1 Performance of MatChain

Blocking and the computation of similarity values are preliminary steps for many matching algorithms. Table 1 presents results for the matching algorithm AutoCal in combination with varying blocking procedures and similarity functions. The table displays the F_1-score (in percentage) and the total time t (in seconds) for six different pairs of datasets (FZ, DA, DS, AG, KG, and DG) that need to be matched.

Table 1: *Comparison of F_1-score (in percentage) and total time t (in seconds) for the original implementation of AutoCal (denoted as OntoMatch) and its efficient implementation and additional variants in MatChain.*

Library	Blocking	Similarity	Mach.		FZ	DA	DS	AG	KG	DG
OntoMatch	token	tok, emb	A100	F_1	80.3	94.9	81.4	61.6	82.9	87.7
				t	32	464	548	113	31	46
MatChain	token	tok, emb	A100	F_1	80.3	94.9	81.4	61.6	83.1	87.6
				t	3.9	15.5	80.2	7.2	6.3	5.4
MatChain	sdt + shg	shg	CPU	F_1	87.6	97.4	88.0	63.3	83.9	85.9
				t	0.7	2.3	21.4	1.0	0.8	1.1
MatChain	sdt + shg	shg, fuz	CPU	F_1	79.4	97.9	89.6	64.5	84.1	85.7
				t	0.8	2.3	22.3	1.2	0.9	1.2
MatChain	faiss + emb	shg	A100	F_1	81.2	96.8	89.3	63.1	84.1	87.1
				t	2.6	9.8	88.1	5.1	5.8	4.7

All experiments were conducted on Google's Colab, utilizing two distinct machines:

- CPU: Equipped with a 2.2 GHz Xeon CPU (1 core), 13.6 GB RAM, and no GPU.
- A100: Equipped with a 2.2 GHz Xeon CPU (6 cores), 89.6 GB RAM, and NVIDIA A100 with 40 GB.

These machines come pre-installed with Ubuntu Linux, Python 3.10, popular machine learning libraries, and have been updated with libraries required by MatChain.

MatChain offers several methods for representing and comparing two strings:

- tok: The method segments each string into words and represents it with a sparse vector of TFIDF weights for its constituent words.
- shg: For each string, the method computes shingles and a sparse vector with TFIDF weights for these shingles. Shingles are character-level n-grams. Example: For $n = 3$ and the string *matchain*, we obtain the shingles mat, atc, tch, cha, hai and ain.

- emb: For each string, the library SentenceTransformer[1] generates a dense embedding vector.
- fuz: The method uses the library thefuzz[2] to derive similarity values between two strings.

In the first three methods, similarity values are determined as cosine similarities between the respective vector representations. By contrast, similarity values for numerical values are computed from their absolute or relative distances.

Next, let's delve into the findings for various configurations in Table 1. The initial line presents the results for the original implementation of AutoCal[3]. The original implementation, denoted as OntoMatch[4], utilizes token blocking which means that two records become a candidate pair if they share at least one token (in this context, a word). Moreover, OntoMatch combines two similarity functions (tok and emb) for string comparison.

MatChain heavily relies on libraries such as NumPy, Pandas, and SciPy, all of which support vectorized data handling. MatChain incorporates an efficient implementation of AutoCal and integrates additional libraries to ensure fast blocking and string comparisons. The second line in Table 1 illustrates MatChain's results for AutoCal in conjunction with token blocking and the same similarity functions employed in the original implementation. The F_1-scores remain nearly unchanged, but the total times are significantly reduced – by a factor of 5 to 30.

However, generating the embedding vectors for string comparison becomes a limiting factor; its share of the total time in line 2 becomes substantial, even when optimizing the batch size for the powerful GPU A100. To address this, we employ shingle vectors instead of embedding vectors for string comparison. Moreover, we also replace token blocking with a blocking procedure on shingle vectors. Consequently, blocking is transformed into a nearest neighbour problem: for each vector representing a string from one dataset, find the nearest - let's say 10 - vectors representing strings from the other dataset. Brute force search is typically too slow. In the third line in Table 1, we chose the library sparse_dot_topn[5] for approximate nearest neighbour search. As evident, the entire code now runs at least four times faster, even without any GPU. Compared to the original implementation in line 1, the total times are reduced by a factor of 25 to 200. Additionally, the F_1-scores for FZ, DA and DS increase significantly. However, the improved F_1-score for FZ should be regarded rather as an "outlier" in the positive direction here.

Lines 4 and 5 of Table 1 vary the configuration in line 3. For line 4, we combine cosine similarity for shingle vectors with similarity values computed by the popular library thefuzz (formerly called fuzzywuzzy). The F_1-scores improve slightly for four dataset pairs, while the total time increases only a little bit. In contrast, line 5 uses embedding vectors for blocking and exclusively leverages shingle vectors for string comparison. In this

[1] https://github.com/UKPLab/sentence-transformers

[2] https://github.com/seatgeek/thefuzz

[3] https://www.sciencedirect.com/science/article/pii/S1570826824000015

[4] https://github.com/cambridge-cares/TheWorldAvatar/tree/main/Agents/OntoMatchAgent

[5] https://github.com/ing-bank/sparse_dot_topn

case, we deploy the GPU-accelerated version of Faiss[6] for approximate nearest neighbour search within the embedding vectors. This setup results in marginal F_1-score improvements for three dataset pairs but embedding vector generation becomes a bottleneck again.

[6] https://github.com/facebookresearch/faiss