Randomized Algorithms (RA-MIRI): Assignment #3

Marc Díaz (marc.diaz.calderon@estudiantat.upc.edu)

December 2024

1 Code Solution

The code solution has been implemented in C++ and can be found in this GitHub repo - (https://github.com/MDCmarc/RA-Assigment3). The python notebook `data.ipynb` has been used to run the simulations and generate the LaTeX tables.

For this assignment, we were asked to implement different cardinality estimator algorithms. To make the code more structured, easier to test and to add new estimators, a base class called `CardinalityEstimator` was created. It implements basic functionalities that specific implementations may need. That is, some functions for bit manipulation and the hashing function. As for cardinality estimation, a good hashing function is required, we used the `xxhash` library [1] port to `C++ 17` implemented by Red Gavin [2] which can be found in GitHub. This base class contains the following:

- The constructor of this base class requires as *seed* which will be used in the hash function to make the results obtained replicable.
- Then the hashing function. In our implementation of the estimators, we used 32-bit hashing, however, it can be modified for 64-bit hashing. This function is just a wrapper for the call to `xxhash`.

```
uint32_t ComputeXXHash32(const string& input) const {
    return xxh::xxhash<32>(input.c_str(), input.size(), seed_);
}
```

- A function to extract the first b bits:

```
uint32_t ExtractHighBits( uint32_t value, const uint8_t pos ) {
    return value >> (32 - pos);
}
```

- A function to extract the lowest b bits:

```
uint32_t ExtractLowBits( uint32_t value, const uint8_t pos ) {
    return value & ((1UL << (32 - pos)) - 1);
}
```

- And lastly, a function to count the position of the first bit set to 1, given an initial position. For this function, we use the built-in compiler function "count leading zeros"[1].

```
uint8_t FindFirstSetBit( const uint32_t tail, const uint8_t start_pos ) {
    int adjusted_clz = std::max((int)__builtin_clz(tail) - static_cast<int>(start_pos), 0);
    return static_cast<uint8_t>(adjusted_clz + 1); // Range 1-32
}
```

[1] When compiling on Linux with the `g++` compiler, appending `l` for long values `__builtin_clzl` returned incorrect results, producing an erroneous range of 64 instead of 1-32. In contrast, on Windows using CLion, the explicit call to `__builtin_clzl` was necessary to obtain the correct range of 1-32.

1.1 Recordinality

The first cardinality estimator implemented is based on the paper by Helmi, Lumbroso, Martínez, *et al.* [3] and the slides of the course. This approach requires k registers to store k hash values and a counter. Our implementation is described in Algorithm 1

Algorithm 1: Estimate Cardinality (Recordinality)

Input : $file$ – File containing stream of words
k – Number of records to use
Output : Estimated cardinality of the set of words

$S \leftarrow \{\}$;
$r \leftarrow 0$;
while $r < k$ ***and read*** $word$ ***from*** $file$ **do**
  $y \leftarrow$ `ComputeXXHash32(`$word$`)`;
  if $y \notin S$ **then**
    $S \leftarrow S \cup \{y\}$;
    $r \leftarrow r + 1$;
  end
end
while ***read*** $word$ ***from*** $file$ **do**
  $y \leftarrow$ `ComputeXXHash32(`$word$`)`;
  if $y >$*min*(S) ***and*** $y \notin S$ **then**
    $S \leftarrow (S \setminus \{\min(S)\}) \cup \{y\}$;
    $r \leftarrow r + 1$;
  end
end
$E \leftarrow k \cdot (1 + \frac{1}{k})^{(r-k+1)} - 1$;
return E;

1.2 HyperLogLog

For the HyperLogLog estimator, we just require m registers of ≤ 8 bits. For the α_m we follow the values proposed in the original paper Flajolet, Fusy, Gandouet, *et al.* [4]. That is: $\alpha_{16} = 0.673$, $\alpha_{32} = 0.697$, $\alpha_{64} = 0.709$, and for $m \geq 128$, $\alpha_m = 0.7213/(1 + 1.079/m)$. After calculating the estimator, we apply the corrections proposed. The algorithm can be seen in 2.

Algorithm 2: Estimate Cardinality (HyperLogLog)

Input : $file$ – File containing stream of words
m – Number of registers to use
α_m – Correcting factor
Output : Estimated cardinality of the set of words

Initialize $R[0 \dots m-1] \leftarrow \{0\}$;
while ***read*** $word$ ***from*** $file$ **do**
  $y \leftarrow$ `ComputeXXHash32(`$word$`)`;
  $head \leftarrow$ `ExtractHighBits(`y, log_m`)`;
  $tail \leftarrow$ `ExtractLowBits(`y, log_m`)`;
  $R[head] \leftarrow \max(R[head],$ `FindFirstSetBit(`$tail$, log_m`))`;
end
$E \leftarrow \alpha_m m^2 \cdot \left(\sum_{j=0}^{m-1} 2^{-R[j]}\right)^{-1}$;
return `ApplyCorrection(`E`)`;

1.3 Probabilistic Counting

Lastly, as an optional estimator, we implemented the Probabilistic Counting, that, as it uses far more memory, may provide better results. This algorithm, shown in 3 uses m registers of 32 bits.

Algorithm 3: Estimate Cardinality (Probabilistic Counting)

Input : $file$ – File containing stream of words
m – Number of registers to use
ϕ – Correcting factor
Output : Estimated cardinality of the set of words

Initialize $bmap[m][32] \leftarrow \{0\}$;
while ***read*** $word$ ***from*** $file$ **do**
  $y \leftarrow$ `ComputeXXHash32(`$word$`)`;
  $head \leftarrow$ `ExtractHighBits(`y, log_m`)`;
  $tail \leftarrow$ `ExtractLowBits(`y, log_m`)`;
  $Zeros \leftarrow$ `FindFirstSetBit(`$tail$, log_m`)` -1;
  $bmap[head][Zeros] \leftarrow 1$;
end
$E \leftarrow m \cdot \phi \cdot 2^{\frac{1}{m} \sum_{j=0}^{m-1} \left(\arg\min_{0 \leq p \leq 31} \{bmap[j][p]=0\} \right)}$;
return E;

1.4 Generating Synthetic Data Streams

Lastly, we created a synthetic data stream generator that follows a Zipfian law of parameter α. The implementation can be seen in Algorithm 4. As the presented algorithms read each word from a file, this generator also writes the output into a file named `random_stream-n-N-`α`.txt`.

Algorithm 4: Synthetic Data Stream Generation

Input : n – Number of unique elements
N – Total size of the stream
α – Law parameter
Output : Synthetic stream of size N

$C_n \leftarrow (\sum_{i=1}^{n} i^{-\alpha})^{-1}$;
for $i \leftarrow 1$ ***to*** n **do**
  $P[i] \leftarrow C_n / i^{\alpha}$;
end
$C \leftarrow$ `PartialSum(`P`)`;
for $j \leftarrow 1$ ***to*** N **do**
  Get $x \in [0.0, 1.0]$ **at random**;
  $c \leftarrow$ `LowerBound(`C, x`)`;
  write c;
end

Where `PartialSum` just accumulates the probabilities and `LowerBound` returns to the first element greater than x.

For all random number generation, we used the `mt19937` random engine, which is based on the classic Mersenne Twister algorithm [5].

2 Experimentation

We initiated our experimentation by analysing the impact of the parameter values, k for Recordinality and m for HyperLogLog and Probabilistic Counting, on the average and standard error divided n of the cardinality estimation. To this end, we conducted 10,000 simulations using the *Dracula* dataset. The seed is just the simulation number $\in [1, 10000]$. The results are summarized in Table 1.

Parameter	**Recordinality**		**HyperLogLog**		**Probabilistic Counting**	
	Avg.	**Error**	**Avg.**	**Error**	**Avg.**	**Error**
16	9442	0.64	9413	0.28	9613	0.20
32	9420	0.40	9416	0.19	9526	0.14
64	9413	0.25	9405	0.13	9464	0.10
128	9432	0.16	9417	0.09	9448	0.07
256	9420	0.10	9418	0.06	9439	0.05
512	9423	0.06	9421	0.04	9430	0.03

Table 1: Comparison of average estimated cardinality (**Avg.**) and empirical standard error divided by n (**Error**) across three methods: Recordinality, HyperLogLog, and Probabilistic Counting. Results are based on 10,000 simulations using the dataset *Dracula*, with a true cardinality of $n = 9425$.

As can be seen, the empirical standard error decreases consistently as the parameter (and consequently the memory usage) increases. This aligns with theoretical expectations, as greater memory allows for a more precise representation of the data, thus reducing variability in the estimates. Also, we see how Probabilistic Counting (PCSA), which inherently uses more memory, achieves the lowest error across all parameter values. As the parameter grows larger, the differences in errors among the three methods become negligible. We can also this pattern in the same experiment conducted with datasets *A Midsummer Night's Dream* (Table 2) and *Don Quijote de la Mancha* (Table 3), which represent smaller and significantly larger cardinalities, respectively.

Parameter	**Recordinality**		**HyperLogLog**		**Probabilistic Counting**	
	Avg.	**Error**	**Avg.**	**Error**	**Avg.**	**Error**
16	3143	0.55	3137	0.28	3202	0.20
32	3135	0.34	3139	0.19	3170	0.14
64	3136	0.22	3136	0.13	3157	0.10
128	3137	0.13	3140	0.09	3146	0.07
256	3132	0.08	3136	0.06	3141	0.05
512	3135	0.04	3137	0.04	3154	0.03

Table 2: Comparison of average estimated cardinality (**Avg.**) and empirical standard error divided by n (**Error**) across three methods: Recordinality, HyperLogLog, and Probabilistic Counting. Results are based on 10,000 simulations using the dataset *A Midsummer Night's Dream*, with a true cardinality of $n = 3136$.

Parameter	Recordinality		HyperLogLog		Probabilistic Counting	
	Avg.	**Error**	**Avg.**	**Error**	**Avg.**	**Error**
16	22968	0.67	23043	0.27	23505	0.20
32	22985	0.43	23073	0.19	23282	0.14
64	23023	0.28	23057	0.13	23142	0.10
128	22992	0.18	23040	0.09	23096	0.07
256	23027	0.12	23050	0.06	23066	0.05
512	23045	0.07	23040	0.05	23056	0.03

Table 3: Comparison of average estimated cardinality (**Avg.**) and empirical standard error divided by n (**Error**) across three methods: Recordinality, HyperLogLog, and Probabilistic Counting. Results are based on 10,000 simulations using the dataset *Don Quijote de la Mancha*, with a true cardinality of $n = 23034$.

An important observation is the consistency in the standard error for HyperLogLog and Probabilistic Counting across all three datasets. This is because the error for these two estimators depends solely on the parameter m, rather than the true cardinality n. On the other hand, for Recordinality, the error is influenced by n, as reflected in the results. For instance, with *Quijote* ($n = 23034$), the initial error is 0.67, while for *A Midsummer Night's Dream* ($n = 3136$), the error begins at 0.55.

Until now, we have focused on comparing the effects of the parameters on the error, but we haven't addressed the accuracy of the estimators. Table 4 presents the results from 10,000 simulations of the average cardinality estimator calculated using each method across all datasets. In this table, we show both the true cardinality and the accuracy of the estimators.

Dataset	Card.	REC		HLL		PCSA	
		Avg.	**Acc.(%)**	**Avg.**	**Acc.(%)**	**Avg.**	**Acc.(%)**
mare-balena	**5670**	5656	99.7%	5662	99.9%	5696	99.5%
dracula	**9425**	9413	99.9%	9405	99.8%	9464	99.6%
quijote	**23034**	23023	100.0%	23057	99.9%	23142	99.5%
crusoe	**6245**	6246	100.0%	6234	99.8%	6274	99.5%
war-peace	**17475**	17516	99.8%	17464	99.9%	17554	99.5%
midsummer nights-dream	**3136**	3136	100.0%	3136	100.0%	3157	99.3%
valley-fear	**5829**	5854	99.6%	5827	100.0%	5858	99.5%
iliad	**8925**	8913	99.9%	8912	99.9%	8964	99.6%

Table 4: Comparison of average estimated cardinality (**Avg.**) and accuracy (**Acc.(%)**) across three methods: Recordinality, HyperLogLog, and Probabilistic Counting. Results are based on 10,000 simulations over all the datasets provided.

From the table, we can observe that all three methods, Recordinality (REC), HyperLogLog (HLL), and Probabilistic Counting (PCSA), demonstrate high accuracy across the datasets. The average

accuracy for each method is generally very close to 100%. Specifically, Recordinality and HyperLogLog maintain accuracy percentages well above 99.6%. On the other hand, although also performing very well, PCSA shows a slight drop in accuracy compared to the other two, which is interesting as it is the one that requires more memory.

Finally, for this algorithms, a random sequence of words is expected. However, for stories like the ones provided in the datasets, we can not say that they are fully random. For this reason, we conducted the same experiments but with datasets formed using the Synthetic data stream generator previously described. Table 5 shows the averages and errors for diferent parameter values for datasets of $N = 40000$ words where there are only $n = 5000$ distinct elements. The alpha value for the generation is $\alpha = 0.4$. Here also 10 000 simulation have been run with the seed being the simulation number.

Parameter	**Recordinality**		**HyperLogLog**		**Probabilistic Counting**	
	Avg.	**Error**	**Avg.**	**Error**	**Avg.**	**Error**
16	4979	0.57	5006	0.28	5097	0.20
32	4998	0.37	4998	0.19	5045	0.14
64	4994	0.23	4993	0.13	5019	0.10
128	5002	0.15	4993	0.09	5004	0.07
256	4994	0.09	4994	0.06	4996	0.05
512	4990	0.05	4989	0.04	4992	0.03

Table 5: Similar experiment for a random stream that contains 5000 distinct elements.

Also, the same experiment but for $N = 100000$ words can be seen in 6 where there are $n = 20000$ distinct elements. As both parameters are much bigger, 30 000 simulations have been run. The alpha value for the generation is $\alpha = 0.5$.

Parameter	**Recordinality**		**HyperLogLog**		**Probabilistic Counting**	
	Avg.	**Error**	**Avg.**	**Error**	**Avg.**	**Error**
16	19397	0.65	19377	0.27	19734	0.20
32	19476	0.42	19347	0.18	19524	0.14
64	19383	0.27	19364	0.13	19440	0.09
128	19370	0.17	19354	0.09	19390	0.07
256	19364	0.11	19348	0.06	19375	0.05
512	19350	0.07	19350	0.04	19360	0.03

Table 6: Similar experiment for a random stream that contains 20 000 distinct elements.

As we can see, for $n = 5000$ we still have a great accuracy of 99%. On the other hand, for 20 000 distinct elements, while the dataset *El Quijote de la Mancha* had 23034 words with an accuracy of 100%, we get an accuracy of $\leq 97\%$, which is still good, but we can see the deterioration.

References

[1] *Xxhash - extremely fast non-cryptographic hash algorithm.* [Online]. Available: https://xxhash.com/#other-languages.

[2] R. Gavin, *Redspah/xxhash_cpp: Port of the xxhash library to c++17.* [Online]. Available: https://github.com/RedSpah/xxhash_cpp.

[3] A. Helmi, J. Lumbroso, C. Martínez, and A. Viola, "Data streams as random permutations: The distinct element problem," *Discrete mathematics & theoretical computer science DMTCS*, vol. 0, Dec. 2012. DOI: 10.46298/dmtcs.3002.

[4] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," *Discrete Mathematics & Theoretical Computer Science*, vol. DMTCS Proceedings vol. AH,... Mar. 2012. DOI: 10.46298/dmtcs.3545.

[5] *Mersenne Twister - Wikipedia.* [Online]. Available: https://en.wikipedia.org/wiki/Mersenne_Twister.

[6] C. Martínez, *Randomized algorithms (ra-miri).* [Online]. Available: https://www.cs.upc.edu/~conrado/docencia/ra-miri.html.