Abstract
In the fields of big data, AI, and streaming processing, we work with large amounts of data from multiple sources. Due to memory and network limitations, we process data streams on distributed systems to alleviate computational and network loads. When data streams with non-uniform distributions are processed, we often observe overloaded partitions due to the use of simple hash partitioning. To tackle this imbalance, we can use dynamic partitioning algorithms that require a sampling algorithm to precisely estimate the underlying distribution of the data stream. There is no standardized way to test these algorithms. We offer an extensible ranking framework with benchmark and hyperparameter optimization capabilities and supply our framework with a data generator that can handle concept drifts. Our work includes a generator for dynamic micro-bursts that we can apply to any data stream. We provide algorithms that react to concept drifts and compare those against the state-of-the-art algorithms using our framework.

1 Introduction
The number of computers grows at a never before seen rate, mainly due to the inclusion of microchips in everyday household items, embedded devices, the availability of personal computers, and mobile devices. Thus, our world is becoming more and more interconnected, creating vast networks of devices producing data at a large scale.

We have to process this constantly generated data in a way that satisfies consumer needs, such as zero downtime and low latency. Algorithms over the incoming data stream cannot be computed locally on one central server due to the bitrate of the stream [18]; furthermore, requests could come in from all over the world. Therefore, using some distributed system is necessary to provide the power to meet these demands [17], however partitioning our computation raises interesting challenges.

Partitioning often occurs with simple hash functions [15, 23], which does not take the weights of the keys into account. This only works well when the distribution of the incoming data stream follows a near-uniform distribution [23].

However, requests over the internet do not follow a uniform distribution; instead, they follow Zip’s law, and can, therefore, be fitted using a Zipfian distribution [1]. This means that the majority of internet traffic may attribute to a small portion of the userbase.

Changing or non-uniform data distributions can cause overloaded partitions, and manually adjusting partitioners is infeasible due to the amount and frequency of incoming data [32]. Unexpected and high loads in a distributed system can cause downtime, which entails significant losses in revenue. Examples include Amazon’s and Lowe’s downtime during Black Friday [29, 30].

The base problem of partitioning is the NP-complete Bin Packing problem’s special case, the Multiprocessor Scheduling Problem [10]. Although it has a trivial polynomial-time solution, this would require tasks with the same lengths [10].

For non-trivial cases, the problem requires the knowledge of every key’s weight, which, in a distributed system, is a hard, if not nearly impossible task.

However, the problem can be alleviated by using the knowledge that the data follows Zip’s law, therefore there exists a cutoff where the incoming requests are so sparse that it will not influence any partitioning algorithm significantly. It becomes possible to count only a subset of the elements with the highest frequencies, called frequent elements problem. Some of the earlier works researching the frequent elements problem were done by Alon et. al [3]; Henzinger et. al. [14]; and Charikar et. al. [5].

Algorithms that can precisely and cheaply estimate the frequencies of such elements are essential. Stalling the data stream with heavy computations is infeasible, so solutions that can process each element in the stream in $O(1)$ time, query the most frequent items in sublinear time, and use sublinear space are well sought after [19, 20, 8, 21, 5, 7, 12].

A major problem in data streaming is changing distribution, also called Concept drift, which requires algo-
gorithms that can detect the changes without explicit knowledge about them \[28\]. Failing to do so causes their accuracy to degrade \[32\]. However overcorrection triggered by mild noise is also a problem, so balancing robustness and flexibility is key \[25\].

Data bursting happens when data packets arrive at their destination more rapidly than intended by the transmitter \[2\]. The phenomenon of micro-bursting is quite common in networks with window-based transport protocols, especially TCP \[2\].

Distributed systems (and most parts of the internet) are built on TCP, but even on non-window based protocols with physically far enough components, where packets are going to be transmitted over several jumps, data burst can occur. Algorithms that are not designed to handle periodical micro-bursting could show significant inaccuracies in their results.

In this paper, our main contributions are: a data generator that can handle concept drifts and data bursts; a mechanism to benchmark and rank sampling algorithms, and a baseline Oracle, which correctly estimates the ground truth based on our data generator; formalization of concepts and concept drifts and proof of the correctness of the Oracle; a hyperparameter optimizer for sampling algorithms; two novel sampling algorithms that can react to concept drifts and analysis of them in conjunction with state-of-the-art sampling algorithms using our framework.

2 Related work

2.1 Sampling algorithms

There are multiple algorithms that try to solve the problem of frequent items with fundamentally different approaches. The common ground for all of these algorithms is that they have to carefully balance the memory usage, run time, and precision simultaneously.

These algorithms can be categorized into the following three categories: Counter-based, Sketch-based and Change respecting.

Counter-based algorithms rely on a counter to update the currently estimated frequencies of each key. Strategies exist to periodically flush or thin the counters to save memory. Three defining algorithms of this category are: Sticky Sampling \[19\], Lossy Counting \[19\] and Space Saving \[21\] \[8\] \[20\].

Sketch-based algorithms work by keeping fewer counters than their counter-based counterparts. Often using a probabilistic approach to keep estimations of the frequency in sub-linear space. However they require expensive calculations when recording a key, so they cannot fit into the tight budget of a data streaming application. Notable algorithms are: Count Sketch \[5\] and Count-Min Sketch \[7\].

Change respecting algorithms have a mechanism in place to detect and conform to the concept drifts unlike static algorithms, which concentrate on either static data or a data stream with a non-changing distribution. We are especially interested in this category, the two main algorithms here are Landmark \[31\] and Frequent \[12\].

2.2 Partitioning algorithms

Hash partitioning works by taking the modulus of the hash of the data key with the number of partitions \[15\]. This is a static partitioning algorithm that only solves the Multi-processor Scheduling problem with uniformly distributed keys.

In distributed systems, the number of partitions can change and migration costs cannot be ignored, as parts of the system could be physically apart. Consistent hash is therefore commonly used \[16\] \[11\]. Its major problem is that it has difficulties with non-uniform distributions \[11\].

Gedik proposed three algorithms that consider non-uniform distributions as well \[11\]. Therefore, we decided to use Gedik’s algorithms in our tests. Furthermore, these algorithms are constructed in such a way that both balance and migration costs can be configured easily.

2.3 Concept Drifts

Widmer defines concept drift as the radical change in a target concept introduced by changes in a hidden context \[28\]. Studying concept drift in data science is a fundamental building block to a good algorithm. Internet traffic is ever-changing due to minor or major real-world events and drifts in the data will be inevitable.

A lot of research went into studying concept drifts, but the emphasis was placed on machine learning, with a categorizer on a finite number of categories.

The term concept drift does not have a clear definition in the field of data streaming or it does not apply to the sampling problem. It is often used interchangeably with the following terms: concept shift, changes of classification, fracture points, fractures between data, since these all refer to the same basic idea \[22\]. Therefore, when we mention concept drift or drifting data stream we refer to the change of the underlying distribution of the stream. For example, the cause of this change in the distribution could be a sporting event, a sale at an online store or sudden hardware failure at a major data centre.

Concept drift can be sudden, also known as abrupt or instantaneous, where a change takes place suddenly; and gradual, where a transition period exists between two concepts \[25\] \[26\]. Sudden concept drift may occur because of critical failure at a major server park, while a gradual drift can occur every evening as different parts of the world start their day, while others go to sleep. Concept drift may also occur during any of our computations, which renders both static partitioning algorithms and static algorithms ineffective.
Data Stream

3.1 Data streams

3 Definitions

3.1 Data streams

We define data streams with a finite cartesian product over the set of keys \( K \). A key could be anything, we define \( K \) over the natural numbers (\( K \subset \mathbb{N} \)).

Definition 3.1 (Data Stream). Given a key-set \( K \) and \( n \in \mathbb{N}^+ \), let \( str \in K^n \) be a data stream.

Although it is not the nature of the stream to have a random accessor to their items, for further definitions let \( str_i \) be the \( i \)-th element of the stream.

3.2 Concepts

Concepts are probability distributions with a start and an end index, between which the concept is considered active.

Definition 3.2 (Concept). Let \( Con \subset \mathbb{N}^+ \times \mathbb{N}^+ \times (\mathbb{N}^+ \to (K \to [0,1])) \) be the set of all concepts, let \( c = (c_s, c_e, c_f) \in Con \), if \( c_s \leq c_e \) and \( \forall i \in D(c_f) : c_f(i) \) is a discrete probability distribution over \( K \), we call this \( c \) a concept.

Given a stream \( str \in K^n \) (\( n \in \mathbb{N}^+ \)), let \( Con^{str} \subset Con \) be the set of all concepts for stream \( str \). We require exactly one concept to be active at every point of the stream.

\[ \forall i \in [1,2,\ldots,n], \exists (c_s, c_e, c_f) \in Con^{str} : c_s \leq i \leq c_e \]

And require the concept’s probability function to cover the range of the concept.

\[ \forall (c_s, c_e, c_f) \in Con^{str} : D(c_f) = [c_s, c_s + 1, \ldots, c_e] \]

Given a stream \( str \in K^n \) (\( n \in \mathbb{N}^+ \)), \( c = (c_s, c_e, c_f) \in Con^{str} \) and \( \Delta \in ([0,1] \times (K \to [0,1]) \times (K \to [0,1])) \to (K \to [0,1]) \) we study the following two types of \( c_f \):

- Constant concept: \( \exists P^c \) discrete probability distribution over \( K \), \( \forall i \in D(c_f) : c_f(i) = P^c \)
- Changing concept: if \( c_e > c_s \) and \( \exists P^1, P^2 \) discrete probability distributions over \( K \), \( \forall i \in D(c_f) : c_f(i) = \Delta(\frac{i-c_s}{c_e-c_s}, P^1, P^2) \)

Function \( \Delta \) can be defined in multiple ways, we will provide further information on the specifics of how we use it in Section 3.1.

Definition 3.3 (Concept Drift). Given a stream \( str \in K^n \) (\( n \in \mathbb{N}^+ \)), let \( Con^{str} \subset Con \) be the set of all concepts for stream \( str \). concept drift occurs in \( i \in [1,2,\ldots,n-1] \) if,

\[ \exists c = (c_s, c_e, c_f), c' = (c'_s, c'_e, c'_f) : c_f(i) \neq c'_f(i+1) \]

Definition 3.4 (Abrupt Concept Drift). Given two consecutive concepts, \( c = (c_s, c_e, c_f), c' = (c'_s, c'_e, c'_f), c_e+1 = c'_s \), abrupt concept drifts occur at \( c'_e \) if \( c_f(e_k) \neq c'_f(e'_k) \).

Gradual concept drift occurs between the start and end of a changing concept.

Definition 3.5 (Gradual Concept Drift). Given a stream \( str \in K^n \) (\( n \in \mathbb{N}^+ \)) and a changing concept \( c = (c_s, c_e, c_f) \in Con^{str} \) gradual concept drift occurs during \([c_s, c_s+1, \ldots, c_e] \)

Given all the concepts that are acting on the stream, the true distribution can be defined at any location.

Definition 3.6 (Concept and True Distribution). Given a stream \( str \in K^n \) (\( n \in \mathbb{N}^+ \)), its concepts \( Con^{str} \subset Con \). At any point \( i \in [1,2,\ldots,n] \) the true distribution of the stream \( P^{str}_i \) is determined by the underlying concept, \( \exists c = (c_s, c_e, c_f) \in Con^{str} : c_s \leq i \leq c_e \)

\[ P^{str}_i = c_f(i) \]

3.3 Sampling algorithm

Our problem is similar to the frequent items problem [6].

Beyond the most frequent items above a threshold, the frequency of those items is also necessary.
Definition 3.7 (Sampling problem). Given a key-set $K$, $n \in \mathbb{N}^+$ and a data stream $str \in K^n$, let the resulting most frequent items with their frequency be $F \in K \rightarrow \mathbb{N}^+$ with threshold $\phi \in [0 \ldots 1]$ defined as
\[
\forall k \in K, (k, f_k) \in F \iff f_k \geq \phi \cdot n
\]
\[f_k = \sum_{i \in \{1,2,\ldots,n\}} 1_k(str_i)\]

Instead of $\phi$, a $k \in \mathbb{N}^+$ can be given, resulting in at most $k$ number of items with the highest frequencies. This $k$ parameter is often called top-$K$ in the literature [20].

Algorithms that solve the sampling problem have to work with heavy limitations. Memory is limited, the run time has to be as low as possible, the stream can only be processed once, it’s elements one-by-one and the length of the stream is hidden or non-existent.

The result of these algorithms are normalized to produce relative frequencies, we will call these sample distributions.

3.4 Oracle

Definition 3.8 (Oracle). An oracle is a sampling algorithm that has access to the data stream’s metadata, which contains the concepts and the $\Delta$ function that was used to generate that data stream. It can trivially calculate the true distribution ($P^t_{str}$) using the method described in definition 3.6.

Using the oracle as a sampling algorithm in a real distributed system is of course impossible.

3.5 Error calculation

3.5.1 Distribution difference

Directly measuring the error of an algorithm in isolation is key to determine how fast and how accurately it reacts to drifts compared to other algorithms. We need the baseline distribution, provided by the oracle sampling, to which we can compare all of the other distributions.

There are multiple ways, to compare probability distributions, but usually, either Kullback–Leibler divergence or Hellinger distance is used [9,27].

Given $P$ target discrete distribution and $Q$ approximate discrete distribution the Hellinger distance is the following formula [13]:
\[
H(P||Q) = \frac{1}{\sqrt{2}} \cdot \sqrt{\sum_x (\sqrt{P(x)} - \sqrt{Q(x)})^2}
\]

Hellinger distance is bounded between 0 and 1. It is a distance function (metric), and unlike Kullback–Leibler divergence, it does satisfy the rule of symmetry and triangle inequality.

3.5.2 Load imbalance

A partitioning problem can be traced back to the Multi-processor scheduling problem originally defined by Garey et. al. in 1979 [10]. It is NP-complete, but pseudo-polynomial solutions exist for any fixed $m$. [10]

When measuring the error of a whole system we use the most commonly used metric, the percent imbalance metric. It measures the lost performance to the imbalanced load, or in other words the performance that could be gained by balancing our partitions [24]. Pearce et. al. (2012) [24] defined this imbalance as follows:

Definition 3.9 (Imbalance). Let $L$ be the loads on our partitions, then $L_{max}$ is the maximum load on any process. Let $L$ be the mean load over all processes. The percent imbalance metric for $L$, $\lambda_L$, is:
\[
\lambda_L = \left( \frac{L_{max}}{L} - 1 \right) \cdot 100
\]

Calculating the $L_{max}$ of the loads is sufficient because that will determine the runtime of our whole computation.

4 Our Framework

To test sampling algorithms we need a flexible, fast and deterministic testbed with low overhead.

We list our findings of what the requirements should be for a ranking and benchmarking framework for algorithms over data streams. We base these requirements on previous frameworks [4] and our own experience while developing sampling algorithms.

- Allow algorithms to only process the data stream at most once [4].
- Provide metrics about the algorithms.
- Provide those metrics on-demand during the computation [4].
- Be Fast with low overhead [4].
- Be Deterministic.
- Include a data generator that can
  - handle concept drifts;
  - allow pre-generation of data streams;
  - generate the metadata of pre-generated streams;
  - be capable of simulating micro-bursting during tests.
- Offer Hyper-parameter optimization.

4.1 Data generator

4.1.1 Concept drift generation

It makes more sense to define concepts and then concept drifts as their consequence. However, for generating concept drifts it is more straightforward to define the points at which concept drifts are occurring and derive the concepts from those.
Let \( D \subseteq \mathbb{N}_0^+ \times \mathbb{Q}^+ \times (K \to [0, 1]) \times (K \to [0, 1]) \), 
\((\text{len}, \text{mid}, P_1, P_2) \in D\) is a drift if \((\text{mid} + \frac{1}{2}) \in \mathbb{N}^+ \land \text{len} > 2\), where \(\text{len}\) is the length, \(\text{mid}\) is the midpoint of the drift and \(P_1, P_2\) are the two probability distributions of the two concepts that the drift is created by.

In further definitions \( \forall d = (\text{len}, \text{mid}, P_1, P_2) \in D \) let 
\( s \in D \to \mathbb{N}^+ : s(d) = \text{mid} - \frac{\text{len}}{2} \) and \( e \in D \to \mathbb{N}^+ : e(d) = \text{mid} + \frac{\text{len}}{2} \).

We have exactly one concept active at any given point, so overlapping drifts should not be allowed either. Given \( \text{str} \in K^n(n \in \mathbb{N}^+) \) and \( \text{Dstr} \subseteq D \), the drifts for that data stream: \( \forall d, d' \in \text{Dstr}, d \neq d' \):
\[
eq e(d') \implies e(d) < s(d')
\]

There also has to be an initial drift starting at index 1,
\[
\exists d \in \text{Dstr} : s(d) = 1
\]

For a \( \text{str} \in K^n(n \in \mathbb{N}^+) \) stream we call \( d = (\text{len}, \text{mid}, P_1, P_2) \in D \) a gradual drift, if \(\text{len} > 0\), \(s(d) > 1\) and \(e(d) < n\).

For a \( \text{str} \in K^n(n \in \mathbb{N}^+) \) stream we call \( d = (0, \text{mid}, P_1, P_2) \in D \) an abrupt drift, if \(s(d) > 1\) and \(e(d) < n\).

Given a stream \( \text{str} \in K^n(n \in \mathbb{N}^+) \) generated by \( \text{Dstr} \subseteq D \) drifts, \( \forall i \in [1, 2, \ldots, n] \), let \( \text{str}_i \) be a random variable following \( P^{\text{str}}_G \) discrete distribution, where \( P^{\text{str}}_G \) is

1. \( \exists d = (d_{\text{len}}, d_{\text{mid}}, d_{P_1}, d_{P_2}) \in \text{Dstr} : \text{len} > 0 \land s(d) \leq i \leq e(d) \) then,
\[
P^{\text{str}}_G = \Delta \left( \frac{i - s(d)}{d_{\text{len}}}, d_{P_1}, d_{P_2} \right)
\]

2. \( \exists d = (0, d_{\text{mid}}, d_{P_1}, d_{P_2}) \in \text{Dstr} : s(d) = i \) then,
\[
P^{\text{str}}_G = d_{P_2}
\]

3. \( \exists d = (l, m, d_{P_1}, d_{P_2}) \in \text{Dstr} : e(d) < i \land 2d' < \text{Dstr} : e(d) < s(d') \leq i \)
\[
P^{\text{str}}_G = d_{P_2}
\]

To make sure that these drift definitions are compliant with the concepts defined in Section 3.2, we prove that their expressive powers are equal [Theorem 1] [Theorem 2]. Therefore the same concept drifts can be described by them.

**Theorem 1** (Drifts and concepts are equal in expressive power). \( \forall \text{str} \in K^n \) and \( \text{Constr} \subseteq \text{Con} \): \( \exists \text{Dstr} \subseteq D \) drifts, where \( |\text{Dstr}| = |\text{Constr}| \) and \( \forall i \in [1, 2, \ldots, n] : P^{\text{str}}_G = P^{\text{str}}_T \)

**Proof.** We construct such \( P^{\text{str}}_G, \forall c = (c_e, c_e, c_f) \in \text{Constr} \) and show that the construction is correct. Then for any \( i \in [1, 2, \ldots, n] \), \( \exists c = (c_e, c_e, c_f) \in \text{Constr} : c_e \leq i \leq c_e \) (based on requirements for \( \text{Constr} \)). We show that for any \( c \) concept, our constructed corresponding \( d \) drift is correct by showing that \( P^{\text{str}}_G = P^{\text{str}}_T \). Please see [Appendix A] for the whole proof.

**Theorem 2** (Drifts and concepts are equal in expressive power II.). \( \forall \text{str} \in K^n \) and \( \text{Dstr} \subseteq D \): \( \exists \text{Constr} \subseteq \text{Con} \) concepts, where \( \forall i \in [1, 2, \ldots, n] : P^{\text{str}}_G = P^{\text{str}}_T \)

**Proof.** We begin by constructing such \( \text{Constr} \), \( \forall d = (d_{\text{len}}, d_{\text{mid}}, d_{P_1}, d_{P_2}) \in \text{Dstr} \), then for any \( i \in [1, 2, \ldots, n] \), we show that for any \( d \) drift, our constructed corresponding \( c \) concept is correct by showing that \( P^{\text{str}}_G = P^{\text{str}}_T \). Please see [Appendix B] for the whole proof.

Both the generator and the concepts depend on a \( \Delta \) function. To allow the most generic way of defining drifts we left \( \Delta \) undefined up until this point. For performance reasons, we only approximate a linear interpolation with the following \( \Delta \) function, and in [Theorem 3] we prove the correctness of this approximation.

Let \( \Delta \in ([0, 1] \times (K \rightarrow [0, 1]) \times (K \rightarrow [0, 1])) \rightarrow (K \rightarrow [0, 1]) \) := \( \forall p \in [0, 1] \)
\[
\Delta(p, P_1, P_2) = \begin{cases} P_1, & \text{if } R > p \\ P_2, & \text{otherwise} \end{cases}
\]

where \( R \) is a random variable following standard uniform distribution.

We use this \( R \) to avoid recalculating a new distribution for every generated item.

To see smooth transitions in the generator’s implementation we also require that the probability distribution at the final point of a drift matches the probability distribution at the first point of the following drift. To achieve this we require the drifts to comply with the following restriction:

Let \( \text{str} \in K^n(n \in \mathbb{N}^+) \) be a data stream and \( \text{Dstr} \) its concepts.
\( \forall d = (d_{\text{len}}, d_{\text{mid}}, d_{P_1}, d_{P_2}) \in \text{Dstr} : \)
\[
\exists d' = (d'_{\text{len}}, d'_{\text{mid}}, d'_{P_1}, d'_{P_2}) \in \text{Dstr},
\]
\[
d'_{\text{mid}} > d_{\text{mid}} \implies d_{P_2} = d'_{P_1}
\]

Using the aforementioned \( \Delta \) it can be shown that this is sufficient precondition.
4.1.2 Dynamic burst generation

When simulating bursts we simulate faulty routers. Before beginning every micro-batch, a burst has a chance to start. During the duration of the burst, the keys have a certain probability that they will be held back. At the end of the burst, which could last a couple of micro-batches, the keys are released back into the stream at once.

Bursts can only be applied to already existing streams, for this, it is a prerequisite for a burst process to already have a data stream prepared. We will use the phrase loading to describe the process of getting the next item of this stream.

Bursts are defined with the following structure: \( B \subset [0, 1] \times [0, 1] \times \mathbb{N}^+ \times \mathbb{N}^+ \), \( b = (b_{sp}, k_{bip}, b_{min}, b_{max}) \in B \) is a valid burst configuration, if \( b_{min} \leq b_{max} \).

Burst start probability \((bsp)\) describes the probability of a burst starting at any micro-batch. Once the burst started, another one cannot start again. Before any bursting could take place, the faulty keys are calculated in advance, with Key burstiness probability \((kbp)\) that gives the probability of each key being held up. This process generates a map (Faulty Keys Map - FKM) in which we can store these keys until the burst is over.

During the burst, the faulty keys are counted in FKM. Whenever a key that is in FKM would be loaded into the stream as a next item, instead will be put into the FKM. If a key is not faulty, then the stream loading can continue as usual.

The burst duration is defined by Burst length in micro-batches, which is a random number between the minimum \((b_{min})\) and maximum \((b_{max})\) duration. At the end of the burst, the held-up data in FKM is released back into the stream.

4.2 Ranking

We use on-demand metric querying to archive the ranking of sampling algorithms. We show two different ways to compare them. One method works with the direct sampling outputs, while the other method indirectly measures them in a simulated environment by the efficiency of that simulated system.

4.2.1 Ranking by reported distribution difference

Sampling algorithms can be measured directly by the samples described in Section 3.5.1. To achieve this, we run the algorithms in separate processes and provide them the same data stream. At the end of each micro-batch, the samples of the algorithms are saved and the sample distributions are calculated from them. We run an Oracle on the same data stream to provide the baseline, the true distribution, at the end of each micro-batch.

The Oracle is a good choice for this, because it can correctly calculate the true distributions even during concept drifts [Theorem 3]. Seeing the difference increase between a sample distribution and the true distribution during concept drifts means the sampling algorithm could not react to the drift fast enough. After this, the speed at which the difference shrinks tells us how fast it can detect and correct the drift. The Oracle’s correctness is based on the synthetic data generator. We showed that the data generator can correctly generate concept drifts in [Theorem 1] and [Theorem 2]. Therefore, it is enough to show that the oracle correctly calculates the true distribution based on the data generator. We have access to all of the drifts \( D^{str} \subset D \) from the metadata, which was used to generate \( str \in K^n \) data stream.

Because there is always at least one drift present (the starting drift) and no two drifts can overlap at any given point, for all \( i \in [1, n] \), \( P^{i, str}_O \) is:

1. \( \exists d = (d_{len}, d_{mid}, d_{p_1}, d_{p_2}) \in D^{str} : d_{len} > 0 \land s(d) \leq i \leq e(d) \) then, 
   \( P^{i, str}_O := (1 - \frac{(i - s(d))}{d_{len}}) \cdot d_{p_1} + \frac{(i - s(d))}{d_{len}} \cdot d_{p_2} \)
2. \( \exists d = (0, d_{mid}, d_{p_1}, d_{p_2}) \in D^{str} : s(d) = i \) then, 
   \( P^{i, str}_O := d_{p_2} \)
3. \( \exists d = (l, d_{mid}, d_{p_1}, d_{p_2}) \in D^{str} : e(d) < i \land \exists d' \in D^{str} : e(d') \leq i \) 
   \( P^{i, str}_O := d_{p_2} \)

Theorem 3 (Oracle is correct). Given \( str \in K^n (n \in \mathbb{N}^+) \) stream generated by \( D^{str} \subset D \) drifts.

\[ \forall i \in [1, 2, \ldots, n] : P^{i, str}_O \text{ correctly estimates } P^{i, str}_G \]

Theorem 3 (Oracle is correct). Given \( str \in K^n (n \in \mathbb{N}^+) \) stream generated by \( D^{str} \subset D \) drifts.

\[ \forall i \in [1, 2, \ldots, n] : P^{i, str}_O \text{ correctly estimates } P^{i, str}_G \]

Proof. There are two cases for each \( i \in [1, 2, \ldots, n] \):

1. \( \exists d = (d_{len}, d_{mid}, d_{p_1}, d_{p_2}) \in D^{str} : d_{len} > 0 \land s(d) \leq i \leq e(d) \) 
   let \( p = \frac{(i - s(d))}{d_{len}} \), the generator defines \( P^{i, str}_G \) to be 
   \( P^{i, str}_G := \Delta \left( \frac{(i - s(d))}{d_{len}}, d_{p_1}, d_{p_2} \right) \)
     we provided the generator with the following \( \Delta \) function,
   \[ \Delta(p, d_{p_1}, d_{p_2}) = \begin{cases} 
   d_{p_1}, & \text{if } R > p \\
   d_{p_2}, & \text{otherwise}
   \end{cases} \]
     where \( R \) is a random variable following standard uniform distribution.
   \[ P(P^{i, str}_G = d_{p_1}) = P(R > p) = (1 - p) \]
   \[ d_{p_2}) = P(R \leq p) = p \]
     therefore, the expected value of \( P^{i, str}_G \),
   \[ E[P^{i, str}_G] = (1 - p) \cdot d_{p_1} + p \cdot d_{p_2} = P^{i, str}_O \]
2. otherwise, \( P_{i,str}^O \equiv P_{i,str}^G \) by definition.

4.2.2 Ranking by load imbalance

In real distributed systems incoming data is processed by multiple nodes. The data arrives at the first stage from a data source, such as Kafka, Couchbase, through a hash partitioner denoted by the blue square. The nodes in the first stage then shuffle this data by a grouping, which is a frequently used operation and could be the result of group-by or join operations. The second stage runs calculations on the shuffled data stream.

In our framework, we simulate a distributed system with multiple nodes by starting multiple sampling algorithms. We process the data stream in micro-batches and for each element of the micro-batch, we select a sampling algorithm with a hash partitioner to feed that element to. After each micro-batch, we gather the outputs of the sampling processes and aggregate them. Based on the aggregated output, we determine whether repartitioning is necessary with a decider strategy. If so, we create a new partitioning algorithm for the imbalance measurement, which will be used by the next micro-batch.

The calculation of the load imbalance happens during each micro-batch. A naive approach would be to calculate the partition loads from the reported data of the sampling processes, however, this would allow cheating, the sampling algorithm could report falsified data and produce perfect results. To make sure this does not happen, we calculate the partition loads during the micro-batch processing. We determine this load by counting the actual number of elements that would be shuffled to certain partitions if we were to continue the computation. If repartitioning happens after processing a micro-batch, the load imbalance caused by the new partitioning algorithm will be calculated during the following micro-batch.

To rank multiple sampling algorithms, we only compare test cases that have the same data stream, number of nodes, decider strategy, repartitioner, and micro-batch size.

4.3 Optimizer

We provide a framework for optimization of sampling algorithms in which a new optimization strategy is easy to implement.

The optimization process works the following way:

1. The initial population is selected as a starting point.
2. The initial population is benchmarked
3. In the selection process, the selectors are used to thin out the population based on their fitness.
4. In the evolution phase, a new population is generated based on the previous generation’s survivors.
5. The survivors are added to the new population

The optimization strategy we provide is a local minimum search, in which we choose the member of the population with the best fitness value in the selection process.

The hyperspace of an algorithm could be high dimensional. Every additional parameter increases the number of neighbours exponentially. Therefore, calculating all possible neighbours of an algorithm is infeasible and instead, we generate a subset of them and only include those in the population. Because of the complexity of the problem domain, the correct steps and ranges of the parameters are not universal. For example, a “probability” parameter only makes sense, if it is between 0 and 1. Furthermore, it often does not make sense to use the whole domain of the parameter as the range. For example, a “window size” parameter should not be bigger than the stream length and should have a sensible minimum size as well.

To evaluate these algorithms with specific hyperparameters, fitness functions are used by the optimizer. They can be defined by the use of the metrics, such as accuracy, run time and memory usage.

If the diversity of the data stream is low, overfitting could occur during the optimization process. To avoid this multiple data streams are used during optimization, and also allow different kinds of bursts to be introduced to the data streams.

5 Our Algorithms

5.1 Temporal Smoothed

Our first algorithm is called Temporal Smoothed. This algorithm is inspired by the Landmark, and it aims to solve a flaw in it. In the Landmark, when the sampler is asked to report shortly after a landmark, the reported output is too small in size and may not show an accurate distribution in its sampled keys.

Our algorithm, like the Landmark, works with windows (threshold), but rather than resetting the whole inner state of the sampling after each window, it starts a new sampling process instead. It maintains the original sampler (main sampler) and the newly created one (secondary sampler) for a predetermined window size (switch thresh-
old). During this window, the incoming data is sampled by both of them and only the main sampler’s results are being reported. After the window is processed, the secondary sampler gains a stable size and becomes the new main sampler.

**Algorithm 1: TemporalSmoothed**

| Input : key: Key |
|------------------|
| Parameter : ms: SamplerBase, main sampler |
| Parameter : ss: SamplerBase, secondary sampler |
| Parameter : t: N^+, threshold |
| Parameter : st: N^+, switch threshold |
| totalProcessedElements ← totalProcessedElements + 1; |
| if ss initialized then |
| ss.recordKey(key); |
| end |
| if ss not initialized and ms.totalProcessedElements = t + st then |
| initialize ss; |
| else if ss initialized and ss.totalProcessedElements = st then |
| ms ← ss; |
| discard ss; |
|

The **TemporalSmoothed** algorithm can encapsulate any sampling algorithm. As a result, the memory usage and run time heavily depend on which algorithm is encapsulated.

**Theorem 4** (Memory usage of **TemporalSmoothed**). Assuming the chosen sampling algorithm has a maximum memory usage of $f(n)$ where $n$ is the current length of the stream, the **TemporalSmoothed** algorithm’s maximum memory usage will be:

$$f(2 \cdot st + t) + f(st)$$

**Proof.** First, it has to be stated the $f$ function is monotone increasing, since the maximum memory usage of a stream cannot decrease with the increase of the stream length. Secondly, with the help of Figure 4, it can be shown that the main sampler during its lifetime will sample $2 \cdot st + t$ times, which means that the maximum memory usage for the main sampler will be $f(2 \cdot st + t)$. During the main sampler’s lifetime, the secondary sampler samples $st$ times, which means a maximum memory usage of $f(st)$.

Assuming the chosen sampling algorithm can initialize itself and sample a key in $O(1)$ time, then the **Temporal Smoothed** can also sample a key in $O(1)$ time. This assumption is not unreasonable, the sampling process has to sample large amounts of data in real-time.

**Theorem 5** (**Temporal Smoothed** drift detection). The **Temporal Smoothed** algorithm can detect a change in the stream after sampling less then $t + 2 \cdot st$ number of times.

**Proof.** In the algorithm a newly created sampler has three phases (Figure 4).

- In phase one, the sampler takes the role of the secondary sampler. The length of this phase is $st$ samples. (The exception to this is the very first sampler which already starts as main sampler)
- In phase two, the sampler becomes a main sampler. The length of this phase is $t$ samples.
- In phase three, the sampler is still the main sampler, but at the end of this phase, it gets discarded and replaced. This phase overlaps the newly created secondary sampler’s phase one. The length of this phase is $st$ samples.

By examining the phase in which the change in the stream happens, we can determine the following cases:

1. The change happens exactly at the start of phase one. The change will be detected when the sampler becomes the main sampler, which is in $st$ samples. (This cannot happen for the very first sampler, because the starting point of the change and the stream would overlap)
2. The change happens in phase one or two. The sampler may not correctly detect the change since it has samples from before the change and after it. On the other hand, the next main sampler will only have samples from after the change and can correctly detect it. This will happen in no more than $t + 2 \cdot st$ samples, which is the upper estimate for the replacement of the current main sampler.
3. The change happens in phase three. This case is covered by the first two cases because this coincides with the phase one of the current secondary sampler.

**5.2 Checkpoint Smoothed**

Our next algorithm is called **Checkpoint Smoothed**. This algorithm works similarly to **Temporal Smoothed**, but rather than periodically and rigidly renewing the main sampler, it aims to only replace it if we are certain enough that the main sampler’s reported results are not accurate due to a possible change in the stream.

Our algorithm works with a main sampler and a secondary sampler. After our main sampler samples enough data (checkpoint window), a new secondary sampler is initialized. The two samplers sample concurrently for
another window (check threshold). After this, the reported results of the main sampler and secondary sampler gets compared using Hellinger distance. If the result is beyond a predetermined threshold (error threshold), the main sampler is replaced by the secondary sampler. We repeat the aforementioned process, either with the original main sampler or the new one.

```
Algorithm 2: CheckpointSmoothed

Input: key: Key
Parameter: ms: SamplerBase, main sampler
Parameter: ss: SamplerBase, secondary sampler
Parameter: cw: N+, checkpoint window
Parameter: ct: N+, check threshold
Parameter: ct: N+, error threshold
Parameter: checkpoint: N+, helper variable

1. totalProcessedElements ←− totalProcessedElements + 1;
2. ms.recordKey(key);
3. if ss initialized then
4.    ss.recordKey(key);
5. end
6. if ss not initialized and
totalProcessedElements > checkpoint + cw then
7.    initialize ss;
8. else if ss initialized and ss.totalProcessedElements > ct then
9.    expected ←− ss.estimateRelativeFrequencies;
10.   actual ←− ss.estimateRelativeFrequencies;
11.   result ←− H(expected, actual);
12. if result > ct then
13.    ms ←− ss;
14. end
15. discard ss;
16. checkpoint ←− totalProcessedElements;
```

The Checkpoint Smoothed algorithm can also encapsulate any sampling algorithm.

**Theorem 6** (Memory usage of Checkpoint Smoothed). Assuming the chosen sampling algorithm has a maximum memory usage of \( f(n) \) where \( n \) is the current length of the stream, the Checkpoint Smoothed algorithm’s maximum memory usage will be:

\[
f(n) + f(ct)
\]

**Proof.** If there is too little or no change in the stream, the main sampler will never be replaced. This means that the main sampler’s maximum memory usage is \( f(n) \). If the main sampler never gets replaced, a new secondary sampler is started periodically and will sample \( ct \) amount of data. This means that the maximum memory usage of the secondary sampler is \( f(ct) \). □

The sampling time of a key for the Checkpoint Smoothed algorithm depends on multiple things:

1. The chosen sampling algorithm: As stated in the Temporal Smoothed algorithms run time, it is not unreasonable to assume that a sampler can initialize and sample a key in \( O(1) \).
2. To calculate the Hellinger distance periodically, the relative frequencies of the main sampler and secondary sampler are needed. The run time is therefore tied to the size of the main sampler’s and the secondary sampler’s output.
3. The calculation of the Hellinger distance can be done in linear time based on the size of its inputs, therefore this does not increase the asymptotic run time.

Based on these, the Checkpoint Smoothed algorithm can sample a key in \( O(g(n) + g(ct)) \) time. Where \( g(x) \) is the maximum number of relative frequencies calculated by the chosen sampling algorithm, given \( x \) samples.

The asymptotic run time can be quite misleading, because the majority of times \((cw + ct - 1)\) out of \( cw + ct \) times) the run time will be \( O(1) \). The algorithm’s sample time can be improved upon. For example, if we run the calculation of Hellinger distance concurrently, we do not have to interrupt the sampling process. This introduces a possible error, which is that we may calculate the errors for a main sampler with \( cw + \epsilon_1 \) samples and for a secondary sampler with \( ct + \epsilon_2 \) samples. If the parameter \( ct \) and \( cw \) are reasonable in size this will not cause a significant change in the algorithm’s output.

### 6 Analysis

We use concept drifts where the midpoint of the concept drift is at the midpoint of the data stream. This is to give the algorithms as much time to react to the concept drifts as they had to estimate the frequencies before the concept drift.

All data sets consist of 5 000 000 elements with a key set of size 100 000. In our tests, we use micro-batches of size 30 000 and a top-K value of 300, because with the chosen Zipfian distributions this top-K should include the keys with meaningful frequencies (> 1%).

The number of test cases grows exponentially with each new algorithm and hyperparameter. Showing all of these would be impractical, because we have limited space, so instead we present only the relevant cases and provide all of our measurements on GitHub [1]. The measurements were made on a 6 core, 12 thread Ryzen 3600 CPU clocked at 3.6GHz with 16GB of RAM.

We only show our algorithms in this paper with Frequent and Lossy Counting. Frequent performed the best in the majority of our tests and only seems to have difficulties with bursts. We use Lossy Counting as the encapsulated sampling algorithm in Temporal Smoothed and Checkpoint Smoothed, due to its speed, low memory usage and accuracy. Lossy Counting is robust, it is resilient.
Table 1: Run time for Frequent, Lossy Counting, Checkpoint Smoothed and Temporal Smoothed algorithms with and without bursts

| Algorithm            | Gradual | Gradual | Gradual | Gradual |
|----------------------|---------|---------|---------|---------|
|                      | \( exp = 1 \) | \( exp = 1 \) | \( exp = 2 \) | \( exp = 2 \) |
| Frequent             | 1355 ms | 1168 ms | 546 ms  | 421 ms  |
| Lossy Counting       | 1493 ms | 1387 ms | 616 ms  | 504 ms  |
| Checkpoint Smoothed  | 2089 ms | 1854 ms | 868 ms  | 760 ms  |
| Temporal Smoothed    | 1979 ms | 1784 ms | 849 ms  | 743 ms  |

against data burst and is amongst the worst performers when it comes to reacting to concept drifts.

We use the same parameters for Temporal Smoothed and Checkpoint Smoothed as for Landmark to allow a fair comparison. After running multiple optimizations over various datasets, we decided to use an error threshold of 0.2 for Checkpoint Smoothed.

- Temporal Smoothed: threshold = 40000, switch threshold = 40000
- Checkpoint Smoothed: checkpoint window = 40000, check threshold = 40000, error threshold = 0.2

Temporal Smoothed and Frequent react the fastest to concept drifts, which is due to their windowed nature. Checkpoint Smoothed can be seen slightly inaccurate during the concept drift, but it quickly corrects itself. During the gradual drift, we can see multiple cuts made by Checkpoint Smoothed, but after the concept drift it rapidly corrects itself.

With the introduction of light burst Checkpoint Smoothed gives the best results on average because it is not hypersensitive to tiny changes. Temporal Smoothed and Frequent however give better results between light bursts compared to Checkpoint Smoothed.

With the introduction of heavy burst Checkpoint Smoothed can better dampen the effects of bursts compared to Frequent and Temporal Smoothed.

We can observe the standard deviation of the memory usage increase when bursts are introduced. Checkpoint Smoothed uses almost twice as much memory than its base, Lossy Counting. Temporal Smoothed uses similar amounts of memory compared to its base but with a greater deviation which becomes more apparent with bursts.

The introduction of bursts does not influence run time significantly. We can see that our algorithms behave the same way as the state-of-the-art algorithms. When the exponent becomes higher, the run time becomes lower.

We chose a partition number of 5 to test our algorithms.

An abrupt concept drift causes a sudden jump in the imbalance error, which is hard to recover from, but Temporal Smoothed can do so quickly. Checkpoint Smoothed is slowly improving. When heavy bursts are introduced the imbalance caused by the concept drift is not significantly greater than before (please note the scale of the diagrams in Figure 7).

The gradual concept drift causes dipping in the imbalance error, which can be observed here as well. The gradual concept drift is easier to recover from, and both Temporal Smoothed and Checkpoint Smoothed can do so quickly. They can also achieve similar results in the load imbalance in heavy burst scenarios compared to Frequent.

In summary, we found that Temporal Smoothed and Checkpoint Smoothed can react fast to concept drifts. These algorithms improved the reaction time to concept drifts compared to the encapsulated Lossy Counting algorithm. The cost of this improvement was the increase in memory usage and run time.

7 Summary and Future Work

In this work, we defined concepts and concept drifts of distributions for data streams. We approached concept drifts from different perspectives and showed that the two approaches have the same expressive power, and used this to provide an intuitive way to define concept drifts in data streams.

We provided a data generator that can handle data bursts and concept drifts, and showed two methods of measuring the error of sampling algorithms. An optimizer was also included in the framework with a random local minimum search that can be used to tune the hyperparameters of algorithms.

We introduced two novel algorithms that could react to concept drifts, and analyzed them in conjunction with the state-of-the-art algorithms using our framework. In our analysis, we found that Frequent reacts fastest to concept drifts, and our algorithms also show good reaction times. Checkpoint Smoothed could achieve better results than Frequent in heavy burst scenarios.

7.1 Future Work

We would like to measure our algorithms on a real world dataset.

We conjecture that concepts and concept drifts can be extended to any number of underlying probability distributions.
Figure 5: *Hellinger distance* between the sample distribution and true distribution at each micro-batch for *Frequent, Lossy Counting, Checkpoint Smoothed (CPS)* and *Temporal Smoothed (TMP)* without burst, and with light and heavy burst introduced.

Figure 6: Memory usage (number of counters) for *Frequent, Lossy Counting, Checkpoint Smoothed (CPS)* and *Temporal Smoothed (TMP)* algorithms with and without bursts with Zipfian distribution of $\exp = 1$ and $\exp = 2$.

Figure 7: Percent imbalance error for *Frequent, Lossy Counting, Checkpoint Smoothed (CPS)* and *Temporal Smoothed (TMP)* algorithms for abrupt and gradual *concept drifts* with and without bursts with Zipfian distribution of $\exp = 1$, using 5 partitions.
8 References

[1] Lada A. Adamic and Bernardo A. Huberman. “Zipf’s law and the Internet”. In: Glottometrics 3 (2002), pp. 143–150.

[2] Mark Allman and Ethan Blanton. “Notes on burst mitigation for transport protocols”. In: Computer Communication Review 35 (Apr. 2005), pp. 53–60. DOI: 10.1145/1064413.1064419.

[3] Noga Alon, Yossi Matias, and Mario Szegedy. “The Space Complexity of Approximating the Frequency Moments”. In: Journal of Computer and System Sciences 58.1 (1999), pp. 137–147. DOI: 10.1006/jcss.1997.1545.

[4] Albert Bifet et al. “MOA: Massive Online Analysis”. In: J. Mach. Learn. Res. 11 (Aug. 2010), pp. 1601–1604. DOI: 10.5555/1756006.1859903.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. “Finding Frequent Items in Data Streams”. In: Proceedings of the 29th International Colloquium on Automata, Languages and Programming. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 693–703. ISBN: 3540438645.

[6] Graham Cormode. “Misra-Gries Summaries”. In: Encyclopedia of Algorithms. Springer US, 2014, pp. 1–5. DOI: 10.1006/jcsl.1999.1545.

[7] Graham Cormode and S. Muthukrishnan. “An improved stream data summary: the count-min sketch and its applications”. In: Journal of Algorithms 55.1 (Apr. 2005), pp. 58–75. DOI: 10.1016/j.jalgor.2003.12.001.

[8] Erik D. Demaine, Alejandro Lopez-Ortiz, and J. Ian Munro. “Frequency Estimation of Internet Packet Streams with Limited Space”. In: Proceedings of the 10th Annual European Symposium on Algorithms. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 348–360. DOI: 10.1007/3-540-45749-6_33.

[9] Eugene Fink, Ankur Sarin, and Jaime Carbonell. “Analysis of Uncertain Data: Smoothing of Histograms”. In: 2009 IEEE International Conference on Systems, Man and Cybernetics. 2009, pp. 2549–2555. DOI: 10.1109/ICSMC.2009.5346324.

[10] Michael R. Garey and David S. Johnson. “Computers and Intractability: A Guide to the Theory of NP-Completeness”. In: W. H. Freeman and Co., 1979, p. 238. ISBN: 0716710447.

[11] Buğra Gedik. “Partitioning Functions for Stateful Data Parallelism in Stream Processing”. In: The VLDB Journal 23.4 (Aug. 2014), pp. 517–539. DOI: 10.1007/s00778-013-0335-9.

[12] Lukasz Golab et al. “Identifying Frequent Items in Sliding Windows over On-Line Packet Streams”. In: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement. New York, NY, USA: Association for Computing Machinery, 2003, pp. 173–178. DOI: 10.1145/948205.948227.

[13] E. Hellinger. “Neue Begründung der Theorie quadratischer Formen von unendlichvielen Veränderlichen.” In: Journal für die reine und angewandte Mathematik 136 (1909), pp. 210–271.

[14] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. Tech. rep. SRC TR 1998-011. Dec. 1998.

[15] Holden Karau et al. Learning Spark: Lightning-Fast Big Data Analytics. 1st. O’Reilly Media, Inc., 2015. ISBN: 1449358624.

[16] David Karger et al. “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing. New York, NY, USA: Association for Computing Machinery, 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[17] Leonard Kleinrock. “Distributed Systems”. In: Commun. ACM 28.11 (Nov. 1985), pp. 1200–1213. DOI: 10.1145/4547.4552.

[18] Wang Lam et al. “Muppet: MapReduce-Style Processing of Fast Data”. In: Proc. VLDB Endow. 5.12 (Aug. 2012), pp. 1814–1825. DOI: 10.14778/2367502.2367520.

[19] G. Manku and R. Motwani. “Approximate frequency counts over data streams”. In: Proceedings of 28th International Conference on Very Large Data Bases 5 (Jan. 2002).

[20] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. “Efficient Computation of Frequent and Top-k Elements in Data Streams”. In: Proceedings of the 10th International Conference on Database Theory. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 398–412. DOI: 10.1007/978-3-540-30570-5_27.

[21] J. Misra and David Gries. “Finding repeated elements”. In: Science of Computer Programming 2.2 (Nov. 1982), pp. 143–152. DOI: 10.1016/0167-6423(82)90012-0.
Appendices

A Drifts and concepts are equal in expressive power I.

Proof. We construct such $D^{str}$, $\forall c = (c_e, c_s, c_f) \in Con^{str}$ and show that the construction is correct.

1. if $c$ is a changing concept, say $\exists P^c_1$, $P^c_2$ discrete probability distributions over $K$, $\forall i \in D(c_f), c_f(i) = \Delta((i - c_e), c_e, P^c_1, P^c_2)$, let
   $$ d := (c_e - c_s, \frac{c_e + c_s}{2}, P^c_1, P^c_2) \in D^{str} $$

2. if $c$ is a constant concept, say $\exists P^c$ discrete probability distribution over $K$, $\forall i \in D(c_f), c_f(i) = P^c$, let $P^c_{str}$ be an arbitrary discrete probability distribution over $K$ (it does not matter)
   $$ d := (0, c_e, P^c_{str}, P^c) \in D^{str} $$

For any $i \in [1, 2, \cdots, n]$, $\exists c = (c_e, c_s, c_f) \in Con^{str} : c_s \leq i \leq c_e$ (based on requirements for $Con^{str}$). We show that for any $c$ concept, our constructed corresponding $d$ drift is correct by showing that $P^c_{str} = P^c_{str}$

1. if $c$ is a changing concept, with $P^c_1$, $P^c_2$ discrete probability distributions over $K$, $\forall i \in D(c_f), c_f(i) = \Delta((i - c_e), c_e, P^c_1, P^c_2)$, from our construction
   $$ d = (c_e - c_s, \frac{c_e - c_s}{2}, P^c_1, P^c_2) \in D^{str} $$
   $$ s(d) = \frac{c_e + c_s}{2} - \frac{c_e - c_s}{2} \leq i $$
   $$ i \leq e(d) = \frac{c_e + c_s}{2} + \frac{c_e - c_s}{2} $$

reduces to
   $$ c_s \leq i \leq c_e $$
Therefore we can use our first generation rule for \(d\) drift,
\[
P_G^{i,\text{str}} = \Delta(i - s(d)) \frac{1}{c_e - c_s}, P_1^c, P_2^c
\]
\[
= \Delta \left( \frac{i - s(d)}{c_e - c_s} \right), P_1^c, P_2^c
\]
\[
= \Delta \left( \frac{i - s(d)}{c_e - c_s} \right), P_1^c, P_2^c = P_T^{i,\text{str}} \]

2. if \(c\) is a constant concept, with \(P^c\) dis\- 

Proportionality distribution over \(K\), \(\forall i \in D(c_f), c_f(i) = P^c\).

The concepts are non-overlapping, so

\[
\forall c' = (c'_s, c'_i, c'_f) \in \text{Con}^{\text{str}} : \exists d' \in D^{\text{str}} : c'_s = s(d') \leq e(d') \leq c_e
\]

\[
\exists P^c (\text{an arbitrary dis\- proportionate distribution over } K).
\]

if \(\exists d = (0, c_s, P^c, P^c) \in D^{\text{str}} : c_s + 0 < i\), we can use our second generation rule for \(d\) drift, that is

\[
P_G^{i,\text{str}} = P^c = c_f(i) = P_T^{i,\text{str}}
\]

if \(\exists d = (0, c_s, P^c, P^c) \in D^{\text{str}} : c_s + 0 < i \land \exists d' \in D^{\text{str}} : c_s + 0 < s(d') \leq i\), we can use our third generation rule for \(d\) drift, that is

\[
P_G^{i,\text{str}} = P^c = c_f(i) = P_T^{i,\text{str}}
\]

\[\square\]

B Drifts and concepts are equal in expressive power II

Proof. We begin by constructing such \(\text{Con}^{\text{str}}\), \(\forall d = (d_{\text{len}}, d_{\text{str}}, d_P, d_{\text{str}}) \in D^{\text{str}}\).

1. if \(d_{\text{len}} > 0\), let

\[
(s(d), e(d), \Delta(i - s(d)) \frac{1}{d_{\text{len}}}, d_P, d_{\text{str}}) \in \text{Con}^{\text{str}}
\]

and if \(\exists \exists d' \in D^{\text{str}} : s(d') = e(d) + 1\) and \(e(d) \neq n\)

\[
(e(d) + 1, \mid d_{\text{len}}, e_{d}, c_f) \in \text{Con}^{\text{str}}
\]

\[
\text{where } \forall i \in D(c_f) : c_f(i) = d_P, \text{ and }
\]

\[
\text{end}_d = \left\{ \begin{array}{ll}
\text{n}, & \text{if } \exists \exists d' \in D^{\text{str}} : d_{\text{str}} > d_{\text{mid}} \\
\text{s(d')} - 1, & \text{if } \exists \exists d' \in D^{\text{str}} : d_{\text{str}} > d_{\text{mid}}
\text{ and } \exists \exists d'' \in D^{\text{str}} : \\
d_{\text{mid}} < d_{\text{mid}} < d_{\text{mid}}
\end{array} \right.
\]

2. if \(d_{\text{len}} = 0\), let

\[
(s(d), end_d, c_f) \in \text{Con}^{\text{str}}
\]

\[
\text{where } \forall i \in D(c_f) : c_f(i) = d_P, \text{ and end_d is the same as before.}
\]

For any \(i \in [1, 2, \ldots, n]\), we show that for any \(d\) drift, our constructed corresponding \(c\) concept is correct by showing that \(P_T^{i,\text{str}} = P_G^{i,\text{str}}\)

1. if \(\exists d = (\text{len}, \text{mid}, P_1, P_2) : s(d) \leq i \leq e(d)\) and \(\text{len} > 0\) then, based on the given construction

\[
\exists c = (s(d), e(d), \Delta(i - s(d)) \frac{1}{\text{len}}, P_1, P_2)
\]

\[
P_T^{i,\text{str}} = \Delta \left( \frac{i - s(d)}{e(d) - s(d)} \right), P_1, P_2
\]

\[
= \Delta \left( \frac{i - s(d)}{\text{len}} \right), P_1, P_2 = P_G^{i,\text{str}}
\]

2. if \(\exists d = (\text{len}, \text{mid}, P_1, P_2) : \text{len} > 0 \land \exists d' \in D^{\text{str}} : e(d) < s(d') \leq i\), then,

(a) if \(\exists d' \in D^{\text{str}} : d_{\text{mid}} > d_{\text{mid}} \land \exists \exists d'' \in D^{\text{str}} : \\
d_{\text{mid}} < d''_{\text{mid}} < d''_{\text{mid}}\) then, based on the given construction

\[
\exists c = (e(d) + 1, s(d') - 1, c_f) \in \text{Con}^{\text{str}}
\]

\[
\text{where } \forall i \in D(c_f) : c_f(i) = P_2
\]

(b) if \(\exists d' \in D^{\text{str}} : d_{\text{mid}} > d_{\text{mid}}\) then, based on the given construction

\[
\exists c = (e(d) + 1, n, c_f) \in \text{Con}^{\text{str}}
\]

\[
\text{where } \forall i \in D(c_f) : c_f(i) = P_2
\]

In both cases,

\[
P_T^{i,\text{str}} = c_f(i) = P_2 = P_G^{i,\text{str}}
\]

3. if \(\exists d = (0, d_{\text{mid}}, d_P, d_{\text{str}}), e(d) \leq i \land \exists \exists d' \in D^{\text{str}} : \\
e(d) < s(d') \leq i\), then,

(a) if \(\exists d' \in D^{\text{str}} : d_{\text{mid}} > d_{\text{mid}} \land \exists \exists d'' \in D^{\text{str}} : \\
d_{\text{mid}} < d''_{\text{mid}} < d''_{\text{mid}}\) then, based on the given construction

\[
\exists c = (s(d), s(d') - 1, c_f)
\]

\[
\text{where } \forall i \in D(c_f) : c_f(i) = d_P
\]

(b) if \(\exists d' \in D^{\text{str}} : d_{\text{mid}} > d_{\text{mid}}\) then, based on the given construction

\[
\exists c = (s(d), n, c_f) \in \text{Con}^{\text{str}}
\]

\[
\text{where } \forall i \in D(c_f) : c_f(i) = d_P
\]

In both cases,

\[
P_T^{i,\text{str}} = c_f(i) = d_P = P_G^{i,\text{str}}
\]

\[\square\]