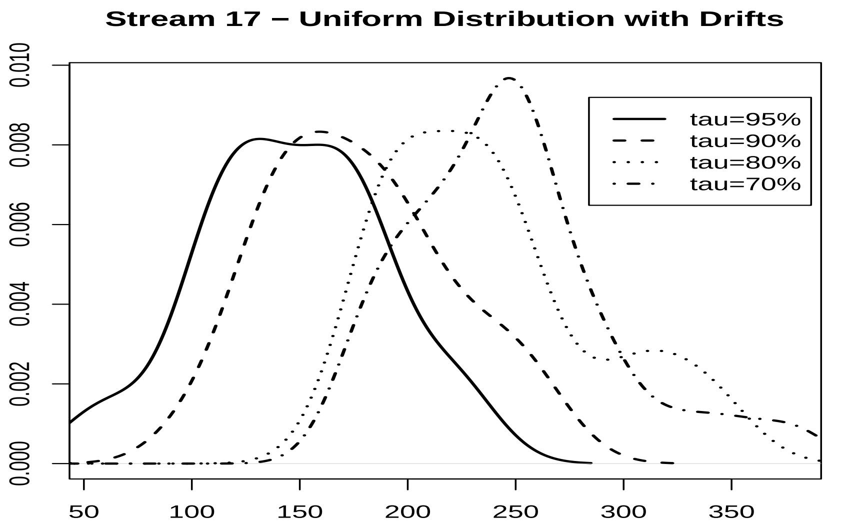
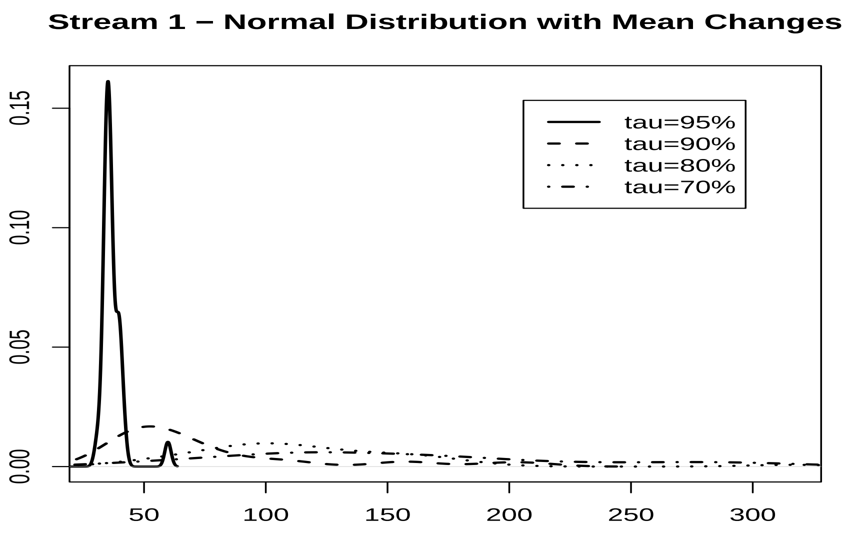


Figure 3.126: Mean duration for detecting true changes in $Stream1$ and $Stream17$ with different τ values



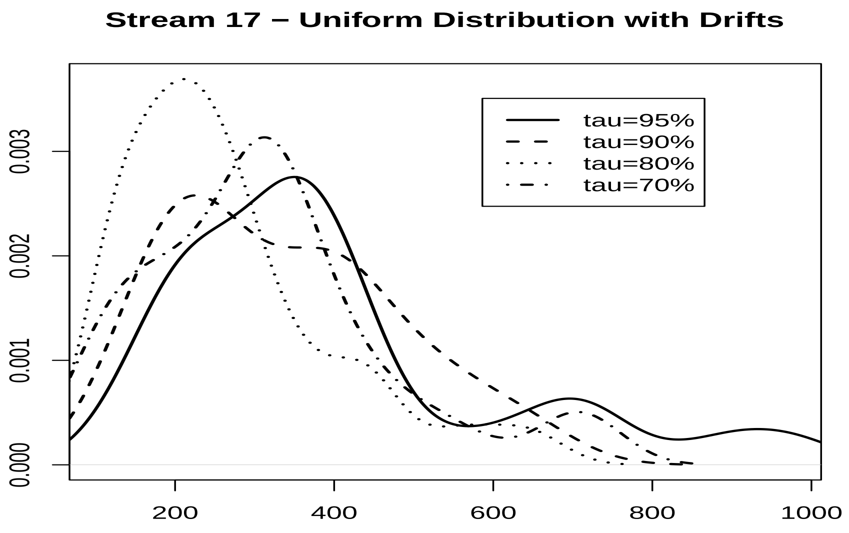
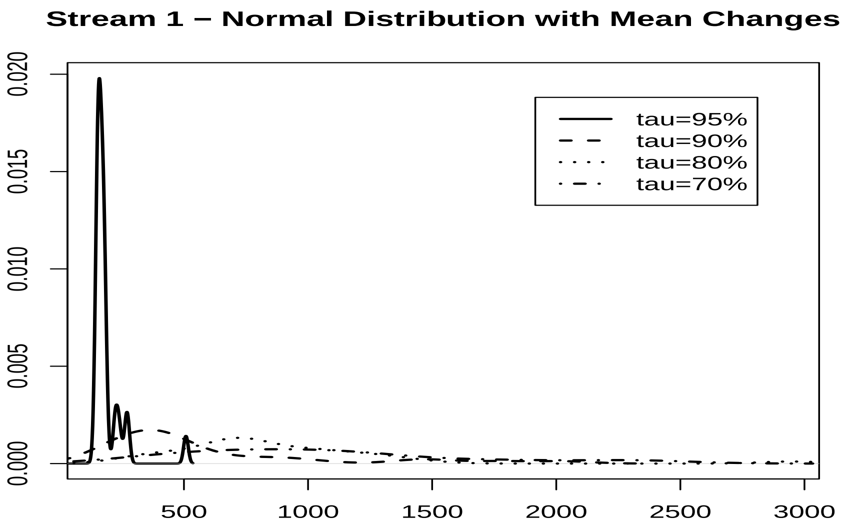


Figure 3.127: Standard deviation of the duration for detecting true changes in $Stream1$ and $Stream17$ with different τ values



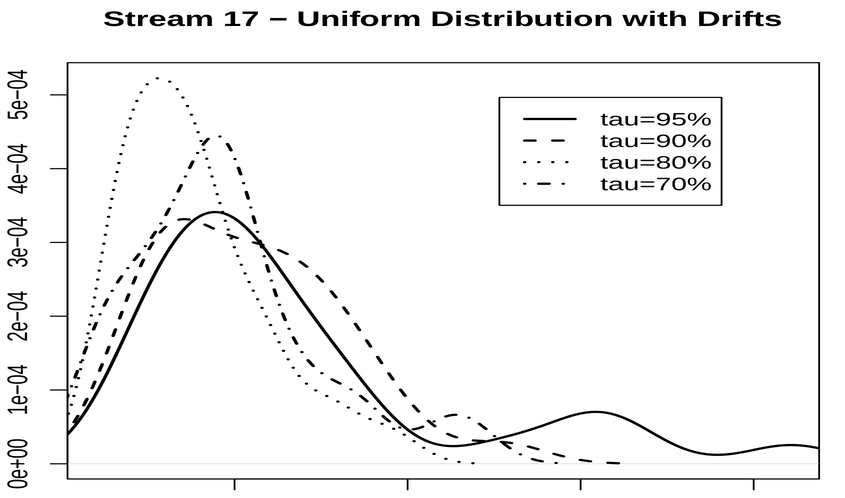


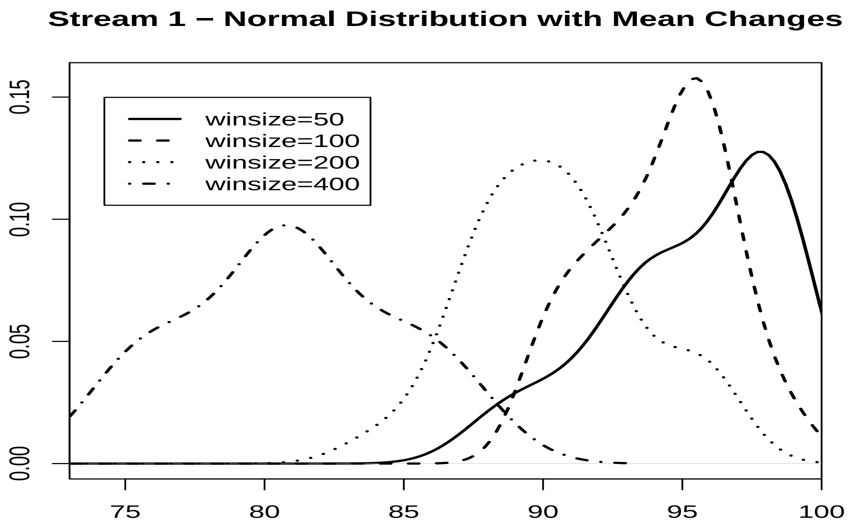
Figure 3.128: Max duration for detecting true changes in $Stream1$ and $Stream17$ with different τ values

These results suggest that, increasing significance level τ can increase the number of true detections. However, the number of false detection will also increase when τ value is larger. This is because, with a high τ value, the control limits UCL and LCL are "tighter", i.e., distance between UCL/LCL and the center line is small. Hence, small changes on the mean values and standard deviations will be reported as a distribution change. It is also worth noting that the durations for detecting true changes are noticeably smaller with higher τ setting. This is because higher τ values increase the weight ω in Equations 3.13 and 3.15, making historical data less important in calculating weighted moving means and standard deviations. Therefore, control charts are more sensitive in changes in the newly arrived data in windows W, resulting in a faster detection.

Effect of window size

As discussed in Section 3.4.3, the size of the tumbling window W may affect the efficiency and accuracy of the proposed technique. To study the effect of window size over the control chart based technique, a set of experiments is conducted by repeating the previous experiments with $|W|$ set as 50 data, 100 data, 200 data, and 400 data. The experimental results on stream types $Stream1$ and $Stream17$ are illustrated in Figures 3.129 – 3.133.

According to these results, a smaller window size will result in larger number of false detections, however, the number of true changes detected is also larger. This may be because with a small sample size (i.e., the number of data in window W is smaller), outliers may greatly affect the mean value and standard deviation, making the control charts more sensitive to outliers. Hence more changes, both true changes and false changes, are detected. The results also show that smaller window sizes make the true changes detected faster, because the weighted moving means and standard deviations are updated and compared with the control limits more frequently. These results confirm our discussions in Section 3.4.3.



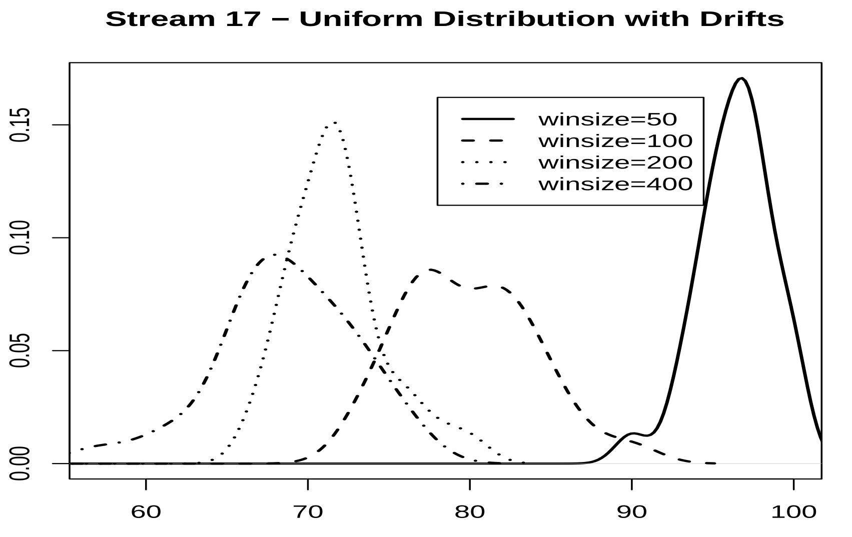
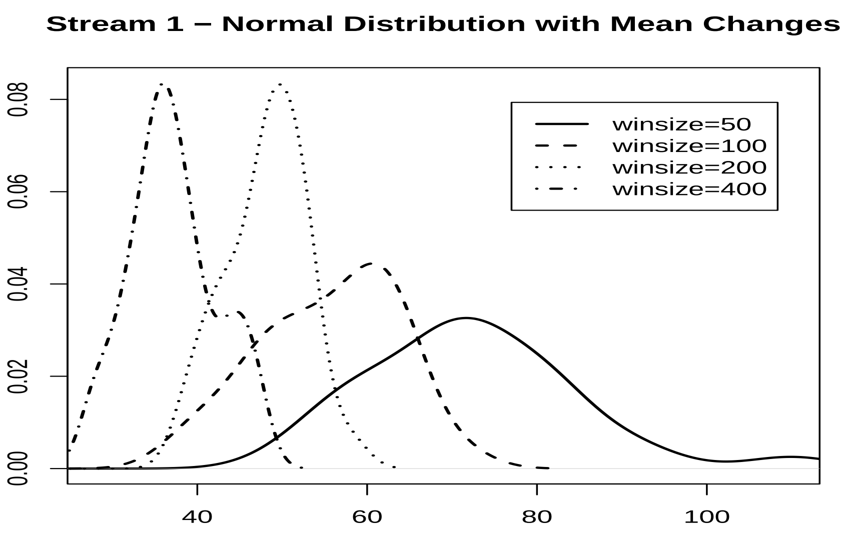


Figure 3.129: Number of true changes detected for $Stream1$ and $Stream17$ with different window size



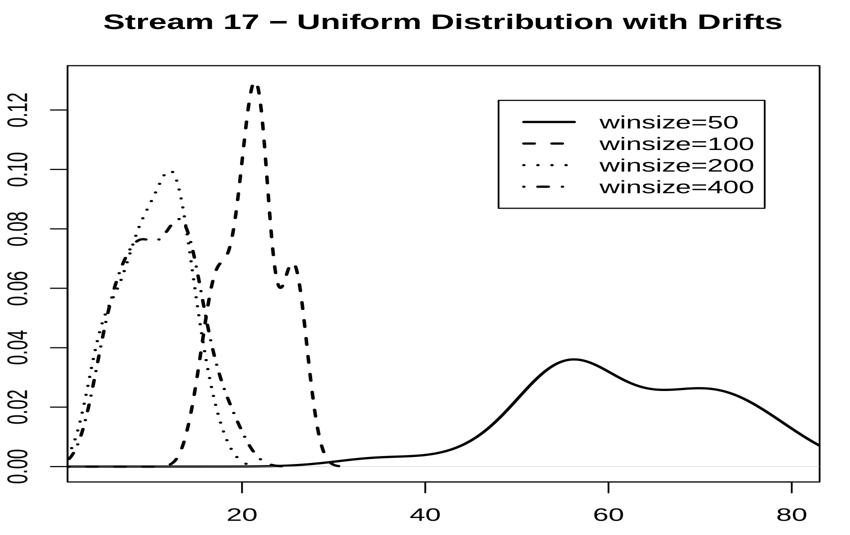
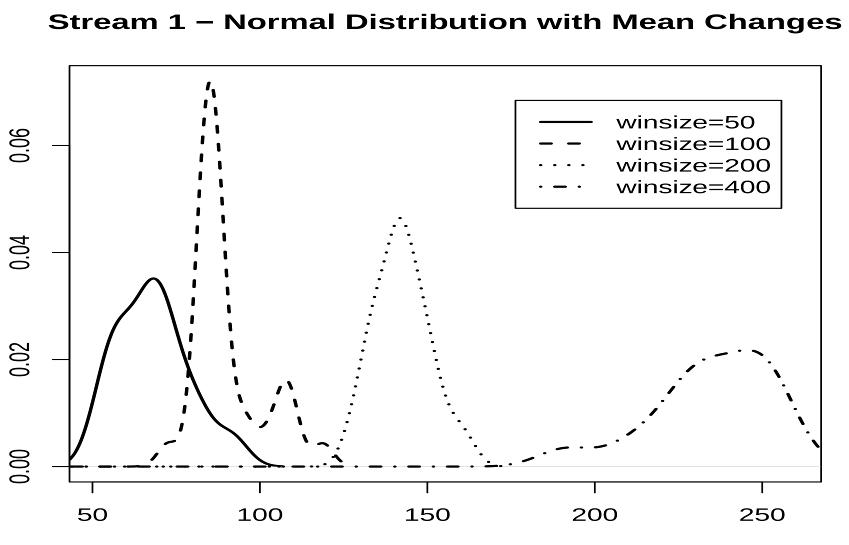


Figure 3.130: Number of false changes detected for $Stream1$ and $Stream17$ with different window size



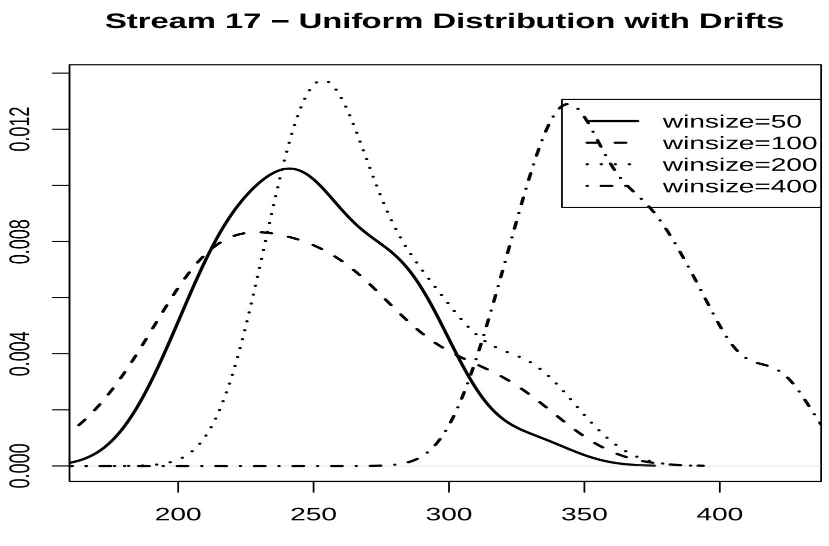
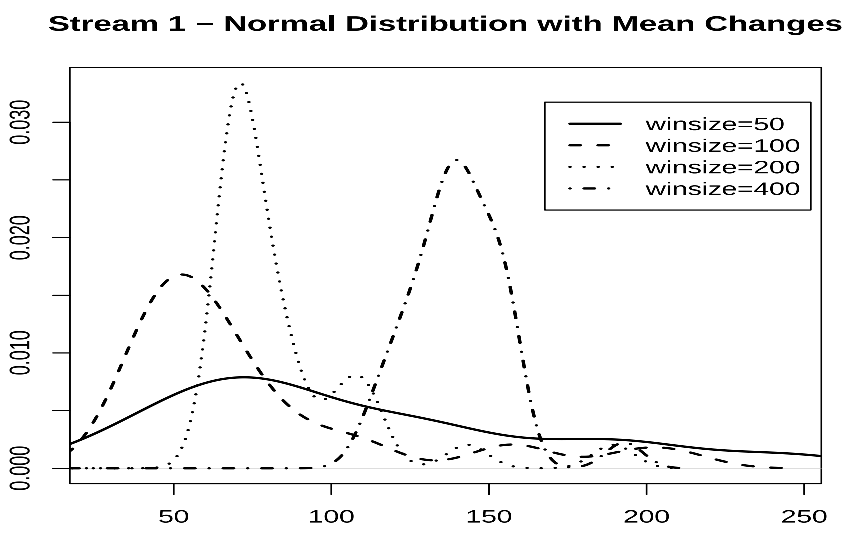


Figure 3.131: Mean duration for detecting true changes in $Stream1$ and $Stream17$ with different window size



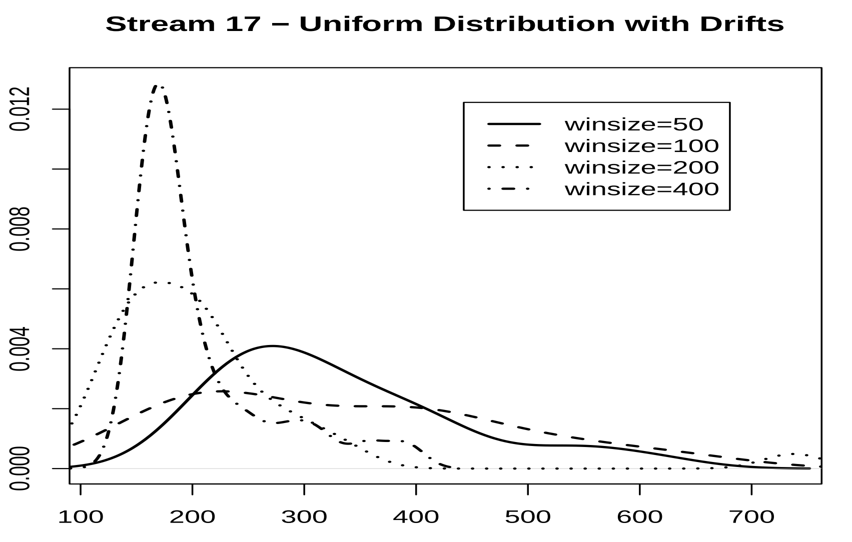
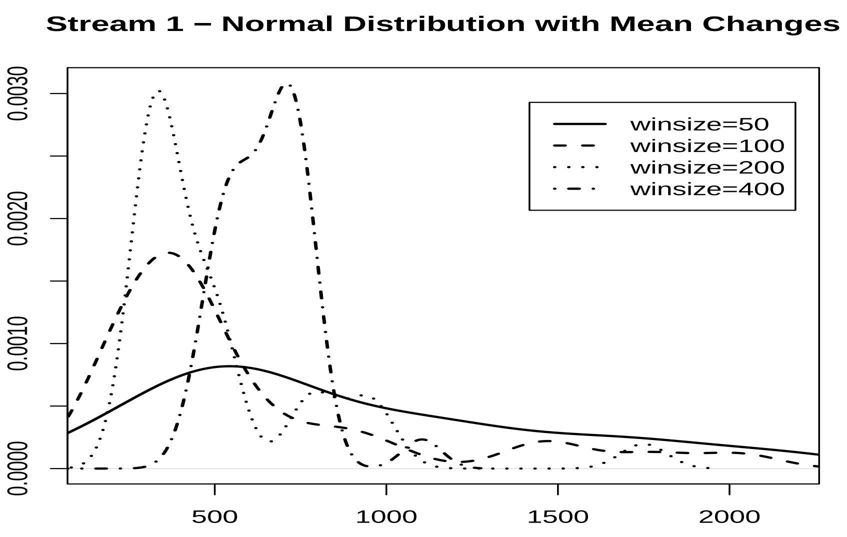


Figure 3.132: Standard deviation of the duration for detecting true changes in *Stream*1 and *Stream*17 with different window size



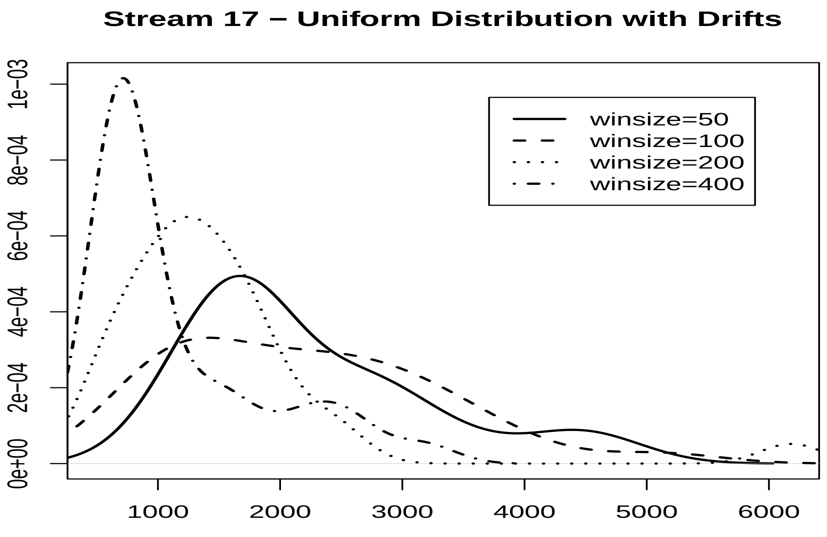


Figure 3.133: Max duration for detecting true changes in $Stream1$ and $Stream17$ with different window size

3.5 Summary

The unboundedness and high arrival rates of data streams and the dynamic variations in their underlying data distribution make processing stream data challenging. Detection of these distribution changes are important to properly and accurately perform various data mining tasks. However, most of the techniques proposed in literature are ad-hoc and cannot be directly applied to all types of streams. General solutions are more difficult to find but are much desired, because they can be plugged in any stream mining applications and can make techniques developed under stable environments suitable for mining dynamic streams.

In this chapter, two new techniques are proposed for detecting distribution changes in dynamic streams. The first technique is designed to solve the problem of presenting (complex) distributions using small data sets with high accuracy. The second technique focuses on detecting the changes in two statistics of a distribution – mean and standard deviation. Both approaches are generic and can be applied to many types of data streams and mining applications. Extensive experiments are conducted and results show the promise of these approaches in detecting distribution changes for ever-changing streams.

Chapter 4

Change Detection in Multi-dimensional Streams

In Chapter 3, two application-independent change detection approaches are proposed. These techniques demonstrate promising performance in the experiments. However, one major constraint of these approaches is that they can only be applied to streams with single dimension, i.e., there is only one attribute of interest in the stream.

The control chart based approach discussed in Section 3.4 only detects the changes of two key features, i.e., the mean and standard deviations, of the stream. Since correlations of these two features can be easily obtained, the proposed control chart based approach can be extended to streams with multi-dimensions. In this chapter, we relax the uni-dimension constraint of the proposed control chart for detecting mean changes by using a T^2 control chart [66].

4.1 Motivation

The elements in a data stream collected from real-world applications usually contain several attributes. Most of the change detection techniques for dynamic data streams assume that there is only one attribute of interest. However, in practice, many stream processing applications need to take more than one attribute into consideration. For example, in modern quality control, several quality characteristics are usually monitored simultaneously. In e-commerce, where each element in the stream is an order placed by customers, a positive linear correlation between items may indicate similar purchase patterns. There has been little attention paid to the problem of extending change detection to multi-dimensional data.

Under the assumption that attributes of interest are not correlated with each other, the multi-dimensionality can be easily addressed by running a set of processes that detect distribution changes simultaneously on each attribute. However, this assumption does not hold for many real-world applications. Take a stream monitoring application that controls the quality of drug products as an example. A drug product contains several substances [40]; most often the substances in the drug are interdependent. One impurity might be formed as a result of the degradation of another one, or two impurities might react to form another impurity. If the correlations among several attributes (variants) in the data are taken into account, such a solution is no longer satisfactory.

Processing multi-dimensional data can be difficult because of the correlations among dimensions. One of the challenges is known as "the curse of dimensionality": In high dimensional space, data may be sparse, making it difficult to find any structure in the data. One solution is to reduce the dimensionality, either by selecting a feature subset [46, 54, 135] or by using a feature transformation that projects high dimensional data onto "interesting" subspaces [18, 70, 73]. Reducing the dimensionality can be a feasible solution when the dimensionality is very high. However, for data sets that have been reduced to only a few dimensions, it is improper to further reduce them, since some important correlation information may be lost during the process.

The correlations of some key features of the data, such as mean, range, variance, among dimensions can be easily obtained. Therefore, a technique that only detects distribution changes over certain key features in uni-dimensional streams should have the potential to be extended to multi-dimensional streams. Based on this insight, we modify the technique proposed in Section 3.4 that detects mean changes to make it suitable for detecting mean changes in multi-dimensional streams.

Recall that the original approach in Section 3.4 is based on control chart. This provides another advantage for multi-dimensional change detection. Unlike many change detection techniques that require a large number of data to get promising results, the control chart-based method can achieve high accuracy with only a small sample set. This is an important feature for detecting changes in the streaming environment, since stream processing techniques are single pass and memory is always limited. As will be discussed, the modified approach has high efficiency so that fast distribution changes in the stream can be captured. However, note that the proposed method is not a generic change detection method. It can only detect changes in mean values and may not perform well for detecting other types of distribution changes in multi-dimensional streams.

4.2 Related Work

Detecting distribution changes in multi-dimensional space is difficult and few techniques have been proposed for this problem. Many approaches try to bypass the problem by transforming the multi-dimensional data to a uni-dimensional space using dimensionality reduction techniques (e.g., [18, 46, 54, 70, 135]). As noted earlier, useful correlation information will be lost during the reduction process.

A statistical test, called *cross-match*, has been proposed for comparing two multivariate distributions [139]. Cross-match uses interpoint distances [140] to construct an optimal non-bipartite matching. The similarity between two distributions is measured by the number of pairs that contain one data from each distribution. This method is computationally expensive and consumes large amount of memory. Therefore, it is not applicable for data stream applications.

Dasu et al. propose a change detection approach in multi-dimensional streams using the Kullback-Leibler (KL) distance [82] to calculate the distance between two distributions [124, 125]. KL distance, defined as $dist_{KL}(P_A, P_B) = \sum P_A(x_i) log \frac{P_A(x_i)}{P_B(x_i)}$, is one of the most fundamental measures for measuring the dissimilarity between two completely determined probability distributions. However, note that since KL distance is a non-symmetric measure, it is not strictly a distance metric. This approach then uses bootstrap methods [30] to determine whether the change is significant or not. This technique relies on a partition of the space. As shown in their experiments, the performance will decrease significantly when the number of the dimensions increases.

Song et al. develop a technique for calculating the discrepancy between the underlying distributions of two large data sets with multiple dimensions [127]. The method is based on multi-dimensional kernel density estimator and uses maximum expectation algorithm [42] to choose the kernel bandwidths that are closest to the true distribution. According to their experiments, this method can detect subtle changes in the data set with few false detections. However, since large amount of data are required for the statistical test and relatively heavy computation is involved, this approach is not suitable for data stream applications that require high efficiency.

4.3 Detecting changes using multi-dimensional control charts

The extended technique for detecting mean changes in multi-dimensional streams has a similar framework as the original technique proposed in Section 3.4. A tumbling window W with tumbling interval Δ is used to store the most up-to-date substream from which the test data are obtained. The multi-dimensional control chart C monitors the mean changes of a stream S. New data are fed into C when W moves. A distribution change is reported when the mean value of a data set falls outside the control limits LCL and UCL of the control chart. The same stopping rule StR and null hypothesis H_0 introduced in Section 3.4 are adopted for the multi-dimensional approach. A significance level τ is defined as the minimum probability of refuting H_0 when stopping rule StR is satisfied.

4.3.1 Building the multi-dimensional control chart

To detect distribution changes in multi-dimensional streams, the covariance matrix is introduced into the proposed control chart. Each entry in the covariance matrix quantifies the degree to which two dimensions vary together (covary). Let S^d be a d-dimensional stream with all elements in space $\Re^d$ and let M be the $d \times d$ covariance matrix. $S^d(t-\Delta, t]$ represent the set of d-dimensional data elements that have arrived in S^d in the last Δ time unit. $\mu^d(S^d(t-\Delta,t])$ denotes the mean of the values in this substream. The covariance matrix M can be calculated as:

$$M = \frac{\sum (S^d(t-\Delta,t] - \mu^d(S^d(t-\Delta,t]))(S^d(t-\Delta,t] - \mu^d(S^d(t-\Delta,t]))^T}{n(t-\Delta,t]} \tag{4.1}$$

where $n(t-\Delta,t]$ is the number of elements in $S^d(t-\Delta,t]$.

Let $\mu^d(S^d_{t'})$ be the mean value of S at time $t' = t - \Delta$. The *distance* between the mean value of substream $S^d(t-\Delta,t]$ and the mean value of the previous data set $S^d_{t'}$ is then defined as follows:

$$\begin{aligned} &dist(\mu^d(S^d(t-\Delta,t], \mu^d(S^d_{t'})) = \\ &\qquad (\mu^d(S^d(t-\Delta,t]) - \mu^d(S^d_{t'}))^T M^{-1} (\mu^d(S^d(t-\Delta,t]) - \mu^d(S^d_{t'})) \end{aligned} \tag{4.2}$$

From this equation, it can be seen that the larger the distance, the more different is the mean value of substream $S^d(t-\Delta,t]$ compared with the previous mean value. If this distance is significant, then a distribution change may occur.

Recall that in Section 3.4, the history of the stream is taken into consideration by introducing a weighting factor into the control chart. The weighting factor ensures that both distribution shifts and distribution drifts can be detected in time. Hence, for the d-dimensional stream S^d, the weight factor is extended to a weight matrix Λ that is a diagonal matrix with $\omega_1, \omega_2, ..., \omega_d$ on its main diagonal, where $\omega_1, \omega_2, ..., \omega_d$ are the weight variables on each dimension $i = 1, ..., d$. Every time a new set of data $S^d(t-\Delta, t]$ arrives, it is assigned weight Λ, and weight of the old data is decreased by factor $(1-\Lambda)$.

By combining the weight matrix Λ and the covariance matrix M, the weighted covariance matrix M_Λ is defined as:

$$M_\Lambda(i,j) = \omega_i\omega_j \frac{1-(1-\omega_i)^n(1-\omega_j)^n}{\omega_i+\omega_j-\omega_i\omega_j} M(i,j) \tag{4.3}$$

where $M_\Lambda(i,j)$ and $M(i,j)$ are the (i,j)-th elements in matrix M_Λ and M, respectively; ω_i and ω_j are the ith and jth members of the diagonal matrix Λ; variable n is the number of elements in substream $S^d(t-\Delta, t]$.

If the weight of each dimension is identical, i.e., $\omega_1 = \omega_2 = ... = \omega_d = \omega$, and if the number of elements in $S^d(t-\Delta, t]$ is sufficiently large, then Equation 4.3 can be simplified as:

$$M_\Lambda = \frac{\omega}{2-\omega} M \tag{4.4}$$

Assigning identical weights for all dimensions can greatly reduce the computational cost and can reduce the work of the user for setting parameters. For many real-world applications, although the attributes in the data stream have different meanings, the degree of significance of historical data is similar for all dimensions. Using the simplified equation for calculating weighted covariance matrix M_Λ is recommended for these applications. Similar to the discussion in Section 3.4, the weight ω is also determined using the significance level τ and the tables provided by Lucas and Saccucci [91].

Based on the above discussion, the weighted moving mean to estimate the mean value of S^d up to time t is defined as follows:

$$\begin{aligned} \mu^d(S^d_t) = {} & (\Lambda * \mu^d(S^d(t-\Delta, t]) + (1-\Lambda) * \mu^d(S^d_{t'}))^T \\ & M_\Lambda^{-1}(\Lambda * \mu^d(S^d(t-\Delta, t]) + (1-\Lambda) * \mu^d(S^d_{t'})) \end{aligned} \tag{4.5}$$

4.3.2 Detecting changes

When data from stream S^d first arrive, a set of data is obtained to be used as learning samples and the control chart C is built using the method discussed above. Once C is built and all the parameters are tuned, a tumbling window W, with tumbling interval Δ, is applied to capture the most current substream and feed it to C as a set of new data.

Let t be the current timestamp, and let W contain data that have arrived in $(t-\Delta, t]$. Each time W moves, the new sample set will be evaluated by C, and the evaluation results will indicate whether S^d is in control (i.e., distribution is stable) or it is out of control (i.e., distribution has changed). After the evaluation, all data inside W are removed, making room for new data.

Tracy et al. [152] have shown that for a T^2 control chart with dimension d, if no distribution change is observed, then the probability of the T^2 statistics having a chi-square distribution is quite high. That is,

$$T^2 \sim \frac{d*(m+1)(m-1)}{m*(m-d)} F_{d,m-d,\rho} \tag{4.6}$$

where m is the number of observations in a preliminary data set with stable distribution, ρ is the probability that T^2 statistics have chi-square distribution, and $F_{d,m-d,\rho}$ is the Fisher-Snedecor distribution [39] with d and $m-d$ degrees of freedoms.

Let t be the current timestamp when W is about to tumble, and t_a be the last time a distribution change was detected. In control chart C, the parameters m and ρ in equation 4.6 are equivalent to $n(t_a, t-\Delta]$, i.e., the number of data in substream $S^d(t_a, t-\Delta]$, and $1-\tau$, respectively. Therefore, the UCL of the proposed multi-dimensional control chart C, denoted as UCL^d, is defined as follows:

$$UCL^d = \frac{d*(n(t_a,t-\Delta]+1)(n(t_a,t-\Delta]-1)}{n(t_a,t-\Delta]*(n(t_a,t-\Delta]-d)} F_{d,n(t_a,t-\Delta]-d,1-\tau} \tag{4.7}$$

If the distribution has not changed since the beginning of S, then

$t_a = t_0$. Null hypothesis H_0 (i.e., distribution of S does not change) is refuted iff:

$$\mu^d(S_t^d) > UCL^d \tag{4.8}$$

The distribution change occurs only when weighted moving mean of S at time t is beyond the upper control limit of the control chart. When H_0 is refuted, a distribution change is noted and t_a is set to t.

Note that $\mu^d(S_t^d)$ have the property of directional invariance, that is, the average run length to detect a change depends only on the direction of the change. The significance of the difference of the mean value between the old data and the new substream is the sole point of interest; hence, only the upper control limit is required for the proposed control chart.

4.3.3 Full algorithm

The process of the proposed technique using multi-dimensional control chart is summarized in Algorithm 2.

The most time consuming part of the algorithm is calculating weighted moving mean using Equation 4.5 (line 27). The time complexity for calculating Equation 4.5, if carried out naively, is $O(d^3)$. For the rest of the algorithm, the time complexity is linear, i.e., $O(n)$, where n is the size of stream S. Therefore, the proposed approach has the worst-case time complexity $O(d^3n)$, which is acceptable for fast arriving streams when the number of dimensions is not extremely large.

4.4 Experimental framework

To evaluate the performance of the proposed approach, an experiment framework that includes synthetic stream generation and experimental results illustration is designed and implemented.

A d-dimensional data stream S^d consists of d single dimensional streams $S_1, S_2, ..., S_d$, where S_i is the stream of the ith dimension. For each data

Algorithm 2 Mean Change Detection for Multi-dimensional Streams

```
 1: INPUT: Data stream $S^d$
 2:             Significance level $\tau$
 3: OUTPUT: Distribution change alarms
 4:             (Interpretation) Set of single dimensional charts that are
    out-of-control
 5:
 6: Determine $\omega$ based on $\tau$;
 7: Record timestamp $t_a = t_0$;
 8: Calculate covariance matrix $M_\Lambda$ using Equation 4.1;
 9: while $t - t_0 < \Delta$ do
10:     Collect data into $W$;
11:     Record timestamp $t$;
12: end while
13: $\mu^d(S^d_{t'}) = \mu^d(S^d[t_0, t])$;
14: while $S$ does not terminate do
15:     $t' = t$;
16:     while $t - t' < \Delta$ do
17:         Collect data into $W$;
18:         Record timestamp $t$;
19:     end while
20:     $\mu^d(S^d_t) = \mu^d(S^d(t', t])$;
21:     Calculate $UCL^d$ using Equation 4.7;
22:     if $\mu^d(S^d_t) > UCL^d$ then
23:         Report distribution change;
24:         $t_a = t'$;
25:         $\mu^d(S^d_t) = \mu^d(S^d(t', t])$;
26:     else
27:         Calculate $\mu^d(S^d_t)$ using Equation 4.5;
28:     end if
29: end while
```

element $X_j = (x_1, x_2, ..., x_d) \in S^d$, the ith dimension x_i of X_j is the jth data element in the single dimensional stream S_i. Hence, we generate d single dimensional streams with total of n data in each.

Let $mChg$ be the number of distribution changes in S^d. We generate $mChg$ random numbers $chg_1, chg_2, ..., chg_{mChg}$ within the range $(1, n)$. These $mChg$ numbers indicate the "location" of the new distribution change occurs. We then generate each single dimensional stream S_i using the framework discussed in Section 3.3.5. The distributions $P_1, ..., P_d$ of streams $S_1, ..., S_d$ are of the same type. For example, all single dimensional streams $S_1, ..., S_d$ of S^d can have normal distribution with changing mean values. As in Section 3.3.5, we generate streams with either distribution shifts or distribution drifts.

A total of 12 types of multi-dimensional data streams are generated using the framework discussed above. These streams are summarized in Table 4.1.

Table 4.1: Stream types generated

Stream type	Dim	Distribution type	Chg speed
$Stream1$	4	Normal distribution	Shifts
$Stream2$	4	Uniform distribution	Shifts
$Stream3$	5	Exponential distribution	Shifts
$Stream4$	5	Half-half mix of one Normal and one Uniform with different mean	Shifts
$Stream5$	15	Normal distribution	Shifts
$Stream6$	15	Uniform distribution	Shifts
$Stream7$	18	Exponential distribution	Shifts
$Stream8$	18	Half-half mix of one Normal and one Uniform with different mean	Shifts
$Stream9$	4	Normal distribution	Drifts
$Stream10$	15	Uniform distribution	Drifts
$Stream11$	5	Exponential distribution	Drifts
$Stream12$	18	Half-half mix of one Normal and one Uniform with different mean	Drifts

For each stream type, there are a total of *numRun* streams generated using the proposed framework. These streams are input into the proposed technique and one set of results is output for each stream. Therefore, there are a total of *numRun* result sets generated for each stream type. Same as Section 3.3.5, these result sets record five important criteria for evaluating the performance of the change detection technique: number of true changes detected; number of false changes detected; mean duration for detecting true changes; standard deviation of durations for detecting true changes; maximal duration for detecting true changes. Density figures are generated to summarize the *numRun* result sets for each stream type, where x-axis in each figure is the value of each criteria and y-axis is the density.

4.5 Experiments

A series of experiments are conducted to evaluate the performance of the proposed multi-dimensional control chart for mean change detection. All experiments are conducted on a PC with 3GHz Pentium 4 processor and 1GB of RAM, running Windows XP system. All programs are implemented in R.

4.5.1 Change detection evaluation

The parameters discussed in the experimental framework are set as follows. The size n of each generated multi-dimensional data stream is set to 100,000 data elements. The number of true distribution changes $mChg$ in each stream is 100. Significance level τ is set to 80%. The total number of streams generated and tested for each stream type is $numRun = 100$. The arrival speed of each stream is stable, and, therefore, a time-based tumbling window is equal to a count-based one. The size of the tumbling window W used in our control chart is set to 100 data elements.

The proposed control-chart approach is applied on stream types $Stream1, ..., Stream12$ described in Table 4.1. The results are grouped into three sets. The first set contains results of detecting changes in stream types $Stream1, ..., Stream4$. These stream types have low dimensions and distribution shifts. Stream types $Stream5, ..., Stream8$ have distribution shifts and relatively high dimensions and, thus, the change detection results on these stream types are grouped into the second set. The last set contain results of detecting distribution drifts in stream types $Stream9, ..., Stream12$.

Figures 4.1 – 4.5 demonstrate the results of the five important criteria for the first set (streams with low dimensions). The results of the second set (streams with high dimensions) are presented in Figures 4.6 – 4.10. The results of the third set (streams with distribution drifts) are illustrated in Figures 4.11 – 4.15. In each figure, x-axis records the values of the five criteria and y-axis shows their densities.

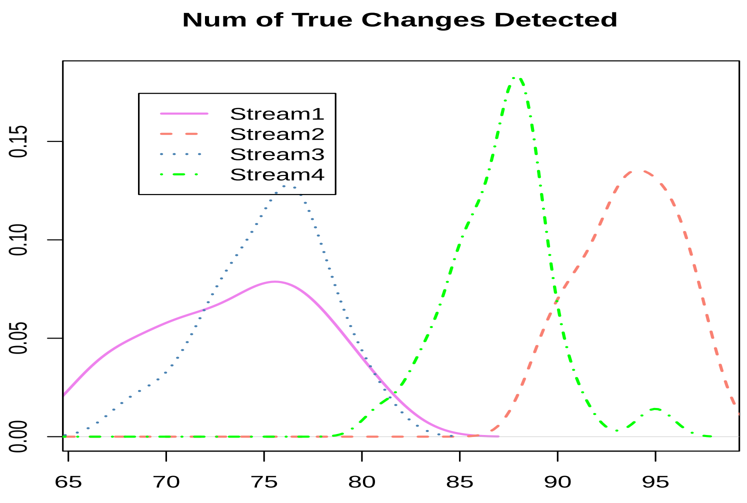


Figure 4.1: Number of true changes detected for $Stream1 - Stream4$

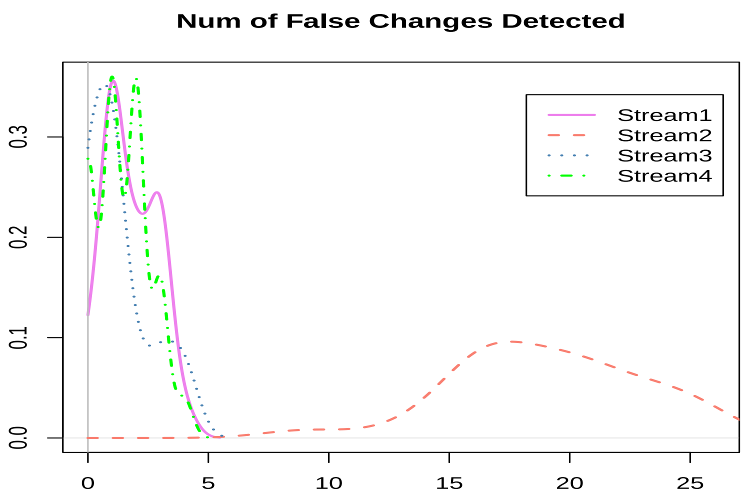


Figure 4.2: Number of false changes detected for $Stream1 - Stream4$

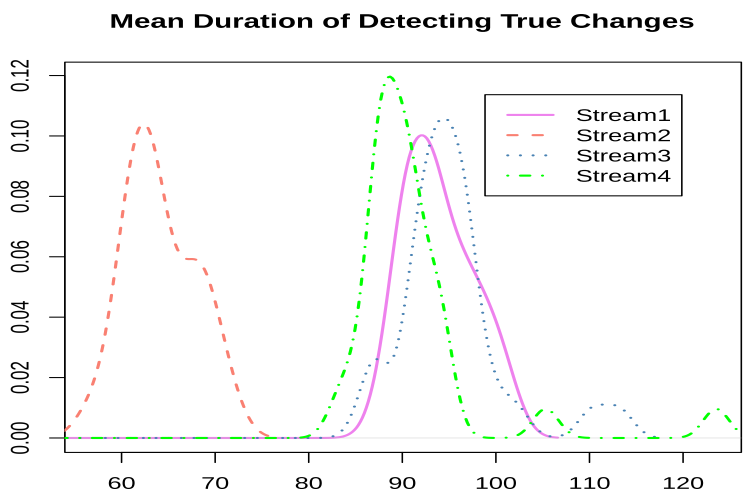


Figure 4.3: Mean duration for detecting true changes in *Stream*1 – *Stream*4

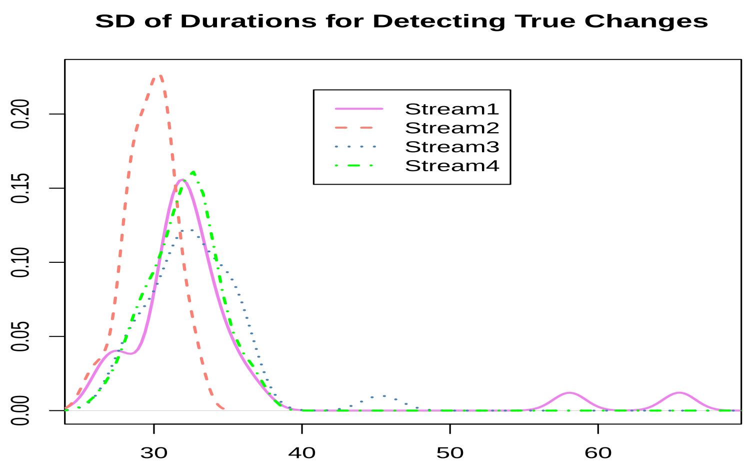


Figure 4.4: Standard deviation of duration for detecting true changes in *Stream*1 – *Stream*4

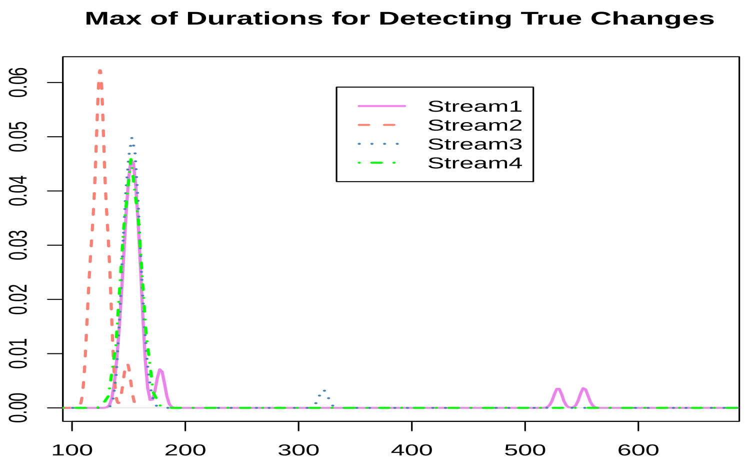


Figure 4.5: Max duration for detecting true changes in $Stream1$ – $Stream4$

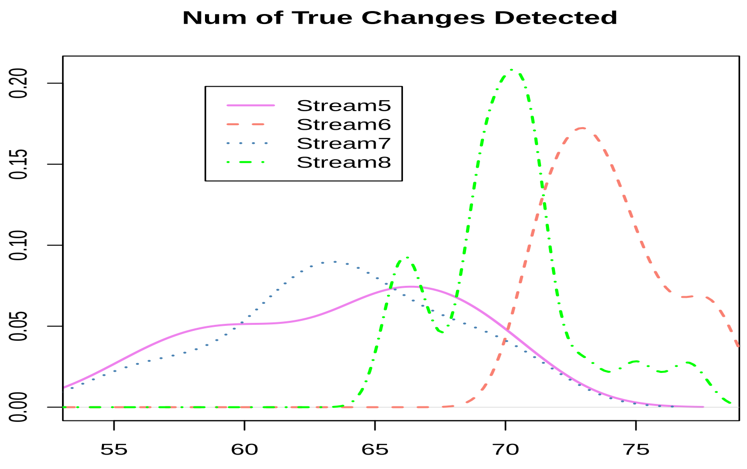


Figure 4.6: Number of true changes detected for $Stream5$ – $Stream8$

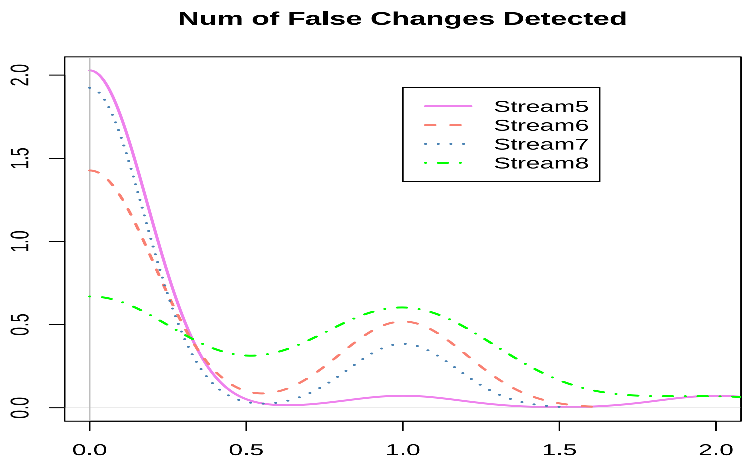


Figure 4.7: Number of false changes detected for *Stream*5 – *Stream*8

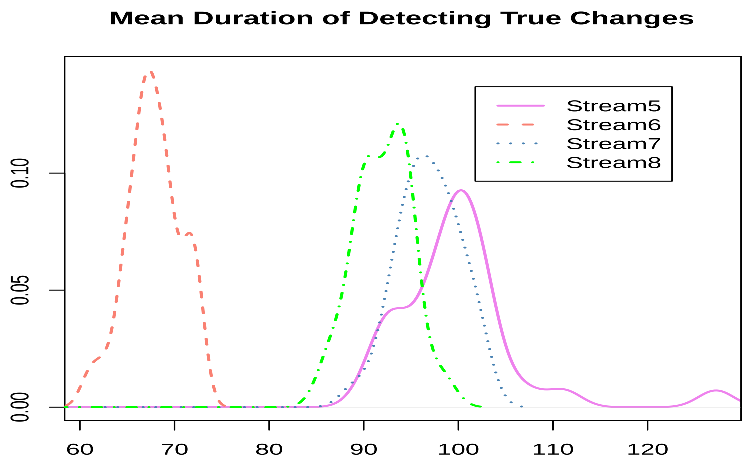


Figure 4.8: Mean duration for detecting true changes in *Stream*5 – *Stream*8

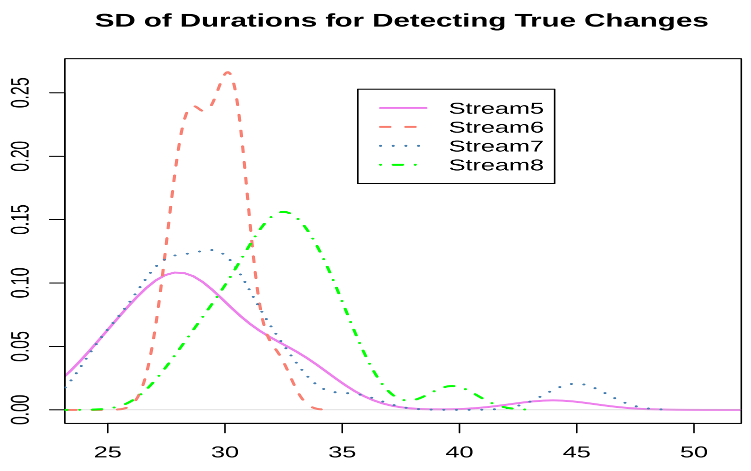


Figure 4.9: Standard deviation of duration for detecting true changes in *Stream*5 – *Stream*8

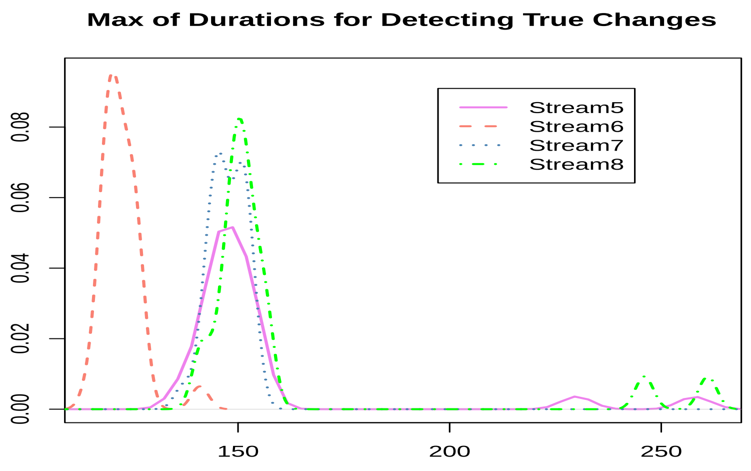


Figure 4.10: Max duration for detecting true changes in *Stream*5 – *Stream*8

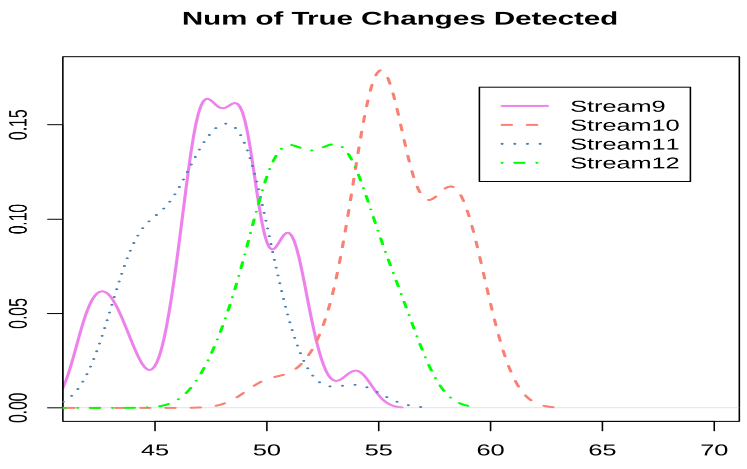


Figure 4.11: Number of true changes detected for $Stream9 - Stream12$

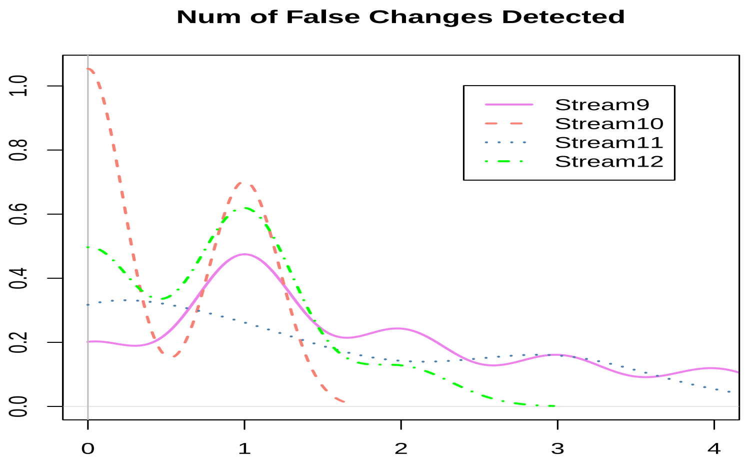


Figure 4.12: Number of false changes detected for $Stream9 - Stream12$

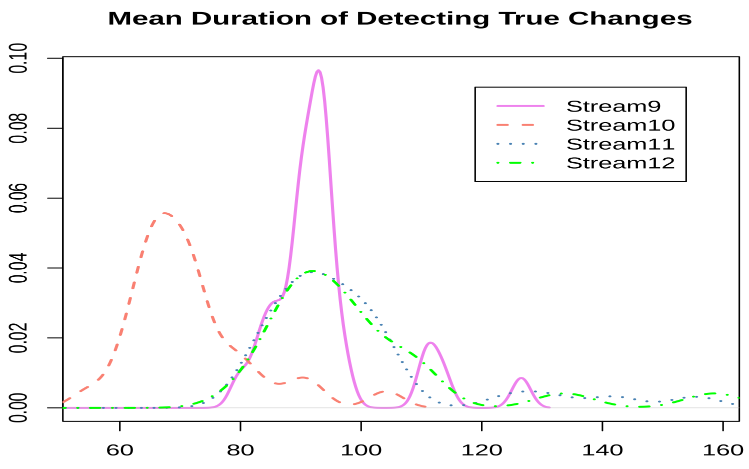


Figure 4.13: Mean duration for detecting true changes in $Stream9$ – $Stream12$

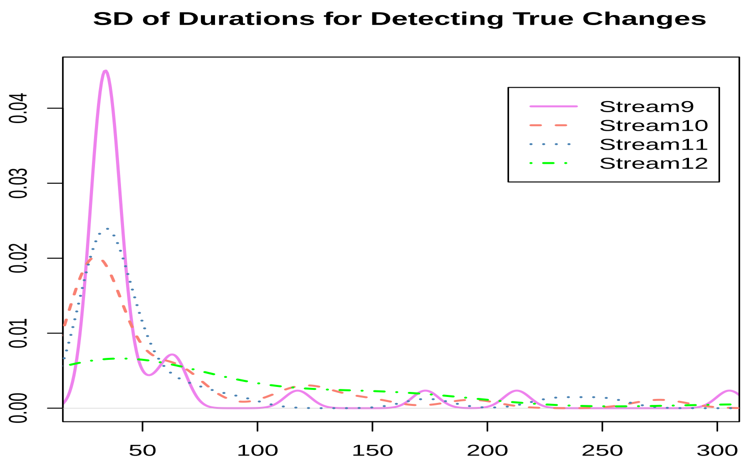


Figure 4.14: Standard deviation of duration for detecting true changes in $Stream9$ – $Stream12$

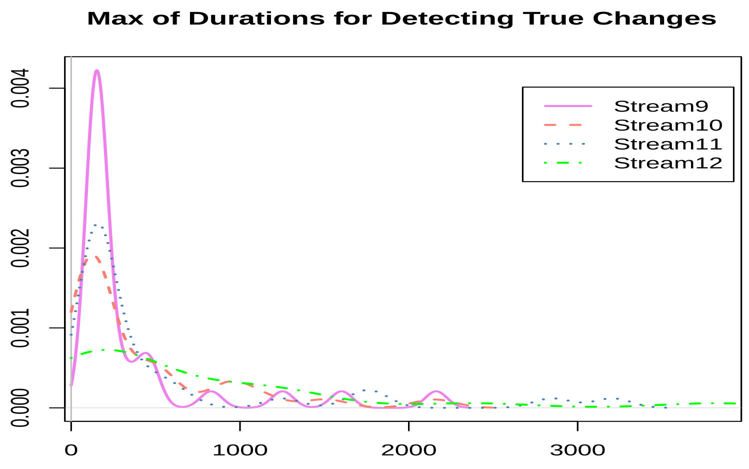


Figure 4.15: Max duration for detecting true changes in *Stream*9 – *Stream*12

These results indicate that the proposed control chart technique performs well on multi-dimensional streams. It can detect most of the true changes in a short time, usually within the same tumbling window where a distribution change occurs. The number of false detection is low for all stream types except *Stream*2.

It can be noted that both the number of true detections and false detections are lower when the streams have higher dimensions. This is because when number the dimensions is higher, the correlations among dimensions make distribution in the stream more difficult to capture. Therefore, the distribution changes are more difficult to detect.

Comparing to distribution shifts, distribution drifts are more difficult to detect by the proposed technique. This is because distribution drifts are gradual and less significant than distribution shifts. Fewer true changes can be detected in streams with distribution drifts. The time taken for detecting drifts are also longer. In several cases it takes more than 2000 data elements until a distribution drift is detected.

4.5.2 Performance comparison with other technique

To further evaluate the performance of the proposed approach, we compare the multi-dimensional control chart approach with a technique proposed by Song et al. (denoted as "KD") that detects distribution changes between two multi-dimensional data sets [127]. The KD technique takes two multi-dimensional data sets S_1 and S_2 as input. It estimates the distribution P_1 of S_1 using multi-dimensional kernel density estimation and then calculates the likelihood of S_2 being generated by P_1. If the likelihood is lower than the significance level, then the null hypothesis H_0 that asserts S_1 and S_2 have identical distribution is revoked, and a distribution change is reported.

Let t_a be the last time a distribution change is reported by the KD approach and let t be the current timestamp. The first data set S_1 used in KD is set as $S_1 = S[t_a, t)$. We set the second data set $S_2 = S[t, t + \Delta]$, where Δ is the same as the size of the tumbling window W in our control chart approach. When W tumbles and a new data set $S(t + \Delta, t + 2 * \Delta]$ arrives, if null hypothesis H_0 is true for S_1 and S_2, then set $S_1 = S_1 + S_2 = S[t_a, t + \Delta]$ and $S_2 = S(t + \Delta, t + 2 * \Delta]$. Otherwise, if H_0 is revoked, then a distribution change is reported and we set $t_a = t$, $S_1 = S(t, t + \Delta]$, and $S_2 = S(t + \Delta, t + 2 * \Delta]$.

We apply both techniques on stream types $Stream2$, $Stream5$, and $Stream12$. All parameter settings are the same as in Section 4.5.1. The experimental results on the five criteria are demonstrated in Figures 4.16 – 4.20, where "CC" is the control chart-based approach and "KD" is the kernel density based approach.

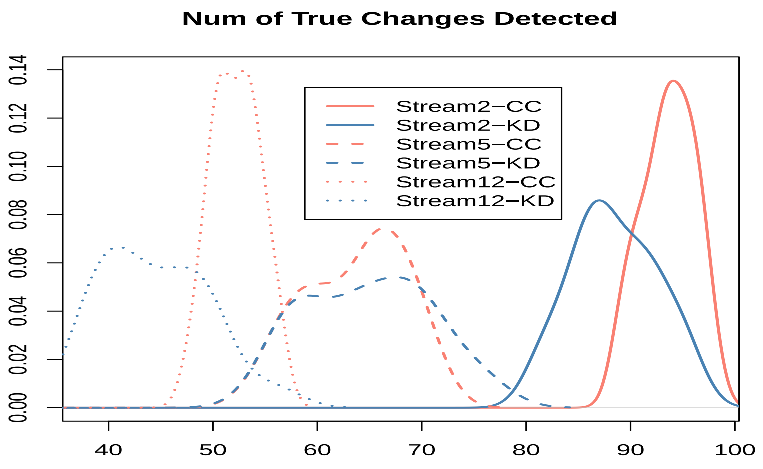


Figure 4.16: Number of true changes detected using CC and KD

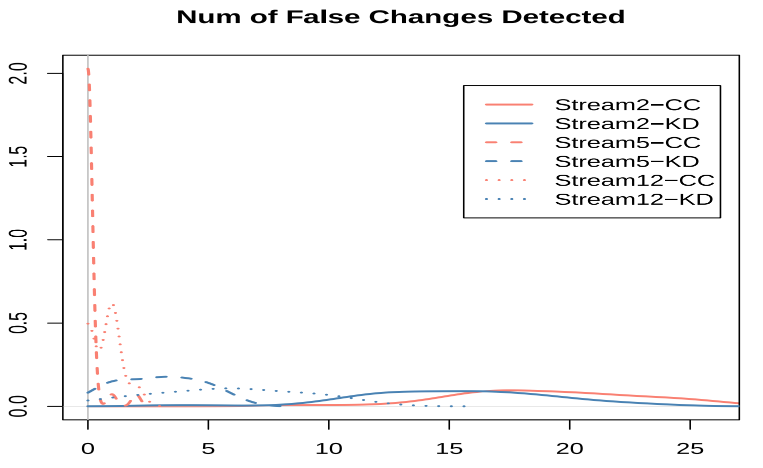


Figure 4.17: Number of false changes detected using CC and KD

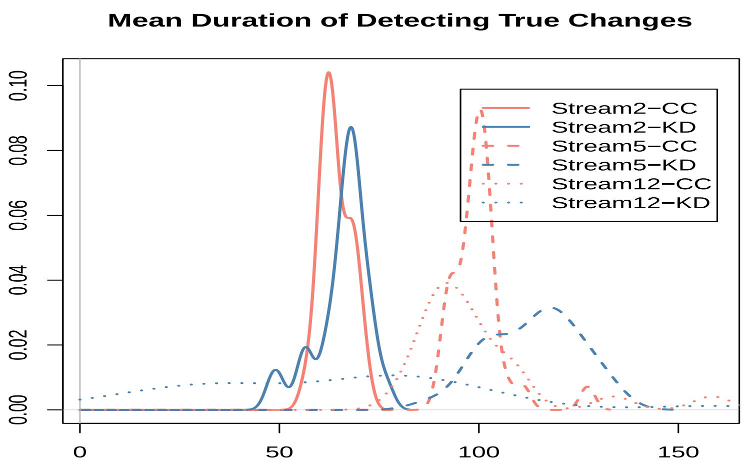


Figure 4.18: Mean duration for detecting true changes using CC and KD

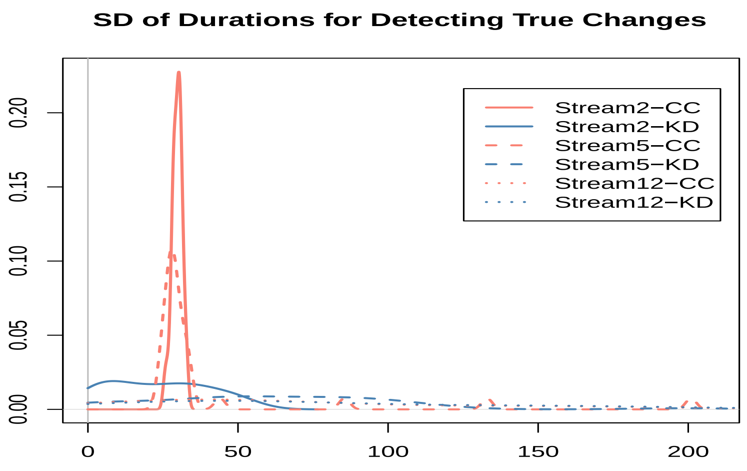


Figure 4.19: Standard deviation of duration for detecting true changes using CC and KD

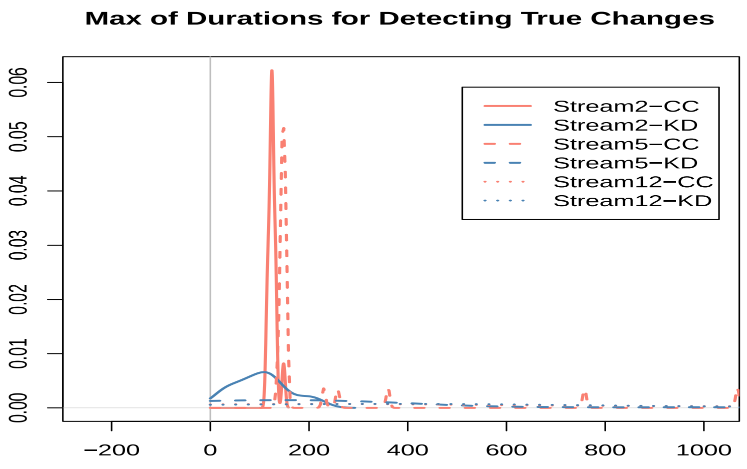


Figure 4.20: Max duration for detecting true changes using CC and KD

From these results, it can be seen that the proposed control chart approach can detect more true distribution changes than KD for all stream types. The control chart approach also detects less false changes than KD for stream types $Stream5$ and $Stream12$. The number of false detections of KD is slightly less than control chart approach. Both techniques have comparable true change detection durations. Hence, it can be concluded that the proposed approach outperforms KD on the overall performance of detecting changes in these three stream types.

One major reason why KD in general detects less true changes and more false changes than control chart approach is, to make an accurate estimation of the kernel density of a data set, the size of this data set must be sufficiently large, e.g., with at least thousands of data. Hence, when a data set contains only 100 data elements, the kernel estimation is highly unreliable and, hence, the performance of KD decreases. In contrast, a control chart based approach that only focuses on key features of the stream can generate change detection results with high precision and low recall using small samples. This is one major advantage of control chart based approach.

Note that KD is a generic approach that can detect all kinds of distribution changes. Hence, it may have better performance than the proposed control chart if the type of changes in the stream is not mean changes. The time complexity of multi-dimensional kernel estimation is $O(dn^2)$, where d is the number of dimensions and n is the size of data set. The worst-case time complexity of detecting changes in a data stream S using KD approach is $O(dn^3)$, where n is the total number of data in S. Hence, KD may not be applicable for data streams with high arrival rate. In contrast, as discussed in Section 4.3.3, the proposed approach has worst-case time complexity $O(d^3n)$. Since n is usually significantly larger than d, the proposed control chart has worst-case run time complexity significantly lower than KD and, thus, is more suitable for streaming data.

4.6 Summary

Distribution change detection for multi-dimensional data streams is important, but difficult. Most of the proposed change detection techniques are only suitable for streams in single-dimensional environments. We extend the control chart approach proposed in Chapter 3 and develop a new technique for detecting mean changes of data streams with multiple dimensions. A multi-dimensional control chart is built to monitor the data stream. If the mean value of a current sample set falls beyond the control limit, then a distribution change is reported. This proposed control chart technique is efficient with promising performance as demonstrated by the experiments.

Chapter 5

Mining Frequent Itemsets in Time-Changing Streams

Mining frequent itemsets in data stream applications is beneficial for a number of purposes such as knowledge discovery, trend learning, fraud detection, transaction prediction and estimation [41, 63, 104]. In this chapter, a false-negative oriented technique, called TWIM, is proposed. The proposed technique can find most of the frequent itemsets, detect distribution changes, and update the mining results accordingly for dynamic data streams.

5.1 Motivation

The problem of mining frequent itemsets has long been recognized as important for many applications such as fraud detection, trend learning, customer management, marketing and advertising. However, the characteristics of stream data – unbounded, continuous, fast arriving, and time-changing – make this a challenging task. Existing mining techniques that focus on relational data cannot handle streaming data well [50].

The problem of mining frequent *items* has been extensively studied [23, 36, 41, 75]. The common assumptions are that the total number of items is too large for memory-intensive solutions to be feasible. Mining frequent items over a data stream under this assumption still remains an open problem. However, the task of mining frequent *itemsets* is more difficult than mining frequent items. Even when the number of distinct items is small, the number of itemsets could still be exponential in the number of items, and maintaining frequent itemsets requires considerable more memory.

Mining frequent itemsets is a continuous process that runs throughout a data stream's life-span. Since the total number of itemsets is exponential, it is impractical to keep statistics for each itemset due to bounded memory. Therefore, usually only the itemsets that are already known to be frequent are recorded and monitored, and statistics of other infrequent itemsets are discarded. However, since the distribution of a data stream can change over time, an itemset that was once infrequent can become frequent if the stream changes its distribution. Detecting such changes is an important task, especially for online applications, such as leak detection, network monitoring, and decision support.

Since it is not feasible to maintain all itemsets, it is difficult to detect frequent itemsets when distribution changes happen. Furthermore, even if these itemsets can be detected, it is not possible to obtain their statistics (supports), since mining a data stream is a one-pass procedure and history information is irretrievable. Distribution changes over data streams might have considerable impact on the mining results; however, few of the previous works have addressed this issue.

A number of techniques have recently been proposed for mining frequent itemsets over streaming data. However, as will be discussed in Section 5.3, these techniques have problems in meeting common requirements for processing dynamic data streams: ability to process large numbers of itemsets in real time, low (preferably minimum) memory usage, and ability to cope with time varying distributions.

A new algorithm, called TWIM, is proposed for mining frequent itemsets in real time. TWIM can also predict the distribution change and update the mining results accordingly. The proposed approach maintains two tumbling windows over a data stream: a maintenance window and a prediction window. All current frequent itemsets are recorded and maintained in the maintenance window, and the prediction window is used to keep track of candidates that have the potential of becoming frequent if the distribution of stream values changes. Every time the windows tumble, we check if new frequent itemsets and candidates should be added, and if some existing itemsets and candidates need to be removed from the lists. Since statistics are not kept for every itemset within the windows, memory usage is limited. Experimental results show that while TWIM is as effective as previous approaches for *non-time-varying* data streams, it is superior to them since it can also capture the distribution change for time-varying streams in real-time.

5.2 Preliminaries

The stream of interest for mining frequent itemsets are transactional data streams, where each data element corresponds to a transaction. Examples of such transaction-based data streams include online commerce, web analysis, banking, and telecommunications applications, where each transaction accesses a set of items from a certain item pool, such as inventory, customer list, or a list of phone numbers.

Let $\mathcal{I} = \{i_1, i_2, ..., i_n\}$ be a set of *items*. A *transaction* $\mathcal{T}_i$ accesses a subset of items $\mathcal{I}_i \subseteq \mathcal{I}$. Let $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_{N_t}\}$ be the set of transactions at time t. N_t is the total number of transactions received up to time t. The data stream that contains $\mathcal{T}$ is denoted as $S_{\mathcal{T}}$. Note that the number of items, n, is finite and usually is not very large, while the number of transactions, N_t, will grow monotonically as time progresses.

Definition 6.1. Given a transaction $\mathcal{T}_i \in \mathcal{T}$, and a subset of items $\mathcal{A}_j \subseteq \mathcal{I}$, if $\mathcal{T}_i$ accesses $\mathcal{A}_j$, i.e., $\mathcal{A}_j \subseteq \mathcal{I}_i$, we say $\mathcal{T}_i$ *supports* $\mathcal{A}_j$.

Definition 6.2. Let $sup(\mathcal{A}_i)$ be the total number of transactions that support $\mathcal{A}_i$. If $SUP(\mathcal{A}_i) = sup(\mathcal{A}_i)/N_t > \nu$, where ν is a predefined threshold value, then $\mathcal{A}_i$ is a frequent itemset in S under current distribution. $SUP(\mathcal{A}_i)$ is called the *support* of $\mathcal{A}_i$.

Example 1. Consider a data stream S with $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_5\}$ at time t and a set of items $\mathcal{I} = \{a, b, c, d\}$. Let $\mathcal{I}_1 = \{a, b, c\}, \mathcal{I}_2 = \{a, b, c, d\}$, $\mathcal{I}_3 = \{c, d\}$, $\mathcal{I}_4 = \{a\}$, and $\mathcal{I}_5 = \{a, c, d\}$. If threshold $\nu = 0.5$, then the frequent itemsets are $\mathcal{A}_1 = \{a\}, \mathcal{A}_2 = \{c\}, \mathcal{A}_3 = \{d\}, \mathcal{A}_4 = \{a, c\}$, and $\mathcal{A}_5 = \{c, d\}$, with supports $SUP(\mathcal{A}_1) = SUP(\mathcal{A}_2) = 0.8$, and $SUP(\mathcal{A}_3) = SUP(\mathcal{A}_4) = SUP(\mathcal{A}_5) = 0.6$.

Let $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ...\mathcal{A}_m\}$ be the complete set of frequent itemsets in S under current distribution. The ultimate goal of mining frequent itemsets in data stream S is to find $\mathcal{A}$ in polynomial time with limited memory space. However, it has been proven that the problem of finding $\mathcal{A}$ off-line is NP-hard [109]. The following theorem proves that on-line updating $\mathcal{A}$ for a data stream that grows in real-time is #P-hard.

Theorem 1 *The problem of finding the complete set of frequent itemsets $\mathcal{A}$ in a given transaction-based data stream S with threshold ν is #P-hard.*

Proof If there exists an algorithm that can list all frequent itemsets $\mathcal{A}$ in polynomial time, this algorithm should also be able to count the total number of such frequent itemsets with the same efficiency. Thus, it suffices to show that counting $|\mathcal{A}|$ for any given S and threshold ν is #P-hard.

Let n be the total number of items in $\mathcal{I}$, and let N_t be the total number of transactions at time t. Construct a $n \times N_t$ matrix M. Each element $M_{k,j}$ in M is a Boolean value: $M_{i,j} = 1$ iff $i_k \in \mathcal{T}_j$, and $M_{i,j} = 0$ is otherwise. Hence, there exists a one-to-one mapping between stream S and matrix M.

Any $n \times N_t$ matrix M can be mapped to a monotone-2CNF formula with N_t clauses and n variables. Therefore, the problem of counting $|\mathcal{A}|$ can be reduced to the problem of counting the number of satisfying assignments for a monotone-2CNF formula using polynomial time.

It has been proven that the problem of counting the number of satisfying assignment of monotone-2CNF formulas with threshold ν is #P-hard [106, 154]. Hence, counting $|\mathcal{A}|$ is a #P-hard problem. $\square$

Note that in the proof, N_t does not have to be infinite and does not even have to be a large number. Therefore, even if techniques such as windowing that can reduce the number of transactions N_t are applied, the problem of mining the complete set of frequent itemsets still remains #P-hard. Furthermore, the size of the complete set of frequent itemsets $\mathcal{A}$ can be exponential. An extreme case is that every transaction $\mathcal{T}_j$ in S accesses $\mathcal{I}$ (i.e., $\forall \mathcal{T}_j \in \mathcal{T}_t$, $\mathcal{I}_j = \mathcal{I}$). For such cases, no algorithm can list $\mathcal{A}$ using polynomial time and space. However, note that this proof holds even for the cases where $|\mathcal{A}|$ is not exponential. Even when the actual size of $\mathcal{A}$ is small, the time taken for *searching for* $\mathcal{A}$ is still exponential.

5.3 Related Work

Mining frequent items and itemsets is challenging and has attracted attention recently. Jiang and Gruenwald [74] provide a good review of the research in frequent itemsets and association rule mining over data streams.

The problem of mining frequent *items* and approximating frequency counts has been extensively studied [23, 36, 41, 75]. Much of the work mainly considers the applications where total number of items in a stream is very large and, therefore, under memory-intensive environments, it is not possible to store a counter even for each of the items. However, the problem of mining frequent items is much easier than the problem of mining frequent itemsets. Even when the number of distinct items is small, which is true for many applications, the number of itemsets could be exponential.

One of the classical frequent itemset mining techniques for relational DBMSs is Apriori [8], which is based on the heuristic that if one itemset is frequent, then its supersets may also be frequent. Apriori requires multiple scans over the entire data and, hence, cannot be directly applied in a streaming environment. Many Apriori-like approaches for mining frequent itemsets over streaming data have been proposed in literature [22, 29, 77], and some of them can be applied on dynamic data streams. However, as will be discussed in Section 5.4.2, Apriori-based approaches suffer from a long delay when discovering large sized frequent itemsets and may miss some frequent itemsets that can be easily detected using TWIM.

Yang and Sanver propose a naive approach that can only mine frequent itemsets and association rules that contain only a few items (usually less than three) [163]. When the sizes of potential frequent itemsets are over three, this algorithm may take an intolerably long time to execute.

Manku and Motwani propose the Lossy Counting (LC) algorithm for mining frequent itemsets [93]. LC quickly prunes itemsets with low frequency and, thus, only frequent itemsets remain. Since LC has a very low runtime complexity and is easy to implement, it is one of the most popular stream mining techniques adopted in real-world applications. However, as experimentally demonstrated by a number of studies, LC may not perform well in practice [29, 36, 166], and is not applicable to data streams that change over time.

Chang and Lee propose an algorithm named estDec for finding recent frequent itemsets by setting a decay factor [21]. It is based on the insight that historical data should play a less important role in frequency counting. This approach does not have the ability to detect any itemsets that change from infrequent to frequent due to distribution drifts.

Chi et al. present an algorithm called Moment [31], which maintains *closed* frequent itemsets [159] using a tree structure named CET. The Moment algorithm provides accurate results within the window and can update the mining results when stream distribution changes. However, Moment is not suitable for streams that change distributions frequently, because there might be a long overhead for updating CET when new nodes are added or an itemset is deleted. Furthermore, if the total number of frequent itemsets is larger or their size is large, Moment could consume much memory to store the tree structure and hash tables. Chang and Lee also adopt a sliding window model to mine recent frequent itemsets [22], which suffers from the same problem of large memory usage and may not be feasible in practice.

Most of the techniques proposed in literature are false-positive oriented, that is, the itemsets they find may not be truly frequent ones. False-positive techniques may consume more memory and are not suitable for many applications where accurate results, even if not complete, are preferred. Yu et al propose a false-negative oriented algorithm, called FDPM, for mining frequent itemsets [166]. The number of itemsets monitored in FDPM is fixed and, thus, memory usage is limited. However, this approach cannot detect distribution changes in the stream, because an itemset could be pruned long before it becomes frequent.

5.4 TWIM: Algorithm for Mining Time-Varying Data Streams

We propose an algorithm, called TWIM, for detecting and and maintaining frequent itemsets for any data stream. The proposed algorithm is false-negative oriented: all itemsets that it finds are guaranteed to be frequent under current distribution, but there may be some frequent itemsets that it will miss. However, TWIM usually achieves high recall according to the experimental results. Since it is a false-negative algorithm, its precision is always 100%.

To detect distribution changes in time, a tumbling windows model is applied on S (Section 5.4.1). When windows tumble, the supports of existing frequent itemsets are updated. If a distribution change occurs during the time span of the window, then some frequent itemsets may become infrequent, and vice versa. In most previous techniques, itemsets that are not frequent at the point when the check is performed are simply discarded. Since supports for only frequent itemsets are maintained, infrequent itemsets that become frequent due to distribution change are difficult to detect. Even if such itemsets can be detected somehow, since the historical information is irretrievable, their estimated supports may be far from the true values, which leads to poor precision. Therefore, TWIM maintains a candidate list that contains a list of itemsets that have the potential to become frequent when the distribution of S changes. Since the supports for the candidates are maintained long before they become frequent, their estimated supports have high accuracy. The problem of predicting candidate itemsets and the procedure for reducing the size of candidate lists to reduce memory usage are discussed in Section 5.4.2.

When windows tumble, the supports of all candidates are updated. If a distribution change occurs, some infrequent itemsets are added to the candidate list and some itemsets will be removed from the candidate list according to certain criteria (Section 5.4.3). Candidates with supports greater than ν are moved to the frequent itemset list.

The main TWIM algorithm is given in Algorithm 3. Each procedure is expanded in the following subsections.

Algorithm 3 TWIM Algorithm

```
1: INPUT: Transactional data stream $S$
2:          Tumbling window $W_M$ and $W_P$
3:          Threshold $\nu$ and $\lambda$
4: OUTPUT: A list of frequent itemsets $\mathcal{A}$ and their supports
5: $\mathcal{A} = \Phi$; $\mathcal{C} = \Phi$; $N_t = 0$;
6: $sup(i_1) = sup(i_2) = ... = sup(i_n) = 0$;
7: for all transaction $\mathcal{T}_k$ that arrives in $S$ do
8:   if $W_M$ is not ready to tumble then
9:     Update the supports for all frequent itemsets and candidates
10:  else
11:    //Windows ready to tumble
12:    Call MAINTAIN_CURRENT_FREQSETS;
13:    //Move infrequent itemsets from $\mathcal{A}$ to candidates
14:    Call DETECT_NEW_FREQSETS;
15:    //Check if any itemset in candidate becomes frequent
16:    Call MAINTAIN_CANDIDATES;
17:    //Add new candidates
18:    Call UPDATE_CANDIDATE_SUP;
19:    //update supports for all candidates
20:    $W_M$ and $W_P$ tumble;
21:  end if
22: end for
```

5.4.1 Tumbling windows design

For most real-life data streams, especially the ones with distribution changes, recent data are more important than historical data. Based on this insight, a tumbling windows model is adopted in TWIM to concentrate on recently arrived data.

A time-based tumbling window W_M, called the *maintenance window*, is used to maintain existing frequent itemsets. A smaller W_M is more sensitive to distribution changes in S; however, it will also incur higher overhead as the interval for updating frequent itemset lists and candidate lists is shorter. While larger W_M reduces the maintenance overhead, it cannot detect sudden distribution changes.

Since data streams are time-varying, a frequent itemset can become infrequent in the future, and vice versa. It is easy to deal with the first case. Since counters for all frequent itemsets are maintained, the supports for these frequent itemsets can be updated periodically (every time W_M tumbles). The counters of those itemsets that are no longer frequent will be removed. However, in the latter case, since there is no information about currently infrequent itemsets, it is difficult to tell when the status changes. Furthermore, even if a new frequent itemset can be detected somehow, its support cannot be estimated, as no history exists for it.

To deal with this problem, a second tumbling window W_P, called the *prediction window*, is defined on the data stream. W_P moves together with W_M, aligning the window endpoints. It keeps history information for candidate itemsets that have the potential to become frequent. The size of W_P is larger than W_M and is predefined based on system resources, the threshold ν, and the accuracy requirement of the support computation for candidates. Note that we do not actually maintain W_P; it is a virtual window that is only used to keep statistics. Hence, the size (time length) of W_P can be as large as required. A large prediction window can ensure high accuracy of the estimated supports for candidate itemsets, resulting in high precision. However, it cannot detect sudden distribution changes and may consume more memory as there are more itemsets maintained in the window. A smaller W_P is more sensitive to distribution changes and requires less memory, but the precision of the mining result may be lower.

Figure 5.1 demonstrates the relationship between maintenance window W_M and prediction window W_P. In Figure 5.1, W_M and W_P are the windows before tumbling, while W'_M and W'_P are the windows afterwards[1].

[1]To discuss the maintenance and prediction windows before and after tumbling, W_M and W_P are used to denote the old windows before tumbling, and

When the end of W_M is reached, it tumbles to the new position W'_M. For every iteration that W_M tumbles, W_P also tumbles. This is to ensure that the endpoints of W_M and W_P are always aligned, so that frequent itemsets and candidate itemsets can be updated simultaneously. Therefore, in Figure 5.1, W_P tumbles to its new position W'_P even before its time interval is fully spanned.

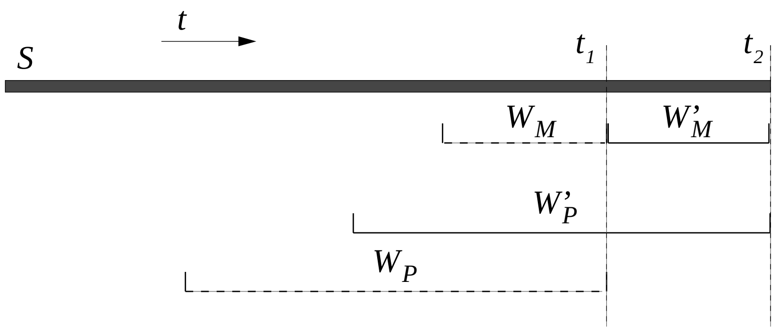


Figure 5.1: Tumbling windows for a data stream

Mining frequent itemsets requires keeping counters for all itemsets; however, the number of itemsets is exponential. Consequently, it is not feasible to keep a counter for all of them and, thus, only counters for the following are maintained:

- A counter for each item $i_j \in \mathcal{I}$. Since the total number of items n is small (typically less than tens of thousands), it is feasible to keep a counter for each item. If each counter is four bytes, then the memory requirement for storing all the counters usually will not exceed 4 MB.
- A counter for each identified frequent itemset. As long as the threshold value ν is reasonable (i.e., not too low), the number of frequent itemsets will not be large.

W'_M and W'_P are used to denote the new windows after tumbling.

- A counter for each itemset that has the potential to become frequent. These itemsets are called *candidate itemsets*[2]. The list of all candidates is denoted as $\mathcal{C}$. The number of candidate itemsets $|\mathcal{C}|$ is also quite limited, as long as the threshold value λ (discussed in Section 5.4.2) is reasonable.

If a frequent itemset becomes infrequent at some point, instead of deleting it right away, it is moved from the set of frequent itemsets $\mathcal{A}$ to $\mathcal{C}$, and its counter is reset. However, this itemset is not removed immediately in the event that it becomes frequent again soon, as will be explained in more detail in Section 5.4.3. The counter for an itemset is removed only when this itemset is removed from candidate list $\mathcal{C}$.

5.4.2 Predicting candidates

To deal with the difficulties of determining which infrequent itemsets may become frequent, a prediction stage is designed to generate a list of candidate itemsets $\mathcal{C}$, which includes itemsets that are most likely to become frequent. The prediction stage happens as W_M and W_P tumble, so that statistics for these candidates can be collected within the new window W'_P.

Any itemset $\mathcal{A}_i$ with $\lambda \leq SUP(\mathcal{A}_i) < \nu$ is considered a candidate and included in $\mathcal{C}$. Here λ is the support threshold for considering an itemset as a candidate. Every time W_P tumbles, all candidates in $\mathcal{C}$ are evaluated. If the counter of one candidate itemset is below λ, it is removed from $\mathcal{C}$ and its counter is released. λ is user defined: smaller λ may result in a higher recall, but consumes more memory since more candidates are generated; a high λ value can reduce memory usage by sacrificing the number of resulting frequent itemsets. Thus, the λ value can be set based on application requirements and available memory.

Every time W_M and W_P tumble, the counters of all candidates and the supports of all items will be updated. If one candidate itemset $\mathcal{A}_i \in \mathcal{C}$ becomes frequent, then $\forall \mathcal{A}_j \in \mathcal{A}$, $\mathcal{A}_k = \mathcal{A}_i \cup \mathcal{A}_j$ might be a candidate.

[2]The question of which itemsets are predicted to have such potential will be discussed in the following sections.

Similarly, if one infrequent item r becomes frequent at the time the windows tumble, then $\forall \mathcal{A}_j \in \mathcal{A}$, $\mathcal{A}_k = \{r\} \cup \mathcal{A}_j$ can be a candidate.

One simple solution is to add all such supersets $\mathcal{A}_k$ into the candidate list $\mathcal{C}$. However, this will result in a large increase of the candidate list's size, since the total number of $\mathcal{A}_k$ for each $\mathcal{A}_i$ or $\{r\}$ can be $|\mathcal{A}|$ in the worst case. The larger the candidate list, the more memory is required for storing counters, and the longer it takes to update the list when W_M and W_P tumble.

As indicated earlier, many existing frequent itemset mining techniques for streams are derived from the popular Apriori algorithm [8]. When an itemset $\mathcal{A}_i$ with size n_i is determined to be frequent, Apriori makes multiple passes to search for its supersets. In the first run (or in the streaming case, the first time W_M and W_P tumble after $\mathcal{A}_i$ is detected), all its supersets with size $n_i + 1$ are added to the candidate list. The size of candidate supersets increases by 1 at every run, until the largest itemset is detected. This strategy successfully reduces the number of candidates; however, in cases when the itemset size $|\mathcal{I}|$ is large, it may take an extremely long time until one large frequent itemset is detected.

Example 2. Let $\mathcal{I} = \{a, b, c, d, e\}$, where $\{a\}, \{b\}, \{c\}$ and $\{d\}$ are frequent itemsets. Assume that, at the point when W_M and W_P tumble, item e becomes frequent; hence, $\{e\}$'s immediate supersets $\{a, e\}, \{b, e\}, \{c, e\}$ and $\{d, e\}$ will be regarded as candidates. If, by next time W_M and W_P tumble, $\{a, e\}$ is detected as frequent, then $\{a, b, e\}, \{a, c, e\}$ and $\{a, d, e\}$ will be added to the candidate list. Assuming the largest itemset $\{a, b, c, d, e\}$ is actually a frequent itemset, it will take time $4 \times |W_M|$ for this itemset to be detected. This delay could be unacceptably long when the maintenance window size is large. Furthermore, the itemset $\{a, b, c, d, e\}$ may never be detected as frequent if the distribution of the stream changes rapidly.

Another problem may occur for such Apriori-like approaches, as demonstrated in the following example.

Example 3. Let $\mathcal{I} = \{a, b, c, d\}$, where a and b are frequent items and itemset $\{a, b\}$ is the only candidate in $\mathcal{C}$. Assume that the next time windows tumble, $SUP(\{a, b\}) < \lambda$ and, hence, itemset $\{a, b\}$ will be discarded from the candidate list $\mathcal{C}$. Assuming t time later c becomes a frequent item, and $\mathcal{A}$ will be $\{\{a\},\{b\},\{c\}\}$, and $\mathcal{C} = \{\{a,c\},\{b,c\}\}$. If by the next run, both $\{a,c\}$ and $\{b,c\}$ are determined to be frequent, then in the end $\mathcal{A}$ will be $\{\{a\},\{b\},\{c\}\ \{a,c\},\{b,c\},\{a,b,c\}\}$. Notice the problem here: itemset $\{a,b\}$ is not included in $\mathcal{A}$. However, since $\{a,b,c\}$ is a frequent itemset, by definition, $\{a,b\}$ must be frequent as well. The problem occurs because $\{a,b\}$ has been discarded long before. When the distribution changes and $\{a,b\}$ turns from infrequent to frequent, it cannot be added to the candidate list if $\{a\}$ and $\{b\}$ are in $\mathcal{A}$ all the time. Although by simply adding all subsets of $\{a,b,c\}$ in $\mathcal{A}$, $\{a,b\}$ can be added back to the frequent itemset list, since Apriori-like approaches only check the supersets of the existing frequent itemsets, the subsets of existing frequent itemsets are not considered.

Continuing with Example 3, assume that item d becomes frequent at time t'. Using Apriori-like approaches, it will take $3 \times |W_M|$ to detect the frequent itemset $\{a,b,c,d\}$. Instead, however, if starting from the current largest itemset in $\mathcal{A}$, that is $\{a,b,c\}$ in this example, then itemset $\{a,b,c,d\}$ is considered a candidate and can be detected as frequent next time windows tumble. Hence, the time for detecting $\{a,b,c,d\}$ is only $|W_M|$. By definition, the complete $\mathcal{A}$ can be obtained by simply computing the power set of $\{a,b,c,d\}$ minus null set ϕ. This approach minimizes both the delay in detection and the size of the candidate list.

Definition 6.3. Given an itemset list $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_m\}$, for $\forall \mathcal{A}' = \{\mathcal{A}'_1, \mathcal{A}'_2, ..., \mathcal{A}'_r\}$, where $\mathcal{A}'_1, \mathcal{A}'_2, ..., \mathcal{A}'_r \in \mathcal{A}$, if $\mathcal{A}'_1 \cup \mathcal{A}'_2 \cup ... \cup \mathcal{A}'_r = \mathcal{A}_1 \cup \mathcal{A}_2 \cup ... \cup \mathcal{A}_m$ and $r < m$, then $\mathcal{A}'$ is a *cover set* of $\mathcal{A}$, denoted as $\mathcal{A}^C$.

For example, given itemset list $\mathcal{A} = \{\{a\},\{b\},\{c\},\{d\},\{a,b\},\{a,b,c\}\}$, $\mathcal{A}^C = \{\{d\}, \{a,b\},\{a,b,c\}\}$ is one cover set.

Definition 6.4. Given an itemset list $\mathcal{A}$ and all its cover set $\mathcal{A}^C_1, \mathcal{A}^C_2, ..., \mathcal{A}^C_q$, if $|\mathcal{A}^C_s| = min(\forall|\mathcal{A}^C_i|)$, where $i = 1, ..., q$, then $\mathcal{A}^C_s$ is the *smallest cover set* of $\mathcal{A}$, denoted as $\mathcal{A}^{SC}$.

For example, the smallest cover set $\mathcal{A}^{SC}$ of itemset list $\mathcal{A} = \{\{a\},\{b\}, \{c\},\{d\},\{a,b\},\{a,b,c\}\}$ is $\{\{d\},\{a,b,c\}\}$.

When a candidate itemset or an infrequent item becomes frequent, the candidate list can be expanded from either direction, i.e., combining the new frequent itemset with all current frequent items in $\mathcal{A}$ or with the smallest cover set of $\mathcal{A}$. The decision as to which direction to follow depends on the application. If the sizes of the potential frequent itemsets are expected to be large, then the smallest cover set could be a better option. In contrast, if small sized frequent itemsets are more likely, then Apriori-like approaches can be applied. However, it is difficult to make such predictions in many real-world scenarios, especially when the distribution of the data streams is changing over time. Hence, a hybrid method is applied in TWIM.

Hybrid approach for generating candidates

The proposed hybrid candidate prediction technique is as follows. At the time W_M and W_P tumble:

- **Step 1.** Detect new frequent itemsets and move them from candidate set $\mathcal{C}$ into set of frequent itemsets $\mathcal{A}$. At the same time, detect any new frequent items and add them into $\mathcal{A}$. This step will be discussed in detail in Section 5.4.3.
- **Step 2.** Update $\mathcal{A} = \mathcal{A} \cup \mathcal{P}(\mathcal{A}^{SC}) - \phi$, where $\mathcal{P}(\mathcal{A}^{SC})$ is power set of $\mathcal{A}$'s smallest cover set. This step is for eliminating the problem discussed in Example 3.
- **Step 3.** Detect itemsets in $\mathcal{C}$ whose supports have dropped below λ. Replace each of these itemsets by its subsets of length one smaller and remove it from $\mathcal{C}$. For example, if itemset $\{a, b, c, d\}$ is no longer a candidate, then itemsets $\{a, b, c\}, \{a, b, d\}, \{b, c, d\}$, and $\{a, c, d\}$ are added into $\mathcal{C}$, and $\{a, b, c, d\}$ is removed. This process can be regarded as the reverse process of a Apriori-like approach.
- **Step 4.** Set $\mathcal{C} = \mathcal{C} - \mathcal{A}$. After Steps 2 and 3, there could be some candidates that are already included in $\mathcal{A}$ and, hence, they are no longer kept in the candidate list $\mathcal{C}$.
- **Step 5.** Let $\mathcal{A}_i$ be a candidate itemset that becomes frequent, or $\{r\}$ where r is an item that turns from infrequent to frequent.

Step 5.1. $\forall \mathcal{A}_k = \{r\} \cup \mathcal{A}_i$, where $r \in (\mathcal{I} - \mathcal{A}_i)$ and $\{r\} \in \mathcal{A}$, if $\mathcal{A}_k$ is not in $\mathcal{A}$, then $\mathcal{A}_k$ is a new candidate.

Step 5.2. $\forall \mathcal{A}_j \in (\mathcal{A} - \mathcal{A}_i)^{SC}$, if $\mathcal{A}_k = \mathcal{A}_j \cup \mathcal{A}_i$ is not in $\mathcal{A}$, then $\mathcal{A}_k$ is a new candidate.

The following example demonstrate the hybrid candidate prediction process:

Example 4. Let $\mathcal{I} = \{a, b, c, d\}$, $\mathcal{A} = \{\{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$, and $\mathcal{C} = \phi$. At the time W_M and W_P tumble:

Step 1. Assume that item d becomes frequent; hence, $\mathcal{A} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$.

Step 2. $\mathcal{A} = \mathcal{A} \cup \mathcal{P}(\mathcal{A}^{SC}) - \phi = \mathcal{A} \cup \mathcal{P}(\{\{d\}, \{a, b, c\}\}) - \phi = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Notice that itemset $\{b, c\}$ is added to $\mathcal{A}$.

Steps 3 and 4. Since currently $\mathcal{C} = \phi$, these two steps are skipped.

Step 5. $\mathcal{C} = \{\{a, d\}, \{b, d\}, \{c, d\}, \{a, b, c, d\}\}$.

After time $|W_M|$, the two windows tumble again:

Case 1: $SUP(\{a, b, c, d\}) \geq \lambda$ and $\{a, b, c, d\}$ becomes frequent.

Step 1.1. $\mathcal{A} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}, \{a, b, c, d\}\}$.

Step 1.2. $\mathcal{A} = \mathcal{P}(\{a, b, c, d\}) - \phi$.

Step 1.3. $\mathcal{C} = \{\{a, d\}, \{b, d\}, \{c, d\}\}$.

Step 1.4. $\mathcal{C} = \mathcal{C} - \mathcal{A} = \phi$.

Step 1.5. All frequent itemsets are detected.

Case 2: $SUP(\{a, b, c, d\}) < \lambda$, and no new frequent itemset detected.

Step 2.1 and step 2.2. $\mathcal{A}$ remains unchanged.

Step 2.3. $\mathcal{C} = \{\{a, d\}, \{b, d\}, \{c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}$.

Step 2.4. Itemset $\{a, b, c\}$ is removed from $\mathcal{C}$, because it is already a frequent itemset.

Step 2.5. No new frequent itemset detected and, thus, this step does not apply.

Property: For each itemset $\mathcal{A}_i$ with size n_i that moves from infrequent to frequent at tumbling point t, let $\mathcal{C}_\mathcal{A}$ be the list of new candidates generated using the hybrid approach at Step 5. Let $|\mathcal{C}_\mathcal{A}|$ be the number of itemsets in $\mathcal{C}_\mathcal{A}$, and $t_\mathcal{A}$ be the total time required for all frequent itemsets in $\mathcal{C}_\mathcal{A}$ to be detected. It can be proven that $|\mathcal{C}_\mathcal{A}| + \frac{2}{|W_M|}t_\mathcal{A} \leq 2m - n_i$, where m is the total number of frequent items in $\mathcal{A}$.

Notice that m, i.e., the number of frequent items, is determined by the nature of the stream and is not related to the chosen mining method. This property indicates that the time and memory usage of the proposed hybrid candidate generation approach are correlated. They are bounded to a constant that is not related to the size of minimal cover set $\mathcal{A}^{SC}$. If, at time t, the size of $\mathcal{C}_\mathcal{A}$ is large (which indicates a large amount of memory consumption), then based on this property, the time for detecting all frequent itemsets in $\mathcal{C}_\mathcal{A}$ will be very short. In other words, large $|\mathcal{C}_\mathcal{A}|$ value indicates a small $t_\mathcal{A}$. Note that once all the frequent itemsets are detected, $\mathcal{C}_\mathcal{A}$ will be removed from $\mathcal{C}$ and, therefore, the large memory usage only lasts for a short time period. In contrast, if it takes longer to detect all frequent itemsets in $\mathcal{C}_\mathcal{A}$, then the memory usage will be quite limited. That is, $|\mathcal{C}_\mathcal{A}|$ must be small when $t_\mathcal{A}$ is large. Hence, this nice property guarantees that the overall memory usage of the proposed hybrid approach is small, and its upper bound is only determined by the number of frequent items in the stream.

Finding smallest cover set

The proposed candidate prediction technique uses smallest cover set of $\mathcal{A}$ to discover the most number of frequent itemsets in the shortest time. An approximate algorithm is proposed in this section that can find a good cover set[3] for a given frequent itemset list $\mathcal{A}$ efficiently in terms of both time and memory. The proposed technique is described as follows:

[3]Informally, a good cover set is one with a small number of itemsets, and the size of each itemset in this cover set is as large as possible. For example, $\{\{a, b, c\}, \{b, c, d\}\}$ is better than $\{\{a, b, c\}, \{d\}\}$, because if $\{b, c, d\}$ is deter-

- **Step 1.** Let $\mathcal{A}^{SC} = \phi$. Build a set of itemsets $\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_m\}$. Let $\mathcal{B}_i = \mathcal{A}_i$ for $\forall \mathcal{A}_i \in \mathcal{A}$.

- **Step 2.** Select the largest itemset $\mathcal{B}_k \in \mathcal{B}$, i.e., $|\mathcal{B}_k| = max(|\mathcal{B}_i|)$, $\forall \mathcal{B}_i \in \mathcal{B}, i = 1, ..., m$. If there is a tie, then select the one with larger corresponding itemset in $\mathcal{A}$. In other words, if $|\mathcal{B}_k| = |\mathcal{B}_r| = max(|\mathcal{B}_i|)$ and $|\mathcal{A}_k| > |\mathcal{A}_r|$, where $\mathcal{A}_k$ and $\mathcal{A}_r$ are the corresponding frequent itemsets of $\mathcal{B}_k$ and $\mathcal{B}_r$ according to step 1, then select itemset $\mathcal{B}_k$. If the tie remains, then randomly select one of the largest itemsets. Set $\mathcal{A}^{SC} = \mathcal{A}^{SC} \cup \{\mathcal{A}_k\}$.

- **Step 3.** For $\forall \mathcal{B}_i \in \mathcal{B}, i = 1, ..., m$, set $\mathcal{B}_i = \mathcal{B}_i - \mathcal{B}_k$. Remove all empty sets from $\mathcal{B}$.

- **Step 4.** Stop if $\mathcal{B} = \phi$. Otherwise, go to step 2.

The following example demonstrates this smallest cover set algorithm:

Example 5. Let $\mathcal{A} = \{\{a, b, c\}, \{a, c, d\}, \{a, d, e\}, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}\}$.

Step 1. $\mathcal{A}^{SC} = \phi$, and $\mathcal{B} = \mathcal{A}$.

Step 2. Select the largest itemset $\mathcal{B}_1 = \{a, b, c\} \in \mathcal{B}$. $\mathcal{A}^{SC} = \{\mathcal{A}_1\} = \{\{a, b, c\}\}$.

Step 3. $\mathcal{B}_2 = \{a, c, d\} - \{a, b, c\} = \{d\}$; $\mathcal{B}_3 = \{a, d, e\} - \{a, b, c\} = \{d, e\}$. All the rest itemsets in $\mathcal{B}$ are empty. Hence, $\mathcal{B} = \{\mathcal{B}_2, \mathcal{B}_3\} = \{\{d\}, \{d, e\}\}$. Go to Step 2.

Step 2-2. Select the largest itemset $\mathcal{B}_3 = \{d, e\} \in \mathcal{B}$. $\mathcal{A}^{SC} = \mathcal{A}^{SC} \cup \{\mathcal{A}_3\} = \{\{a, b, c\}, \{a, d, e\}\}$.

Step 3-2. $\mathcal{B} = \phi$. Algorithm terminates. The final cover set of $\mathcal{A}$ is $\{\{a, b, c\}, \{a, d, e\}\}$.

The run time of this algorithm in the worst case is $(|\mathcal{A}| - n) \times |\mathcal{A}^{SC}|$, where n is the total number of frequent items in the stream. Hence, this algorithm is very efficient in practice.

---

mined to be frequent, many subsets can be added into $\mathcal{A}$.

Updating candidate support

For any itemset that changes its status from frequent to infrequent, instead of being discarded immediately, it remains in the candidate list $\mathcal{C}$ for a while in the event that distribution drifts back quickly and it becomes frequent again.

Every time W_M and W_P tumble, $\mathcal{C}$ is updated: any itemset $\mathcal{A}_i \in \mathcal{C}$ with $SUP(\mathcal{A}_i) < \lambda$ along with its counter is removed, and new qualified itemsets are added resulting in the creation of new counters for them.

For an itemset $\mathcal{A}_i$ that has been in the candidate list $\mathcal{C}$ for a long time, if it becomes frequent at time t_j, its support may not be greater than the threshold ν immediately, because the historical transactions (i.e., the transactions that arrive in the stream before t_j) dominate in calculating $SUP(\mathcal{A}_i)$. Therefore, to detect new frequent itemsets in time, historical transactions need to be eliminated when updating $SUP(\mathcal{A}_i)$ for every $\mathcal{A}_i \in \mathcal{C}$.

Every time W_P tumbles, some of the old transactions expire from W_P. For any itemset $\mathcal{A}_i$ that remains in $\mathcal{C}$, $SUP(\mathcal{A}_i)$ is updated to eliminate the effect of those historical transactions that are no longer in W_P.

W_M and W_P tumble every $|W_M|$ time units. At the time W_M and W_P tumble, the transactions that expire from W_P are those transactions that arrived within the oldest $|W_M|$ time span in W_P. Hence, a checkpoint can be maintained every $|W_M|$ time intervals in W_P, denoted as $chk_1, chk_2, ..., chk_q$, where chk_1 is the oldest checkpoint and $q = \lfloor |W_P|/|W_M| \rfloor$. For each $\mathcal{A}_j \in \mathcal{C}$, the number of transactions arriving between chk_{i-1} and chk_i that access $\mathcal{A}_j$ are recorded, denoted as $sup_i(\mathcal{A}_j)$. At the time W_M and W_P tumble, transactions before checkpoint chk_1 are expired from W_P and $sup(\mathcal{A}_j)$ is updated as $sup(\mathcal{A}_j) = sup(\mathcal{A}_j) - sup_1(\mathcal{A}_j)$. Note that after tumbling, a new checkpoint is added and chk_2 becomes the oldest checkpoint.

The procedures for maintaining candidate list $\mathcal{C}$ and updating candidate counters are given in Algorithm 4 and Algorithm 5, respectively.

Algorithm 4 MAINTAIN_CANDIDATES

```
1: $\mathcal{A} = \mathcal{A} \cup \mathcal{P}(\mathcal{A}^{SC}) - \phi$;
2: for all $\mathcal{A}_i \in \mathcal{C}$ do
3:    if $S(\mathcal{A}_i) < \lambda$ then
4:       for all $j \in \mathcal{A}_i$ do
5:          $\mathcal{C} = \mathcal{C} \cup (\{\mathcal{A}_i - \{j\}\})$
6:       end for
7:       $\mathcal{C} = \mathcal{C} - \{\mathcal{A}_i\}$;
8:       remove $sup(\mathcal{A}_i)$; remove $offset(\mathcal{A}_i)$;
9:       //the concept of offset is presented in the next section
10:   end if
11: end for
12: $\mathcal{C} = \mathcal{C} - \mathcal{A}$;
13: for all $\mathcal{A}_i$ = detect_new_freqset() do
14:    for all $\{j\} \in \mathcal{A}$ and $j \notin \mathcal{A}_i$ do
15:       $\mathcal{A}_k = \{j\} \cup \mathcal{A}_i$;
16:       if $\mathcal{A}_k \notin \mathcal{A}$ then
17:          $\mathcal{C} = \mathcal{C} \cup \{\mathcal{A}_k\}$; $sup(\mathcal{A}_k) = 0$; $offset(\mathcal{A}_k) = N_t$;
18:       end if
19:    end for
20:    for all $\mathcal{A}_j \in \mathcal{A}^{SC}$ do
21:       $\mathcal{A}_k = \mathcal{A}_j \cup \mathcal{A}_i$;
22:       if $\mathcal{A}_k \notin \mathcal{A}$ then
23:          $\mathcal{C} = \mathcal{C} \cup \{\mathcal{A}_k\}$; $sup(\mathcal{A}_k) = 0$; $offset(\mathcal{A}_k) = N_t$;
24:       end if
25:    end for
26: end for
```

Algorithm 5 UPDATE_CANDIDATE_SUP

```
1: for all $\mathcal{A}_i \in \mathcal{C}$ do
2:     $sup(\mathcal{A}_i) = sup(\mathcal{A}_i) - sup_1(\mathcal{A}_i)$;
3:     $offset(\mathcal{A}_i) = N_t$;
4: end for
5: for all $j = 2$ to $q = \lfloor |W_P|/|W_M| \rfloor$ do
6:     $chk_{j-1} = chk_j$; //expire the oldest point $chk_1$
7: end for
8: Set all the records in $chk_q$ to 0;
9: //every time $W_M$ tumbles, a new checkpoint is added to $W_P$
```

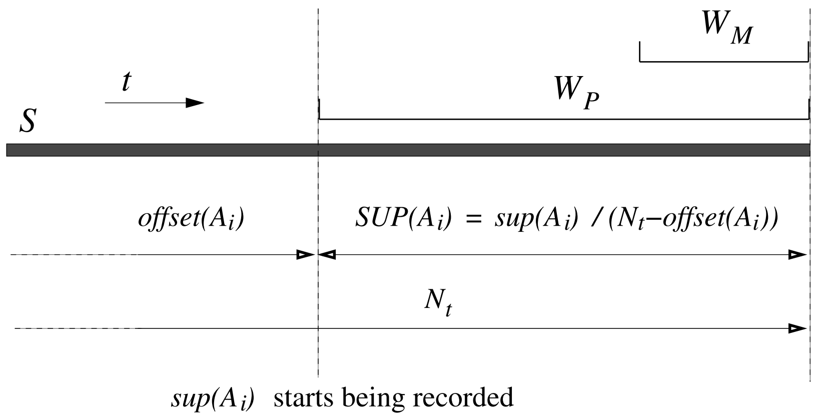
5.4.3 Maintaining current frequent itemsets and detecting new frequent itemsets

Every time W_M tumbles, support values for all the existing frequent itemsets are updated. If the support of an itemset $\mathcal{A}_i$ drops below ν, then it is moved from the set of frequent itemsets $\mathcal{A}$ to the candidate list $\mathcal{C}$. The counter used to record its frequency will be reset to zero, i.e., $sup(\mathcal{A}_i) = 0$. This is to ensure that $\mathcal{A}_i$ may stay in the candidate list for some time if the distribution change is not rapid, as its history record plays a dominant role in its support. By resetting its counter, the effect of historical transactions is eliminated and its support is mainly determined by the most recent transactions. This ensures that the decrease in its support can be detected in a shorter time. During the time-span of W_M, $sup(\mathcal{A}_i)$ will be updated as each new transaction arrives. If $SUP(\mathcal{A}_i) < \lambda$ at the time W_M and W_P tumble, $\mathcal{A}_i$ will be removed from the candidate list.

New frequent itemsets will come from either the infrequent items or the candidate list. Since counters for all items $i \in \mathcal{I}$ are maintained, when an item becomes frequent, it is easy to be detected and its support is accurate. However, for a newly selected frequent itemset $\mathcal{A}_i$ that comes from candidate list $\mathcal{C}$, its support will not be accurate, as most of its historical information is not available. If its support is still calculated as $SUP(\mathcal{A}_i) = sup(\mathcal{A}_i)/N_t$, where N_t is the number of *all* transactions received so far, this $SUP(\mathcal{A}_i)$ will not reflect $\mathcal{A}_i$'s true support. Hence, an offset for $\mathcal{A}_i$, denoted *offset*$(\mathcal{A}_i)$, is applied that represents the number of transactions that are missed in counting the frequency of $\mathcal{A}_i$. $\mathcal{A}_i$'s support at any time $t' > t$ should be modified to $SUP(\mathcal{A}_i) = sup(\mathcal{A}_i)/(N_{t'} - \mathit{offset}(\mathcal{A}_i))$, where $N_{t'}$ is the total number of transactions received at time t', as the data stream monotonically grows.

Because the counters of candidate itemsets are updated every time W_M and W_P tumble to eliminate the history effect (as mentioned in Section 5.4.2), their offsets also need to be reset to the beginning of the new W_P.

Figure 5.2 demonstrates the process of the offset being calculated. Assume that an itemset $\mathcal{A}_i$ is added to the candidate list at the beginning of W_P (time t) and a counter is created for it. At the time W_M and W_P tumble (time t'), $SUP(\mathcal{A}_i)$ is evaluated to check if $\mathcal{A}_i$ should be moved to the set of frequent itemsets $\mathcal{A}$. Since $\mathcal{A}_i$'s historical information before time t is not available,$\mathcal{A}_i$'s offset is adjusted to N_t. Hence, $\mathit{offset}(\mathcal{A}_i) = N_t$, where t is the timestamp when $\mathcal{A}_i$ starts being recorded.



W_M
t
W_P
S
$offset(A_i)$
$SUP(A_i) = sup(A_i) / (N_t - offset(A_i))$
N_t
$sup(A_i)$ starts being recorded

Figure 5.2: Offset for itemset $\mathcal{A}_i$

Note that the supports for such itemsets are no longer based on the entire history, unlike all items that have been tracked throughout the entire life-span of the stream. However, using supports that only depend on recent history should not affect TWIM's effectiveness. This is because the data stream is continuous with a distribution that changes over time and, hence, the mining results over such data stream is temporary – the result at time t_1 may not be consistent with the result at time t_2 ($t_1 < t_2$). Therefore, calculating supports using the entire history may not reflect the **current** distribution correctly or promptly, not to mention the huge amount of memory required for tracking the entire history for each itemset. Experiments demonstrate that the proposed approach is sensitive to both steady and slow changes, and rapid and significant changes, while the existing techniques cannot perform well, especially for the latter case.

The procedures for maintaining $\mathcal{A}$ and detecting new frequent itemsets are given in Algorithms 6 and 7, respectively.

Algorithm 6 MAINTAIN_CURRENT_FREQSETS

```
1: for all $\mathcal{A}_i \in \mathcal{A}$ do
2:   if $sup(\mathcal{A}_i)/(N_t - offset(\mathcal{A}_i)) < \delta$ then
3:     $\mathcal{C} = \mathcal{C} \cup \{\mathcal{A}_i\}$; $\mathcal{A} = \mathcal{A} - \{\mathcal{A}_i\}$;
4:     $sup(\mathcal{A}_i) = 0$; $offset(\mathcal{A}_i) = N_t$;
5:   end if
6: end for
```

Algorithm 7 DETECT_NEW_FREQSETS

```
1: for all $\mathcal{A}_i \in \mathcal{C}$ do
2:   if $sup(\mathcal{A}_i)/(N_t - offset(\mathcal{A}_i)) > \delta$ then
3:     $\mathcal{A} = \mathcal{A} \cup \{\mathcal{A}_i\}$; $\mathcal{C} = \mathcal{C} - \{\mathcal{A}_i\}$;
4:     RETURN $\mathcal{A}_i$;
5:   end if
6: end for
7: for all $j \in \mathcal{I}$ do
8:   if $\{j\} \notin \mathcal{A}$ and $sup(j)/N_t > \delta$ then
9:     $\mathcal{A} = \mathcal{A} \cup \{j\}$; $\mathcal{A}_i = \{j\}$;
10:    RETURN $\mathcal{A}_i$;
11:  end if
12: end for
```

5.5 Experiments

To evaluate TWIM's performance, a set of experiments is designed. TWIM is compared with three techniques: SW method [22], which is a sliding window based technique suitable for dynamic data streams; FDPM [166] approach, which is a false-negative algorithm; and Lossy Counting (LC) [93] algorithm, which is a widely-adopted false-positive algorithm. Details of these approaches have been discussed in Section 5.3. Since neither FDPM nor LC has the ability to detect distribution changes, the experiments are conducted in two stages. In the first stage, these algorithms are compared over data streams without distribution change. In the second stage, dynamic data streams are introduced.

The experiments are carried out on a PC with 3GHz Pentium 4 processor and 1GB of RAM, running Windows XP. All algorithms are implemented using C++.

5.5.1 Effectiveness over streams with stable distribution

In these experiments, the effectiveness of the four algorithms is evaluated over four data streams S_1, S_2, S_3, S_4 that is used in FDPM [166]. The total number of different items in $\mathcal{I}$ is 1000, the average size of transactions in $\mathcal{T}$ is eight, and the number of transactions in each data stream is 100,000. Note that in real-world a data stream can be unbounded. However, none of the algorithms will be affected by the total number of transactions as long as the stream is sufficiently large.

The four streams S_1, S_2, S_3, and S_4 have Zipf-like distributions [171]. The lower the Zipf factor, the more evenly distributed are the data. A stream with higher Zipf factor is more skewed. Since FDPM cannot deal with time-varying streams, these testing data streams that we adopted from FDPM do not have distribution changes. The objective of these experiments is to test the performance of TWIM over streams with stable distribution.

The sizes of the tumbling windows used in TWIM are user determined based on the arrival rate of a data stream. The effect of different window sizes is studied in Section 5.5.4. For ease of representation, the transaction arrival rate for all data streams is fixed in these experiments. Hence, the sizes of W_M and W_P can be represented using transaction counts.

The sizes of the two tumbling windows are $|W_M| = 500$ transactions and $|W_P| = 1500$ transactions. The threshold values ν and λ are set to 0.8% and 0.5%, respectively. The effect of these threshold values is discussed in Section 5.5.2 and Section 5.5.4. The size of the sliding window used in SW is the same as the size of W_M in TWIM, i.e., 500 transactions. The error parameter and reliability parameter used in FDPM and LC are set to $\nu/10$ and 0.1, respectively[4]. According to earlier experiments, this setting can produce the best performance for FDPM and LC [93, 166]. The recall (R) and precision (P) results for all four techniques are shown in Table 5.1.

Table 5.1: Recall and precision comparison of TWIM

Stream	Zipf	TWIM		SW		FDPM		LC	
		R	P	R	P	R	P	R	P
S_1	0.8	0.68	1	0.71	0.74	0.69	1	1	0.52
S_2	1.2	0.87	1	0.79	0.83	0.80	1	1	0.62
S_3	2.0	0.93	1	0.92	0.95	0.95	1	1	0.84
S_4	2.8	1	1	1	1	1	1	1	0.88

These results demonstrate that, FDPM and SW perform slightly better than TWIM when the distribution of a data stream is close to uniform. However, when Zipf is higher, the performance of TWIM is comparable to FDPM and better than SW. When the stream is very skewed, TWIM, SW, and FDPM can all find the exact answer. Note that although LC always has a recall of 100%, its results are unreliable, especially for streams with lower Zipf. These results demonstrate that TWIM performs at least as well

[4]The error parameter ϵ is used to control error bound. Smaller ϵ can reduce errors and increase the recall of FDPM and LC. The memory consumption of FDPM is reciprocal of the reliability parameter [166].

as existing algorithms on streams *without* distribution change. Although the recall of FDPM is claimed to approach 100% at infinity [166], this only holds when the stream has no distribution change during its entire lifespan, which is a very strong and usually incorrect assumption for most real-world applications.

5.5.2 Effect of threshold ν

This set of experiments evaluate the effectiveness of the four algorithms with different values of threshold ν. For this set of experiments, λ is set as $\nu - 0.3\%$. Note that threshold λ is mainly used to control the size of candidate list $\mathcal{C}$. As λ becomes smaller, more candidate itemsets are selected, leading to a higher memory consumption. In contrast, a larger λ may cause TWIM's recall to decrease when mining a dynamic data stream, because there are fewer candidates available. The effect of different λ values on mining time-varying streams is demonstrated in Section 5.5.4. Since the testing data stream in this set of experiment has a steady distribution, the size of $\mathcal{C}$ should not affect the performance of TWIM.

TWIM, SW, FDPM and LC are applied to data stream S_2 (as in the previous experiments) with Zipf 1.2, and ν value varies from 0.4% to 2%. The results are shown in Table 5.2, which demonstrate that the effectiveness of TWIM is comparable with FDPM when ν varies. TWIM's recall is improved with higher ν. This is because a high ν value indicates that only itemsets with extremely high supports are considered frequent itemsets. Such itemsets are distinctive from the rest and, thus, are easier to be detected. Although SW always has a better recall than TWIM and FDPM, its precision never reaches 100%. LC has a low precision even when ν is high (2%).

5.5.3 Effectiveness over dynamic streams

To evaluate the effectiveness of these four algorithms over time-varying data streams, several experiments are conducted.

Table 5.2: Results for varying ν value

ν	TWIM		SW		FDPM		LC	
	R	P	R	P	R	P	R	P
0.4%	0.62	1	0.76	0.57	0.65	1	1	0.44
0.8%	0.83	1	0.85	0.81	0.80	1	1	0.62
1.2%	0.94	1	1	0.87	0.93	1	1	0.74
2%	0.98	1	1	0.99	1	1	1	0.77

We adopted the two data streams S_5 and S_6 used in SW [22]. Each stream contains 1,000,000 transactions. On average, the number of items in the transactions of S_5 and S_6 is five. Both of the streams change their distributions every 20,000 transactions. The distribution changes of S_5 are steady and slow. It takes 4000 transactions for S_5 to complete one distribution change. Whereas S_6 has faster and more noticeable changes: only 800 transactions to change. The sizes of the two tumbling windows are $|W_M| = 400$ transactions and $|W_P| = 1500$ transactions. Threshold values ν and λ are 0.8% and 0.5%, respectively. The mining results after each distribution change for S_5 and S_6 are given in Table 5.3 and Table 5.4.

Table 5.3: Mining results over S_5

change #	TWIM		SW		FDPM		LC	
	R	P	R	P	R	P	R	P
change 1	0.91	1	0.85	0.87	0.82	0.93	1	0.66
change 2	0.93	1	0.86	0.92	0.73	0.87	1	0.51
change 3	0.88	1	0.74	0.84	0.69	0.77	1	0.44
change 4	0.88	1	0.77	0.93	0.72	0.68	1	0.46
change 5	0.92	1	0.83	0.86	0.60	0.68	1	0.35

These results reveal that TWIM and SW adapt to time-varying data streams, while neither FDPM nor LC is sensitive to distribution changes. The more severe the changes, the worse is the performance of FDPM and LC. Moreover, FDPM and LC's performance decrease when more

Table 5.4: Mining results over S_6

change #	TWIM		SW		FDPM		LC	
	R	P	R	P	R	P	R	P
change 1	0.95	1	0.72	0.82	0.87	0.82	1	0.58
change 2	0.97	1	0.71	0.77	0.78	0.81	1	0.51
change 3	0.93	1	0.69	0.80	0.65	0.74	1	0.38
change 4	1	1	0.74	0.71	0.67	0.66	1	0.41
change 5	0.88	1	0.71	0.89	0.53	0.64	1	0.32

distribution changes occur in a stream, whereas TWIM and SW are not affected by the number of changes. SW performs worse than TWIM in both experiments. Mining results of TWIM over the stream with faster and more noticeable distribution changes (S_6) are better than the one that changes slower (S_5), while SW appears to be more suitable to slower and mild changes. Note that as mentioned in Section 5.4.1, the mining results of TWIM can be improved for such slow-drifting data streams by reducing the sizes of W_M and W_P.

5.5.4 TWIM Parameter Settings

Effect of threshold λ

We test TWIM on the time-varying streams S_5 and S_6 described in Section 5.5.3, and vary λ from 0.4% to 1%. The sizes of W_M and W_P are 400 transactions and 1500 transactions, respectively. Threshold value ν is fixed at 1.2%. The results are presented in Table 5.5 and Table 5.6.

According to the results, the performance of TWIM can be improved by decreasing λ. However, as discussed in Section 5.4.2, a low λ value may result in higher memory consumption. The extreme case is $\lambda = 0$. In this case, all infrequent itemsets will be treated as candidates and, thus, the total number of counters is exponential.

Table 5.5: Results for varying λ over S_5

Change #	λ (%)							
	0.4		0.6		0.8		1	
	R	P	R	P	R	P	R	P
change 1	0.96	1	0.97	1	0.88	1	0.72	1
change 2	0.95	1	0.95	1	0.83	1	0.74	1
change 3	0.93	1	0.89	1	0.85	1	0.68	1
change 4	0.98	1	0.94	1	0.88	1	0.75	1
change 5	0.89	1	0.87	1	0.74	1	0.69	1

Table 5.6: Results for varying λ over S_6

Change #	λ (%)							
	0.4		0.6		0.8		1	
	R	P	R	P	R	P	R	P
change 1	0.98	1	0.92	1	0.89	1	0.83	1
change 2	1	1	1	1	0.92	1	0.87	1
change 3	0.93	1	0.91	1	0.84	1	0.77	1
change 4	1	1	0.95	1	0.91	1	0.85	1
change 5	0.95	1	0.95	1	0.90	1	0.83	1

Varying window sizes

To evaluate the effect of tumbling window sizes, TWIM is tested on S_5 and S_6 with $|W_M|$ varying from 200 to 1000 transactions, and $|W_P|$ varying from 1000 to 4000 transactions. Threshold values ν and λ are 0.8% and 0.5%, respectively. The experimental results are shown in Tables 5.7 and 5.8. Since the precisions are always 100% for the proposed false-negative approach, only the recall values are demonstrated in the tables.

Table 5.7: Varying $|W_M|$ and $|W_P|$ over S_5

$\|W_M\|$	$\|W_P\|$	chg 1	chg 2	chg 3	chg 4	chg 5
200	1000	0.93	0.88	0.89	0.91	0.95
400	1500	0.87	0.92	0.85	0.88	0.90
600	2000	0.82	0.88	0.79	0.74	0.82
800	3000	0.75	0.73	0.72	0.67	0.69
1000	4000	0.68	0.72	0.66	0.64	0.61

Table 5.8: Varying $|W_M|$ and $|W_P|$ over S_6

$\|W_M\|$	$\|W_P\|$	chg 1	chg 2	chg 3	chg 4	chg 5
200	1000	0.99	0.97	0.93	0.95	0.88
400	1500	0.94	0.97	0.91	1	0.87
600	2000	0.89	0.92	0.85	0.89	0.86
800	3000	0.82	0.84	0.79	0.86	0.77
1000	4000	0.78	0.81	0.81	0.75	0.73

These results reveal that larger windows size may reduce TWIM's recall, since sudden distribution changes will be missed. However, note that as mentioned in Section 5.4.1, large windows ensure high accuracy of the estimated supports for candidate itemsets.

5.5.5 Memory usage

The major memory requirements for TWIM are the counters used for all items, frequent itemsets, and candidates. To reflect the memory usages of the proposed approach, the maximal numbers of counters that are create for each experiment are reported as follows.

Table 5.9 presents the memory usage of TWIM, FDPM, and LC for mining data sets S_1, S_2, S_3 and S_4. Given that each counter takes four bytes, the memory requirement for mining these data streams using TWIM is around 60 KB. According to Table 5.9, the memory consumed by SW is about four times of TWIM's memory usage. TWIM uses slightly more memory than FDPM, and LC consumes the least memory.

Table 5.9: Maximal counters for mining S_1 - S_4

Stream	Maximal Counters			
	TWIM	SW	FDPM	LC
S_1	11892	47606	8478	7129
S_2	14533	59438	10128	8722
S_3	18002	71040	13502	11346
S_4	16115	56442	11764	10098

Table 5.10 indicates TWIM's memory usage for the experiments in Section 5.5.4. It demonstrates that the memory consumption of TWIM is inversely correlated to threshold λ. The maximum memory consumption is around 228 KB for S_5 and 191 KB for D_6.

Table 5.10: Maximal counters when λ varies

Stream	λ (%)			
	0.4	0.6	0.8	1
Max Counter-S_5	64432	47210	36778	32002
Max Counter-S_6	51301	42676	35209	28123

To evaluate the effect of window sizes on memory usage, we present in Table 5.11 the maximum number of counters created for experiments in Section 5.5.4.

Table 5.11: Maximum counters when $|W_M|$ and $|W_P|$ varies

$\|W_M\|$	$\|W_P\|$	max Counter - S_5	max Counter - S_6
200	1000	42398	39901
400	1500	50006	44872
600	2000	56020	51922
800	3000	59891	54646
1000	4000	65335	59043

The maximum memory usage for mining S_5 and S_6 are around 251KB and 225KB, respectively. These results show that larger windows sizes result in more counters to be used. Furthermore, the number of counters used for a stream with slow distribution changes is greater than the number of counters for a stream that changes fast.

5.5.6 CPU time analysis

Since TWIM is a window-based approach while neither FDPM nor LC use windows, it is difficult to fairly compare their CPU times. However, to demonstrate that TWIM is efficient for high-speed data streams, a set of experiments is conducted.

By analyzing Algorithm 3, it is clear that TWIM performs the greatest amount of work when $|W_M|$ and $|W_P|$ tumble. Hence, the average run time of TWIM at each tumble point is tested for streams S_1 to S_6. The results are demonstrated in Table 5.12. These results indicate that TWIM is an efficient algorithm suitable for online streams. Notice that streams with distribution changes (S_5 and S_6) require slightly longer processing time, because $\mathcal{A}$ and $\mathcal{C}$ are updated more frequently.

Table 5.12: CPU time for TWIM

Streams	S_1	S_2	S_3	S_4	S_5	S_6
Average run time (ms)	3.3	4.0	2.5	3.7	5.3	5.9

5.6 Summary

Mining frequent itemsets is important for many real-world applications. Many existing techniques do not support dynamic data streams. A novel false-negative orientated algorithm, called TWIM, for change detection and mining frequent itemsets is proposed in this chapter. This approach has the ability to detect distribution changes in a data stream and update mining results in real-time.

TWIM uses two tumbling windows to maintain current frequent itemsets and to predict distribution changes. A list of candidate itemsets is generated and updated during mining. The candidates are the itemsets that have the potential to become frequent if distribution changes. Every time the two tumbling windows move, both the candidate list and frequent itemset list are updated. Candidates that become frequent are moved to the frequent itemset list, new candidates are added, and itemsets that no longer have supports greater than threshold value λ are removed. Unlike most existing algorithms that are false-positive oriented, the proposed approach produces only true frequent itemsets and requires less memory. Experimental results demonstrate that TWIM has promising performance on mining data streams with or without distribution changes.

Chapter 6

Conclusions

Many of today's applications generate data in the form of continuous, fast-arriving, and time-changing streams. Mining dynamic data streams for knowledge discovery has become increasingly important. However, traditional data mining techniques that make multiple passes over data or that ignore distribution changes are not suitable for dynamic data streams. New techniques that are efficient to run in real-time, that only require one-pass of the stream, and that are sensitive to distribution changes are desired.

6.1 Summary of work

This thesis focuses on developing techniques for mining time-changing data streams. One of the major differences that distinguish data stream mining from traditional data mining is that mining data streams is a continuous process that lasts over the entire life-span of the stream. The ever-changing distribution of a dynamic data stream makes the stream mining task difficult, because once the underlying distribution that generates the data in the stream changes, the data mining model built previously may no longer be accurate or efficient, and the previous mining results may be invalid for the new distribution. Hence, distribution change detection is a fundamental problem for dynamic data stream mining.

Two techniques are proposed in this thesis aiming to solve the problem of distribution change detection in streaming data (Chapter 3). These techniques are not tied to specific stream mining tasks and can be generalized to detect changes in generic data streams. Existing stream mining approaches that are only suitable for streams with stable distributions can support dynamic data streams by adopting the proposed techniques.

The first approach is designed to represent the distribution that generates the data in the stream using a small data set. Two windows, a reference window and a observation window, are used to maintain data sets that represent the current and new distributions of the stream. An intelligent merge-and-select sampling approach is proposed that can dynamically update the reference window. The small data set in the reference window can represent the current distribution of the stream with high accuracy. Since many existing statistical tools require a large amount of sample data to estimate the distribution, with this proposed approach, the accuracy of change detection techniques that adopt these statistical tools can be improved.

We also proposed a framework for mining streams with periodically occurring distributions. Once a distribution change is detected, the new distribution is compared with all preserved ones to determine if it is a reoccurring distribution. Identifying such periodic distributions can greatly reduce the data mining time, since previous mining results on the same distribution can be directly output as the new mining results.

The second approach aims to detect mean and standard deviation changes in the distributions of a stream. This technique detects distribution changes efficiently with high accuracy by using control charts. Data elements of the stream are continuously fed into the control charts and the distribution change is monitored by control limits. This control chart-based approach can detect both fast and severe distribution shifts and slow and steady distribution drifts with satisfying performance.

Data elements in different data streams can have various forms and each data element may contain multiple attributes (dimensions). In many real-world applications, dimensions in the stream are correlated. Detecting distribution changes for such multi-dimensional streams is more difficult because of these correlations. Very few multi-dimensional change detection techniques over data streams have been proposed in literature. We observe that, by using a covariance matrix to represent the correlations among all dimensions, the proposed control chart-based approach can be extended to detect distribution changes in multi-dimensional streams (Chapter 4).

Although ad-hoc approaches lack the flexibility to be plugged into any data mining processes, they can obtain better performance by taking advantages of previous mining results and of special features of the streams. Therefore, we study one of the most important mining tasks, frequent itemset mining, and develop a mining technique for this task.

The major difficulty of mining frequent itemsets in dynamic data streams is that data stream can be scanned in only one-pass. Non-frequent itemsets are not monitored due to memory concerns and, hence, once the distribution changes, it is impossible to check if a non-frequent itemset has become frequent following the change. A frequent itemset mining technique that maintains a list of candidate itemsets is proposed (Chapter 5). The candidates are the non-frequent itemsets that have the potential to become frequent if the distribution changes. These candidates are generated by a set of heuristics and are updated when new sets of data elements arrive. Candidates that become frequent are output as new mining results, new candidates are added according to the heuristics, and itemsets that are no longer qualified as candidates are removed from the list. This false-negative approach demonstrates promising performance in the experiments.

6.2 Directions for future research

This research can be extended along the following directions.

- Automatic window size setting.

  Many of the proposed techniques in the thesis adopt window models to obtain data elements and to control the time intervals at which change detection and mining processes to be triggered. The sizes of the windows used in these techniques are important parameters that affect the performance. In the current work, these parameters are assigned to be user-specific. This requires the users to have certain level of knowledge on the data stream and the mining technique. Ideally, the window size should be set automatically and can be adjusted based on the mining results. Although the issue is discussed and experiments are presented, there is still human involvement. Hence, a possible future research topic is to automatically tuning the windows size.

- Determining number of partitions for representative set selection.

  Currently, the number of partitions k for selecting representative set in Chapter 3 is user defined. However, as has been discussed in Chapter 3, k value is determined by the "shape" of the distribution. Smaller k should be chosen if the distribution is less "complex", i.e., with less peaks and valleys, and vice versa. Hence, an algorithm that estimates the "complexity" in the distribution would be useful for setting k value. The design of such an algorithm is a possible future research topic.

- Recognizing and eliminating noise.

  The control chart-based change detection technique proposed in Chapter 3 and its multi-dimensional extension in Chapter 4 are both sensitive to noises. One of the possibilities for improving these approaches is to develop effective noise recognition and elimination algorithms. However, it should be noted that outliers are not always noise and may contain critical information for some streaming applications. Hence, different noise recognition algorithms should be applied for certain types of streams.

- Mining k-th most frequent itemsets.

  It has been proven in Chapter 5 that mining frequent itemsets in dynamic data streams is an #-P problem and, hence, only approximate solutions exist. However, if the problem of finding *all* frequent itemsets is reduced to the problem of finding the top-k most frequent itemsets, then a polynomial algorithm that can generate exact results might be found. Further research work needs to be done to provide theoretical proof of the existence (or non-existence) of such solution, and to find the algorithm if it exists.

- Designing an experimental framework for evaluating the performance of the proposed frequent itemsets mining technique.

  Currently only six streams are used in the experiments of evaluating the performance of the proposed frequent itemsets mining technique. The number of sample streams is too few to make the experimental results conclusive. An experimental framework is required to generate different stream types and large amount of streams of each stream type for extensively studying the performance of the proposed technique. The framework may have a similar form as the ones designed in Chapter 3 and Chapter 4, where several stream types are defined and 100 streams for each stream type is automatically generated. The number of distribution changes, the time when the changes occur, the type of distribution in a stream, the type of distribution changes (i.e., distribution shifts or drifts), and the number of items in each transaction should be set as parameters that can be tuned dynamically.

Bibliography

[1] G. Abdulla, T. Critchlow, and W. Arrighi. Simulation data as data streams. *ACM SIGMOD Record*, 33(1):89–94, 2004.

[2] D. Adams. A model for parallel computations. *Parallel Processor Systems, Technologies, and Applications*, pages 311–333, 1970.

[3] C. Aggarwal. A framework for diagnosing changes in evolving data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 575–586, 2003.

[4] C. Aggarwal. On change diagnosis in evolving data streams. *IEEE Trans. Knowledge and Data Eng.*, 17(5):587–600, 2005.

[5] C. Aggarwal, J. Han, J. Wang, and P. Yu. A framework for clustering evolving data streams. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 81–92, 2003.

[6] C. Aggarwal, J. Han, J. Wang, and P. Yu. A framework for projected clustering of high dimensional data streams. pages 852–863, 2004.

[7] C. Aggarwal, J. Han, J. Wang, and P. Yu. On demand classification of data streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 503–508, 2004.

[8] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 487–499, 1994.

[9] F. Aparisi and J. Garcia-Diaz. Design and optimization of ewma control charts for in-control, indifference, and out-of-control regions. *Computers and Operations Research*, 34(7):2096–2108, 2007.

[10] M. Ayers, D. Wolock, G. McCabe, L. Hay, and G. Tasker. Sensitivity of water resources in the delaware river basin. *US Geological Survey, Open-File Report 92-52*, 1994.

[11] S. Babu, K. Munagalat, J. Widom, and R. Motwani. Adaptive caching for continuous queries. pages 118–129, 2005.

[12] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Sys.*, 33(1), 2008.

[13] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the 1st conf. on Symposium on Networked Systems Design and Implementation*, page 15, 2004.

[14] M. Basseville and I. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, 1993.

[15] R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[16] P. Bosman. Linkage information processing in distribution estimation algorithms. In *Proc. the Genetic and Evolutionary Computation Conf. (GECCO)*, pages 60–67, 1999.

[17] L. Carbonara, H. Roberts, and B. Egan. Data mining in the telecommunications industry. *Principles of Data Mining and Knowledge Discovery*, 1263:396, 1997.

[18] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Sys.*, 27, 2002.

[19] S. Chandrasekaran and M. Franklin. Psoup: A system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, 2003.

[20] J. Chang and H. Kum. Frequency-based load shedding over a data stream of tuples. *Journal of Information Science: an Int. Journal*, 179(21):3733–3744, 2009.

[21] J. Chang and W. Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 487–492, 2003.

[22] J. Chang and W. Lee. A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineering*, 20:753–762, 2004.

[23] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. Int. Colloquium on Automata, Languages, and Programming*, pages 693–703, 2002.

[24] M. Charikar, K. Chen, and R. Motwani. Incremental clustering and dynamic information retrieval. In *Proc. ACM Symp. on Theory of Computing*, pages 626–635, 1997.

[25] M. Charikar, L. O'Callaghan, and R. Panigrahy. Better streaming algorithms for clustering problems. In *Proc. ACM Symp. on Theory of Computing*, pages 30–39, 2003.

[26] J. Chen, D. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 379–390, 2000.

[27] L. Chen and R. Ng. On the marriage of edit distance and lp-norms. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 792–803, 2004.

[28] L. Chen, T. Ozsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 491–502, 2005.

[29] J. Cheng, Y. Ke, and W. Ng. Maintaining frequent itemsets over high-speed data streams. In *Proc. Pacific-Asia Conf. on Knowledge Discovery and Data Mining* PAKDD, pages 462–467, 2006.

[30] M. Chernick. *Bootstrap Methods, A practitioner's guide.* Wiley Series in Probability and Statistics, 1999.

[31] Y. Chi, H. Wang, P. Yu, and R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proc. 2004 IEEE Int. Conf. on Data Mining*, pages 59–66, 2004.

[32] Y. Chi, P. Yu, H. Wang, and R. Muntz. Loadstar: A load shedding scheme for classifying data streams. In *Proc. SIAM International Conference on Data Mining*, pages 1302–1305, 2005.

[33] F. Chiew and T. McMahon. Detection of trend or change in annual flow of australian rivers. *Int. Journal of Climatology*, 13(6):643–653, 1993.

[34] K. Chuang, H. Chen, and M. Chen. Feature-preserved sampling over streaming data. *ACM Trans. Knowledge Discovery from Data*, 2(4), 2009.

[35] G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. pages 13–24, 2005.

[36] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *Proc. 22nd ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 296–306, 2003.

[37] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55:29–38, 2004.

[38] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. pages 25–36, 2005.

[39] F. David. The moments of the z and f distributions. *Biometrika*, 36:394–403, 1949.

[40] M. Gonzalez de la Parra and P. Rodriguez-Loaiza. Application of the multivariate t2 control chart and the mason-tracy-young decomposition procedure to the study of the consistency of impurity profiles of drug substances. *Quality Engineering*, 16(1):127–142, 2003.

[41] E. Demaine, A. Lopez-Ortiz, and J. Munro. Frequency estimation of internet packet streams with limited space. In *Proc. 10th Annual European Symposium on Algorithms*, pages 348–360, 2002.

[42] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society.*

[43] J. B. Dennis. First version of a data flow procedure language. *Programming Symp., Lecture Notes Computer Science*, 19:362–376, 1974.

[44] Q. Ding and W. Perrizo. Decision tree classification of spatial data streams using peano count trees. In *Proc. 2002 ACM Symp. on Applied Computing*, pages 413–417, 2002.

[45] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 71–80, 2000.

[46] J. Dy and C. Brodley. Feature subset selection and order identification for unsupervised learning. In *Proc. Int. Conf. on Machine Learning*, pages 247–254, 2000.

[47] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery proto cols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[48] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions.* Wiley-Interscience, 2000.

[49] W. Fan, Y. Huang, and P. Yu. Decision tree evolution using limited number of labeled data items from drifting data streams. In *Proc. 2004 IEEE Int. Conf. on Data Mining*, pages 379–382, 2004.

[50] M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2):18–26, 2005.

[51] J. Gama, P. Medas, and P. Rodrigues. Learning decision trees from dynamic data streams. In *Proc. 2005 ACM Symp. on Applied Computing*, pages 573–577, 2005.

[52] V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining data streams under block evolution. *SIGKDD Explorations*, pages 1–10, 2002.

[53] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look a tutorial. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 635, 2002.

[54] E. Gatnar. A wrapper feature selection method for combined tree-based classifiers. *From Data and Information Analysis to Knowledge Engineering*, 3:119–125, 2006.

[55] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *ACM Symposium on Parallel Algo. and Architectures*, pages 281–291, 2001.

[56] L. Golab and M. Ozsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[57] L. Golab and M. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. pages 500–511, 2003.

[58] H. Gonzalez, J. Han, and D. Klabjan. Warehousing and analyzing massive rfid data sets. In *Proc. 22nd Int. Conf. on Data Engineering*, page 83, 2006.

[59] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182, 1996.

[60] X. Gu, S. Papadimitriou, S. Yu, and S. Chang. Online failure forecast for fault-tolerant data stream processing. pages 1388–1390, 2008.

[61] S. Guha, A. Meyerson, N. Mishra, and R. Motwani. Clustering data streams: Theory and practice. *IEEE Trans. Knowledge and Data Eng.*, 15(3):515–528, 2003.

[62] F. Gustafsson. *Adaptive filtering and change detection*. Wiley, 2000.

[63] M. Halatchev and L. Gruenwald. Estimating missing values in related sensor data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 83–94, 2005.

[64] Q. Hart and M. Gertz. Querying streaming geospatial image data: The geostreams project. In *Proc. 17th Int. Conf. on Scientific and Statistical Database Management*, pages 147–150, 2005.

[65] J. Hellerstein, W. Hong, and S. Madden. The sensor spectrum: Techonology, trends, and requirements. *ACM SIGMOD Record*, 32(4):22–27, 2003.

[66] H. Hotelling. Multivariate quality control - illustrated by the air testing of bombsights. *Technics of Statistical Analysis*, pages 111–184, 1947.

[67] H. Huang and F. Chen. A synthetic control chart for monitoring process dispersion with sample standard deviation. *Computers and Industrial Engineering*, 49(2):221–240, 2005.

[68] S. Huang and Y. Dong. An active learning system for mining time-changing data streams. *Intelligent Data Analysis*, 11(4):401–419, 2007.

[69] G. Hulten, L. Spencer, and P. Domingos. Mining time-chaning data streams. In *Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

[70] A. Hyvarinen. Survey on independent component analysis. *Neural Computing Surveys*, 2:94–128, 1999.

[71] P. Indyk, N. Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 363–372, 2000.

[72] C. Isert and K. Schwan. Acds: Adapting computational data streams for high performance. In *Proc. of Int. Symposium on Parallel and Distributed Processing*, page 641, 2000.

[73] J. Jackson. *A User's Guide to Principal Components.* John Wiley and Sons, 1991.

[74] N. Jiang and L. Gruenwald. Research issues in data stream association rule mining. *ACM SIGMOD Record*, 35(1):14–19, 2006.

[75] R. Jin and G. Aggrawal. Efficient decision tree constructions on streaming data. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 571–576, 2003.

[76] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. pages 341–352, 2003.

[77] R. Karp, C. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in sets and bags. *ACM Trans. Database Sys.*, pages 51–55, 2003.

[78] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 180–191, 2004.

[79] P. R. Kosiniski. A data flow programming language for operating systems. *Proc. ACM Sigplan-Sigops Interface Meeting*, 8(9):89–94, 1973.

[80] J. Kramer and B. Seeger. Pipes - a public infrastructure for processing and exploring streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 925–926, 2004.

[81] J. Kramer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Sys.*, 34(1), 2009.

[82] S. Kullback and R. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[83] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. of the VLDB Endowment*, 1(1), 2008.

[84] P. J. Landin. The next 700 programming languages. *Comm. of the ACM*, 9(3):157–166, 1966.

[85] P. Larranaga and J. Lozano. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation.* Kluwer Academic Publishers, 2002.

[86] M. Last. Classification of nonstationary data streams. *Intelligent Data Analysis*, 6(2):129–147, 2002.

[87] R. Lawrence. Early hash join: A configurable algorithm for the efficient and early production of join results. pages 841–852, 2005.

[88] A. Lerner and D. Shasha. The virtues and challenges of ad hoc + streams querying in finance. *Q. Bull. IEEE TC on Data Engineering*, 26(1):49–56, 2003.

[89] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proc. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 2–11, 2003.

[90] X. Lin and J. Xu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. pages 362–374, 2004.

[91] J. Lucas and M. Saccucci. Exponentially weighted moving average control schemes: Properties and enhancements. *Technometrics*, 32:1–29, 1990.

[92] A. Manjhi, S. Nath, and P. Gibbons. Tributaries and deltas: efficient and robust aggregation in sensor network streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 287–298, 2005.

[93] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 346–357, 2002.

[94] G. Manku, S. Rajagopalan, and B. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 251–262, 1999.

[95] A. Markov. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. *Dynamic Probabilistic Systems*, 1:552–577, 1971.

[96] A. Arasu et al. Stream: The stanford stream data manager. *Q. Bull. IEEE TC on Data Engineering*, 26(1):19–26, 2003.

[97] A. Arasu et al. Linear road: A stream data management benchmark. pages 480–491, 2004.

[98] A. Arasu et al. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[99] A. Bifet et al. New ensemble methods for evolving data streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 139–148, 2009.

[100] B. Babock et al. Models and issues in data stream systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 1–16, 2002.

[101] B. Babock et al. Maintaining variance and k-medians over data stream windows. In *Proc. Symposium on Principles of Database Systems*, pages 234–243, 2003.

[102] B. Park et al. Reservoir-based random sampling with replacement from data stream. In *Proc. SIAM International Conference on Data Mining*, pages 492–496, 2004.

[103] D. Abadi et. al. The design of the borealis stream processing engine. pages 277–289, 2005.

[104] D. Cai et al. Maids: Mining alarming incidents from data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 919–920, 2004.

[105] D. Carney et al. Monitoring streams - a new class of data management application. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 215–226, 2002.

[106] D. Gunopulos et al. Discovering all most specific sentences. *ACM Trans. Database Sys.*, 28(2):140–174, 2003.

[107] D. Terry et al. Continuous queries over append-only databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 321–330, 1992.

[108] E. Rundensteiner et al. Cape: Continuous query engine with heterogeneous-grained adaptivity. pages 1353–1356, 2004.

[109] F. Angiulli et al. On the complexity of inducing categorical and quantitative association rules. *Theoretical Computer Science*, 314:217–249, 2004.

[110] J. Daniel et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[111] J. Hwang et al. High-availability algorithms for distributed stream processing. pages 779–790, 2005.

[112] M. Chen et al. Path-based failure and evolution management. In *1st Symposium on Network Systems Design and Implementation*, pages 309–322, 2004.

[113] M. Gertz et al. A data and query model for streaming geospatial image data. *Lecture Notes in Computer Science*, 4254:687–699, 2006.

[114] M. Hammad et al. Nile: A query processing engine for data streams. page 851, 2004.

[115] M. Mazzucco et al. Merging multiple data streams on common keys over high performance networks. In *Proc. of ACM/IEEE Conf. on Supercomputing*, pages 1–12, 2002.

[116] M. Shah et al. Flux: An adaptive partitioning operator for continuous query systems. pages 25–36, 2003.

[117] N. Tatbul et al. Load shedding in a data stream manager. pages 309–320, 2003.

[118] N. Tatbul et al. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[119] R. Agrawal et al. An interval classifier for database mining applications. In *Proc. 18th Int. Conf. on Very Large Data Bases*, pages 560–573, 1992.

[120] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. pages 245–256, 2003.

[121] S. Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proc. 1st Biennial Conf. on Innovative Data Systems Research*, pages 269–280, 2003.

[122] S. Jeffery et al. A pipelined framework for on-line cleaning of sensor data streams. page 140, 2006.

[123] S. Subramaniam et al. Online outlier detection in sensor data using non-parametric models. In *Proc. 32nd Int. Conf. on Very Large Data Bases*, pages 187–198, 2006.

[124] T. Dasu et al. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *Proc. Symp. on the Interface of Statistics, Computing Science, and Applications*, pages 1–24, 2006.

[125] T. Dasu et al. Change (detection) you can believe in: Finding distributional shifts in data streams. *Lecture Notes in Computer Science*, 5772:21–34, 2009.

[126] T. Johnson et al. Streams, security and scalability. In *Proc. 19th Annual IFIP Conf. on Data and Applications Security*, pages 1–15, 2005.

[127] X. Song et al. Statistical change detection for multi-dimensional data. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 667–676, 2007.

[128] L. Meyer-Waarden. The effects of loyalty programs on customer lifetime duration and share of wallet. *Journal of Retailing*, 83(2):223–236, 2007.

[129] D. Montgomery. *Introduction to statistical quality control.* John Wiley and Sons, 1996.

[130] S. Muthukrishnan, E. van den Berg, and Y. Wu. Sequencial change detection on data streams. In *Proc. IEEE Int. Conf. on Data Mining Workshops*, pages 551–560, 2007.

[131] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 563–574, 2003.

[132] C. Ordonez. Clustering binary data streams with k-means. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 12–19, 2003.

[133] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[134] M. Pelikan. Hierarchical bayesian optimization algorithm: Toward a new generation of evolutionary algorithms. *Studies in Fuzziness and Soft Computing*, 170, 2005.

[135] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1226–1238, 2005.

[136] Perlman and Java. Predictive mining of time series data in astronomy. *Astronomical Data Analysis Software and Systems XII ASP Conf. Series*, 295:431–434, 2003.

[137] Q. Pham, N. Mouaddib, and G. Raschia. Data stream synopsis using saintetiq. *Lecture Notes in Computer Science*, pages 530–540, 2006.

[138] B. Plale. Using global snapshots to access data streams on the grid. In *Lecture Notes in Computer Science*, pages 191–201, 2004.

[139] P. Rosenbaum. An exact distribution-free test comparing two multivariate distributions based on adjacency. *Journal of the Royal Statistical Society: Series B*, 67(4):515–530, 2005.

[140] J. Ross and N. Cliff. A generalization of the interpoint distance model. *Psychometrika*, 29(2):167–176, 2006.

[141] C. Rueda and M. Gertz. Modeling satellite image streams for change analysis. In *Proc. of ACM Int. Symposium on Advances in Geographic Info. Systems*, 2007.

[142] S. Salvador and P. Chan. Fastdtw: Toward accurate dynamic time warping in linear time and space. In *Proc. KDD Workshop on Mining Temporal and Sequential Data*, pages 561–580, 2004.

[143] J. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *Proc. 22th Int. Conf. on Very Large Data Bases*, pages 544–555, 1996.

[144] W. Shewhart. *Statistical Method from the Viewpoint of Quality Control*. Dover Publications Inc., 1987.

[145] G. Shorack and J. Wellner. *Empirical processes with applications to statistics*. John Wiley & Sons Inc, 1986.

[146] B. Silverman. *Density Estimation*. Chapman & Hall, 1986.

[147] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[148] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[149] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. USENIX Annual Technical Conf.*, page 2, 1998.

[150] Y. Tao and T. Ozsu. Efficient decision tree construction for mining time-varying data streams. In *Proc. Conf. of the Centre for Advanced Studies on Collaborative research* (CASCON), pages 43–57, 2009.

[151] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. pages 333–344, 2003.

[152] N. Tracy, J. Young, and R. Mason. Multivariate control charts for individual observations. *Journal of Quality Technology*, 24(2):88–95, 1992.

[153] Y. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. pages 787–798, 2006.

[154] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[155] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proc. 18th Int. Conf. on Data Engineering*, pages 673–684, 2002.

[156] A. Wald. *Sequential Analysis.* Dover Publications, 2004.

[157] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 226–235, 2003.

[158] H. Wang, C. Zaniolo, and C. Luo. Atlas: A small but complete sql extension for data mining and data streams. pages 1113–1116, 2003.

[159] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 236–245, 2003.

[160] W. Wang, M. Sharaf, S. Guo, and M. Ozsu. Potential-driven load distribution for distributed data stream processing. In *Proc. of the 2nd Int. Workshop on Scalable Stream Processing System*, pages 13–22, 2008.

[161] K. S. Weng. Stream orientated computation in recursive data flow schemas. In *Project MAC Technical Memo 69*, 1975.

[162] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418, 2006.

[163] L. Yang and M. Sanver. Mining short association rules with one database scan. In *Proc. Int. Conf. on Information and Knowledge Engineering*, pages 392–398, 2004.

[164] M. Yang and J. Yang. Control chart pattern recognition using semi-supervised learning. In *Proc. 7th WSEAS Int. Conf. on Applied Computer Science*, pages 272–276, 2007.

[165] Y. Yao and J. Gehrke. Query processing for sensor networks. In *Proc. 1st Biennial Conf. on Innovative Data Systems Research*, pages 233–244, 2003.

[166] J. Yu, Z. Chong, H. Lu, and A. Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 204–215, 2004.

[167] M. Fazel Zarandi, A. Alaeddini, and I. Turksen. A hybrid fuzzy adaptive sampling - run rules for shewhart control charts. *Journal of Information Science: an Int. Journal*, 178(4):1152–1170, 2008.

[168] P. Zhang, X. Zhu, and Y. Shi. Categorizing and mining concept drifting data streams. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 812–820, 2008.

[169] S. Zhang and Z. Wu. Designs of control charts with supplementary runs rules. *Computers and Industrial Engineering*, 49(1):76–97, 2005.

[170] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 358–369, 2002.

[171] G. K. Zipf. *Human behavior and the principle of least-effort.* Addison-Wesley, 1949.