Incremental Sketch Algorithms for Frequent Item Finding in Data Streams

Capstone Project Report

Lara Santos Bastos

Bachelor in Informatics and Computing Engineering

U.Porto Tutor: Pedro Nuno Ferreira da Rosa da Cruz Diniz
Proponent: Bruno Miguel Delindro Veloso

June 27, 2024

Contents

1 Introduction

This report aims to offer an overview of the Curricular Internship undertaken at INESC TEC for the Capstone Project Curricular Unit. The task proposed for the internship consisted of the study, implementation and validation of Incremental Sketch Algorithms for data streams.

1.1 Motivation and Context

Due to my growing interest in artificial intelligence and machine learning, I aim to culminate my three-year bachelor's degree by exploring these fields further. This interest and my desire to engage in research led me to seek an internship that combined both areas. I wanted an opportunity to gain experience in the fields I intend to follow for my Master's degree and professional career.

The Curricular Internship was conducted at LIAAD, the Laboratory of Artificial Intelligence and Decision Support within INESC TEC, a research institute on the Faculty of Engineering campus of the University of Porto. LIAAD specializes in Data Science, Machine Learning, Statistics, and Mathematics. One of the research areas within LIAAD is Data Mining and the study of Sketch Algorithms.

At the start of the internship, I was presented to River[12], an online Machine Learning Framework designed for working with streaming data. It offers a variety of algorithms and tools tailored to facilitate the extraction of insights from data streams. River contains a dedicated section for Sketch Algorithms, which is used for counting frequent items in data streams, the topic on which my internship focused.

1.2 Goals and expected results

Given the significance of counting algorithms for streaming data, the primary goal of the Internship was to implement a set of Incremental Sketch Algorithms for the River framework. The implementation aimed to provide the framework with extra features for data stream analysis.

The expected outcomes of this Internship included:

- Learn sketch algorithms, understand their principles, and evaluate their relevance.
- Implementing the algorithms within the framework.
- Documenting the implementations made per the framework's guidelines.
- Performing testing and validation of the implemented algorithms, comparing their performance with existing sketch algorithms.
- Understanding the algorithms' performance across various data streams and problem contexts.

1.3 Report Structure

The Internship report is organized into five main sections, each serving a specific purpose:

- Introduction: Introduces the entity where the Internship was conducted and provides information on the project's purpose and objectives, setting the stage for the subsequent exposures and discussions.
- Literature Review: Contextualizes the Frequent Item Sets problem and Incremental Sketch Algorithms, aiming to establish the background and motivation for the research over this particular problem. It introduces the current state of the art in this field of data science, presenting some classic implementations, appropriate metrics for evaluation, and some real-world applications of these algorithms.
- Methodology and Activities Developed: Outlines the methodology utilized throughout the project and the roles of all involved parties and presents an overview of the activities undertaken during the Internship.
- Solution Developed: Exposition of project requirements and needs, technologies used, and the River Framework structure. Explain the selected algorithms, the implementations made, and the tests conducted to assess the effectiveness and reliability of the algorithms.
- Conclusions: Summarizes the project's outcomes and achieved results. Reflect on the lessons learned throughout the Internship and propose future improvements.

2 Literature Review

This section presents an overview of the existing research related to the Frequent Item Finding Problem and Incremental Sketch Algorithms. The following subsections will explain key concepts, present classic Incremental Sketch Algorithms along with their evaluation metrics, and discuss their real-world applications.

2.1 The Frequent Item Finding Problem and Incremental Sketch Algorithms

The Frequent Item Finding Problem [3] differs from traditional batch processing, which analyzes entire datasets. Instead, it focuses on processing data streams to identify items above a specified threshold, known as heavy hitters, the most frequently occurring in the dataset. Algorithms tackling this problem must handle updates quickly without excessive memory usage.

Incremental Sketch algorithms constitute one of the algorithms used for frequently finding items in a data stream. These are designed to efficiently process large volumes of streaming data while maintaining a fixed and limited memory space. Their internal structures, designated as sketches, can be seen as a linear input projection and are updated as new data arrives.

By refining their output incrementally, these algorithms avoid reprocessing the entire dataset at the arrival of new data. They are particularly suited for real-time analysis in scenarios involving large data volumes that cannot fit into memory.

2.2 Classic Incremental Sketch Algorithms

In the realm of the frequent item-finding problem, several Sketch algorithms have emerged as tools for analyzing streaming data efficiently without requiring extensive computational resources:

- Count Sketch algorithm: The *Count Skecth* [2] algorithm employs pairwise hash functions and a two-dimensional array of counters to approximate frequency counts of items in data streams with controlled error rates.
- Count-Min Sketch: The *Count-Min Skecth*[5] algorithm uses a two-dimensional array of counters and multiple pairwise hash functions as well, used to map items into one or more entries of the array. Suitable for applications requiring more accurate frequency estimation.
- Lossy Count: The *Lossy Count* [9] algorithm maintains a data structure that stores tuples with items and their respective upper and lower bounds. These bounds are updated as new items arrive.

Other classic sketch algorithms, such as HyperLogLog and Space Saving, will be explored in subsequent sections, as they were implemented during the internship.

2.3 Evaluation Metrics

A set of measures should be taken into account for the proper analysis and validation of the performance of sketch algorithms. These are:

- Space Complexity: Efficient memory usage is crucial, as one of the main advantages of this class of algorithms is their ability to operate within strict memory constraints.
- Speed: In particular, update throughput, given by the time taken to update the sketches. Measures the algorithm's efficiency in processing new elements in the data stream.
- Accuracy: Measures how closely the sketch algorithm's results approximate the true counts of heavy hitters. It is calculated as the number of true heavy hitters identified over the total number of heavy hitters.
- Recall: Measures the proportion of true heavy hitters identified by the algorithm compared to the actual number of heavy hitters in the data stream.
- Error: Quantifies the difference between the estimated and actual frequencies. There are multiple error metrics, such as Mean Absolute Error, Relative Error and Average Relative Error.

These measures will serve as the benchmarks for evaluating the performance and correctness of the implementations developed during the internship.

2.4 Real-World Applications

Incremental Sketch algorithms serve various purposes in the real-world, making their study and research justified.

One of their key applications is in network traffic monitoring. By analyzing streaming data in large-scaling networks, these algorithms can identify unusual patterns and anomalies, which may indicate security breaches or network failures. This may be crucial for maintaining the stability and security of networks, whereas other methods of analysis may need to be more practical due to the high volume of data.

Another application is in web analytics. Incremental Sketch algorithms can enable real-time processing of website user interactions and behaviour. By continuously updating sketches with each new interaction, these algorithms can identify popular pages and trending topics, useful information for optimizing website performance and enhancing user experience.

Among these, other applications demonstrate why Incremental Sketch Algorithms are so relevant and why they were chosen as the topic for the internship.

3 Methodology and Activities Developed

3.1 Methodology

The internship was conducted based on the Scrum [15] methodology to ensure iterative development and effective work management. At the beginning of the semester, an initial set-up meeting was held to introduce the project and the River framework and give initial development guidelines. All tasks and requirements were defined and scheduled throughout the internship timeline.

The work was held every Friday from 9 AM to 5 PM online, with frequent feedback from the INESC TEC Proponent. These meetings alternated between online sessions and in-person meetings at the INESC TEC offices. After the completion of each phase, the work was presented in person, feedback was received, and the following phase was planned.

An individual GitHub repository was utilized for version control management to house all algorithms and their respective tests. Each new algorithm or fix was handled in a new branch to maintain a structured development process. The code developed to support this work can be found in [1].

The final work was merged into the River repository to integrate with the framework.

3.2 Stakeholders, roles and responsibilities

Lara Santos Bastos fully developed the project.

Bruno Miguel Delindro Veloso, researcher at LIAAD and assistant professor at the Faculty of Economics of the University of Porto, served as the proponent of INESC TEC. He was responsible for closely overseeing the project through frequent meetings and providing technical support and feedback. He also served as the Product Owner, delineating the project objectives and requirements.

Pedro Nuno Ferreira da Rosa da Cruz Diniz, Full Professor at the Faculty of Engineering of the University of Porto, served as the U.Porto Tutor. He provided academic support for the internship proposal and was responsible for grading the report.

3.3 Activities developed

Once proposed, the project was delineated into five main phases, each subdivided into specific tasks. The internship followed all the planned activities with some additions, resulting in the completion of all listed activities:

- Searching and Planning
  - Requirements Gathering: Identify the specific project requirements, including the algorithms to be implemented, desired levels of accuracy and any performance constraints.

  - Analysis of the River Framework: Study the structure of the River framework, understand the key concepts, and identify integration points for implementing sketch algorithms.

- Environment Setup
  - Installation of the River Framework: Set up the development environment by installing the River framework and its dependencies.
  - Development Environment Configuration: Configure an appropriate development environment, including the IDE, version control and other necessary tools.

- Implementation
  - Research of the Algorithms: Study and understand the workings of the algorithms chosen for implementation in the project context.
  - Development of Incremental Sketch Algorithms: Implement the chosen algorithms in the River framework, following good coding practices.
  - Testing: Conduct testing to validate the correctness and robustness of the implemented algorithms.
  - Documentation of the Algorithms: Document each algorithm developed.
  - Integration with the Framework: Integrate the implemented algorithms into the River framework, ensuring compatibility and compliance with its standards.

- Validation and Optimization:
  - Validation of Results: Validate the results of the implemented algorithms by comparing them with reference implementations or known data.

- Conclusion and Delivery:
  - Preparation of the Report and Video: Prepare and finalize the report to share the results and the implementation process. Prepare the video for delivery.

The internship was over five months, as depicted in the Gantt Chart in Figure 1.

After the Searching and Planning Phase, a list of three algorithms to implement and their corresponding desired performance constraints was finalized. The subsequent phase involved setting up the IDE and framework.

Phase 3 was characterized by an iterative process for each algorithm involving study, implementation, and testing. After completing these tasks, the algorithms were properly documented and adjusted to adhere to River's coding standards. By the end of Phase 3, all three algorithms were prepared for validation.

Phases 4 and 5 proceeded concurrently. The project report was drafted while the algorithms underwent validation and comparison with existing implementations. Following the report's finalization, the video presentation was recorded, fulfilling the requirements of the Curricular Unit.

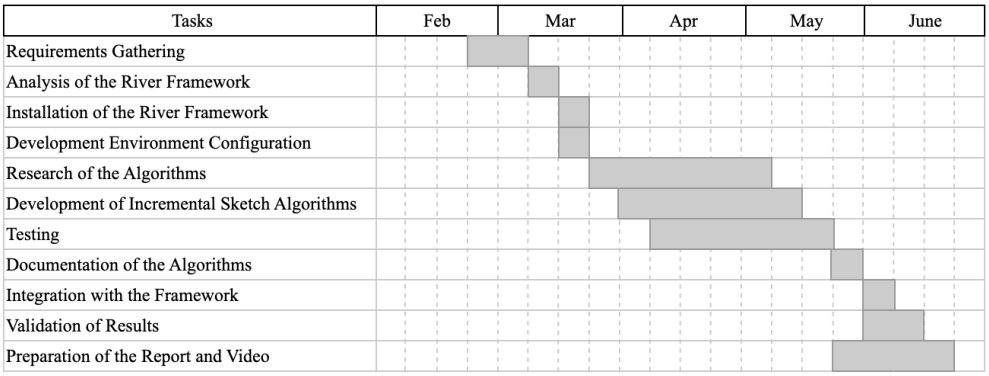

Figure 1: Gantt Chart: Project Phases and Tasks

Throughout the project, meetings were held with the INESC TEC Proponent at the end of each phase to review progress and strategize the following phase.

4 Solution Development

4.1 Requirements and Restrictions

The first phase of the internship involved analyzing the project to be implemented and then gathering the requirements and constraints. These steps ensure that the work ahead is adequately outlined and that the final solution meets the expected standards.

The functional requirements identified for the internship are as follows:

- Implementation of three Incremental Sketch Algorithms under the River Framework guidelines:
  - Space Saving;
  - HyperLogLog;
  - A solution to the Hierarchical Heavy Hitters problem.

  These algorithms were selected based on their distinct functionalities: Space Saving for identifying heavy hitters and counting elements, HyperLogLog for estimating dataset cardinality, and HHH for organizing heavy hitters hierarchically.
- Documentation of all parameters, arguments, and methods implemented.
- Integration of the algorithms into the framework.
- Obtaining clear metrics on the efficiency of each algorithm.

In terms of non-functional requirements, the following were identified:

- Correctness of algorithms.
- Efficiency comparable to or better than existing implementations (in terms of speed, memory usage, and accuracy).
- Usability: The algorithms should be well-organized and documented for ease of use by the River framework users.
- Scalability: The algorithms should be designed to facilitate easy expansion.

The only restriction was that the project and the corresponding report should be completed by June 28th, 2024.

4.2 Architecture and technologies

The algorithms implemented were integrated into the River framework, an online Python machine learning framework designed to build machine learning models on data streams. The primary purpose of River is to enable machine learning tasks to be performed in a streaming manner, utilizing online processing.

Unlike traditional machine learning models, which are trained on large batches of data simultaneously, online processing involves handling one data element at a time. This approach provides efficient algorithms for real-time data stream processing, making River an ideal tool for applications that require incremental learning.

The River framework is divided into three main sections:

- Recipes: Tutorials on how to perform simple machine-learning tasks.
- Examples: Notebooks demonstrating traditional machine learning problems.
- API Reference: Modules, classes, and functions in River, including a section dedicated to sketch, shown in Figure 2, where the implemented algorithms will be added.

The framework includes various sketch algorithms that cater to different needs. These algorithms utilize parameters like thresholds and error control factors to tune their operations and are updated by data streams. Users can query them for specific metrics using well-defined interfaces. Figure 3 depicts the simplified workings of a river sketch algorithm.

The structure of a single sketch algorithm will be further explored in subsequent sections.

The project was developed using Python 3.8.5 within Visual Studio Code. Several Python libraries were utilized to improve the algorithm's functionalities and support the testing and performance analysis tasks:

- Math: Used for its extra mathematical operations.
- Typing: Used for type annotations.
- Numpy [13]: Used for generating data streams with a Zipf distribution.
- Matplotlib [10]: Used for plotting speed, memory usage, accuracy, recall, and relative error metrics.
- Pympler[14]: Used to measure the memory usage of Python objects.

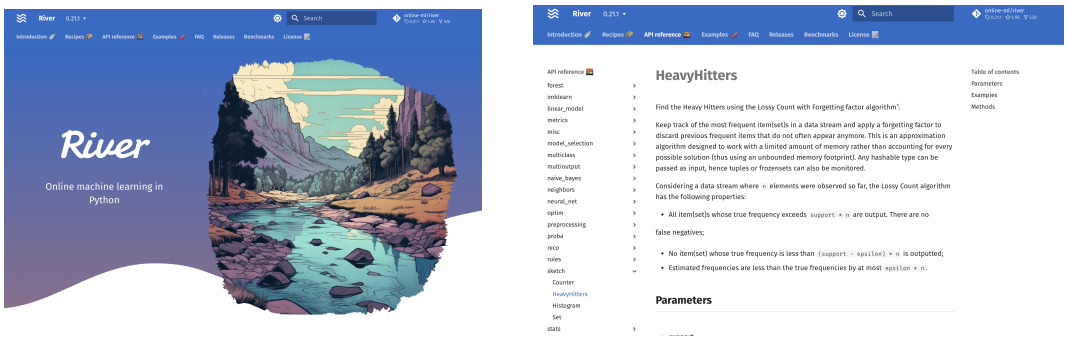

Figure 2: River Framework and its Sketch Section

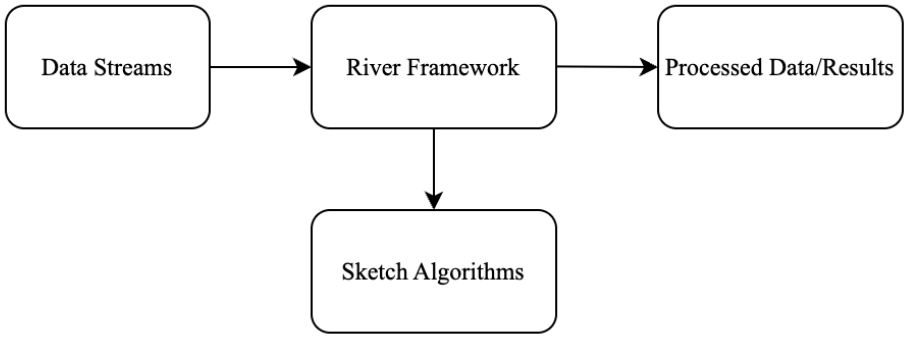

Figure 3: Sketch Algorithms in River.

- Time: Used for measuring the execution time of operations in the stream processing algorithms.

Their easy nature primarily drove the decision to use these libraries. The familiarity with them helped conclude the project without encountering significant technical difficulties.

4.3 Algorithms

In response to the project's objectives, the following three algorithms were proposed for implementation:

4.3.1 Space Saving Algorithm

The *Space Saving* algorithm [11] comprises a counter-based methodology that provides a space-efficient approach to identifying the top-k most frequent elements within a data stream.

The underlying principle revolves around maintaining a data structure containing only the k most common elements at any given time. For this purpose, the algorithm initializes the data structure with k tuples, each consisting of an element along with its associated counter, serving as an estimate of its frequency ((*element, counter*)).

Upon receiving a new item from the data stream, the algorithm increments its counter if it corresponds to an already tracked element. Otherwise, the algorithm finds the tuple with the smallest counter value and replaces its element with the new element, incrementing its counter.

The Space Saving algorithm provides a trade-off between memory usage and accuracy.

Increasing values of k can reduce the error bound, which stands at approximately $\frac{N}{k}$, where N is the total number of items seen in the stream, as there are fewer

Algorithm 1 Space Saving Pseudocode

```
1: C ← {}
2: for each element e in stream S do
3:     if e exists in C then
4:         count_e ← count_e + 1
5:     else
6:         if |C| < k then
7:             C ← C ∪ {(e, 1)}
8:         else
9:             (min, count_min) ← element with min count in C
10:            C ← C \ {(e_min, minCount)}
11:            C ← C ∪ {(e, minCount + 1)}
12:        end if
13:    end if
14: end for
```

substitutions of elements. However, this comes at the expense of higher memory consumption, as the space complexity of the algorithm is $O(k)$.

The time complexity is dependent on the data structure's ability to find items. Other operations, such as retrieving an item's count or determining the number of heavy hitters, have a constant time complexity $O(1)$.

4.3.2 Hyper Log Log Algorithm

LogLog [6] is a probabilistic algorithm designed to estimate the cardinality of a data set with the aid of m bytes of auxiliary memory, known as registers. Note that neither the replication of items nor the ordering of data is relevant, as the algorithm should only be concerned about the number of distinct items present in the dataset.

Each element in the data set starts by being hashed into a binary string, ensuring data is uniformly distributed and simulating random distribution. The algorithm then proceeds by organizing these binary representations into registers. The number of registers is given by m, denoted by 2^b, b being the precision parameter given by the user. Elements are mapped to a register based on the value encoded in its first b bits of the binary representation.

For each binary string, $\rho(x)$ is defined as the position of the first bit with value '1' on the $\log_2(n/m)$ leftmost bits left. For instance, *ρ(001...) = 3*. This allows the algorithm to discard every element's initial $\log_2(n/m)$ bits. The algorithm then computes for each register the parameter R, the maximum number of leading zeros observed in the binary strings mapped to that register (see Figure 4).

To estimate the cardinality n of the dataset, where n is the number of distinct elements, the algorithm computes an average of these R parameters across all registers. This average approximates $\log_2(n/m)$.

The cardinality n is then estimated using the formula:

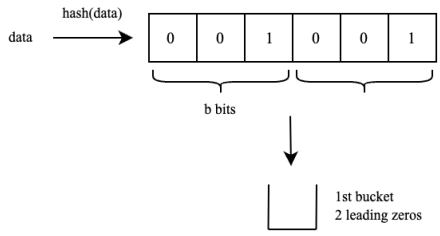

Figure 4: Hashing and Register Assignment in HyperLogLog

$$E = \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

Here, $M[j]$ represents the register values obtained from each bucket. The bias correction constant α_m adjusts the estimated cardinality to correct any systematic biases inherent in the algorithm based on the number of registers m.

Hyper LogLog [7], represents an improvement over the original LogLog algorithm. It employs the same principle but uses a harmonic mean technique to estimate the cardinality.

After adding all elements to the respective registers, the algorithm computes,

$$Z := \sum_{j=1}^{m} X^{2^{-M(j)}}$$

For the calculation of the mean of $2^{M(j)}$, it then returns a normalized version of the harmonic mean,

$$\alpha_m m^2 E := \frac{1}{\sum_{j=1}^{m} 2^{-M(j)}} \cdot \frac{Z}{\left(\sum_{j=1}^{\infty} 2^{2^{-u}m} - 1 \right)^{-1}}$$

with

$$\alpha_m := \int_1^{2^u} \frac{m}{\log_2(1+u)\, du} - 1$$

that can be, however, approximated by the formula:

$$\alpha_m \approx \begin{cases} 0.673, & \text{for } m = 16; \\ 0.697, & \text{for } m = 32; \\ 0.709, & \text{for } m = 64; \\ \frac{0.7213}{1+1.079/m}, & \text{for } m \geq 128. \end{cases}$$

As proved in the paper referenced above, *Hyper LogLog* assures the following properties:

(i) The estimate E is asymptotically almost unbiased.

(ii) The standard error $n_1 \cdot P[V_n(E)]$ satisfies as $n \to \infty$.

Algorithm 2 HyperLogLog Algorithm

```
M ← {}
for each element e in stream S do
    x ← h(e)
    j ← 1 + ⟨x_1 x_2 ... x_b⟩_2
    w ← x_{b+1} x_{b+2} ...
    M[j] ← max(M[j], ρ(w))
end for
Z ← ∏_{j=1}^{m} X^{2^{-M[j]}};
return E ← α_m m^2 Z
```

The computation of the hash function mainly determines the time complexity of HLL. Additionally, its relative error is $\frac{1.04}{\sqrt{m}}$, and it requires $O(\epsilon^{-2} \log \log n + \log n)$ space, where n represents the set cardinality and m the number of registers.

4.3.3 Hierarchical Heavy Hitters

The third and last algorithms proposed have different purposes from the previous ones. In addition to providing an approximate count of the most frequent element in the stream, it hierarchizes these values according to a specific attribute.

Hierarchization algorithms are considered one-dimensional when a single attribute is considered to compute the relation between elements.

Full Ancestry [4] solves the One Dimensional Hierarchical Heavy Hitters problem. Given a new node, it uses the information stored about its ancestors to predict the frequency of the node more accurately.

The algorithm maintains a trie data structure with a set of tuples consisting of the element prefix e and the auxiliary information $(g_e, \Delta e, m_e)$.

The frequency difference between the element and its descendants, denoted as *gp*, is saved for each element. This makes the algorithm "hierarchy aware" as it only needs to insert values until a parent is found. The Δe of each element represents the upper bound on the uncertainty of the count. Given the existent information about the ancestor nodes, we can improve its bounds by keeping track of $m_e = \max_{d \in d(e)}(g_d + \Delta d)$ after every deletion of a child ($\{d(e)\}$ are the deleted children).

The algorithm has two principal phases: *insertion* and *compression*.

For every new data element e received, the *insertion* phase recursively tries to find e in the tree. If present, it increments the counter of the element by its weight. Otherwise, its parent is recursively called until the closer one or the root is reached.

After every w updates, the *compression* phase reduces space by merging unnecessary values, meaning values that satisfy $g_e + \Delta e \leq \lfloor \epsilon N \rfloor$, into the parent node and delete the child. Thus, the input is divided into buckets of $w = \frac{1}{\epsilon}$, where the current bucket is given as $\lfloor \frac{N}{w} \rfloor$.

The *output* operation, on the other hand, is purely informative and given ϕ, outputs e that are an HHH by comparing $F_e + g_e + \Delta e$ to ϕN.

Algorithm 3 Full Ancestry Algorithm

procedure INSERT(e, c)
  if te exists in T **then**
    $ge \leftarrow ge + c$
  else
    create te;
    $ge \leftarrow c$;
    if $anc(e)$ exists in T **then**
      $\Delta e \leftarrow= m_e = m_{anc(e)}$
    else
      $\Delta e \leftarrow m_e = b_{current} - 1$
    end if
  end if
end procedure
procedure COMPRESS
  for each element te in tree T **do**
    if $(ge + \Delta e \leq b_{current})$ **and** te has no descendants **then**
      $m_{par(e)} \leftarrow \max(m_{par(e)}, ge + \Delta e)$
      $g_{par(e)} \leftarrow ge$
      delete te
    end if
  end for
end procedure
procedure OUTPUT(ϕ)
  $Fe \leftarrow 0$
  $f_e \leftarrow 0$
  for each element e in postorder **do**
    if $(ge + \Delta e + Fe > \phi N)$ **then**
      print$(e, f_e + ge, f_e + ge + \Delta e)$
    else
      $F_{\text{par}(e)} \leftarrow F_{\text{par}(e)} + F_e + g_e$
    end if
    $f_{\text{par}(e)} \leftarrow f_{\text{par}(e)} + f_e + g_e$
  end for
end procedure

4.4 Implementation Details

For adding the three explained algorithms to the framework, each was implemented using Python based on the papers referenced.

Efforts were made to ensure the implementations were as similar as possible to allow for better comparison and testing. Furthermore, the code was developed using the framework's coding conventions and standards, with documentation provided for

each function and class to facilitate ease of understanding.

Each Sketch algorithm is defined by a unique set of parameters and methods aimed at managing memory usage, error margins, and other specific requirements. Upon receiving each new element, it invokes an "update" function that updates its internal data structures.

Additionally, each algorithm provides output functions that enable users to retrieve desired metrics such as element counts, estimates of frequencies, cardinality, and other relevant statistics.

Next, it is detailed the implementation of each algorithm:

Space Saving: This algorithm utilizes a dictionary to maintain item counts and accepts a parameter k to specify the maximum number of heavy hitters to track. It allows users to retrieve the count of a specific element, the total number of elements stored, the heavy hitters, and the respective counters.

```
>>> from river import sketch
>>> spacesaving = SpaceSaving(k=10)
>>> for i in range(100):
...     spacesaving.update(i % 10)
>>> spacesaving.total()
100
>>> spacesaving['0']
10
>>> spacesaving.heavy_hitters
{0: 10, 1: 10, 2: 10, 3: 10, 4: 10, 5: 10, 6: 10, 7: 10, 8: 10, 9: 10}
```

Hyper Log Log: This algorithm takes a parameter b to compute the number of registers (m) and α, as well as to initialize an array with m elements representing registers. The algorithm includes a single output function, the "count" method, which returns an estimated cardinality of the data processed so far.

```
>>> hll = sketch.HyperLogLog(b=15)
>>> for i in range(100):
...     hll.update(i)
>>> hyperloglog.count()
100
```

Hierarchical Heavy Hitters: This algorithm takes parameters k for the number of heavy hitters to track and ϵ for error tolerance. It optionally accepts a parent function and a root value to customise the attribute to base the hierarchy. A Node class was created to facilitate the tree structure simulation. Upon receiving new elements, the update function invokes insert to update the tree, and compression is triggered after every w update. The output functionalities include identifying heavy hitters within the hierarchical tree (the output method as explained in [4]), along with options to print the count of a specific element or output the entire tree.

```
>>> def func(x, i):
...     parts = x.split('.')
...     if i >= len(parts):
...         return None
...     return '.'.join(parts[:i+1])

>>> hll = sketch.HierarchicalHeavyHitters(k=10, epsilon=0.001, parent_func=func)

>>> for line in ["123.456","123.123", "456.123"]:
...     hierarchical_hh.update(str(line))

>>> print( hierarchical_hh["123.456"])
1

>>> print(hierarchical_hh)
ge: 0, delta_e: 0, max_e: 0
123:
    ge: 0, delta_e: 0, max_e: 0
    123.456:
        ge: 1, delta_e: 0, max_e: 0
    123.123:
        ge: 1, delta_e: 0, max_e: 0
456:
    ge: 0, delta_e: 0, max_e: 0
    456.123:
        ge: 1, delta_e: 0, max_e: 0

>>> print(hierarchical_hh.output(phi=0.1))
[('123.456', 1), ('123.123', 1), ('456.123', 1)]
```

4.5 Tests and Validations

The final activity in implementing the algorithms was the testing and validation task. Various tests were conducted with two main objectives: validating the correctness of the algorithms and evaluating their performance to conclude.

The validation of the algorithms was carried out continuously during their development. Whenever errors or the worst performance were encountered, such as excessive processing time for a small amount of data, the algorithms were revised and adjusted accordingly. This iterative methodology ensured continuous improvement of the algorithms, leading to a better final product.

The process began with the use of small, manually created data streams. The results obtained from the algorithm implementations were compared against the actual values in these datasets.

Afterwards, larger data streams were utilized to verify the accuracy of the algorithms

and other metrics such as processing speed, memory usage, and recall.

4.5.1 Speed, Memory and Accuracy

To measure these metrics, we varied two parameters on the data streams used: the size and the asymmetry in the distribution of values (skew).

The data streams were created using a Zipf distribution, generated with the function `generate_zipf_data(size, skew)`, which produces data based on the specified skew parameter, allowing for the simulation of real-world data scenarios.

The metrics were calculated as follows:

- **Speed:** The time taken to update the sketches with all the elements received in the stream.
- **Memory:** The memory is occupied by the sketches of the algorithms.
- **Accuracy:**
  - For Space Saving and Hierarchical Heavy Hitters: The proportion of true heavy hitters correctly identified by the algorithm.
  - For HyperLogLog: The cardinality ratio estimated by the algorithm to the actual cardinality of the dataset.

To calculate the true values of heavy hitters and cardinality, exact brute force algorithms were implemented.

Figures 5, 6, and 7 depict the Speed, Memory, and Accuracy metrics for skew values of 1.1, 1.5, and 2.0 respectively.

The update time remained consistent regardless of the skew of the data streams and increased with size across all three implementations. As anticipated, hierarchical Heavy Hitters consistently exhibited significantly higher update times than the others. For Space Saving, its update time varied notably with different skew values, higher for a skew of 1.1 and lower for a skew of 2.0. In moderately skewed distributions, like 1.1, where several items have similar frequencies, the algorithm requires more frequent updates to adjust the counts. In contrast, in highly skewed distributions, where few items dominate, Space Saving can maintain efficient updates with fewer adjustments needed.

HyperLogLog's and Space Saving's memory usage remains constant across different sizes and skews as they allocate a fixed memory space determined by the user. In contrast, Hierarchical Heavy Hitters exhibits variable memory usage depending on the skewness of the data stream. Specifically, for a skew of 1.1, HHH requires a larger memory footprint compared to skew values of 1.5 and 2.0. This difference arises because fewer items dominate the frequency counts in distributions with higher skew. HHH can efficiently compress nodes with lower frequencies into their parent nodes, thereby focusing its memory resources on maintaining a small set of heavy hitters.

In terms of accuracy, it is evident that Hierarchical Heavy Hitters maintains satisfactory accuracy, starting at 0.7 but growing up to 0.95 for larger data sizes. HyperLogLog achieves an accuracy of 0.8 for lower skew values but approaches near-perfect accuracy in estimating cardinality for higher skew values.

On the other hand, Space Saving shows poorer performance, particularly noticeable for skew 1.1. Although its accuracy increases as skew increases, it stabilizes around 0.7 for higher skew values. This could be attributed to the algorithm's reliance on a fixed-size data structure for capturing heavy hitters, which may need help to effectively find them of moderately skewed distributions where frequencies are more evenly spread. In these cases, the decisions on which elements to maintain in the data structures become more arbitrary compared to distributions with very distinct frequencies, where it is obvious which elements are the most frequent.

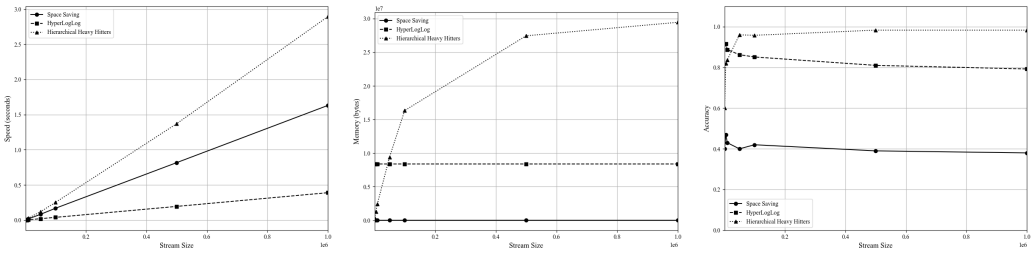

Figure 5: Speed, Memory, and Accuracy for a skew of 1.1

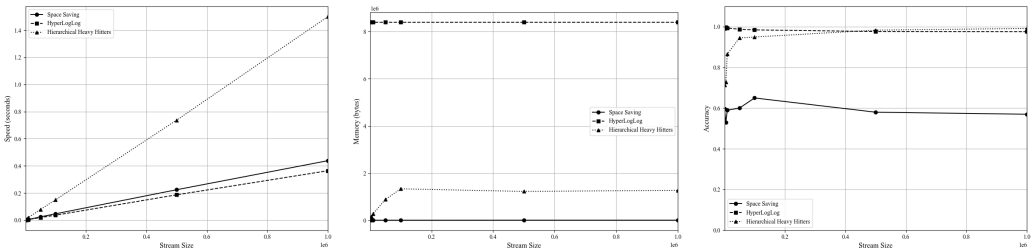

Figure 6: Speed, Memory, and Accuracy for a skew of 1.5

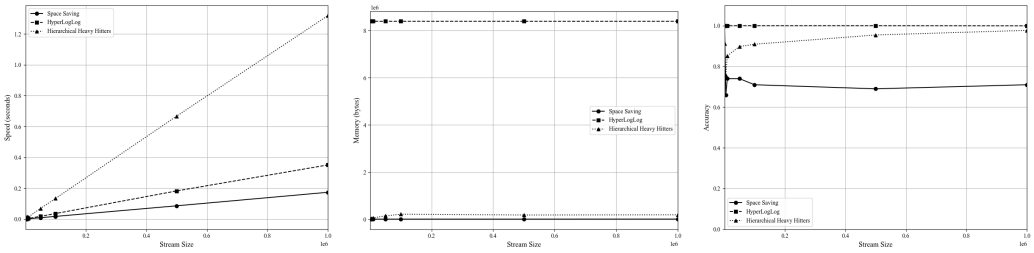

Figure 7: Speed, Memory, and Accuracy for a skew of 2.0

4.5.2 Recall and Relative Error

For the skew value of 2.0, which exhibited the best performance across all algorithms, additional metrics such as Recall and Relative Error were measured to understand these implementations' effectiveness better.

This was assessed as follows:

- **Recall:** The proportion of true heavy hitters/cardinality correctly identified by each algorithm.
- **Relative Error:** The percentage difference between the estimated heavy hitters/cardinality provided by each algorithm and the actual heavy hitters/cardinality of the dataset

Figure 8 illustrates Recall and Relative Error for skew 2.0.

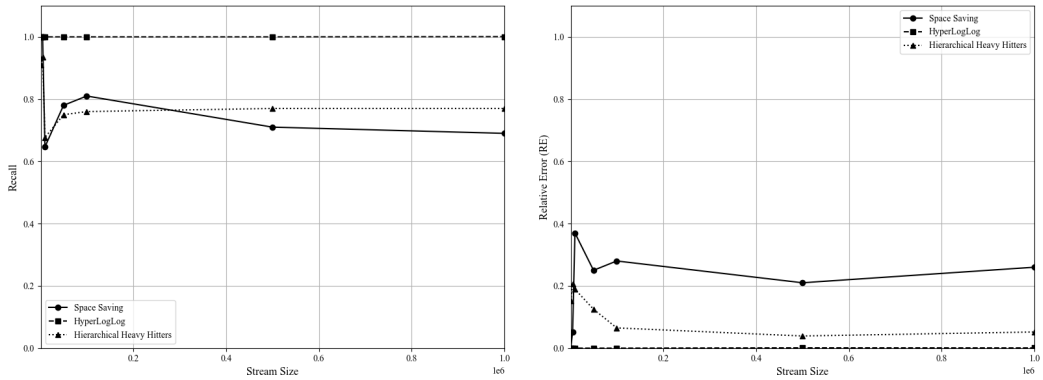

Figure 8: Recall and Relative Error for a skew of 2.0

Once again, HyperLogLog demonstrates an optimal performance, achieving perfect Recall and null Relative Error. Hierarchical Heavy Hitters and Space Saving both achieve satisfactory Recall scores around 0.8. However, Space Saving exhibits a notable Relative Error of 0.3, indicating it is the least accurate algorithm among the three.

4.5.3 Comparison with Base Lines

The focus here was to align the performance of our algorithms with established baseline implementations of frequent item-counting algorithms available online. To achieve this, we utilized the Frequent Itemset Mining Dataset Repository [8], which provides access to five distinct datasets and multiple algorithm implementations.

Each dataset underwent testing to evaluate both speed and memory metrics compared to those reported by existing online implementations. This approach ensured that the implementations developed during the internship aligned with established performance standards.

All the tests conducted are available on the GitHub repository [1] created to store the code.

5 Conclusions

5.1 Achieved Results

All outlined requirements were successfully fulfilled during the internship. The selected algorithms were implemented and integrated into River as planned, ensuring accessibility for new users. Their efficiency was later assessed through performance evaluation.

The algorithms were evaluated based on speed, memory usage, and accuracy across different data sizes and skew values. It was found that each algorithm showed distinct behaviour depending on the characteristics of the data streams, with HyperLogLog demonstrating optimal performance in terms of accuracy, Hierarchical Heavy Hitters showcasing efficient memory management and high accuracy, particularly in highly skewed distributions, and Space Saving providing mediocre results in certain scenarios, due to its limitations.

The methodology employed proved successful and greatly contributed to the project's completion. In particular, dedicating Fridays to the Capstone Project ensured the project remained on schedule without any delays. The ease of communication with the INESC TEC Proponent ensured that any doubts were quickly addressed, allowing a smooth workflow.

5.2 Lessons Learned

The internship allowed me to learn multiple lessons in terms of both technical skills and work ethic.

Firstly, it highlighted the significance of maintaining consistent work and dedicating weekly time slots to work towards the project. Nonetheless, this approach was only as effective with the correct preparation beforehand, highlighting the critical role of defining clear objectives and scheduling tasks throughout the available time.

On a technical level, the importance of thoroughly analyzing papers and scientific articles to understand algorithmic concepts and their properties became evident, a practice not emphasized during the Bachelor in Informatics and Computing Engineering but greatly valued during the internship. Furthermore, the task of utilizing existing implementations for comparison and validation of work was proved to be crucial.

The project also underscored the importance of adapting algorithms to varied datasets and problem domains. Each dataset presented different challenges, necessitating iterative refinement and adjustments. This adaptability ensured that solutions could effectively perform across diverse data distributions and volumes.

Lastly, the importance of following good coding conventions, especially when developing frameworks or APIs, became evident, as the ultimate goal is to improve the user's experience and enhance usability.

5.3 Future work

The fulfilment of all requirements continued the consideration of new ideas and potential improvements in the study and implementation of these algorithms.

Firstly, gaining a deeper understanding of these algorithms and exploring recent advancements could lead to optimizations or further customization options, as the internship mainly focused on the classic instances of the solutions. For example, tackling challenges like the Hierarchical Heavy Hitter problem could involve solutions such as the Partial Ancestry [4] algorithm, which sacrifices some accuracy for reduced space usage, or exploring multidimensional implementations. Space Saving could also be further optimized to obtain improved results.

Exploring, implementing and testing other sketch algorithms could also provide deeper insights into the Frequent Item Finding problem and how different approaches perform under various conditions.

Although this project could be ongoing and there is always room for improvement, the final results and conclusions were highly satisfactory.

References

[1] Lara Bastos. Feup capstone project. `https://github.com/laraabastoss/feup-capstoneproject`. Accessed: 2024-06-17.

[2] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2002.

[3] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *AT&T Labs–Research Technical Report*, 2003. `http://www2.research.att.com/~graham/papers/cormode2003finding.pdf`.

[4] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in streaming data. *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010.

[5] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[6] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). *Algorithms Project, INRIA–Rocquencourt*, 2003.

[7] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Algorithms Project, INRIA–Rocquencourt*. `http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf`.

[8] Bart Goethals and Mohammed J. Zaki. Frequent itemset mining dataset repository. `http://fimi.uantwerpen.be/data/`, 2003. Accessed: 2024-06-17.

[9] Gurmeet Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *International Conference on Very Large Data Bases*, pages 346–357, 2002.

[10] Matplotlib Contributors. Matplotlib. `https://matplotlib.org/stable/`. Accessed: 2024-06-17.

[11] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. *The Computer Journal*, 27(2):97–111, 2005.

[12] Jacob Montiel, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, Heitor Murilo Gomes, Jesse Read, Talel Abdessalem, et al. River: machine learning for streaming data in python. 2021.

[13] NumPy Contributors. Numpy. `https://numpy.org`. Accessed: 2024-06-17.

[14] Pympler Contributors. Pympler. `https://pythonhosted.org/Pympler/`. Accessed: 2024-06-17.

[15] Ken Schwaber and Jeff Sutherland. The scrum guide. `https://scrumguides.org/scrum-guide.html`, 2020. Accessed: 2024-06-17.