Data Science and Engineering (2021) 6:20–38
https://doi.org/10.1007/s41019-020-00146-w

Blocking Techniques for Entity Linkage: A Semantics-Based Approach

Fabio Azzalini[1,2] · **Songle Jin**[1] · **Marco Renzi**[1] · **Letizia Tanca**[1]

Received: 27 June 2020 / Revised: 26 September 2020 / Accepted: 5 October 2020 / Published online: 3 November 2020
© The Author(s) 2020

Abstract
Nowadays, data integration must often manage noisy data, also containing attribute values written in natural language such as product descriptions or book reviews. In the data integration process, Entity Linkage has the role of identifying records that contain information referring to the same object. Modern Entity Linkage methods, in order to reduce the dimension of the problem, partition the initial search space into "blocks" of records that can be considered similar according to some metrics, comparing then only the records belonging to the same block and thus greatly reducing the overall complexity of the algorithm. In this paper, we propose two automatic blocking strategies that, differently from the traditional methods, aim at capturing the semantic properties of data by means of recent deep learning frameworks. Both methods, in a first phase, exploit recent research on tuple and sentence embeddings to transform the database records into real-valued vectors; in a second phase, to arrange the tuples inside the blocks, one of them adopts approximate nearest neighbourhood algorithms, while the other one uses dimensionality reduction techniques combined with clustering algorithms. We train our blocking models on an external, independent corpus, and then, we directly apply them to new datasets in an unsupervised fashion. Our choice is motivated by the fact that, in most data integration scenarios, no training data are actually available. We tested our systems on six popular datasets and compared their performances against five traditional blocking algorithms. The test results demonstrated that our deep-learning-based blocking solutions outperform standard blocking algorithms, especially on textual and noisy data.

Keywords Data integration · Entity linkage · Blocking · Deep learning

1 Introduction

The integration of data coming from different sources is today of paramount importance: companies, hospitals, government agencies, banks and many other actors, in order to carry out their everyday activities, need to merge several datasets, e.g. customers databases or patient and pathology records.

✉ Fabio Azzalini
fabio.azzalini@polimi.it

Songle Jin
songle.jin@mail.polimi.it

Marco Renzi
marco.renzi@mail.polimi.it

Letizia Tanca
letizia.tanca@polimi.it
http://tanca.faculty.polimi.it/

1 Politecnico di Milano, Milan, Italy

2 Center for Analysis Decisions and Society, Human Technopole, Milan, Italy

Integrating data in these scenarios may be relatively simple, especially when the data sources have clean and standard attributes, but with the increased use of internet-based services like e-commerce, web sites for comparing products or online libraries, data integration is becoming more challenging. These services deal with data that is typically noisy and that contains attribute values written in natural language, such as product descriptions or book reviews. Indeed, integrating such data is hard because of the difficulties in managing dirty values and in extracting semantics out of long textual values written in natural language. In particular, a very challenging stage of the integration lies in identifying which records from the several source datasets represent the same concept, or *the same entity*, i.e. the activity known as *Entity Linkage*, or *Entity Resolution.* In the past, this task has been addressed by applying pairwise matching algorithms over the Cartesian product of the records provided by two input sources. However, the current disruptive growth in dataset sizes makes the problem intractable, since, when the number and the sizes of the datasets

Table 1 Database A

ID	Name	Surname	Gender
a1	John	Smith	Man
a2	Robert	Sandy	Man
a3	Christine	Faulkner	Woman

Table 2 Database B

ID	Name	Surname	Gender
b1	Robertt	Sandy	Male
b2	kristine	Fawkner	Female
b3	Johnny	Smith	Male

to be integrated increase, the memory space, and the time needed to apply this approach become rapidly prohibitive; already in the simple case of the integration of two databases the number of comparisons grows quadratically with respect to the data sets sizes. Modern Entity Resolution methods, in order to reduce the dimension of the problem, partition the initial search space into *blocks* within which the comparisons are performed, thus greatly reducing the number of matches and the overall complexity of the algorithm. Blocking methods apply functions and algorithms to filter out the tuple pairs that are clearly not matching from the potential comparisons. Traditional blocking schemes use hand-tuned functions to generate the blocks and place the tuples inside them accordingly. In other words, all the records are passed through a *blocking function(s)*, and each tuple is assigned to a bucket based on its *blocking key value* (BKV). One can clearly understand that the quality of the entity linkage (and thus ultimately of the entire data integration activity) is profoundly influenced by the blocking phase, both in efficiency, as the blocking phase should grant better time and memory consumptions with respect to a naïve Cartesian product approach, and in effectiveness, since this phase should find as many true matching pairs as possible.

To better understand the traditional blocking process, let us consider the sample datasets reported in Tables 1 and 2.

In this case, a traditional blocking method using as blocking function $BKV = f(Gender) = Gender$ would group together only the tuples belonging to the same source database, since the two datasets A and B use two different sets of values for expressing the attribute *Gender*. This example shows a big problem that affects traditional blocking schemes: they ignore the semantics of the attribute values and leverage only the lexicon. Other shortcomings of traditional blocking methods include: (i) the sensitivity to morphological variations and data quality issues, (ii) the fact that designing appropriate blocking functions is time-consuming and cumbersome and (iii) the need to design dedicated blocking strategies for each new dataset. With the ever-increasing need to process, analyze and integrate large datasets that contain noisy and long textual values (especially if in natural language), new solutions that exploit also semantic information can be of great help.

The two blocking systems we propose are based on the fact that, in case the datasets to be merged contain noisy values or texts written in natural language, a method that leverages only the morphological aspects of the attributes is not enough and not effective; therefore, we aim to capture word and sentence meanings. Both our algorithms consist of two key phases: first we transform the records of the datasets to be integrated into real-valued vectors, exploiting techniques such as word and sentence embeddings[1] [10, 24], and then, to generate the blocks, in one case we apply to these vectors an innovative blocking technique, based on locality-sensitive hashing (LSH) [33], while in the second case we first project the numeric representation of the tuples onto a lower-dimensional space and then exploit conventional clustering algorithms to create the blocks.

In this work, we want to keep the entire blocking phase within an unsupervised setting, thus without training the neural networks responsible for building the embeddings on the actual datasets to be merged. This choice stems from the evidence that a labelled training set generated on the actual data to be merged is typically difficult to find, and thus, an unsupervised approach makes the task more feasible. To achieve this scope, we exploit pre-trained word embeddings such as *GloVe* [28] and *fastText* [23] and train the recurrent neural network (RNN) [14] responsible for composing the sentence embeddings on a separate available big dataset: the *SNLI corpus* [6].

Our work builds upon research published in DeepER [12], where the authors were among the first to investigate the application of tuple embeddings along the entire entity linkage pipeline. Our system, however, is focused on the blocking phase and adopts a different paradigm for this specific task. More specifically, our contributions are:

– Differently from DeepER [12], we apply an unsupervised approach to blocking: the models are trained on an external corpus and then used on new datasets directly, without the need of being retrained on the specific datasets to be integrated. This choice has the great advantage of not needing any labelled dataset, or in the case it is not available, to ask the user to manually label the training data.
– Differently from traditional blocking algorithms, our systems implement an automated approach to blocking: our

[1] These are natural language processing (NLP) well-known techniques that encode semantic information in the numeric representation they produce.

embedding models, after being trained on the external corpus, simply require to scan the input tuples to produce the blocks. Our solutions, consequently, ease the difficult and cumbersome definition of the blocking functions that are used in traditional blocking schemes and make the systems more portable across the datasets.

- We present two unsupervised blocking algorithms able of exploiting the semantics of the records on which they operate.
- Combining well-tested and widely adopted data mining, machine learning and deep learning methods, we devise two automated unsupervised blocking techniques that consistently outperform five of the most used blocking methods [7].
- We compare the performances of our systems with some of the most consolidated blocking paradigms on six popular real-world datasets, commonly used to evaluate blocking schemes. Additionally, we provide a wide range of experimental results to study empirically the differences of the architectural choices of our systems.

2 State of the Art

We now describe the traditional blocking algorithms that can be considered the standard and most frequently adopted approaches [7, 26].

Standard blocking is the easiest of the traditional algorithms. Once the blocking function is defined, this method positions inside the same bucket all the records with equal BKV. The peculiarity of this method is that each tuple is inserted into one bucket only, while the other traditional techniques can potentially put a record into several blocks.

This method presents two main drawbacks: (i) it is not robust w.r.t. noisy values, (ii) it is not suitable when the distribution of BKVs is very skewed as the buckets sizes would be too diverse.

Sorted neighbourhood blocking [15] uses the BKVs generated by the blocking function(s) to sort the tuples in the databases. Once the datasets are sorted, a sliding window with a fixed size w ($w > 1$) is passed over the source records and the blocks are generated by the tuples that fall in the window step by step. This method presents two main problems: (i) dataset sorting is sensitive to the prefixes of the BKVs; for instance, if the sorting function corresponds to the *name* attribute of the records, then very similar values such as 'carl' and 'karl' that may refer to the same person but with a typographical error will end up in distant positions in the sorting scheme, (ii) choosing a proper value for w is not trivial.

Q-gram blocking is specifically designed to face the errors and variations in attribute values [3]. To account for them, the method generates variations of the BKVs. Each record is then inserted into both its original bucket (the original BKV) and in all the derived buckets (variations on the original BKV). To build the new BKVs, the algorithm generates all the q-grams (substring of length q) of the original BKVs. To generate the actual new BKVs, a recursive approach is used: from the list l (of size k) of the q-grams of the original BKV, new lists are generated by iteratively removing from l one q-gram at a time. These new lists will be of size $k - 1$. This procedure is repeated on each of the newly generated lists, and it stops when the last built lists are shrunk to a minimum length s defined as $s = max(1, \lfloor k * t \rfloor)$. In this formula, k is the size of q-gram list of the original BKV, t ($0 \leq t < 1$) is a user-defined parameter to decide the minimum length of the newly generated q-gram sub-lists. Finally from each of the generated sub-lists, a new BKV is obtained by the concatenation of its q-gram elements. This algorithm, even if it can capture variations in attribute values, is very expensive in both time and memory consumption, especially when the original BKVs are long.

Suffix Blocking [2] is based on ideas similar to the q-gram blocking. Here, however, instead of considering all the q-grams, only some suffixes of the original values are used. To build the new blocks, two parameters are needed: l_{min} which corresponds to the minimum length of the suffix substrings that are generated, and b_{max}, used to discard blocks whose number of records exceed this threshold.

Canopy cluster blocking [22] adopts a clustering approach to blocking: it groups together the records based on how similar their BKVs are. This blocking scheme is specified by two elements: the similarity function used to express the closeness of the BKVs and the actual algorithm used to group the records. The most commonly used similarity function is the *Jaccard* measure, which basically indicates the percentage of q-grams two BKVs have in common. Relatively to the clustering algorithm, an iterative clustering procedure is used.

All algorithms presented can be considered traditional blocking methods, and they all suffer, to different extent, from the problems presented in the introduction.

3 Methodology

Our blocking schemes are logically composed of two phases: first, an embedding module is responsible for transforming the records into real-valued vectors, and second, a blocking module actually produces the buckets from these numerical vectors (Fig. 1).

The embedding architectures we present are very similar to the ones presented in [12]; however, there are important variations in the implementation of the embedding layer, the main difference being that our RNNs are trained on an external corpus, the SNLI corpus, and not on the actual

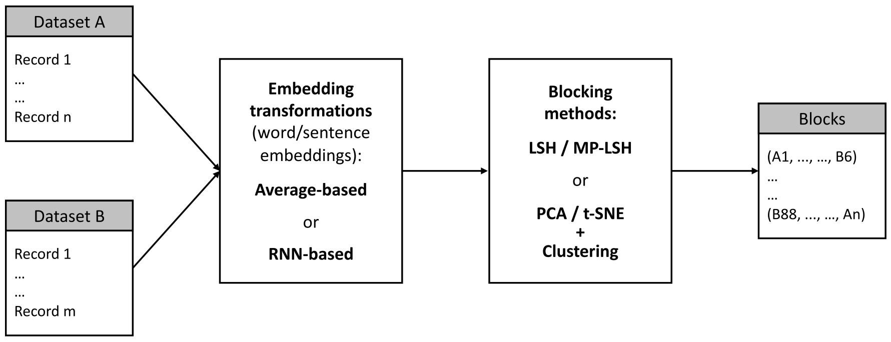

Fig. 1 Structure of our blocking systems

datasets to be merged. This unsupervised approach is motivated by the following considerations. First, as noticed, in many applications no training data are available, and this choice makes our technique more realistic. Secondly, this paradigm ensures fairer evaluation tests because the traditional blocking methods to which we compare ours are not tuned on the datasets.

In our work, we also want to study from an empirical point of view the differences in applying word and character-level embedding models. In particular, fastText is used as word embedding in the paper [25] but in that work the blocking phase is not faced. In this work, we decide to include both fastText and GloVe. The authors of [12] consider only the latter instead.

Finally, we decide to implement several RNNs with different sizes because we want to see whether different levels of complexity lead to noticeable variations in the performances.

4 Embedding Architectures

We now illustrate the two solutions that we adopt to implement the embedding layer. This layer receives as input the records of the datasets to be merged and outputs their vectorial representation.

In the rest of the paper, we use the following notation: given a tuple t with n attributes $\{A_i\}_{i=1,\dots,n}$, the symbol $\mathbf{v}(t[A_k])$ represents the numerical representation of the attribute value $t[A_k]$, while the numerical value of the entire tuple is $\mathbf{v}(t)$; given a generic word w, its embedding representation will be indicated with $\mathbf{v}(w)$.

At their core, both the embedding architectures we propose make use of pre-trained word embeddings, i.e. fastText or GloVe: each word w of each attribute value $t[A_k]$ is transformed into a real-valued vector $\mathbf{v}(w)$. The fastText model we use is *crawl-300d-2M-subword* [23] where each word is represented as a 300-dimensional vector and the corpus on which it was trained is *Common Crawl*[2]. Regarding GloVe, we use the model *glove.840B.300d* [28] and again each word is represented as a 300-dimensional vector and the training corpus was still *Common Crawl*. Once the single words are transformed, what differentiates the two architectures concerns how the single word embeddings are composed first to represent the attribute value (i.e. how to build $\mathbf{v}(t[A_k])$) and eventually the entire record (how to produce $\mathbf{v}(t)$).

4.1 Average-Based Architecture

With this approach, also adopted in [12], when a tuple t with n attributes arrives at the embedding layer, each of its attributes $\{A_k\}_{k=1,\dots,n}$ is treated independently from the others. First, we use a tokenizer to split the attribute value $t[A_k]$ into its component words $\{w_i\}_{i=1,\dots,p}$. In our implementation, we use the standard *NLTK* tokenizer available in the *NLTK* module[3]. In the case one attribute is empty (i.e. there is a missing value) we insert the placeholder word "*unk*". Then, each of these words is mapped to a 300-dimensional vector by applying the pre-trained word embedding model. The attribute vector representation $\mathbf{v}(t[A_k])$ is given by simply averaging the vectors of the component words. The tuple vector $\mathbf{v}(t)$ is finally obtained by the concatenation of all the n attribute vectors, $(\mathbf{v}(t[A_1]),\dots,\mathbf{v}(t[A_n]))$. Speaking of tuple size, $|\mathbf{v}(t)| = d * n$ where d is the attribute vector dimension (in our case it is $d = 300$) and n is the number of attributes.

This composition method has the advantage of being easy to implement, but it can lead to huge vector sizes when the data sources have a long list of attributes.

4.2 RNN-Based Architecture

This second approach to embedding is more refined and its use is motivated by some considerations.

2 https://commoncrawl.org/the-data/.

3 https://www.nltk.org/_modules/nltk/tokenize.html.

Table 3 Tuple t_A

ID	Name	Surname	Address	Postcode
a1	John	Smith	42 Miller St	2602

Table 4 Tuple t_B

ID	Name	Surname	Address	Postcode
b3	Johnny	Smith	42 Miller street 2602	

First, representing an attribute value by simply averaging its component words embeddings—as the previous solution—means ignoring the order of the words. This is generally not an issue when there are atomic values or few words in an attribute, but when long textual elements are present it becomes more important. Indeed, the order of words helps when there are attributes that encapsulate multiword content such as descriptions or specifications of products. The RNN architecture embeds words order naturally in its implementation.

Second, the previous embedding approach cannot capture semantic relationships among attributes: to build the final tuple vector $\mathbf{v}(t)$, it employs a simple concatenation of the component attribute vectors. In some cases, however, being able to link the semantics of nearby attributes can be useful to better capture the overall representation of an entity. Consider the two records representing the same person reported in Tables 3 and 4.

Information about the postcode value "2602" is formally placed in different attributes, and the second record has a missing value. By adopting a simple averaging approach, vectors $\mathbf{v}(t_A[Address])$ and $\mathbf{v}(t_B[Address])$ (as well as $\mathbf{v}(t_A[Postcode])$ and $\mathbf{v}(t_B[Postcode])$) will likely be very different and by later using the concatenation of such vectors the relationship between postcode and address would not be exploited. The RNN architecture as will be shortly shown, instead, considers the attributes as a sequence and so if adjacent attributes have related meanings it can encapsulate better the dependencies.

Third, another shortcoming of the average-based solution that motivates the RNN-based approach concerns the embedding sizes. As said, when using an average approach the tuple embeddings are proportional to the number of attributes n of the sources, and this can easily lead to very big embedding vectors. As we will analyze more in detail later, working on high-dimensional data requires caution and some algorithms do not perform well in such setting [13, 33]. With the RNN-based architecture, the size of the resulting tuple vectors is fixed a priori and is independent of the number of attributes.

As suggested in [12], we use both uni- and bidirectional recurrent neural networks (RNNs) with long short-term memory (LSTM) cells [16]. In the following discussion, we refer to this family of models with the term RNN-LSTM architectures. However, to actually implement these nets we do not follow the approach described in the paper [12] because, as anticipated, the authors train the models directly on the datasets to be integrated in a supervised fashion.

To devise our models, we are instead inspired by the recent studies made by the Facebook AI research team in their paper [10] about sentence embedding models. Among the several solutions they investigate, they propose a model, named *Infersent*, a bi-LSTM net which transforms sentences written in natural language into their corresponding numerical representation. The authors show the strong results of this model on several tasks and in particular for semantic textual similarity applied in unsupervised settings. Our uni- and bi-LSTM nets are consequently implemented with the same architectures described in that paper for analogous models, and partially readapting the code the authors kindly release at [9]. In our work, however, we extend the analysis of these models with both fastText and GloVe and with a greater set of network sizes.

This is interesting because *fastText* and *GloVe* handle words not present in their vocabulary in different ways. *GloVe* replaces the missing words with the default value "*unk*", while *fastText*, exploiting sub-word embeddings, is able to assign a numeric representation also to words not present in its vocabulary; for this reason, *fastText* is usually regarded as a *character level embedding* method.

4.2.1 Preliminary Information on RNN

We now provide the ideas and the description of the two RRN models we use, to understand how they work, and then, we explain how they are applied to our task.

In Fig. 2, the architecture of a uni-LSTM net is presented.

In the picture, w_i represents the word embedding obtained by applying either fastText or GloVe on the corresponding source word. The $\overrightarrow{h_i}$ rectangle is the hidden state of the LSTM layer when input is word i. This layer comprises n LSTM cells that implement the memory of the net. It is this memory that captures the dependencies among the processed vectors.

As can be seen from the picture, when a new word vector $\overrightarrow{w_{n+1}}$ is presented at the net also the $\overrightarrow{h_n}$ internal state is considered to produce the new hidden state $\overrightarrow{h_{n+1}}$. This means that the net keeps track of the input vectors that have passed previously through its layers and past computations influence the present evaluation. The final representation of the sentence $\vec{u}$ is given by the last hidden state of the net $\overrightarrow{h_m}$:

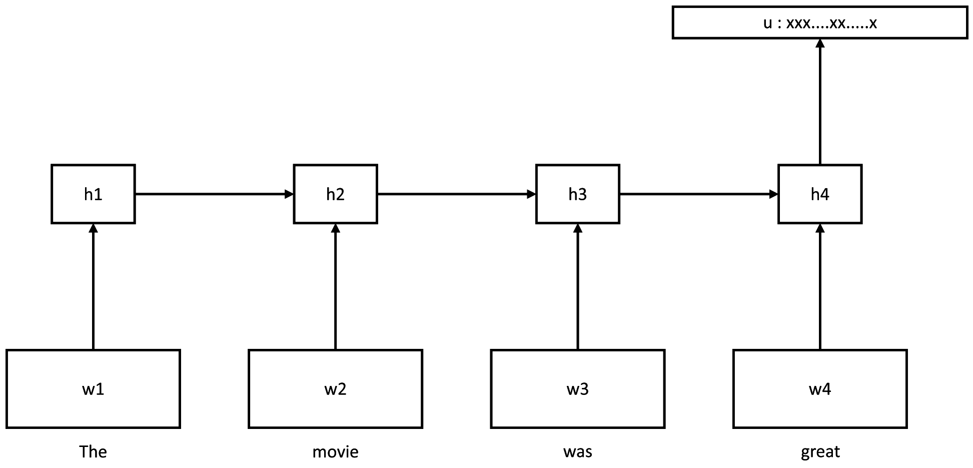

Fig. 2 uni-LSTM network

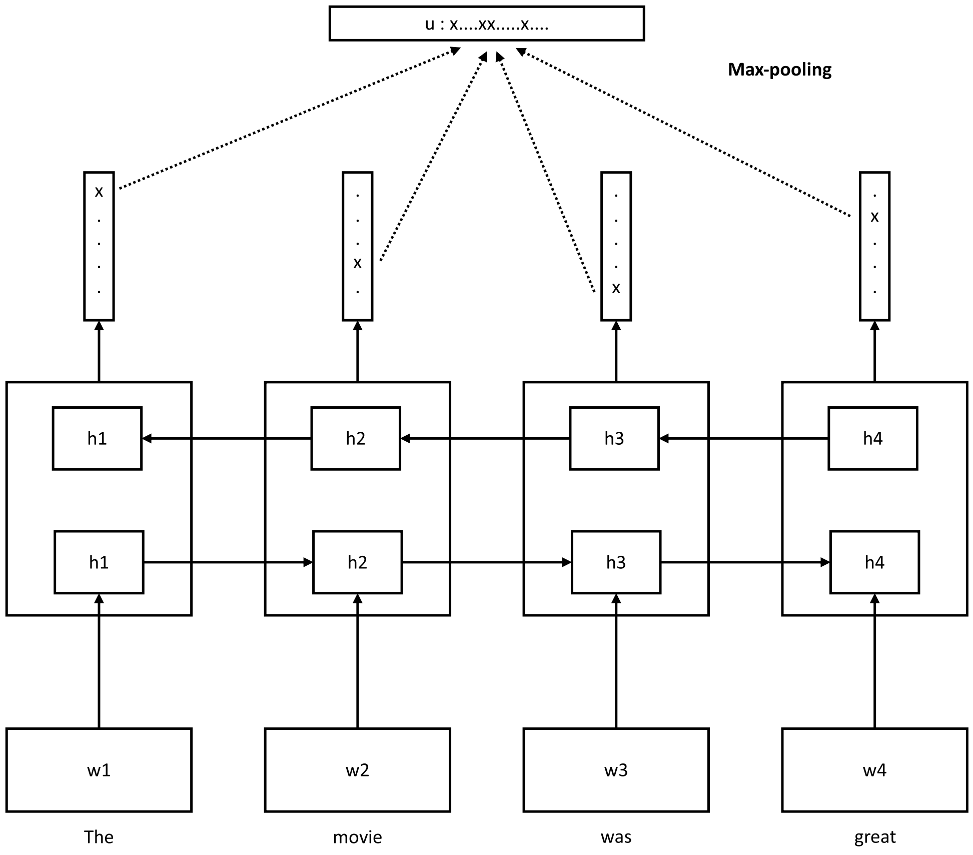

Fig. 3 bi-LSTM network

$$\overrightarrow{h_m} = \overrightarrow{LSTM_m}(w_1, \dots, w_m) \quad (1)$$

Let us now analyze the second RNN-based reference architecture, the bi-LSTM net.

The bi-LSTM net (Fig. 3) is composed of two uni-LSTM nets: a *forward* and a *backward* uni-LSTM net. The former scans the source sentence in direct order (from left to right), whereas the latter in reverse order (from right to left). The forward NN generates the hidden state vectors $\overrightarrow{h_i}$ and the backward produces $\overleftarrow{h_i}$. At each step, the overall hidden state $\underline{h_i}$ of the bi-LSTM net is given by the concatenation of $\overrightarrow{h_i}$ and $\overleftarrow{h_i}$. Scanning m words thus produces a sequence of m states $\{h_i\}|_{i=1,\dots,m}$.

Finally, to obtain the sentence embedding $\vec{u}$ a max-pooling function is applied across $\{h_i\}|_{i=1,\dots,m}$: simply the maximum value over each dimension, across the all steps i, is selected.

$$\begin{aligned}\overrightarrow{h_m} &= \overrightarrow{LSTM_m}(w_1, \dots, w_m)\\ \overleftarrow{h_m} &= \overleftarrow{LSTM_m}(w_1, \dots, w_m) \qquad (2)\\ h_m &= [\overrightarrow{h_m}, \overleftarrow{h_m}]\end{aligned}$$

When using the bi-LSTM net, the size of the resulting embedding vectors is exactly the double of the one produced by the uni-LSTM because of the double scan of the source sentence.

4.2.2 Sentence Embeddings Composition Methods

Given these premises, it is now possible to describe how these models actually perform our embedding task. When a tuple t with n attributes arrives at the embedding layer, each attribute value $t[A_k]$ undergoes the tokenizer which splits it into its component words $\{w_i\}_{i=1,\dots,p}$. Each of these words is mapped to a 300-dimensional vector by applying the pre-trained word embedding model (fastText or GloVe), which yields the words embeddings $\{\mathbf{v}(w_i)\}_{i=1,\dots,p}$. After each word vector $\mathbf{v}(w)$ is generated, it is given as input to the RNN-LSTM, which processes it and produces an internal hidden state. At the end of the current attribute value $t[A_k]$, the process is repeated from the first word of the adjacent attribute A_{k+1} seamlessly. This continuous feeding of the net ensures the first two appealing "properties" discussed above: since the tuple is processed as a single long sentence, the process guarantees to encode both the words order and the possible semantic relationships among adjacent attributes. The final tuple embedding vector $\mathbf{v}(t)$ is obtained after the scan and process of the last word of the last attribute value, and its computation follows the procedure previously described for the uni-LSTM or bi-LSTM networks: in the uni-LSTM case, the tuple vector coincides with the last hidden state $\overrightarrow{h_m}$, while when using a bi-LSTM net it is the output of the max pooling function applied over the dimensions across all internal states. For a deeper understanding of how uni- and bidirectional LSTM nets work, we refer the reader to [10] and [12].

Another important advantage with RNN-LSTM architecture is that it is possible to control and fix the size of the embedding tuples a priori. This holds because the dimensionality of the vectors is determined by the number of LSTM cells we put in the RNN hidden layer, and this is a design decision one takes before actually applying the model. In our work, we implement RNNs with 300, 1024 and 2048 LSTM cells, and so in the tests we will evaluate the results with tuple embeddings whose sizes are 300, 1024, 2048 when applying the uni-LSTM nets and 600, 2048 and 4096 when using bi-LSTMs.

In order to use these RNN-LSTM nets, a prior training phase is necessary to estimate their parameters. The RNN-LSTM nets of our embedding layer are trained on the labelled external *SNLI* corpus [6] before being applied to the test datasets. This dataset is one of the largest labelled sources specifically designed to force semantics understanding and thus a good candidate for our solution.

5 Blocking Algorithms

Now that we have a numerical representation we need to detect similar tuples and put them in the same block. This is the goal of the second step of our blocking systems, where, exploiting two different methods, we actually group records into the buckets. Since the first step transforms the records into vectors with the appealing property that tuples related in the meaning are projected onto vectors that are close to each other, the algorithms we use exploit this geometrical proximity to block them. We now present two blocking methods that take as input the vectors constructed in the previous step and produce groups of semantically similar records.

The first one is based on locality-sensitive hashing (LSH) [33] and multiprobe LSH [20], two popular techniques, that belong to the family of algorithms used in approximate nearest-neighbour (ANN) search problems.

The second blocking method we present first exploits dimensionality reduction techniques such as principal component analysis (PCA) [17] and t-distributed stochastic neighbour embedding (t-SNE) [21] to create a low-dimensional searching space and then employs standard and well-adopted clustering algorithms to arrange records into blocks.

5.1 Approximate Nearest-Neighbour-Based Blocking

ANN search is the problem of finding similar vectors in a n-dimensional space. ANN techniques are typically preferred over traditional *exact* nearest-neighbour algorithms when the dimensionality of the vectors to be compared is high like in our case and therefore poses efficiency problems [33].

5.1.1 Preliminary Information on LSH Methods

Locality-Sensitive Hashing: this algorithm aims at finding a hash function h that satisfies two requirements: *(i)* items that are (semantically) close should have the same hash value $h(x)$ with "high" probability P_1 and *(ii)* points that are distant should have the same value $h(x)$ with "low" probability P_2. The ultimate goal of the h function is indeed to group together (with the same hash value) those items that are close, or equivalently, similar. A popular paradigm is to choose as h a probabilistic binary function, for example $h(x) \in \{-1, +1\}$. We then generate K distinct hash functions $h_1, h_2, \dots, h_K$ with $h_i \in \mathcal{H}$ and apply these functions in order, to each tuple x to be analyzed. Each tuple x is consequently assigned to the sequence of the outputs $g(x) = \{h_1(x), h_2(x), \dots, h_K(x)\}$. This ordered sequence $g(x)$ is named the *hash code* of tuple x, and it is used to identify that record.

Since for large values of K the likelihood that similar records get the same K-dimensional hash code is reduced, typically several hash codes are computed for each tuple. To do so, one builds L hashes $g_1(x), g_2(x), \dots, g_L(x)$, where each $g_i(x)$ is a K-sized vector obtained applying the ith sequence of probabilistic binary functions on tuple x. The set of hash codes $\{g_i(y)\}|_{y=1,\dots,N}$ applied to entire set of N vectors is generally named *hash table.*

Multiprobe LSH: in the standard LSH implementation described above, one considers as similar all the records that fall in the same group (that is those having the same hash code in any of the L tables). Using several hash tables increases the likelihood that similar records fall into the same block but it also comes with the side effect that more tuples, possibly distant or unrelated, are put in the same bucket.

Multiprobe LSH [20] is a variation of standard LSH that aims at reducing the number L of hash tables but without the loss of nearest neighbours. Instead of building a large number of hash tables, multiprobe LSH builds *probing* sequences of the hash codes. The first step consists in building the hash tables as in standard LSH but in a smaller number. Then, in each hash table, all the "similar" hash codes are grouped together, and the elements contained in them are merged into the same bucket list because they are considered "similar". What is crucial is how to locate similar hash codes. To do so, the probing sequences are used: for each target hash code $g_i(x) = \{h_1(x), h_2(x), \dots, h_K(x)\}$ contained in a hash table $\mathcal{T}$, its probing sequence is composed of those hash codes $g_m(x)|_{m=1,\dots,p} \in \mathcal{T}$ whose hamming distance from $g_i(x)$ is at maximum s, where s is a specified scalar parameter. In simple words, multiprobe LSH clusters the K-dimensional hash codes generated by the application of the K hash functions of a hash table $\mathcal{T}$ based on their hamming distances. This is not a pure partitioning clustering, however, because the same hash code can be put into several groups.

5.1.2 Blocks Building

The critical point of the LSH algorithm is to define proper hashing functions $\{h_i\}|_{i=1,\dots,K}$. As the authors of [12], we derive these hash functions using the *random hyperplanes* method. This method consists in choosing K random hyperplanes through the origin in the N-dimensional space in which the embedding tuples are projected. Each hyperplane divides the space into two parts, an "upper" and a "lower" region. Depending on the relative position of the tuple vector x with respect to the ith hyperplane, the vector is assigned a value $+1$ ("upper" region) or -1 ("lower" region) in position i of its hash code. The position of vector x is evaluated with respect to each of the K hyperplanes, and consequently, a K-sized hash code is generated. To increase the likelihood that similar records get the same hash code, we compute several hash codes are for each tuple. Given the hash tables, then all the records sharing the same K-dimensional code in any of the L tables are considered "similar" and put into the same group.

The approach exploiting multiprobe LSH is strictly related to the one just described. We still use the random hyperplanes method, but in this case a smaller number of hash tables are generated. Alternatively, in order to increase the likelihood that similar records are placed in the same block, all the hash codes whose distance is less than or equal to a pre-specified parameter s are grouped and their lists of tuples are merged.

5.2 Clustering-Based Blocking

The second blocking method we present uses conventional clustering algorithms to partition the embedding vectors received as input into buckets of similar records. In order to choose the most appropriate clustering algorithm, in our experimental analysis we test and compare four of them: hierarchical clustering, K-Means, DBSCAN, and Birch.

Before presenting how the different clustering algorithms place the input vectors in distinct blocks, we need to address the challenges posed by the high dimensionality of the data and therefore explain how to pre-process the embedding vectors and transform them in such a way that the four clustering algorithms can effectively analyze them.

The biggest challenge faced while clustering high-dimensional data is that traditional distance measures often used in conventional clustering algorithms become useless [19] due to the problem known as the *Curse of Dimensionality* [4]. This term is actually used to refer to four sub-problems: (i) multiple dimensions are difficult to visualize and enumerate due to the exponential number of possible values related to each dimension; (ii) measures like proximity, distance, or neighbourhood risk to become meaningless in high-dimensional spaces, since in some applications it can happen that the relative distance of the farthest point and the nearest point converges to 0 as the dimensionality grows [1]; (iii) among the many available features, a high number of irrelevant ones is to be expected: these useless dimensions can be related to "noise" and thus interfere with the clustering discovery task making it more difficult and prone to errors; *(iv)* due to the high number of attributes in the dataset, some of them may be correlated.

The above-stated challenges make it difficult to cluster the embedding vectors into blocks of records; in particular,

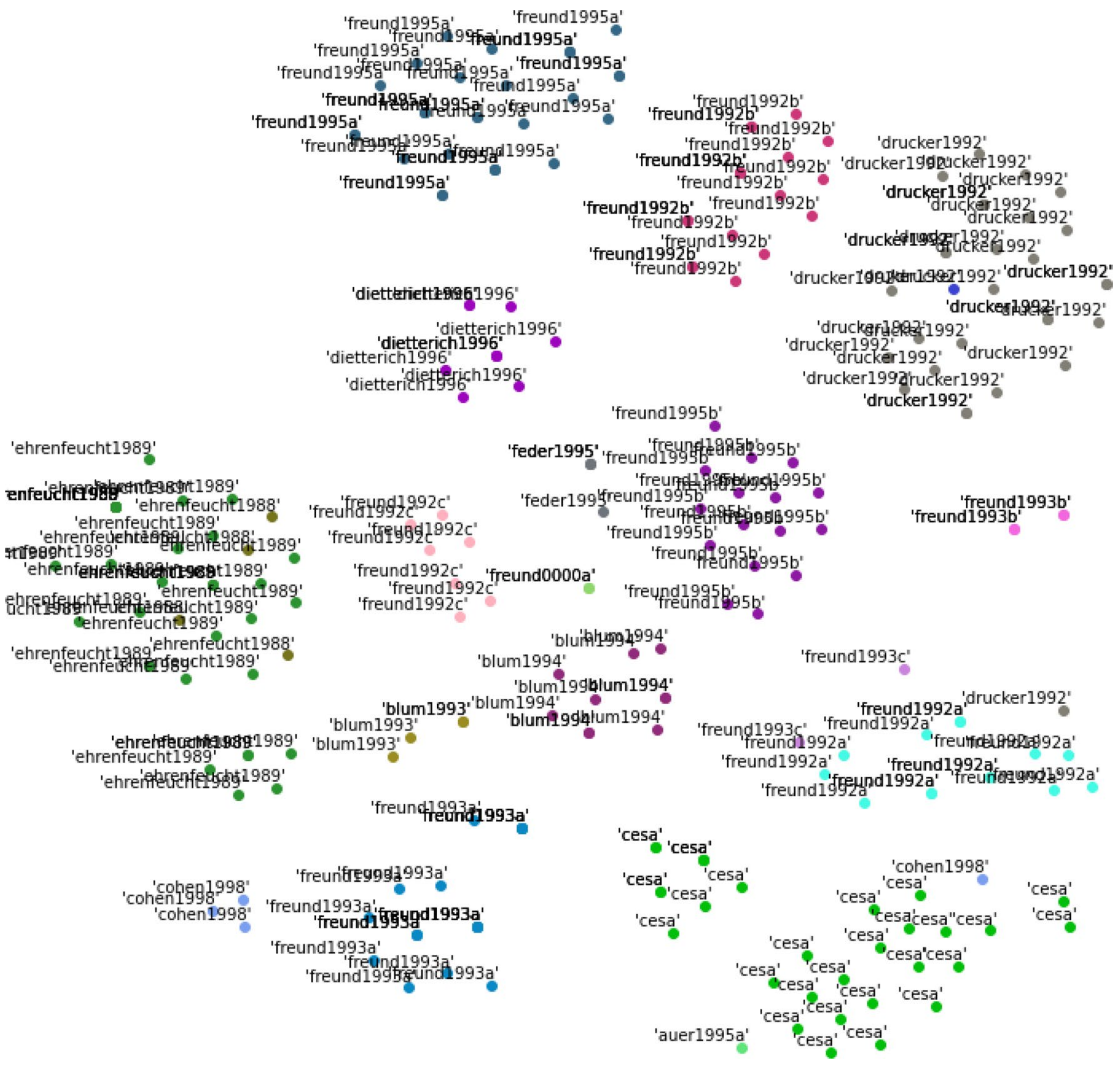

Fig. 4 Visualization of the t-SNE dimensionality reduction technique applied to the Cora dataset

the fact that distance measures become less effective in high-dimensional spaces somehow neutralizes the appealing property of our vectors: they are created in such a way that semantically similar tuples are placed close to each other in the high-dimensional space. In order to exploit this appealing characteristic, we need to project the vectors onto a lower-dimensional space, which can better describe the similarity of records belonging to the same group.

We now present the two dimensionality reduction techniques used in our methodology: principal component analysis (PCA) and t-distributed stochastic neighbour embedding (t-SNE).

Principal Component Analysis: this method performs an orthogonal projection of the data onto a linear space with a lower number of dimensions; the lower-dimensional space is called *principal sub-space*. The projection is performed in such a way that the variance of the projected data is maximized and thus maintaining as much as possible the information present in the original dataset, in our case the embedding vectors [5, 17].

t-distributed Stochastic Neighbour Embedding: this method consists in a nonlinear dimensionality reduction transformation of a high-dimensional input dataset into a low-dimension output space of two or three dimensions. This is achieved by minimizing the divergence between the distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points [21].

Contrary to traditional linear dimensionality reduction techniques, such as PCA, t-SNE is capable of capturing both the local and the global structures of the high-dimensional data; this results in a transformation where similar vectors are modelled by nearby points and dissimilar vectors are modelled by distant points.

Figure 4 shows the result of t-SNE applied to the embedding vectors generated from the Cora dataset[4]. We can see how well t-SNE managed to project the original embeddings, whose size is 4096, onto a bi-dimensional space. (For clarity only the first 300 records of the dataset are represented.)

Although t-SNE delivers overall good results, a deeper analysis shows some weakness of this algorithm: for instance, if we take into consideration the green cluster in the bottom right section of Fig. 4, where most of the “cesa” records have been placed, we can notice two criticalities: (i) the cluster is split into two parts and (ii) it contains an outlier: “cohen1998”, that should have been positioned in the light-blue cluster in the bottom left part of the figure. This is a problem because, once we have placed an element into a block, it will not be compared with elements outside this block, and in this case we would not be able to link the misplaced “cohen1998” record to its true entity. These are well-known problems of t-SNE; in fact, it has been shown that: (i) the visualized clusters produced by t-SNE can be strongly influenced by the parametrization of the algorithm, (ii) such clusters may even appear in non-clustered data. To overcome these challenges, in our methodology, we merge the clusters discovered by t-SNE into bigger cluster, with the ultimate goal of placing all the records that refer to the same entity in the same block; at the same time, we are careful not to generate too large blocks, which would jeopardize the usefulness of the blocking phase. Despite the problems we have just analyzed, as can be seen in the figure, the dimensionality reduction operated by t-SNE still represents a good starting point to be followed by a clustering algorithm and creates the final blocks.

Clustering Phase: once we have projected the embeddings onto a lower-dimensional space, either by using PCA or by using t-SNE, we can apply clustering algorithms to partition the records into groups of similar tuples.

We now briefly present how each of the four conventional clustering algorithms we tested in our methodology can arrange the records inside the buckets.

- *Hierarchical Clustering:* in its agglomerative variant [31], this algorithm assembles the clusters by recursively partitioning the vectors in a bottom-up fashion. Initially, each record is assigned to a cluster of its own; then, at each iteration of the algorithm, clusters are merged until the desired hierarchical structure is achieved. The resulting structure, describing how the clusters have been combined through the iterations, and their respective similarity are called a *dendrogram*. A clustering of the data is obtained by cutting the dendrogram at a height corresponding to the desired similarity level or to the desired number of clusters. In order to decide which clusters should be merged at each iteration, two measures are needed, the first specifying the distance between two records and the second specifying how far are two clusters placed from each other; we selected the *Euclidean distance* for the former and the *Ward Linkage* criterion [34] for the latter .
- *K-Means:* this algorithm partitions the records into K clusters, each identified by its centre, computed as the mean of the vectors’ coordinates assigned to that cluster. Initially, the algorithm randomly chooses K centres; then, at each iteration, the records are assigned to the nearest cluster centre, and once all of them have been assigned, the cluster centres are recomputed and the algorithm continues with the following iterations until convergence is reached [31]. We use the Euclidean distance to determine how far records are from the cluster centres.
- *DBSCAN:* this algorithm can detect clusters of arbitrary shape by analyzing the density surrounding each record. More specifically, a cluster is defined as an area of high density delimited by areas of low density. To discover the clusters, two parameters are needed: the *neighbourhood size*, specifying the minimum number of records required to form a dense region, and *eps*, the maximum distance between two records in order for them to be considered as belonging to the same neighbourhood [31].
- *Birch:* thanks to its ability to find good clustering solutions with just one scan of the data, this clustering algorithm represents a very effective solution for dealing with very large datasets. This algorithm is composed of two main phases: it first loads the records into the memory by building the cluster feature tree (CF tree), a lossless compression of the data, that through the use of appropriate summary statistics, manages to preserve the clusters structure originally present in the records, then any of the existing clustering algorithms can be applied to the leaves of the CF tree to obtain the actual clusters [36].

6 Experimental Results

We tested our blocking systems on six popular datasets with two objectives: first to compare their performances against five traditional blocking algorithms and second to understand how the several architectural variants we implemented differ among each other. Since every blocking algorithm has its own set of parameters, we also define how these are set for the tests.

[4] The Cora dataset, containing information about research articles—represented by their bibliographic references—is presented and analyzed in the experimental section of this paper.

Table 5 Specifications of the target datasets

Data set	Task	#Tuples	#True matches	#Attributes	Cartesian size
Restaurant	Deduplication	864	112	5	372816
Cora	Deduplication	1295	17184	12	837865
Census	Deduplication	841	327	5	353220
DBLP-ACM	Linkage	2616-2294	2224	5	6001104
AMZN-GP	Linkage	1363-3226	1300	5	4397038
ABT-BUY	Linkage	1081-1092	1097	4	1180452

6.1 Datasets

The blocking algorithms are tested on six publicly available datasets that are commonly adopted to evaluate entity linkage and blocking schemes [7, 12, 18, 27]. These are: *Restaurant* dataset [30], *Census* dataset [30], *Cora* dataset [30], *DBLP-ACM* datasets [11], *Amazon-Google Products* datasets [11] and *Abt-Buy* datasets [11].

The first two datasets represent the "easy" task: they are structured and mostly clean with very few typos and missing values. On these datasets, traditional blocking techniques show strong results as reported in [7].

Cora is still a well-structured dataset, but it has some quality issues as will be discussed later in the results. *Restaurant* is a set of tuples with restaurant names, addresses, cities, phones and food styles taken from Fodor and Zagat restaurant guides. *Census* is a dataset generated by the US Census Bureau including information such as first and last names, middle initials, zip codes and street addresses. *Cora* contains records about machine learning articles with many attributes as publication name, publication year, authors name, venue, etc. *DBLP-ACM* are well-structured bibliographic data sources regarding computer science conferences. Among the shared fields there is a relatively long textual attribute *title*, but its values are fixed and the same between the two sources so on these sets traditional blocking models are expected to perform well anyways.

The last two pairs of data sources are "challenging": they have long textual attributes (such as product description) written in natural language with plenty of variations. Both tasks deal with e-commerce products and the single sources have also duplicates inside each of them. In Table 5, we summarize some of the key characteristics of the datasets.

The "Task" column specifies the type of activity to be performed: *(i) Deduplication* for finding duplicate tuples that all belong to a single source and *(ii) Linkage* for finding related tuples across two or more data sources. The "Cartesian size" column contains the number of comparisons one would perform without blocking ($n * m$ for the (entity) linkage task, $\frac{n*(n-1)}{2}$ for the deduplication case, where n and m are the number of tuples in the (two) source(s)).

6.2 Metrics

Two metrics are considered to assess the performances of the blocking algorithms: *Reduction ratio* (RR) and *Pair completeness* (*PC*), which are the measures typically used in the literature [3, 7, 8]. Following the notation of [8], *RR* and *PC* are defined as:

$$RR = 1.0 - \frac{s_M + s_N}{n_M + n_N} \qquad PC = \frac{s_M}{n_M} \tag{3}$$

where

- n_M is the number of matching pairs generated without blocking.
- n_N is the number of non-matching pairs generated without blocking.
- s_M is the number of matching pairs generated with blocking.
- s_N is the number of non-matching pairs generated with blocking.

RR measures how much the blocking technique can reduce the number of pair comparisons with respect to a naïve full comparison approach.

The second metric, *PC*, measures instead the effectiveness of the blocking method at not removing true matches from the set of comparisons.

These metrics range in [0, 1], and they are typically in a trade-off [8]: a high reduction ratio may come at the cost of some missed true matches or vice versa a large pair completeness may require to compare also many unnecessary pairs.

In some works, RR and *PC* are combined to assess the overall blocking result. In [27], for example, the metrics are multiplied $\alpha = RR \cdot PC$. In our tests to select the best result, we use the *harmonic mean* of the two, $\alpha = 2 \cdot \frac{RR \cdot PC}{RR + PC}$.

6.3 Design of the Evaluation

We design our tests in order to analyze two aspects of our blocking systems: first how they perform with respect to

Table 6 Blocking algorithms comparison on Restaurant, Cora and Census datasets

	Restaurant			Cora			Census		
	RR	PC	α	RR	PC	α	RR	PC	α
Standard	0.9848	0.9464	0.9652	**0.9659**	0.7881	0.8679	**0.9829**	0.7703	0.8637
Suffix	**0.99**	0.875	0.9289	0.989	0.4441	0.6129	**0.9829**	0.7703	0.8637
Sorted N.	0.9848	0.9464	0.9652	**0.9659**	0.7881	0.8679	0.8976	0.8721	0.8847
Q-gram	0.9848	0.9464	0.9652	0.9464	0.8322	0.8856	**0.9829**	0.7703	0.8637
Canopy	0.9848	0.9464	0.9652	**0.9659**	0.7881	0.8679	**0.9829**	0.7703	0.8637
Emb–LSH	0.9432	**0.9792**	0.9608	0.9233	0.7626	0.8352	0.8945	0.9542	0.9234
Emb–Clust	0.9859	0.9464	**0.9657**	0.9635	**0.887**	**0.9241**	0.9006	**0.9622**	**0.9304**

traditional blocking methods and second how the architectural variants presented in methodology section of the paper influence the final results.

We now review how the parameters are set in our tests, distinguishing between the traditional methods and our blocking schemes. For what concerns traditional blocking algorithms, parameter setting is done by hand, the best results are obtained experimentally and then used in comparison with our blocking algorithm.

Traditional algorithms require a *blocking function* to operate, and in our tests, for each dataset; we run the experiments by applying iteratively one of the following functions: *attribute value*, *first n chars*, *last n chars* (with $n \in \{2, 3, 4\}$) and *Soundex* phonetic encoding. The blocking function is applied to every single attribute of the current dataset. In *standard* blocking, the blocking function is the only variable to be tested as no further parameters are defined for this algorithm. In *sorted neighbourhood* blocking, for each dataset the parameter window size w is varied in the range $w \in \{2, 3, 4, 5\}$. *Q-gram* blocking is tested with q-gram size $q \in \{3, 4, 5\}$ and threshold $t \in \{0.6, 0.8\}$. In *suffix* blocking, the minimum suffix length $l_{min} \in \{3, 4, 5\}$ and maximum block size $b_{max} \in \{20, 50\}$. Finally, the *canopy cluster* blocking algorithm is applied with the q-gram size $q = 3$.

Regarding our blocking system, we evaluate both the average-based and the RNN-LSTM-based models described in the methodology above, with the following parameters: the number of LSTM cells $n_{cells} \in \{300, 1024, 2048\}$ and as word embeddings both GloVe and fastText.

For what regards the first of our blocking methods, to set the hash code size K and the number of hash tables L of the LSH algorithm, for each dataset we apply the theoretical formulae presented in [33]. While for choosing the hash functions, we proceed similar to [12]. For multiprobe LSH, we use the same K and L values but one more parameter is needed: the Hamming distance value s, this is set by hand and we test the algorithm with $s \in \{1, 2, 3\}$. The final results for LSH and multiprobe LSH are obtained by averaging the outcomes of 5 independent runs.

For what regards the second of our blocking methods, we set the number of components in PCA equals to 2. t-SNE being a more complex dimensionality reduction method needs three parameters to be accurately set: (i) as for PCA we set the number of components equal to 2, (ii) the *perplexity score*, a parameter controlling the number of nearest neighbours considered from the algorithm, varied in the range $\{30, 35, 40, 45, 50\}$ *and (iii)* the *early exaggeration score*, a parameter controlling the space among the discovered clusters, varied in the range $\{9, 12, 12, 15, 18\}$. Concerning the conventional clustering algorithms, we varied the number of clusters between 5 and 50 for *hierarchical clustering*, *K-Means* and *Birch*, while we for *DBSCAN* we assigned *eps* equal to 0.5 and varied the *neighbourhood size* between 2 and 10.

All the algorithms are implemented in Python programming language, and the tests are run on a Google Compute Engine[5] instance with the following specifications:

- Operating system: Ubuntu 16.04.1 LTS (Xenial Xerus)
- CPU: 4 hyper-threaded cores Intel Xeon Processor @2Ghz
- RAM: 32 GB
- SSD storage: 60 GB

6.4 Test Results

We now first provide the results of the tests between our blocking systems and the traditional methods, and then, we investigate the impact of different architectural choices of our model.

6.4.1 Our System vs Traditional Blocking Algorithms

Reduction ratio, *pair completeness* and *alpha*: Tables 6 and 7 illustrate the best results in terms of *RR*, *PC* and $\boldsymbol{\alpha}$ obtained by each blocking algorithm on each of the six target datasets. Our methods are named:

[5] https://cloud.google.com/compute/.

Table 7 Blocking algorithms comparison on DBLP-ACM, AMZN-GP and ABT-BUY datasets

	DBLP-ACM			AMZN-GP			ABT-BUY		
	RR	PC	α	RR	PC	α	RR	PC	α
Standard	**0.9996**	0.8826	0.9374	0.9865	0.4661	0.6331	0.9793	0.6272	0.7646
Suffix	0.9964	0.9528	0.9741	**0.9979**	0.1792	0.3038	**0.9935**	0.4011	0.5714
Sorted N.	0.9974	0.9834	0.9903	0.9604	0.4308	0.5947	0.9371	0.8386	0.8851
Q-gram	0.0	0.0	0.0	0.9865	0.4661	0.6331	0.9424	0.8049	0.8682
Canopy	0.9959	0.9645	0.9799	0.9865	0.4661	0.6331	0.9793	0.6272	0.7646
Emb–LSH	0.9873	**0.9946**	**0.9909**	0.9436	0.7885	0.8591	0.9343	**0.9088**	**0.9213**
Emb–Clust	0.999	0.9820	0.9904	0.9277	**0.8407**	**0.8821**	0.9217	0.9024	0.9120

– *Emb–LSH* the blocking system based on LSH and multiprobe LSH.
– *Emb–Clust* the blocking system based on dimensionality reduction techniques and conventional clustering algorithms.

The experiments on *Restaurant* and *Census* datasets show competitive results between traditional methods and our blocking systems, proving that embedding-based models keep up on clean and simple datasets. On these datasets, traditional blocking algorithms seem to have an edge on *RR* but on the other hand embedding-based models provide better *PC*. The only exception is on *Cora* dataset where the *q-gram* blocking algorithm achieves a 6.9% better score for *PC* with respect to the *Emb–LSH* model. We explain this result by inspecting the dataset. This data source has serious quality issues on the majority of the attributes: the column *volume* has 76% of missing values, the attribute *address* 77%, field *note* reaches 90% of missing values. The only clean attribute is *title* and its values are mainly fixed with very little variations. All the traditional blocking methods select this field to filter the tuples, and without a prior imputation strategy it becomes really hard to leverage semantic information out of so many blank values. Despite the poor performance of our first blocking method on the *Cora* dataset, our second method, *Emb–Clust*, manages to outperform the *q-gram* blocking algorithm; this is thanks to t-SNE, that, also in this challenging conditions is able to build a bi-dimensional projection where the points representing the same entity are overall correctly grouped together.

Overall, the performances of traditional blocking algorithms on these sets are aligned with those reported in [7] and in [27].

Even though on *DBLP-ACM* the results are still balanced, *q-gram* shows null values because it was not possible to conclude the test: time and memory consumptions were prohibitive. This is a known limitation of this type of blocking algorithm, as computing all the q-grams of long textual values is expensive.

The most significant differences between our blocking systems and the traditional approaches are on the tests on the last two “challenging” datasets: *AMZN-GP* and *ABT-BUY*. Our embedding-based models are slightly less efficient at reducing the number of pair comparisons, but *PC* is much higher than traditional methods. Even though a *PC* of 0.84 (obtained by one of our blocking methods on the *AMZN-GP* dataset) is still a relatively poor result, the analysis suggests that our models are good at capturing the semantic information out of data. Supporting this insight are also the results on *ABT-BUY* datasets on which we record the best performances in two scenarios:

– Traditional models are free to choose the attribute and blocking function giving the best result (Table 7). In this case, they all go for the *name* attribute. This attribute is relatively clean and easy for them to block on.
– Traditional models are forced to use as attribute the *description* of products (Fig. 5). The values of this attribute are long textual descriptions written in natural language plenty of variations.

As can be seen in Fig. 5, the *PC* performances are completely different, with an absolute win for the embedding-based models. Overall, we consistently obtain the best *PC*, while maintaining a *RR* close to the best result obtained by traditional approaches. This is confirmed by the α score, where our systems rank first on all the six datasets. We

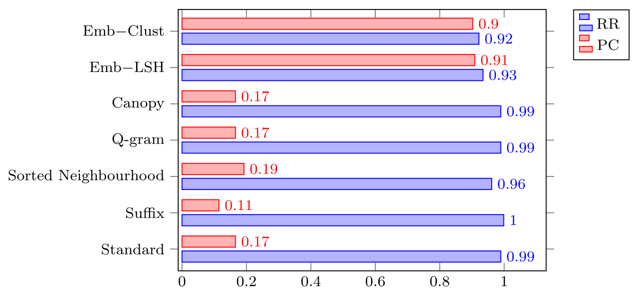

Fig. 5 RR and PC on ABT-BUY dataset forcing on description attribute

Table 8 Average RR, PC and α over all the datasets

Method	RR	PC	α
Standard	0.9832	0.7468	0.8386
Suffix	0.9916	0.6037	0.7091
Sorted N.	0.9572	0.8099	0.8646
Q-gram	0.9686	0.764	0.8305
Canopy	0.9825	0.7604	0.8457
Emb–LSH	0.9377	0.898	0.9151
Emb–Clust	0.9463	0.9212	0.933

also think that often a high *PC* should be preferred w.r.t. a high *RR*, since performing a few more comparisons could be less expensive than actually missing data that were positioned in the wrong block. The results match our intuition that traditional blocking solutions work poorly on noisy and textual datasets because of their inability to leverage semantic information. By contrast, embedding-based models exploit the meaning of words and sentences and overall provide a more appropriate solution.

We can see that the conclusions we have just drawn are confirmed by the results displayed in Table 8 where we reported the average *reduction ratio*, *pair completeness* and α that the blocking methods achieved over all the datasets. In particular, it is possible to notice the considerable margin that our two algorithms have over the traditional blocking methods in terms of the harmonic mean α between *reduction ratio* and *pair completeness*.

As far as our comparison between the two blocking systems is concerned, they exhibit fairly similar performances, with *Emb–Clust* taking an edge over *Emb–LSH* because it scores slightly better in four of the six datasets, and overall gaining 1.79% in terms of α score. The only exception to this evidence is constituted by the *Cora* dataset, where *Emb–Clust* scores significantly better than *Emb–LSH*.

Comparison with DeepER: comparing our approach with the supervised method implemented by DeepER [12], we can notice two different behaviours:

– When the systems are tested on the "easy" datasets, the two approaches show similar results both in terms of pair completeness and in terms of reduction ratio.
– When the systems are tested on the "challenging" datasets, DeepER [12] outperforms our unsupervised approach, consistently achieving a pair completeness higher than 0.95, while our methods manage to obtain a pair completeness between 0.85 and 0.90; on the contrary, the reduction ratios remain comparable.

These results are to be expected, confirming that if labelled data are available, or the user is willing to produce them, a supervised approach can be beneficial. We do not see this difference in performance as a big drawback, in fact, when comparing our methods against the traditional methods we can see a much greater gap in the performances: when the traditional methods are tested on the "challenging" datasets, none of them is able to exceed a pair completeness of 0.5. Our methods perform significantly better, confirming that the semantics learned from the external independent corpus used for training our models actually made a substantial difference in improving the results over the traditional methods. Moreover, hand-labelling data is very expensive, and we believe that in many scenarios sacrificing 5 to 10% of the pair completeness can be acceptable when the cost of creating a manually labelled training set is taken into consideration.

Execution Time: despite the good results of our blocking schemes, tests confirmed that deep-learning-based models are very expensive in terms of time (and memory consumption); this is a well-known pitfall of these models [25].

We show in Table 9 the average time needed (in seconds) to complete the blocking phase on each dataset. Despite both our methods taking longer than traditional blocking algorithms, we can see a significant improvement in the time required by *Emb-Clust* to produce the blocks, with respect to time needed by *Emb-LSH*.

Since our tests are run on a single computing instance, we believe that by adopting a parallel and distributed computation paradigm this blocking scheme can increase its efficiency by a wide margin.

Table 9 Execution times of the blocking methods

Method	Restaurant	Cora	Census	DBLP-ACM	AMZN-GP	ABT-BUY
Standard	0.6888	2.7359	0.6638	3.1501	3.7234	1.3336
Suffix	0.7033	2.7442	0.6797	4.0099	3.1015	1.1457
Sorted N.	0.7056	2.667	0.7303	3.3683	6.1487	2.1025
Q-gram	7.5938	2.9335	7.6657	Unbounded	10.599	1.8767
Canopy	0.2925	2.9954	0.6993	1449.9015	4.1971	2.7381
Emb–LSH	19.1221	163.6499	142.2958	1080.6682	815.4097	402.2910
Emb–Clust	18.8836	33.1579	19.0854	254.7286	204.0034	73.7883

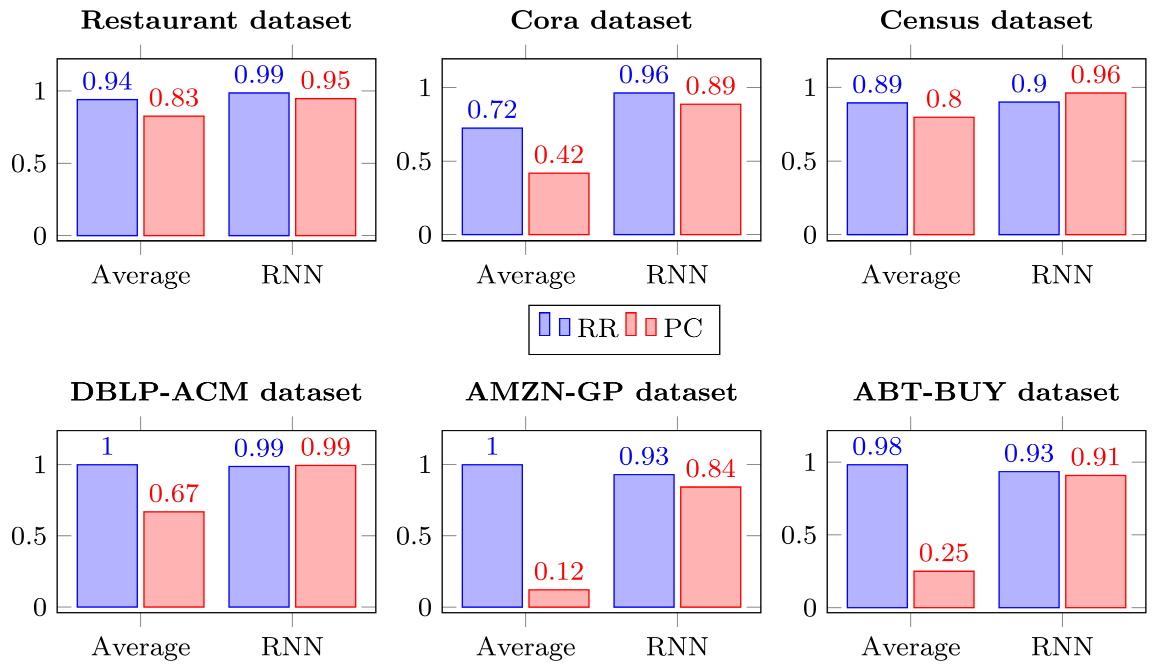

Fig. 6 RNN-LSTM vs average architecture performances

We need also to take into consideration that our methods consistently achieve the best *pair completeness* over all the datasets, and often, forsaking time efficiency for a higher *pair completeness* is considered acceptable, especially in those cases where the quality of the information is regarded as a crucial property of the result. In fact, after the blocking phase, only the records inside the same block are compared with each other, and thus, all the misplaced records cannot be linked to their entity anymore. Additionally, the integration of the data sources is typically done at the beginning of a data analysis project; as a result, we believe that it is worth spending more time (few minutes) on the integration phase, with the scope of having more precise and complete data and hence significantly better performance during the data analysis task.

6.4.2 Results for Different Architectural Choices

RNN-LSTM vs Average: one of the clearest results is the difference in performances when comparing the RNN-LSTM architectures with the simpler average scheme (Fig. 6). The average-based architecture provides sufficient results only on *Restaurant* and *Census* datasets where the attribute values are clean and atomic or at most composed of few words. When the textual values are longer, however taking into account words order guarantees more refined embeddings. This observation is particularly evident by considering the outcomes on the “challenging” datasets, *Amazon-Google Products* and *Abt-Buy*. On those sources, the neural nets are capable of encoding the dependencies among words and adjacent fields more effectively, thus obtaining a substantial improvement on *PC*. Conversely when many word embeddings are averaged, the resulting vector is less discriminative.

LSTM vs bi-LSTM: another key evidence resulting from the tests concerns the superiority of the bi-LSTM architecture over the uni-LSTM model for the current task (Fig. 7). bi-LSTM nets outperform the single LSTMs on every dataset, especially in terms of *PC*, suggesting that they generate better tuple representations. Similarly to the previous set of tests, the gain in *PC* is more significant when dealing with the complex datasets.

fastText vs *GloVe:* the two word embedding approaches show similar results on *Restaurant*, *DBLP-ACM* and *Census*. On the remaining datasets, however, GloVe is ahead regarding the *PC* (Fig. 8). We explain this trend by recognizing greater generalization capabilities of GloVe on these data sources.

The datasets about e-commerce products in particular contain codes, commercial names and brands which are handled differently by the two embedding paradigms: GloVe ignores the majority of them because they are not in the dictionary of known words, whereas fastText constructs new words embeddings by considering their *n*-grams. However, brand names rarely convey semantics about real-world entities, and this should explain why fastText is not able to enrich the expressiveness of the embeddings.

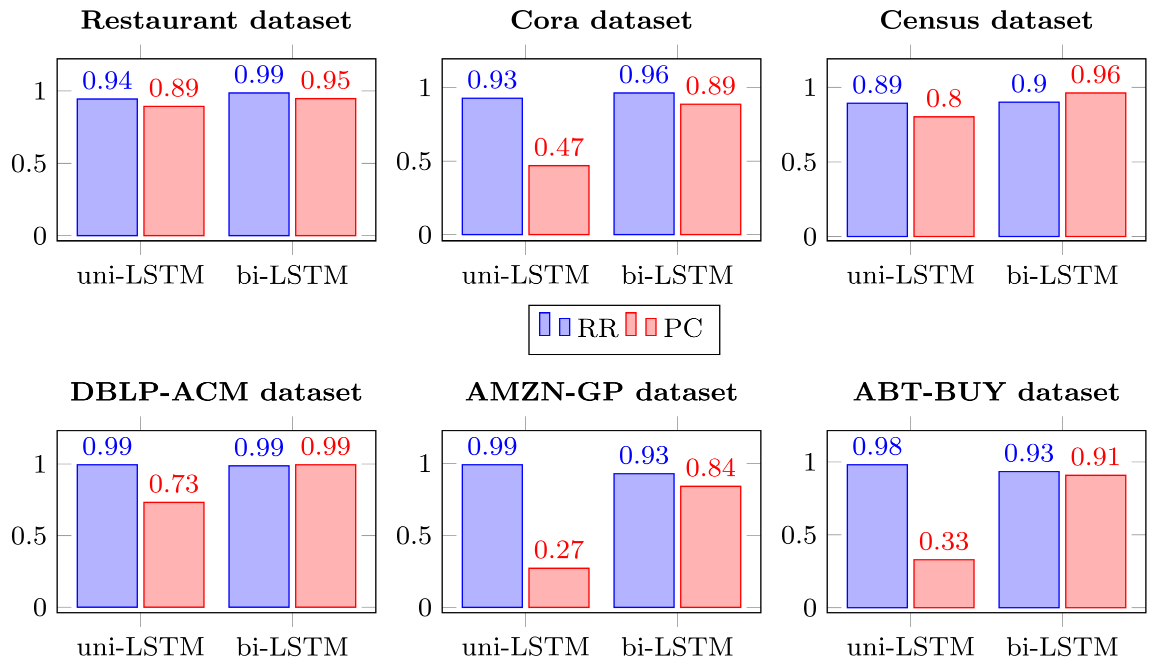

Fig. 7 uni-LSTM vs bi-LSTM performances

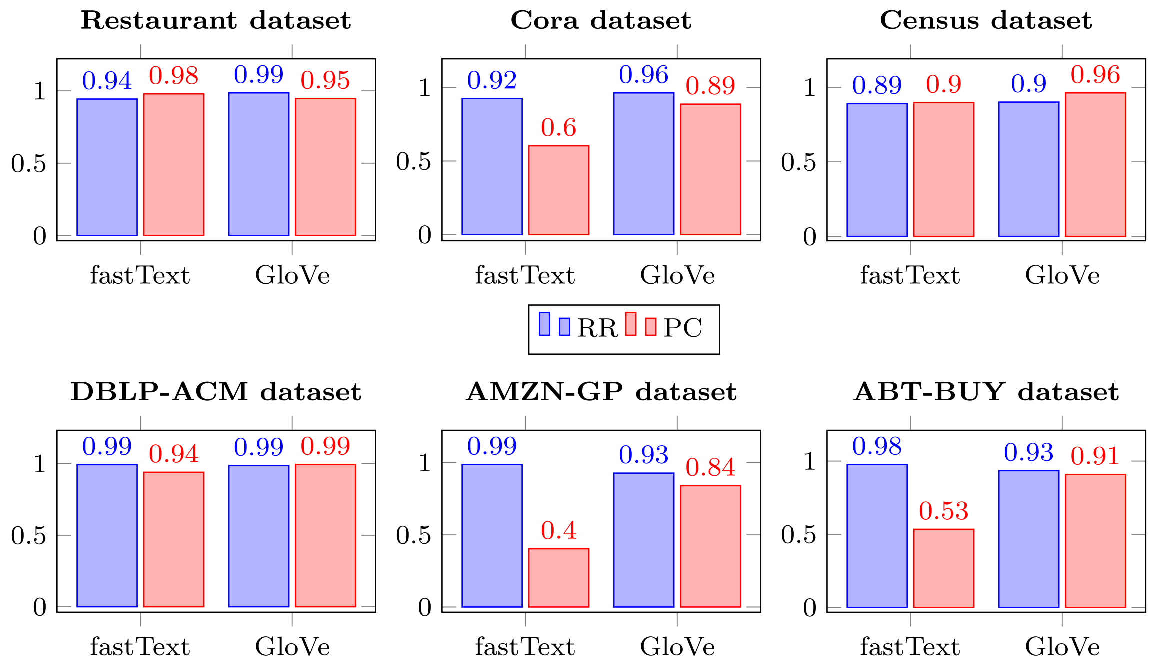

Fig. 8 fastText vs GloVe performances

In other words, on these typologies of datasets a more relaxed model that promotes generalization seems to be more appropriate.

RNN-LSTM Sizes: test results presented in Fig. 9 show the best performances obtained by using RNN-LSTM nets with the following number of LSTM cells: 300, 1024 and 2048. As can be seen, most of the top scores are associated with the 2048 variants of the nets, but in general the differences with respect to sizes are not crucial. Noting that the number of LSTM cells defines the size of the embedding tuples, this suggests that even with the smaller vector sizes we can obtain good blocking results.

PCA vs t-SNE: Figure 10 displays the best results obtained by our system when using the two dimensionality-reduction

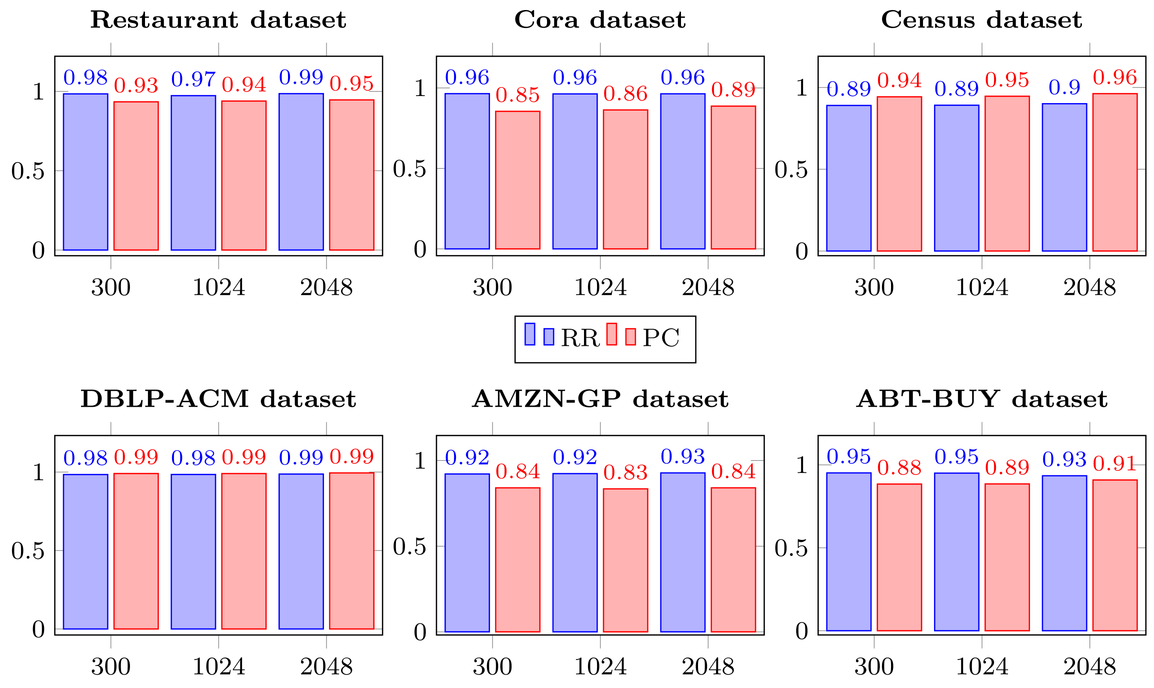

Fig. 9 RNN size performances

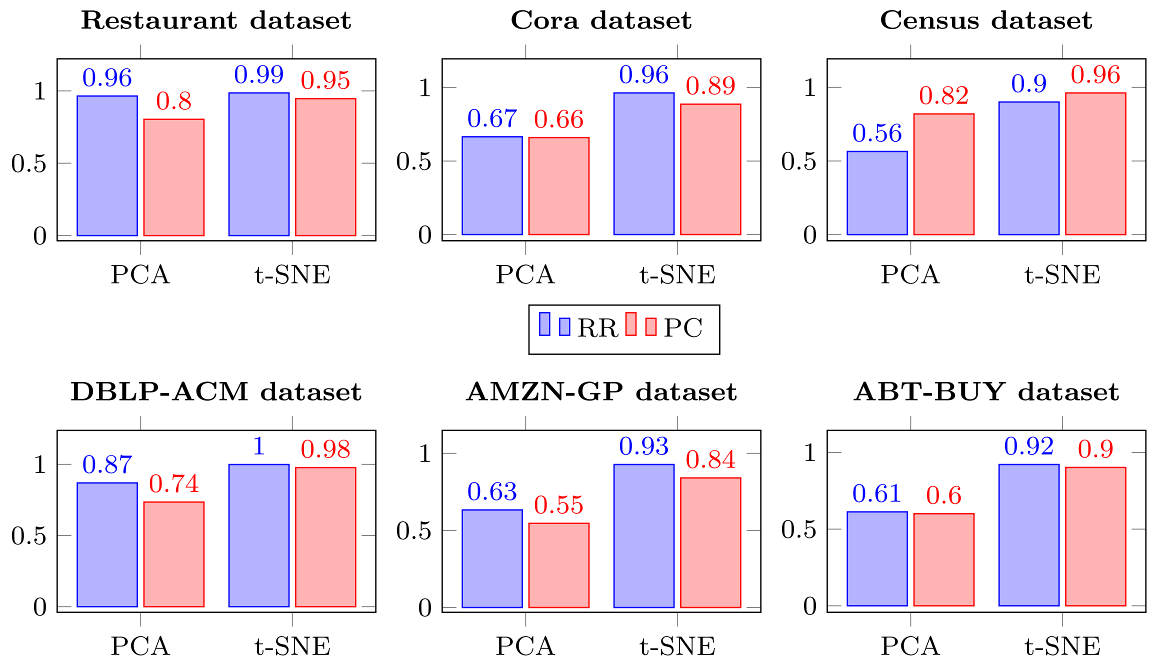

Fig. 10 PCA vs t-SNE dimensionality reduction performances

techniques. It is clear that the results obtained by *t-SNE* are superior to the ones achieved by *PCA*; we attribute the difference in performances to two factors: *(i)* as opposed to *PCA*, the nonlinear projection performed by *t-SNE* better manages to maintain the information encoded in the embedding vectors , *(ii)* the clusters automatically discovered by *t-SNE* represent a better starting point for the conventional clustering algorithms than the linear projection returned by *PCA*.

Conventional Clustering Algorithms Compared: the test results presented in Fig. 11 show how overall the four clustering methods achieve similar results, with two exceptions: *(i)* the *AMZN-GP* dataset, where *hierarchical clustering*

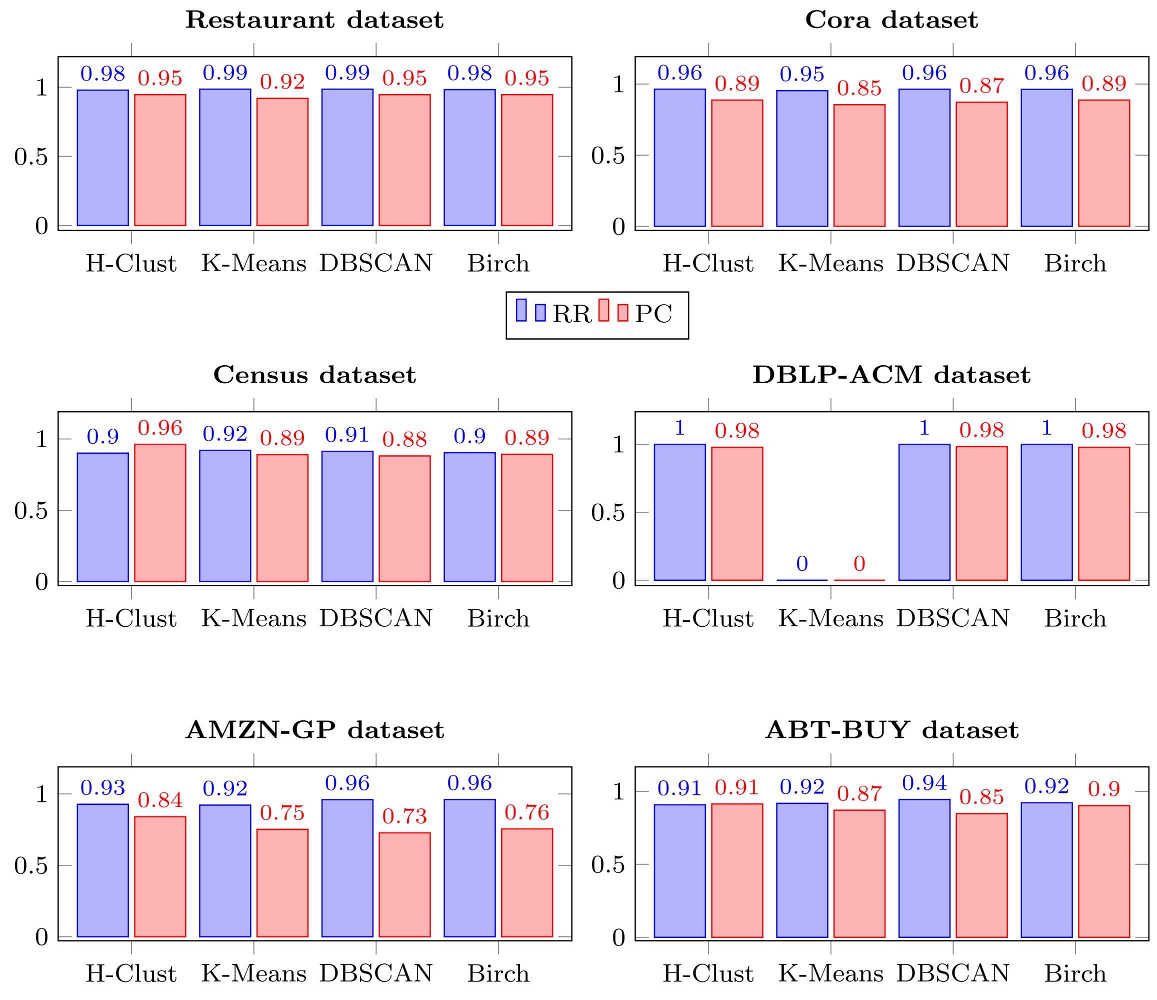

Fig. 11 Performances of the conventional clustering algorithms compared

scores a better *reduction ratio* and *DBSCAN* and *Birch* obtain a better *pair completeness and (ii)* the *DBLP-ACM* dataset, where *K-Means* did not manage to reach convergence in acceptable time. A deeper inspection of these experimental results showed a substantial drawback of *K-Means*: on all the datasets it required a significantly longer time to complete its computation, compared to the other three algorithms. Specifically, with the scikit-learn[6] implementation of the four algorithms used in our system, *Hierarchical Clustering*, *DBSCAN* and *Birch* found the clusters in less than 1 s on all the datasets, while *K-Means* required a time spanning from 13 and 164 s. Given these considerations and being the time complexity of *Birch* [36] linear, we suggest to use this last algorithm in the final implementation of the *Emb–Clust* blocking system.

6.4.3 Final Remarks on the Experiments

Given the experimental results, the final architecture we suggest comprises the following components:

- Embedding architecture: RNN-based
  - Network type: bi-LSTM
  - RNN-LSTM size: 2048
  - Word embeddings: GloVe
- Blocking method: clustering-based
  - Dimensionality reduction technique: t-SNE
  - Clustering method: Birch

Given the good results of our methods, we foresee their application to a wide variety of scenarios; especially when no labeled training datasets are provided and when asking a user to manually create one is considered prohibitively expensive.

[6] https://scikit-learn.org/stable/modules/clustering.html.

7 Conclusions and Future Work

We presented two unsupervised blocking systems based on leveraging the data semantics. Experimental results demonstrated that our deep-learning-based blocking solutions outperform traditional algorithms, especially on textual and noisy datasets. Additionally, our tests showed that training the neural networks on external corpora and then plugging them in the blocking system to build tuple embeddings produces good results.

Possible future work may include: (i) trying other, newly released sentence-embedding models such as [29], (ii) reducing the execution time of our blocking scheme adopting a parallel and distributed computation paradigm, (iii) studying the applicability of our unsupervised approach on a broader range of scenarios [32], (iv) experimenting with more sophisticated clustering algorithms [35].

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Aggarwal CC, Hinneburg A, Keim DA (2001) On the surprising behavior of distance metrics in high dimensional space. In: International conference on database theory
2. Aizawa A, Oyama K (2005) A fast linkage detection scheme for multi-source information integration. In: WIRI
3. Baxter R, Christen P, Churches T (2003) A comparison of fast blocking methods for record linkage. In: KDD
4. Bellman RE (2015) Adaptive control processes: a guided tour. Princeton University Press, Princeton
5. Bishop CM (2006) Pattern recognition and machine learning. Springer, Berlin
6. Bowman SR, Potts C, Manning CD, Angeli G, A large annotated corpus for learning natural language inference. https://nlp.stanford.edu/projects/snli/
7. Christen P (2011) A survey of indexing techniques for scalable record linkage and deduplication
8. Christen P (2012) Data matching. Concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer, Berlin
9. Conneau A, Kiela D, Schwenk H, Barrault L, Bordes A. https://github.com/facebookresearch/InferSent
10. Conneau A, Kiela D, Schwenk H, Barrault L, Bordes A (2018) Supervised learning of universal sentence representations from natural language inference data. arXiv:1705.02364
11. Creative Commons license. https://dbs.uni-leipzig.de/en/research/projects/object_matching/benchmark_datasets_for_entity_resolution
12. Ebraheem M, Thirumuruganathan S, Joty S, Ouzzani M, Tang N (2018) Distributed representations of tuples for entity resolution. In: Proc. VLDB Endowment
13. Gionis A, Indyk P, Motwani R (1999) Similarity search in high dimensions via hashing. In: Proc. VLDB Endowment
14. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT press, Cambridge
15. Hernandez MA, Stolfo SJ (1995) The merge-purge problem for large databases. ACM SIGMOD
16. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780
17. Hotelling H (1933) Analysis of a complex of statistical variables into principal components. J Educ Psychol 24(6):417
18. Koepcke H, Thor A, Rahm E (2010) Evaluation of entity resolution approaches on real-world match problems. In: Proc. VLDB Endowment
19. Kriegel HP, Kröger P, Zimek A (2009) Clustering high-dimensional data: a survey on subspace clustering, pattern-based clustering, and correlation clustering. ACM Trans Knowl Discov Data (TKDD) 3(1):1–58
20. Lv Q, Josephson W, Wang Z, Charikar M, Li K (2007) Multi-probe lsh: efficient indexing for high-dimensional similarity search. In: Proc. VLDB Endowment
21. Maaten LVD, Hinton G (2008) Visualizing data using t-SNE. J Mach Learn Res 9:2579–2605
22. McCallum A, Nigam K, Ungar LH (2000) Efficient clustering of high-dimensional data sets with application to reference matching. In: ACM SIGKDD
23. Mikolov T, Grave E, Bojanowski P, Puhrsch C, Joulin A. https://fasttext.cc/docs/en/english-vectors.html
24. Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems
25. Mudgal S, Li H, Rekatsinas T, Doan A, Park Y, Krishnan G, Deep R, Arcaute E, Raghavendra V (2018) Deep learning for entity matching: a design space exploration. In: ACM SIGMOD
26. Papadakis G, Skoutas D, Thanos E, Palpanas T (2019) A survey of blocking and filtering techniques for entity resolution. arXiv:1905.06167v1
27. Papadakis G, Svirsky J, Gal A, Palpanas T (2016) Comparative analysis of approximate blocking techniques for entity linkage. In: Proc. VLDB Endowment
28. Pennington J, Socher R, Manning CD. https://github.com/stanfordnlp/GloVe
29. Perone CP, Silveira R, Paula TS (2018) Evaluation of sentence embeddings in downstream and linguistic probing tasks. arXiv:1806.06259
30. RIDDLE repository. www.cs.utexas.edu/users/ml/riddle/data.html
31. Rokach L, Maimon O (2005) Clustering methods. Data mining and knowledge discovery handbook
32. Thirumuruganathan S, Parambath SAP, Ouzzani M, Tang N, Joty S (2018) Reuse and adaptation for entity resolution through transfer learning. arXiv:1809.11084
33. Wang J, Shen HT, Song J, Ji J (2014) Hashing for similarity search: A survey. arXiv preprint arXiv:1408.2927
34. Ward JH Jr (1963) Hierarchical grouping to optimize an objective function. J Am Stat Assoc 58(301):236–244
35. Xu R, Wunsch D (2005) Survey of clustering algorithms. IEEE Trans Neural Netw 16(3):645–678
36. Zhang T, Ramakrishnan R, Livny M (1996) BIRCH: an efficient data clustering method for very large databases. ACM Sigmod Rec 25(2):103–114