PyResolveMetrics: Appendix

Andrei Olar (andrei.olar@ubbcluj.ro)
Laura Dioşan (laura.diosan@ubbcluj.ro)

Faculty of Mathematics and Computer Science
Babeş-Bolyai University
Cluj-Napoca

February 20, 2024

1 PyResolveMetrics Technology

The proposed library is implemented using the Python programming language (version 3.11) [Foundation, 2022]. It makes use of the Numpy library [Harris et al., 2020] to accelerate the expensive computations required by some of the implemented metrics. The library does not pose any additional hardware or software portability issues over the ones that are specific to Python and Numpy. In fact, the library design values portability and a high level of interoperability above other desirable traits such as performance and does so first and foremost by abiding to existing standards.

Python propagates standards throughout the community using protocols. Python protocols [Ivan Levkivskyi, 2017] provide a clean and effortless way to integrate new components, libraries and applications within the Python ecosystem. While other libraries [Virtanen et al., 2020] promote their own API by requiring users to adapt their data to the strictures of the library design, our library takes full advantage of the Python ecosystem by leveraging protocols, such as `Iterable` and `Hashable`.

All our library functions expect `Iterable` sequences as their input. In fact, the ground truth and ER result can be any `Iterable` data structure that complies with the ER model of choice. For the statistical model, the library accepts `Iterable[tuple]` input arguments, where the inner tuple must contain only two items. For the algebraic model, the library represents partitions as `Iterable[set]` instances. Since sets operate on `Hashable` items, any instance that implements the `Hashable` protocol is compatible with the library.

Adhering to the `Hashable` protocol has some less obvious advantages over imposing design choices onto the user to compute metrics:

- it works out of the box with basic types such as numbers or strings, but also with any `Hashable` types such as tuples or lists;
- if custom objects are used to represent the data that describes entities, the objects can be reused when computing metrics thus saving the memory that would

otherwise be allocated for labeling the data;

- the only adaptation required to use this library is implementing the well-known `Hashable` protocol, leading to a much lower implementation effort in the case when the data is not already compatible with the library

Choosing Python deserves special attention also because of a limitation it imposes on the design. Normally, a partition P over a set X is defined as a set of non-overlapping subsets containing all the elements from X, that is:

$$P(X) = \{C_1, C_2, \ldots, C_n | \bigcap_{i=1}^{n} C_i = \emptyset \wedge \bigcup_{i=1}^{n} C_i = X\} \tag{1}$$

The library implements metrics that compare partitions, but it requires the partition to be specified as an `Iterable[set]` instead of a set of sets.

The reason behind this choice is purely technological and lies with the choice of the Python core team to not make `set` instances `Hashable`. This entails an obligation on the part of the library to verify that, when partitions over a set are expected as the input arguments of a library function, that function will verify that the input arguments are partitions over the same set.

Using the `frozenset` data type (which is `Hashable`) instead of the `Iterable` protocol would decrease the interoperability of the library. The gains of choosing the `frozenset` in terms of performance and maintainability of the library would be marginal because checking that two iterable sequences of sets are partitions over the same set is a trivial and efficient algorithm that needs to be implemented only once.

While the library does focus on interoperability and portability, it does not disregard performance. The most notable design choice that increases the computational and memory efficiency of the library has to do with adopting Numpy as a primary dependency. Numpy helps by externalizing resource intensive calculations (especially those that are required for clustering metrics) to subroutines that are compiled to native machine code.

2 Example Usage: Plots for Metrics

This section contains the plots of the metrics computed after running the PPJoin algorithm on DG1 and DG2 for values of t in the interval $[0,1)$ at increments of 0.01.

The plots for the probabilistic metrics are available in Figure 1.

Pairwise metrics are depicted in Figure 2.

Cluster metrics can be viewed in Figure 3.

We showcase the algebraic indexes in Figure 4.

3 Performance Evaluation Data

This section contains figures and tables concerning the performance of the PyResolve-Metrics library.

Precision Recall F_1

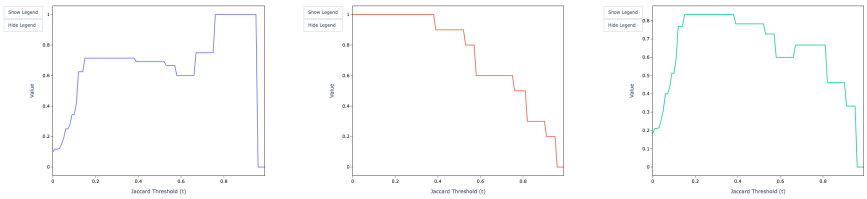

Figure 1: Fellegi-Sunter Metrics

Pairwise Precision Pairwise Recall Pairwise F Measure

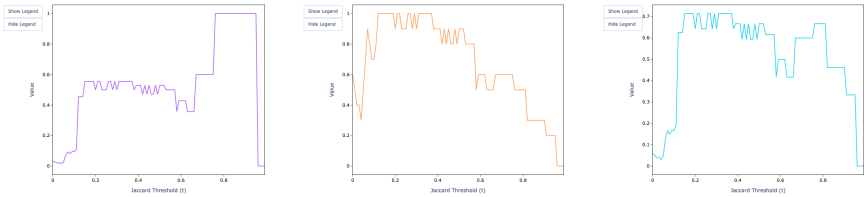

Figure 2: Pairwise Metrics

Cluster Precision Cluster Recall Cluster F Measure

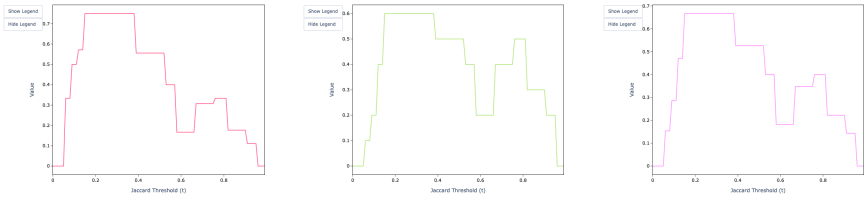

Figure 3: Cluster Metrics

Talburt-Wang Index Rand Index Adjusted Rand Index

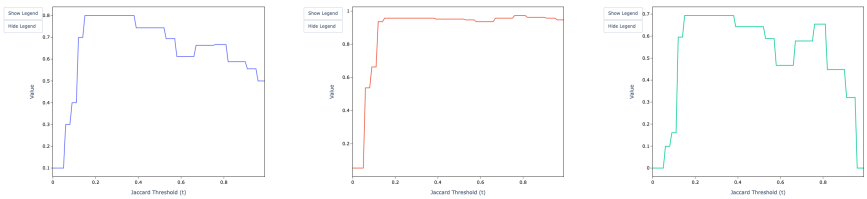

Figure 4: Algebraic Model Indexes

We use flamegraphs [Gregg, 2013] to represent the CPU usage patterns in the library and the `cProfile` package from the Python standard library to trace the execution of the functions that compute metrics. The flame graph from Figure 5 shows that the Fellegi-Sunter metrics take up less CPU than the validation of the input and output of data.

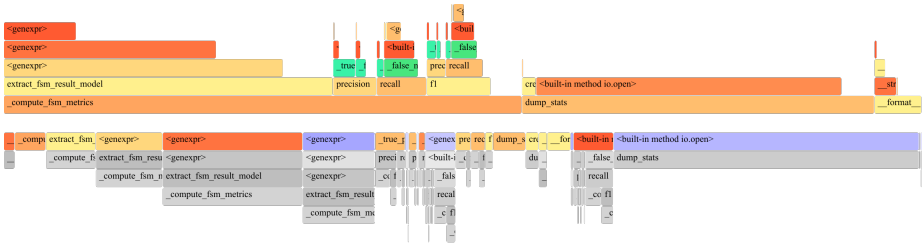

Figure 5: Fellegi-Sunter CPU Performance

The main consumers are shown in Table 1.

Metric	Cumulative Time	Number of Calls	Time per Call
`recall`	7.208e-06	2	3.604e-06
`f1`	5.291e-06	1	5.291e-06
`precision`	4.75e-06	2	2.375e-06

Table 1: Fellegi-Sunter Metrics

The best lesson we can learn here is that computing the statistical metrics is not resource-intensive at all.

The same cannot be said about the metrics that work with partitions over an input set. Table 2 presents how the algebraic metrics perform.

The pairwise and cluster metrics still perform well as indicated in Figure 6, but the algebraic model indexes are a lot more computationally intensive, dominating the flamegraph.

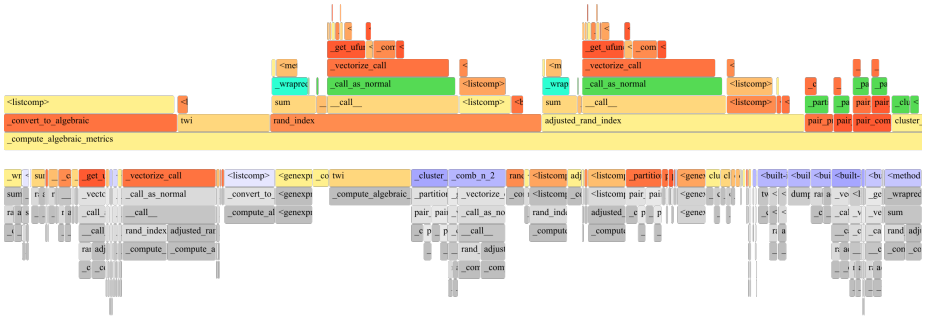

Figure 6: Algebraic CPU Performance

Metric	Cumulative Time	Number of Calls	Time per Call
rand_index	0.0004027	1	0.0004027
adjusted_rand_index	0.0001944	1	0.0001944
twi	9.146e-05	1	9.146e-05
cluster_comparison_measure	5.35e-05	1	5.35e-05
cluster_recall	5.154e-05	2	2.577e-05
cluster_precision	4.271e-05	2	2.135e-05
pair_precision	2.971e-05	2	1.485e-05
pair_recall	2.613e-05	2	1.306e-05
pair_comparison_measure	2.596e-05	1	2.596e-05

Table 2: Algebraic Metrics

Notice that computing the Rand Index and the Adjusted Rand Index are the most onerous operations by far.

We take note that the _convert_to_algebraic function is also a large CPU time consumer. This function checks whether the metrics are computed on two partitions over the same set. Since this function is executed nine times (once for computing each metric) we declare ourselves content to file a note to improve its performance at a later time.

References

[Foundation, 2022] Foundation, P. S. (2022). Python language reference, version 3.11. https://www.python.org. Online; Accessed: 18.11.2023.

[Gregg, 2013] Gregg, B. (2013). Blazing performance with flame graphs. In *Blazing Performance with Flame Graphs*, Washington, D.C. USENIX Association.

[Harris et al., 2020] Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. (2020). Array programming with numpy. *Nature*, 585(7825):357–362.

[Ivan Levkivskyi, 2017] Ivan Levkivskyi, Jukka Lehtosalo, L. L. (2017). Pep-544 - protocols: Structural subtyping (static duck typing). https://peps.python.org/pep-0544/. Online; Accessed: 26.11.2023.

[Virtanen et al., 2020] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.