e-PGPathshala
Subject : Computer Science
Paper: Data Analytics
Module No 18: CS/DA/18 - Data Analytics - Mining Streams - I
Quadrant 1 – e-text

1.1 Introduction

Data streaming is the transfer of data at a steady high-speed rate. Data Stream Mining is the process of extracting knowledge structures from continuous, rapid data records. A data stream is an ordered sequence of instances that in many applications of data stream mining can be read only once or a small number of times using limited computing and storage capabilities. This chapter gives an overview of basics of data stream model, challenges of handling stream, sampling approaches etc.

1.2 Learning Outcomes

- Learn the basics of Data Stream and stream model
- Understand the challenges of handling stream
- Know sampling approaches for stream

1.3 Data stream

Streaming is commonly used when referring to shared media. Data streaming is the transfer of data at a steady high-speed rate sufficient to support such applications as high-definition television (HDTV) or the continuous backup copying to a storage medium of the data flow within a computer. Data streaming requires some combination of bandwidth sufficiency and, for real-time human perception of the data, the ability to make sure that enough data is being continuously received without any noticeable time lag.

Streaming Data is data that is generated continuously by thousands of data sources, which typically send in the data records simultaneously, and in small sizes (order of Kilobytes). Streaming data includes a wide variety of data such as log files generated by customers using your mobile or web applications, ecommerce purchases, in-game player activity, information from social networks, financial trading floors, or geospatial services, and telemetry from connected devices or instrumentation in data centers.

This data needs to be processed sequentially and incrementally on a record-by-record basis or over sliding time windows, and used for a wide variety of analytics including correlations, aggregations, filtering, and sampling. Information derived from such analysis gives companies visibility into many aspects of their business and customer activity such as – service usage (for metering/billing), server activity, website clicks, and geo-location of devices, people, and physical goods –and enables them to respond promptly to emerging situations. For example, businesses can track changes in public sentiment on their brands and products by continuously analyzing social media streams, and respond in a timely fashion as the necessity arises.

1.3 Challenges in Working with Streaming Data

Streaming data processing is beneficial in most scenarios where new, dynamic data is generated on a continual basis. It applies to most of the industry segments and big data use cases. Companies generally begin with simple applications such as collecting system logs and rudimentary processing like rolling min-max computations. Then, these applications evolve to more sophisticated near-real-time processing. Initially, applications may process data streams to produce simple reports, and perform simple actions in response, such as emitting alarms when key measures exceed certain thresholds. Eventually, those applications perform more sophisticated forms of data analysis, like applying machine learning algorithms, and extract deeper insights from the data.

1.4 Stochastic Gradient Descent (SGD)

Stochastic gradient descent (often shortened in SGD), also known as incremental gradient descent, is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. In other words, SGD tries to find minimums or maximums by iteration.

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent (e.g. lasagne's, caffe's, and keras' documentation). These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by.

- **In Machine Learning we call this: Online Learning**
  - Allows for modeling problems where we have a continuous stream of data
  - We want an algorithm to learn from it and slowly adapt to the changes in data
- **Idea: Do slow updates to the model**
  - **SGD** (SVM, Perceptron) makes small updates
  - **So:** First train the classifier on training data.
  - **Then:** For every example from the stream, we slightly update the model (using small learning rate)

1.5 Data Stream processing model

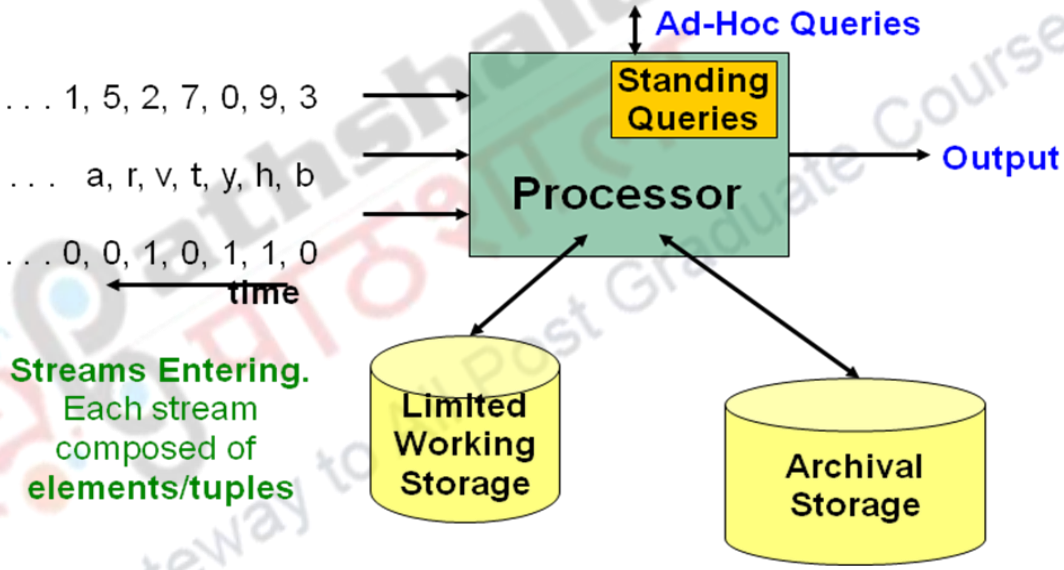

Figure 1. Stream processing model

In data stream processing model, we can view a stream processor as a kind of data-management system, the high-level organization of which is suggested in Fig. 1. Any number of streams can enter the system. Each stream can provide elements at its own schedule; they need not have the same data rates or data types, and the time between elements of one stream need not be uniform. The fact that the rate of arrival of stream elements is not under the control of the system distinguishes stream processing from the processing of data that goes on within a database-management system. The latter system controls the rate at which data is read from the disk, and therefore never has to worry about data getting lost as it attempts to execute queries.

Streams may be archived in a large archival store, but we assume it is not possible to answer queries from the archival store. It could be examined only under special circumstances using time-consuming retrieval processes. There is also a working store, into which summaries or parts of streams may be placed, and which can be used for answering queries. The working store might be disk, or it might be main memory, depending on how fast we need to process queries. But either way, it is of sufficiently limited capacity that it cannot store all the data from all the streams.

1.6 Problems on data streams

There are two ways that queries get asked about streams. One is **standing queries**, these queries, these are in a sense, permanently executing, and produce outputs at appropriate times.

Example 1: The stream produced by the ocean-surface-temperature sensor have a standing query to output an alert whenever the temperature exceeds 25 degrees centigrade. This query is easily answered, since it depends only on the most recent stream.

The other form of query is **ad-hoc**, a question asked once about the current state of a stream or streams. If we do not store all streams in their entirety, as normally we can not, then we cannot expect to answer arbitrary queries about streams. If we have some idea what kind of queries will be asked through the ad-hoc query interface, then we can prepare for them by storing appropriate parts or summaries of streams.

If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a sliding window of each stream in the working store. A sliding window can be the most recent n elements of a stream, for some n, or it can be all the elements that arrived within the last t time units, e.g., one day. If we regard each stream element as a tuple, we can treat the window as a relation and query it with any SQL query. Of course the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

Example 2: Web sites often like to report the number of unique users over the past month. If we think of each login as a stream element, we can maintain a window that is all logins in the most recent month. We must associate the arrival time with each

login, so we know when it no longer belongs to the window. If we think of the window as a relation Logins(name, time), then it is simple to get the number of unique users over the past month.
Note that we must be able to maintain the entire stream of logins for the past month in working storage. However, for even the largest sites, that data is not more than a few terabytes, and so surely can be stored on disk.

1.7 Data stream applications

Sensors in transportation vehicles, industrial equipment, and farm machinery send data to a streaming application. The application monitors performance, detects any potential defects in advance, and places a spare part order automatically preventing equipment down time.

A financial institution tracks changes in the stock market in real time, computes value-at-risk, and automatically rebalances portfolios based on stock price movements.

A real-estate website tracks a subset of data from consumers' mobile devices and makes real-time property recommendations of properties to visit based on their geo-location.

A solar power company has to maintain power throughput for its customers, or pay penalties. It implemented a streaming data application that monitors of all of panels in the field, and schedules service in real time, thereby minimizing the periods of low throughput from each panel and the associated penalty payouts.

A media publisher streams billions of clickstream records from its online properties, aggregates and enriches the data with demographic information about users, and optimizes content placement on its site, delivering relevancy and better experience to its audience.

An online gaming company collects streaming data about player-game interactions, and feeds the data into its gaming platform. It then analyzes the data in real-time, offers incentives and dynamic experiences to engage its players.

1.8 Sampling from a Data Stream

Perhaps the most basic synopsis of a data stream is a sample of elements from the stream. Since we can not store the entire stream, one obvious approach is a sample. A key benefit of such a sample is its flexibility: the sample can serve as input to a wide variety of analytical procedures and can be reduced further to provide many additional data synopses.

Data-stream sampling problems require the application of many ideas and techniques from traditional database sampling, but also need significant new innovations, especially to handle queries over infinite-length streams. Indeed, the unbounded nature of streaming data represents a major departure from the traditional setting.

Example:

A search engine receives a stream of queries, and it would like to study the behavior of typical users. We assume the stream consists of **tuples (user, query, time).** Suppose that we want to answer queries such as "**What fraction of queries by an average search engine user are duplicate?"** Assume also that we wish to store only 1/10th of the stream elements.

However, this scheme gives us the wrong answer to the query asking for the average number of duplicate queries for a user. Suppose a user has issued *s* search queries one time in the past month, d search queries twice, and no search queries more than twice. If we have a 1/10th sample, of queries, we shall see in the sample for that user an expected s/10 of the search queries issued once. Of the d search queries issued twice, only d/100 will appear twice in the sample; that fraction is d times the probability that both occurrences of the query will be in the 1/10th sample. Of the queries that appear twice in the full stream, 18d/100 will appear exactly once. To see why, note that 18/100 is the probability that one of the two occurrences will be in the 1/10th of the stream that is selected, while the other is in the 9/10th that is not selected. The correct answer to the query about the fraction of repeated searches is d/(s+d). However, the answer we shall obtain from the sample is d/(10s+19d).To derive the latter formula, note that d/100 appear twice, while s/10+18d/100 appear once.

Thus, the fraction appearing twice in the sample = $\frac{\frac{d}{100}}{\frac{d}{100}+\frac{s}{10}+\frac{18d}{100}}$

This ratio is $\frac{d}{(10s+19d)}$

For no positive values of s and d is $\frac{d}{(s+d)}$ **=** $\frac{d}{(10s+19d)}$

Solution: Pick 1/10th of users and take all their searches in the sample and use a hash function that hashes the user name or user id uniformly into 10 buckets

Generalized Solution: Consider the stream of tuples with keys, where Key is some subset of each tuple's components. For e.g., tuple is (user, search, time) and key is user. The choice of key depends on application.

To get a sample of *a/b* fraction of the stream, we can hash each tuple's key uniformly into *b* buckets and pick the tuple if its hash value is at most *a*

For example, consider a hash table with b buckets, pick the tuple if its hash value is at most a. Suppose we want to generate a 30% sample, then hash into b=10 buckets and take the tuple if it hashes to one of the first 3 buckets.

1.8.1 Sampling from a Data Stream: Sampling a fixed-size sample

Suppose we need to maintain a random sample *S* of size exactly *s* tuples, e.g., main memory size constraint. At times we may not know length of stream in advance. Suppose at time *n* we have seen *n* items, and each item is in the sample *S* with equal probability ***s/n.***

Say if s = 2 and the stream is a x c y z k c d e g…, then,

At n= 5, each of the first 5 tuples is included in the sample S with equal probability.

At n= 7, each of the first 7 tuples is included in the sample S with equal probability.

Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

Reservoir Sampling is a family of randomized algorithms for randomly choosing *k* samples from a list of *n* items, where *n* is either a very large or unknown number. Typically *n* is large enough that the list doesn't fit into main memory.

The steps in the algorithm (a.k.a Reservoir Sampling) is as given below:

1. Store all the first s elements of the stream to s

2. Suppose we have seen n-1 elements,and now the n^{th} element arrives (n>s)

   I. With probability **s/n**,keep the **n^{th}** element,else discard it

   II. If we picked the **n^{th}** element,then it replaces one of the **s** elements in the sample **s**,picked uniformly at random

This algorithm maintains a sample contains each element **s** so far with probability **s/n**, which may be proved by Induction as given below:

Proof by induction:

Assume that after *n* elements, the sample contains each element seen so far with probability *s/n.* We need to show that after seeing element *n+1* the sample maintains the property. Sample contains each element seen so far with probability s/(*n+1)*

Base case: After we see n=s elements the sample S has the desired property. Each out of n=s elements is in the sample with probability *s/s = 1.*

Inductive hypothesis: After n elements, the sample s contains each element seen so far with probability s/n. Now if element n+1 arrives:

Inductive step: For elements already in s, probability that the algorithm keeps it in s is

$$\underbrace{\left(1 - \frac{s}{n+1}\right)}_{\text{Element n+1 discarded}} + \underbrace{\left(\frac{s}{n+1}\right)}_{\text{Element n+1 not discarded}}\underbrace{\left(\frac{s-1}{s}\right)}_{\text{Element in the sample not picked}} = \frac{n}{n+1}$$

So at time n, tuple in s were there with probability s/n. at time n $\rightarrow$ n+1,tuple in s is with probability (n/n+1). So the probability tuple is in s the time n+1 is

$$= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$$

	**Case Studies**
A) http://insights.axtria.com/case-study-real-time-stream-data-processing-and-validation	
Real-Time User Data Stream Processing and Validation for US based Auto Assistance Company Improved and faster data management process of real time data Data quality for downstream processes improved Advanced analytics done on new buckets of	

information for improving service offerings
B http://www.peppersandrogersgroup.com/view.aspx?docid=35659
Streaming Data Opens the Door to New Customer Insights Spotify and other leading digital companies are mining data streams for a deeper understanding of customer behavior.

Summary

- Sampling a fixed proportion of a stream where the sample size grows as the stream grows
- Sampling a fixed-size sample using Reservoir sampling