AutoBlock: A Hands-off Blocking Framework for Entity Matching

Wei Zhang*
zhangwei@cs.wisc.edu
University of Wisconsin-Madison

Hao Wei
wehao@amazon.com
Amazon.com

Bunyamin Sisman
bunyamis@amazon.com
Amazon.com

Xin Luna Dong
lunadong@amazon.com
Amazon.com

Christos Faloutsos
christos@cs.cmu.edu
Carnegie Mellon University

David Page
david.page@duke.edu
Duke University

ABSTRACT

Entity matching seeks to identify data records over one or multiple data sources that refer to the same real-world entity. Virtually every entity matching task on large datasets requires blocking, a step that reduces the number of record pairs to be matched. However, most of the traditional blocking methods are learning-free and key-based, and their successes are largely built on laborious human effort in cleaning data and designing blocking keys.

In this paper, we propose `AutoBlock`, a novel hands-off blocking framework for entity matching, based on similarity-preserving representation learning and nearest neighbor search. Our contributions include: (a) **Automation**: `AutoBlock` frees users from laborious data cleaning and blocking key tuning. (b) **Scalability**: `AutoBlock` has a sub-quadratic total time complexity and can be easily deployed for millions of records. (c) **Effectiveness**: `AutoBlock` outperforms a wide range of competitive baselines on multiple large-scale, real-world datasets, especially when datasets are dirty and/or unstructured.

CCS CONCEPTS

• **Information systems** → **Entity resolution**; **Deduplication**;
• **Theory of computation** → **Data integration**;

KEYWORDS

Entity Matching; Blocking; Deep Learning; Embedding

ACM Reference Format:
Wei Zhang, Hao Wei, Bunyamin Sisman, Xin Luna Dong, Christos Faloutsos, and David Page. 2020. AutoBlock: A Hands-off Blocking Framework for Entity Matching. In *The Thirteenth ACM International Conference on Web Search and Data Mining (WSDM '20), February 3–7, 2020, Houston, TX, USA.* ACM, Anchorage, Alaska, USA , 10 pages. https://doi.org/10.1145/3336191.3371813

*Work performed during internship at Amazon.

WSDM '20, February 3–7, 2020, Houston, TX, USA

ACM ISBN 978-1-4503-6822-3/20/02…$15.00
https://doi.org/10.1145/3336191.3371813

1 INTRODUCTION

Entity matching seeks to identify data records over one or multiple data sources that refer to the same real-world entities. In the era of Big Data and data science, entity matching is playing an increasingly critical role as the value of the data expands exponentially when they are linked to other data to create a unified repository [9]. An exhaustive pairwise comparison grows quadratically with the number of records, which is unaffordable for datasets of even moderate size. As a result, virtually every entity matching task on large datasets requires *blocking*, a step that effectively reduces the number of record pairs to be considered for matching without potentially ruling out true matches.

A successful application of blocking to an entity matching task should fulfill the following four desiderata: First, blocking, ideally, should not leave out any true matches (i.e., high *recall*), since only the candidate record pairs generated by blocking will be further examined in the downstream matching step. Second, the number of candidate pairs should be small so that the cost of applying a usually computationally-expensive matching algorithm is controlled. Therefore, it is desired to have a small ratio of the number of candidate pairs to the number of entities (Pair-Entity ratio, or *P/E ratio*). Third, *human effort* should not be overspent during the whole blocking process; man-hours on cleaning data and tuning the configuration for blocking algorithms need to be minimized. Last but not least, the blocking algorithm should be *scalable* enough to handle millions of records.

Although the problem of blocking has been studied for decades, to the best of our knowledge, the dominant and most widely used methods in practice are key-based methods. The main idea of these methods is to divide records into a collection of blocks based on several human-crafted *blocking keys* such that we only perform comparisons only among records co-occurring in the same blocks. To improve recall, many efforts have been focusing on generating multiple customized blocking key [1, 11] on individual attributes or aggregated attributes [20].

Challenges The foremost challenge for blocking is the *unnormalization*, or namely the various types of noise, prevalent in the real-world data. As an illustrative example, consider two matched record pairs in Table 1 for songs. While each tuple in the pair resembles the other, a few common cases of unnormalization can still be observed: (a) "Blowin' " is *misspelled* as "Blowing"; (b) *missing values* appear on many attributes; (c) "Michael Bublé" is moved from Composer to Song Writer, which may have resulted from the

Table 1: An example of two matched pairs. Various cases of unnormalization are observed.

ID	Title	Album	Composer	Song Writer
1	Me and Mrs. Jones	Call Me Irresponsible	Michael Bublé	
2	Me and Mrs. Jones [remix]			Michael Bublé
3	Blowin' in the Wind	The Freewheelin' Bob Dylan	Bob Dylan	
4	Blowing in the Wind		Bob Dylan	

ambiguity in schema definition; (d) the title of Record 2 contains an extra version description "[remix]", possibly due to *imperfect extraction*.

The result of the prevalence of unnormalization in real-world data is that blocking becomes rather challenging with traditional key-based blocking methods. This is because these methods rely on exact matching of blocking keys; thus, to deal with the unnormalized data one would have to carefully choose among a large number of combinations of different data cleaning strategies and blocking key design [7, 10, 24]. It is often the case that these decisions are *dataset-specific* and not obvious even to domain experts, and many iterations of trial-and-error have to be implemented [8].

As a concrete example, let us consider a typical blocking process for the song records in Table 1. A user may start with Title as a blocking key, then realize it covers few true matches because of the prevalence of the unnormalized texts in titles. Next, the user may try various ways to clean the titles (such as removing all punctuation and version descriptions) and generate multiple customized blocking keys (such as using prefixes, suffixes and/or character/token n-grams of the titles). These attempts, however, need to be made incrementally, and usually cannot be applied altogether, since combining all of them often becomes overkill and results in an unaffordably large P/E ratio. Furthermore, the user typically has to replicate all these efforts with the other attributes, as Title alone cannot produce high enough recall. Yet other attributes may have their own issues, such as low coverage and extremely large frequencies of particular attribute values (e.g., "Bob Dylan" in Composer). Even worse, the user may need to manually recognize the correlation among a set of attributes, and create an aggregated attribute to assist the blocking (e.g., combining Composer and Song Writer).

Some non-key-based blocking methods (such as MinHash blocking [14]) can partially handle the unnormalization issue by supporting fuzzy-matching on attribute values. But these methods rely purely on lexical evidence, so they can still fall short on recall, or obtain reasonable recall but sacrifice P/E ratio, thus leading to a high comparison cost in the downstream matching step.

Therefore, the process of blocking on large-scale, unnormalized real-world data can be costly in human labor; even a well-educated domain expert often needs to spend *days or weeks* in order to achieve satisfactory blocking results.

Our Solution In this paper, we seek to build a general blocking approach that achieves high recall, low P/E ratio, scalability, and minimum human effort, simultaneously.

We begin with an intuition as follows: *if we had a good similarity metric $\sigma(\cdot,\cdot)$ for quantifying the similarity of any record pair, and could afford to apply the metric σ to all possible pairs in the data source, blocking would be done by simply retrieving the nearest neighbors (NNs) for each record.* However, substantiating this scheme is rather challenging, since a good metric σ for blocking is usually unknown a priori, and finding NNs is inefficient for most non-trivial σ's. Two design questions thus arise naturally:

(Q1) Similarity Metric: *Is it possible to automatically identify a good similarity metric for blocking*?

(Q2) Fast NN Search: *Given the identified, potentially non-trivial similarity metric, can we find nearest neighbors for each record efficiently*?

To answer these two questions, we propose `AutoBlock`, a hands-off blocking framework on tabular records (tuples). To automatically identify a good similarity metric, `AutoBlock` utilizes a set of pairwise labels that indicates which record pairs are matched, and learns a neural network architecture that produces, for similar tuple pairs, similar real-valued representations (named *signatures*), measured under some standard metric. Thus, a similarity metric σ for tuples is implicitly learned as the composition of the signature function (the neural network architecture) and the standard similarity metric for signatures. To further enable efficient approximate NN search, we choose the metric for signatures to be cosine and apply cross-polytope locality-sensitive hashing (LSH) [2]—a theoretically optimal LSH family for cosine similarity—to retrieve the NNs for each tuple in sublinear time.

Contributions We now underscore our main contributions:

- **Automation**: We propose a novel hands-off blocking framework, `AutoBlock`, that frees users from the tedious and laborious processes of data cleaning, and designing and tuning blocking keys.
- **Scalability**: We show that `AutoBlock` has a sub-quadratic total time complexity for generating the candidate pairs for all tuples and thus can be easily deployed for millions of tuples.
- **Effectiveness**: We evaluate `AutoBlock` on multiple large-scale, real-world datasets of various domains, and show that our method outperforms a wide range of competitive baselines on dirty and unstructured datasets, with minimum human effort involved.

The rest of the paper is organized as follows. We start with notation and problem definition in Section 2. Then we further elaborate our intuition—blocking as NN search—and give an overview of the architecture of `AutoBlock` in Section 3. We formally present the five major steps of `AutoBlock` in Section 4. Section 5 shows our experimental results, and Section 6 discusses the related work. Finally, we conclude and list several future directions in Section 7.

2 PROBLEM DEFINITION

Suppose our dataset consists of n tuples, and each tuple has m attributes. We denote the i-th tuple by $\mathbf{x}_i \triangleq [\mathbf{a}_{i1}, \mathbf{a}_{i2}, \ldots, \mathbf{a}_{im}]$, where $\mathbf{a}_{ij}$ is the j-th attribute value for $\mathbf{x}_i$ and $\triangleq$ stands for "is defined as." We use $[n]$ as a shorthand for the set $\{1, 2, \ldots, n\}$. In this way, each attribute value $\mathbf{a}_{ij}$ can be represented as a sequence of tokens $[w_{ijk}]_{k=1}^{l_{ij}}$, where l_{ij} is the sequence length for $\mathbf{a}_{ij}$ and w_{ijk} is the k-th token. We assume that all tokens are drawn from a unified vocabulary $\mathcal{V}$. We emphasize two important properties of the vocabulary $\mathcal{V}$ for unnormalized text: (a) openness—$\mathcal{V}$ can

contain out-of-vocabulary tokens and have infinite cardinality; and (b) prevalence of missing values—many l_{ij}'s can be zero.

We now give a formal definition of blocking as follows.

Definition 1 (Blocking). Given a data source $\mathbf{X} \triangleq [\mathbf{x}_1, \ldots, \mathbf{x}_n]$ of n tuples, *blocking* outputs a subset of candidate pairs $C \subseteq [n] \times [n]$, such that for any $(i, i') \in C$, tuple x_i and tuple $x_{i'}$ are likely to refer to same entity.

Remark. As noted in the definition, high recall is the foremost requirement for blocking; nevertheless, it is also important to control the size of C to achieve the goal of prescreening for matching.

In addition to the tuple set $\mathbf{X}$, we also assume that we are able to access a *positive label set* $\mathcal{L} \subseteq [n] \times [n]$, such that $(\mathbf{x}_i, \mathbf{x}_{i'})$ is a match for all $(i, i') \in \mathcal{L}$. These positive labels can be generated with certain strong keys whenever available (UPC code for grocery products, ISBN numbers for books, SSN for residents, etc.), or obtained by manual annotation. Note that this $\mathcal{L}$ can be reused for training the downstream matching algorithm, which requires collecting positive labels anyway; thus requiring such $\mathcal{L}$ in blocking does not incur additional human effort.

3 BEYOND BLOCKING: NEAREST NEIGHBOR SEARCH

We hypothesize that if there is a perfect similarity metric, and efficiency is not a concern, blocking can be achieved by NN search. We refer to this scheme as *NN blocking* and specify it in Algorithm 1.

Algorithm 1: Nearest neighbor (NN) blocking

Input: tuple set $\mathbf{X}$, a similarity metric $\sigma(\cdot, \cdot)$, and threshold θ
Output: candidate pairs C

$C := \emptyset$;
for $i = 1, \ldots, n$ **do**
  $C := C \cup \{(i, i') | \sigma(\mathbf{x}_i, \mathbf{x}_{i'}) > \theta, \forall i' < i\}$;

In fact, a wide range of existing blocking methods can be viewed as special cases of NN blocking, with their own similarity metrics. Behind the traditional key-based blocking methods, for example, are binary similarity metrics (row 2–5 in Table 2). This observation exposes the key reason why these methods are susceptible to unnormalized data: their similarity metrics rely on exact string matching and are too coarse-grained.

Another example is MinHash blocking, based on set-based similarity (row 6 in Table 2). MinHash blocking first converts each tuple into a set of representative pieces that typically comprise individual tokens and token n-grams, then measures the similarity of tuples based on their set representations. Jaccard similarity and its LSH family—MinHash—are used to provide efficient approximate NN search. However, Jaccard similarity only captures the lexical similarity between tuples and thus can be suboptimal for difficult domains where syntactic or semantic similarity is required.

Overview of Our Architecture Our framework `AutoBlock` follows the scheme of NN blocking and leverages a positive label set to implicitly learn the similarity metric. The overall architecture of `AutoBlock` comprises five steps, as illustrated in Figure 1, in which *steps (1)–(4) together form our solution for design question* ***Q1*** *and step (5) is our solution for* ***Q2***. We briefly describe the five steps below and explain them in details in the next section.

(1) **Token embedding**: A word-embedding model transforms each token to a token embedding (Section 4.1).
(2) **Attribute embedding**: For each attribute value of a tuple, an attention-based neural network encoder converts the input sequence of token embeddings to an attribute embedding (Sectio 4.2).
(3) **Tuple signature**: Multiple signature functions combine the attribute embeddings of each tuple and produce multiple tuple signatures (one per signature function) (Sectio 4.3).
(4) **Model training**: Equipped with the positive label set, the model is trained with an objective that maximizes the differences of the cosine similarities between the tuple signatures of matched pairs and between unmatched pairs (Section 4.4).
(5) **Fast NN search**: The learned model is applied to compute the signatures for all tuples, and an LSH family for cosine similarity is used to retrieve the nearest neighbors for each tuple to generate candidate pairs for blocking (Section 4.5).

Table 2: Common blocking methods and their corresponding similarity metrics

Method	Key (Example)	Similarity Metric
Single Key	$f(x) = x.\text{title}$	$\mathbb{I}(f(x) = f(y))$
Conjunctive Key [4, 17]	$f(x) = (x.\text{title}, x.\text{album})$	$\mathbb{I}(f(x) = f(y))$
Disjunctive Key [4, 17]	$f_1(x) = x.\text{title}$ $f_2(x) = x.\text{album}$	$\mathbb{I}(f_1(x) = f_1(y))$ $\vee\, \mathbb{I}(f_2(x) = f_2(y))$
Customized Key [1, 11, 20]	$f(x) = \text{firstTwoToken}(x.\text{title})$	$\mathbb{I}(f(x) = f(y))$
MinHash [14]	$f(x) = \text{nGrams}(x.\text{title})$	$\text{JaccardSim}(f(x), f(y))$

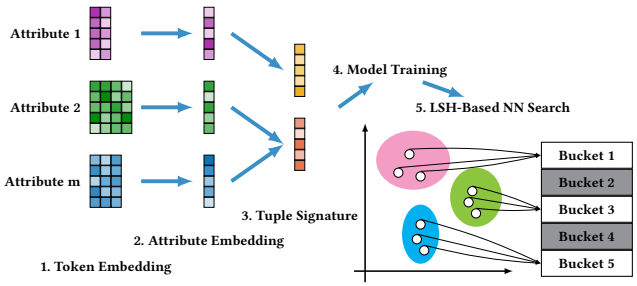


Figure 1: Overall architecture of `AutoBlock`

4 PROPOSED: `AUTOBLOCK`

In this section, we present the five steps of `AutoBlock` in details.

4.1 Token Embedding

The first step of `AutoBlock` is to convert each token into a low-dimensional embedding vector using a word embedding model. We use fastText [5] to obtain embeddings for tokens. Unlike other word embedding models [18, 25] that learn a distinct embedding vector for each word, fastText learns embeddings for character n-grams and computes the embedding for a word as the sum of the embeddings of all n-grams appeared in that word. As a result, fastText

can naturally handle rare tokens, whereas other word embedding models often regard these tokens as out-of-vocabulary tokens and replace them with a special token such as "UNK". We have emperically observed that fastText is more robust than alternative methods to common typos and misspelling, and can produce similar embeddings for homomorphically similar tokens. As a result, we choose fastText as our way to convert tokens to token embeddings.

4.2 Attention-based Attribute Embedding

The second step of `AutoBlock` takes the sequence of token embeddings for each attribute as input and outputs an embedding that encodes the information of that attribute. This step is related to phrase/sentence embedding learning in NLP; but the major challenges are that the nature of different attributes varies regarding their length, word choice, and usage, and that the sequential order of sentences in natural language is missing in tabular data.

We propose an attention-based attribute encoder (called *attentional encoder* for short) for computing attribute embeddings. The main idea behind attentional encoders is *averaging*—the embedding of an attribute is represented by a weighted average of its token embeddings. But rather than fixing a weight for each token a priori, the attentional encoder learns the weight for each token depending on its semantics, position, and surrounding tokens in the input token sequence. This capability is especially useful when attributes are long and exhibit clear structural patterns. For example, the extra version descriptions in the song titles often (but not always) appear at the end of the titles and are enclosed by parenthesis or square brackets (see tuple 2 in Table 1). Given enough positive pairs in the training data in which one member of the pair has such a version description but the other does not, the attentional encoder is able to recognize such patterns and pay less "attention" (i.e., assigning lower weights) to the tokens that form the version description at the end of music title.

Formally, let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_l \in \mathbb{R}^d$ be the sequence of token embeddings, where l is the sequence length and d is the dimension of token embeddings. An attentional encoder computes the weights of the input tokens as follows:

$$\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_l = \text{SeqEnc}(\mathbf{v}_1, \ldots, \mathbf{v}_l), \tag{1}$$

$$\alpha_1, \alpha_2, \ldots, \alpha_l = \text{SoftMax}(\mathbf{w}^T\mathbf{h}_1, \ldots, \mathbf{w}^T\mathbf{h}_l), \tag{2}$$

$$\beta_k = \rho\alpha_k + (1-\rho)\frac{1}{l}, \quad \forall k \in [l]. \tag{3}$$

Here, SeqEnc$(\cdot)$ can be any neural network architecture that takes a sequential input and generates an output for every input position. Possible choices include the standard recurrent neural network (RNN), bidirectional long short-term memory network (Bi-LSTM), 1D convolutional neural network, and transformers [27]. The hidden states are then transformed into *attention weights* in (2), which are further smoothed with uninformative weight $1/l$ in (3), controlled by a hyper-parameter $\rho \in [0, 1]$. Finally, the attribute embedding is defined as

$$g(\mathbf{v}_1, \ldots, \mathbf{v}_l) \triangleq \sum_{k=1}^{l} \beta_k \mathbf{v}_k. \tag{4}$$

4.3 Tuple Signature

In the third step of `AutoBlock`, we would like to combine the attribute embeddings and generate representations at the tuple level, such that representations for matched tuples have large cosine similarity. Before a deep dive into which model to use, let us first consider a more fundamental question: *what would happen if we compress the information in a tuple into a single representation for blocking*?

Example 4.1. Consider three tuples for the same song with attributes on Title, Album, and Composer:

$\mathbf{x}_1$ = (Me and Mrs. Jones, $\emptyset$, $\emptyset$),

$\mathbf{x}_2$ = (Me and Mrs. Jones, Call Me Irresponsible, Michael Bublé),

$\mathbf{x}_3$ = (Me & Mrs., Call Me Irresponsible, Michael Bublé),

where $\emptyset$ denotes missing value. Intuitively, the embeddings need to be dominated by Title in order to ensure $emb(\mathbf{x}_1) \approx emb(\mathbf{x}_2)$. This would, however, imply that the similarity between $emb(\mathbf{x}_1)$ and $emb(\mathbf{x}_3)$ is not high (as $\mathbf{x}_1$ and $\mathbf{x}_3$ differ on Title).

This example indicates that when tuples contain a wide variety of attributes and can possibly have many missing values, representing each tuple with only *one* embedding vector would result in low similarity for certain positive pairs. Consequently, one would have to lower the similarity threshold θ in order to retrieve pairs such as both $(\mathbf{x}_1, \mathbf{x}_2)$ and $(\mathbf{x}_2, \mathbf{x}_3)$. A small θ, however, would also incur many false positive pairs, making the P/E ratio unaffordably large.

To address this issue, we propose to generate *multiple signatures* such that each signature only captures a partial, distinct aspect for tuples, and two tuples are considered similar (and thus regarded as a candidate pair for blocking) as long as they are similar for one signature. With this design, we are able to overcome the issue in Example 4.1, as shown in the next example.

Example 4.2. Continue Example 4.1. Now suppose we have two signature functions $sig_1(\cdot)$ and $sig_2(\cdot)$ applied on Title, and on Album and Composer, respectively. Then we will have $sig_1(\mathbf{x}_1) = sig_1(\mathbf{x}_2)$, $sig_2(\mathbf{x}_2) = sig_2(\mathbf{x}_3)$. Thus, regardless that $sig_1(\mathbf{x}_1) \neq sig_1(\mathbf{x}_3)$ and $sig_2(\mathbf{x}_1) \neq sig_2(\mathbf{x}_3)$, the two candidate pairs $(\mathbf{x}_1, \mathbf{x}_2)$ and $(\mathbf{x}_2, \mathbf{x}_3)$ can still be retrieved with large threshold α by $sig_1(\cdot)$ and $sig_2(\cdot)$, respectively.

Formally, let $\mathbf{g}_1, \ldots, \mathbf{g}_m \in \{\emptyset\} \cup \mathbb{R}^d$ denote the embeddings of the m attributes of a tuple. We define the s-th signature function to be a weighted average over non-missing attributes, i.e.,

$$f^{(s)}(\mathbf{g}_1, \ldots, \mathbf{g}_m) \triangleq \sum_{j=1}^{m} \mathbb{I}(\mathbf{g}_j \neq \emptyset) w_{sj} \mathbf{g}_j, \tag{5}$$

where $\mathbf{w}_s \triangleq [w_{sj}]_{j=1}^m \geq \mathbf{0}$ is a nonnegative weight to be estimated, $\mathbb{I}(\cdot)$ is the indicator function, and $f^{(s)}$ is set to be $\emptyset$ when $\mathbb{I}(\mathbf{g}_j \neq \emptyset) w_j$ is zero for all $j \in [m]$. Given S such signature functions $\{f^{(s)}(\cdot)\}_{s=1}^S$, and denoting the signature computed by $f^{(s)}(\cdot)$ for tuple $\mathbf{x}_i$ by $\mathbf{f}_i^{(s)}$, the final similarity metric used in `AutoBlock` is the maximum cosine similarity over S pairs of signatures, namely

$$\sigma(\mathbf{x}_i, \mathbf{x}_{i'}) \triangleq \max_{s=1,\ldots,S} \cos\left(\mathbf{f}_i^{(s)}, \mathbf{f}_{i'}^{(s)}\right), \tag{6}$$

where $\cos(\mathbf{f}, \mathbf{f}') \triangleq \langle \mathbf{f}, \mathbf{f}' \rangle / (\|\mathbf{f}\|_2 \cdot \|\mathbf{f}'\|_2)$ is the cosine similarity, and we set the cosine similarity to zero if either $\mathbf{f}$ or $\mathbf{f}'$ is $\emptyset$. Note that the cosine similarity is scale-invariant; we thus require $\|\mathbf{w}_s\|_2 = 1$ for all $s \in [S]$ without loss of generality.

4.4 Model Training

Now we describe how the proposed attentional encoders and signature functions are trained with the given positive label sets $\mathcal{L}$. Our training algorithm is based on the following idea: *for any* $(i, i') \in \mathcal{L}$, *the tuple pair* $(\mathbf{x}_i, \mathbf{x}_{i'})$ *should be more similar than pairs* $(\mathbf{x}_i, \star)$ *and* $(\mathbf{x}_{i'}, \star)$, *where* $\star$ *denotes an irrelevant tuple.*

Embracing this intuition we design an auxiliary multi-class classification task for training. Specifically, for each positive pair $(i, i) \in \mathcal{L}$, we randomly sample a small set of indices $U_{i,i'} \subset [n] \setminus \{i, i'\}$. Since typically $n \gg |U_{i,i'}|$ and duplicates in $\mathbf{X}$ are rare, one can reliably assume that the tuples corresponding to $U_{i,i'}$ are irrelevant to $\mathbf{x}_i$ and $\mathbf{x}_{i'}$. Then given signature function $f^{(s)}$, the probability of choosing $(\mathbf{x}_i, \mathbf{x}_{i'})$ to be the only positive pair among all $2|U_{i,i'}| + 1$ pairs from $\{i, i'\} \cup U_{i,i'}$ that involve $\mathbf{x}_i$ or $\mathbf{x}_{i'}$ is defined as

$$P_s\left[(\mathbf{x}_i, \mathbf{x}_{i'}); U_{i,i'}\right] \triangleq \frac{e^{\sigma(\mathbf{f}_i^{(s)}, \mathbf{f}_{i'}^{(s)})}}{e^{\sigma(\mathbf{f}_i^{(s)}, \mathbf{f}_{i'}^{(s)})} + \sum_{j \in U_{i,i'}} \left[e^{\sigma(\mathbf{f}_i^{(s)}, \mathbf{f}_j^{(s)})} + e^{\sigma(\mathbf{f}_{i'}^{(s)}, \mathbf{f}_j^{(s)})}\right]}. \quad (7)$$

The attentional encoders $\{g^{(j)}\}_{j=1}^m$ and signature function weights $\{\mathbf{w}_s\}_{s=1}^S$ can thus be learned end-to-end by maximizing the summed log-probability over all signatures and all positive pairs, namely

$$\max_{\{g^{(j)}\}_{j=1}^m, \{\mathbf{w}_s\}_{s=1}^S} \frac{1}{|\mathcal{L}|} \sum_{(i,i') \in \mathcal{L}} \sum_{s=1}^{S} \log P_s\left[(\mathbf{x}_i, \mathbf{x}_{i'}); U_{i,i'}\right] \quad (8)$$

We apply Adam, a variant of stochastic gradient descent (SGD) algorithms, to optimize the objective. After each update, we further project all signature weights into the feasible region to ensure non-negativity and unit norm.

The optimization problem defined in (8), however, does not impose any regularization on signature weights and thus may end up with S identical, individually optimal signature functions. Ideally signatures should be independent, or orthogonal: $\mathbf{W}^T\mathbf{W} = \mathbf{I}_S$, so that each signature reflects a distinct aspect of tuples. Thus we could incorporate into the optimization problem a penalty such as $\left\|\mathbf{W}^T\mathbf{W} - \mathbf{I}_S\right\|$, or use augmented Lagrangian methods [19]. Nevertheless, when the optimization problem is situated into a larger task as here, tuning the penalty coefficient or related hyperparameters becomes unwieldy and impractical.

We instead propose a simple sequential algorithm to achieve orthogonality. The main idea is that signature functions are trained one at a time, and when training the current signature function, all attributes used (i.e., associated with positive weights) by the previously identified signature functions are marked as unusable. In this way, the attributes used by different signature functions will not overlap and thus satisfy orthogonality naturally. Eventually, the algorithm terminates when either all attributes have been used, or S signature functions have been learned. In fact, this sequential algorithm not only eliminates the need for introducing new hyperparameters (as required by standard constrained optimization algorithms), it may also eliminate the need for tuning S: one could set the initial value of $\bar{S}$ as large as m and let the sequential algorithm end up with an appropriate S.

Algorithm 2: Model Training

Input: tuple set $\mathbf{X}$, label set $\mathcal{L}$, maximum # of iteration T, and maximum # of signatures $\bar{S}$.
Output: attribute encoders $\{g^{(j)}\}_{j=1}^m$ and signature function weights $\{\mathbf{w}_s\}_{s=1}^S$

```
initialize {g^(j)}_{j=1}^m and {w_s}_{s=1}^S;
initialize attribute set M := {1, ..., m};
for s = 1, ..., S̄ do
    for t = 1, ..., T do
        sample a mini-batch L_B ⊂ L;
        sample a U_{i,i'} for each (i, i') ∈ L_B;
        update {g^(j), w_sj}_{j∈M} to improve
          (1/|L_B|) Σ_{(i,i')∈L_B} log P_s [(x_i, x_i'); U_{i,i'}] ;
        project w_s to {w ∈ R^m_{≥0} | w_M = 0, ||w||_2 = 1};
    M := M \ {j ∈ [m] | w_sj > 0};
    if M = ∅ then
        S := s; break;
```

Algorithm 2 sketches the final training procedure.

4.5 Fast NN Search

In the last step of `AutoBlock`, our goal is to efficiently retrieve, for each query tuple, the nearest neighbors whose similarities to the query are above a specified threshold $\theta > 0$ according to the metric σ defined by (6) with the computed signatures. Note that σ in (6) takes a maximum form; thus we can conduct NN search for each signature function with threshold θ separately, then take the union of all candidate pairs found for each signature function (followed by a de-duplication step) as the final candidate pairs.

So the task is reduced to a classic, high-dimensional NN search problem with cosine similarity. We choose to solve this problem with locality-sensitive hashing (LSH), an effective technique for this problem that offers *provable* sublinear query time. Specifically, we apply cross-polytope LSH, a state-of-the-art LSH family for cosine similarity that not only enjoys the theoretically optimal query time complexity but also allows an efficient implementation.

With cross-polytope LSH, one can prove that the NN search problem can be approximately solved in a sublinear query time, as shown in Theorem 4.3.

Theorem 4.3. *Given an n-point dataset $\mathbf{X} \subset \mathbb{R}^d$, there exists an algorithm based on cross-polytope LSH satisfying: for any query $\mathbf{x}$ and similarity thresholds $-1 < \theta' < \theta < 1$, if there exists a point $\mathbf{x}^* \in \mathbf{X}$ such that $\cos(x, x^*) \geq \theta$, the algorithm will with probability at least $1 - \varepsilon^K$ retrieve a point $\mathbf{x}' \in \mathbf{X}$ with $\cos(x, x') \geq \theta'$ in query time $O(K \cdot d \cdot n^\rho)$, where $\varepsilon < \frac{1}{3} + \frac{1}{e}$ and $\rho = \frac{1-\theta}{1-\theta'} \cdot \frac{1+\theta'}{1+\theta} + o(1)$.*

Proof. The proof is in the Appendix. □

Careful readers may have noticed that Theorem 4.3 only guarantees the query time complexity for approximate instead of exact

NN search. So why is a technique for approximate NN search sufficient in our case? This is because by learning multiple signature functions, each of which focuses on only a particular aspect of a tuple, we empirically observe that resultant signatures are fairly similar for most positive pairs with similarities rarely falling below 0.8. In contrast, the similarities of most random pairs center around 0.2 and seldom exceed 0.4. That means in our case, we can afford to set $\theta = 0.8$ and $\theta' = 0.4$ without incurring too many false positives. This would correspond to a sublinear query time complexity $O(K \cdot d \cdot n^{1/3.86})$. The empirical evaluation in Section 5.4 further supports the effectiveness and scalability of the cross-polytope LSH.

5 EMPIRICAL EVALUATION

In this section, we empirically compare `AutoBlock` against an array of competitive baselines on three large-scale, real-world datasets.

5.1 Experiment Setup

Datasets We consider three real-world datasets: `Movie`, `Music`, and `Grocery`, crawled and sampled from various public websites (see Table 3). `Music` is from Amazon and Wikipedia, `Movie` from IMDb and WikiData, and `Grocery` from Amazon and ShopFoodEx (an online grocery store). The three datasets represent three distinct dataset types in the entity matching problem:

(1) *Clean* (`Movie`): Attributes are properly aligned and their values are relatively clean, i.e., each attribute rarely contains irrelevant information.
(2) *Dirty* (`Music`): Certain attributes such as Title may contain significant amount of irrelevant information. In addition, attributes are imperfectly aligned and thus attribute values may be misplaced at the wrong attributes (see Composer and Song Writer in Table 1).
(3) *Unstructured* (`Grocery`): Records are unstructured; that is, all information is mixed into a raw, relatively long, textual attribute.

Table 3: Three real-world datasets for our experiments

Dataset	Type	Table A	Table B	# Pos.	# Attr.
`Movie`	Structured	465,893	202,162	135,275	8
`Music`	Dirty	2,190,080	105,446	2,298	7
`Grocery`	Unstructured	1,292,848	5,886	4,437	1

Positive Label Generation For all three datasets we generate positive labels using the available strong keys. They are tconst (an alphanumeric unique identifier) for `Movie`, ASIN (Amazon Standard Identification Number) for `Music`, and UPC code for `Grocery`. Note that not every record in the three datasets has such a strong key. As these strong keys are used for constructing the positive labels for both training and evaluation, we exclude them from the attribute set to avoid overfitting. [1]

Training/Test Set Split First, we randomly divide the positive labels into two parts, 80% for training and 20% for testing, and ensure that tuples sharing the same strong keys are put into the same part. Then we create a training set that includes all tuples appearing in the training labels. We also add to the training set 20% of the tuples that do not appear in any positive labels; they serve as irrelevant tuples to facilitate the training. A test set is created in a similar manner, which includes all the tuples that appear in the test labels, as well as all remaining tuples. We repeat this procedure and create five training/test set pairs for each dataset.

Methods for Comparison We compare our method to a wide range of competitive baselines:

- Key-based blocking: Two blocking choices are considered, i.e., single key (only Title is used as blocking key), disjunctive key (all attributes are used as blocking keys).
- MinHash blocking: This method retrieves all pairs whose Jaccard similarities on a particular attribute are above θ as candidate pairs. All attributes are considered, and θ is set to be 0.4, 0.6, or 0.8.
- DeepER [10]: As a state-of-the-art, DL-based method for blocking, this method can be viewed as a special case of `AutoBlock` by setting all attribute encoders to unweighted averaging and letting each attribute be a signature.

To understand the impact of the components of `AutoBlock`, we also include two sub-model baselines:

- Unweighted averaging encoder: It takes Title as the only signature and applies unweighted averaging to Title.
- Attentional encoder: It also takes Title as the only signature but applies the attentional encoder to Title.

`AutoBlock` Configuration We use the pretrained fastText model with the word embedding size $d = 300$. We choose the SeqEnc$(\cdot)$ module in our attentional encoder to be a single-layer Bi-LSTM with 64 hidden units, although we find our method is robust to the choice of neural network architecture and other factors affecting optimization such as batch size and initial learning rate. For each positive pair, we randomly sample $|U_{i,i'}| \equiv 10$ irrelevant tuples on the fly during training to construct the loss defined in (7). We set the maximum number of signatures S to be the number of attributes and let the Algorith 2 to determine the appropriate S. We set the attention smoothing parameter ρ to 1 for attribute Title and to 0 for other attributes. Finally, for NN search we set the similarity threshold $\theta = 0.8$ and limit the maximum number of retrieved NNs for each tuple to $\max(1000, \lfloor\sqrt{n_1}\rfloor)$, where n_1 is the size of the larger table.

Minimum Preprocessing For all methods, we only perform the same, minimum preprocessing to the datasets, as one of goals is to minimize human effort in blocking. In fact, the only preprocessing we use is to convert English letters to lowercase and tokenize attributes into token sequences using the standard TreeBank tokenizer [15]. All punctuation, stop-words, non-English characters, typos, abbreviations, and so forth are kept as they are.

Evaluation Metrics We evaluate the effectiveness of each blocking method with two metrics— recall and P/E ratio. Let $\mathcal{T} \subseteq [n] \times [n]$ be the unknown set of all true matched pairs, and recall that X is the tuple set and C is the set of candidate pairs. The two metrics are defined as follows:

$$recall \triangleq \frac{|C \cap \mathcal{T}|}{|\mathcal{T}|},$$

[1] In practice, one can always add those pairs matched on these strong keys to the candidate blocking pairs.

$$P/E\ ratio \triangleq \frac{|C|}{|\mathbf{X}|} = \frac{|C|}{n}.$$

The true label set $\mathcal{T}$, however, is never known beforehand; we therefore approximate it with the collected positive labels $\mathcal{L}$. We report the average performances on the five test sets for each dataset.

Additional Setups Due to the limit of space, we include additional experiment setup such as the implemention notes for our method and other baselines in the Appendix.

5.2 Effectiveness

We begin with investigating the effectiveness of AutoBlock. Table 4 summarizes the recall and P/E ratios of AutoBlock and other baseline methods on the three datasets. The results in the table support the following conclusions.

First, AutoBlock performs best overall, especially when datasets are dirty and/or unstructured. On Grocery, AutoBlock not only surpasses all baselines in recall by a substantial margin (18.3 percentage points, or 25.8%) but also attains the smallest P/E ratio. On Music, AutoBlock has higher recall than the leading baseline (MinHash with $\theta = 0.4$), and its P/E ratio is only 1/5 of the MinHash's P/E ratio. On Movie, AutoBlock achieves a close second recall, but its P/E ratio is about 20 times smaller than the best baseline (MinHash with $\theta = 0.4$).

Second, key-based blocking fails to attain high recall on all three datasets. This disadvantage is most evident on Grocery, where not a single positive pair exactly matches on Title, because the two data sources (i.e., Amazon and ShopFoodEx) differ in their ways of concatenating different aspects of a grocery product (e.g., brand name, package size, and flavor) into a single attribute. As a result, key-based methods are unable to retrieve any true positive pairs as candidate pairs (and thus have a recall of 0.0).

Third, MinHash requires a low similarity threshold to achieve high recall but at a cost of unaffordably large P/E ratio. In fact, only when $\theta = 0.4$ MinHash achieves comparable recalls to AutoBlock on Movie and Music, but its P/E is substantially larger than AutoBlock's; when θ increases, MinHash's recall drops significantly. This sensitivity to θ thus demands considerable amount of tuning in practice to achieve a balance between recall and P/E ratio. Moreover, MinHash's recall is still much lower than AutoBlock on Grocery even when $\theta = 0.4$, suggesting that Jaccard similarity, which leverages only lexical evidence, is less effective than the similarity metric learned by AutoBlock on this challenging domain.

Fourth, the attention mechanism contributes significantly to AutoBlock's recall gain. This is best seen from the comparison between attentional encoder and unweighted averaging encoder, where the former outperforms the latter on Music and Grocery by 16.9 and 18.3 percentage points, respectively. In addition, AutoBlock outperforms DeepER in recall on all three datasets, which further demonstrates the benefits of the attention mechanism.

Last but not least, learning multiple signatures further boosts the AutoBlock's recall. This is shown by the recall gain of AutoBlock over attentional encoder on Movie and Music.[2]

[2]Since Grocery only has one attribute, AutoBlock is the same as attentional encoder and thus both methods share identical performance on this dataset. This convergence also happens to DeepER and the unweighted average encoder.

5.3 Automation

We now explain how AutoBlock saves manual work but still obtains high recall.

One major source of the original manual work is concerned with how to iteratively try out different combinations of cleaning and blocking key customization strategies. This task is now alleviated by AutoBlock's ability to assign different weights to tokens through attentional encoders. Figure 2 visualizes the attention weights for the titles of sampled positive pairs on Music (Figure 2a) and Grocery (Figure 2b 2c). Several patterns stand out:

(1) The tokens at starting positions tend to enjoy large weights. This is consistent to our observation that positive pairs typically match on the first few tokens; these tokens may also encode important information such as brand in Grocery (e.g., "bertolli" in Figure 2b and "la choy" in Figure 2c).
(2) Common stop words (e.g., "the" and "and" in Figure 2a) and uninformative punctuation (e.g., most commas and periods) are properly ignored. This is also expected since these tokens are often irregularly injected into tuples and result in avoidable mismatches.
(3) The tokens in special positional relationship to "functional" punctuation marks (e.g., parenthesis) tend to get special attention. For example, the "digitally" and "remastered" in Figure 2a are surrounded by parenthesis and get small weights.
(4) Many discriminative tokens (e.g., those regarding the package size, and flavor in Grocery) are ignored. We were initially surprised at this because these tokens are usually useful in the downstream matching step. Later we realize that AutoBlock's choice may be reasonable because these tokens are often randomly missing or expressed in different forms; ignoring them in the blocking step avoids missing positive pairs.

Manual work is also saved by AutoBlock's ability to automatically combine different attributes to generate signatures. Figure 3 shows the learned signature weights for Music. The combination of Ablums and Performers in sig_2 is likely due to the fact that they are often matched or unmatched simultaneously, and therefore combining them would reduce the chance of collision and reduce the P/E ratio. The selection of Composer, Lyricist, and SongWriter by sig_3 allows an approximate cross-attribute matching among these three attributes, which is useful to handle the attribute misplacement cases in this dataset.

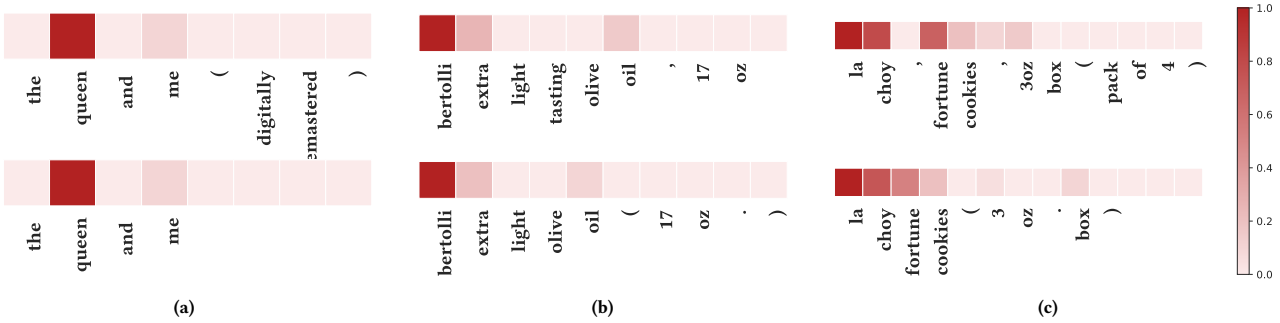
5.4 Scalability

Finally, we investigate the empirical performance of cross-polytope LSH. Figure 4 shows how much speedup of average query time (measured with single CPU core) cross-polytope LSH can achieve over brute force with different the number of points in the retrieval set. Huge speedup of cross-polytope over brute force is observed: when the number of points reaches 10^6, the former is about 40–80 times faster than the latter. In addition, the speedup improves sub-linearly as the number of points increase, which is expected since brute force has a strictly $O(n)$ query complexity, and so the speedup should scale $O(n^{1-\rho})$ where $\rho < 1$.

Theorem 4.3 suggests a potential recall loss of cross-polytope LSH over exact NN search, as the algorithm may not retrieve any neighboring points with a small probability, or the retrieved points

Table 4: Performance comparison of different methods on different datasets. The best and the second best recalls on each dataset are emboldened and italicized, respectively. `AutoBlock` achieves the highest recall on dirty (`Music`) and unstructured (`Grocery`) datasets, and a close second recall with a magnitude smaller P/E ratio on clean dataset (`Movie`).

	Movie		Music		Grocery	
Method	Recall	P/E ratio	Recall	P/E ratio	Recall	P/E ratio
Single Key (Title)	58.6	0.10	72.3	0.40	0.0	0.01
Disjunctive Key (All)	85.5	9.13	92.5	39.11	0.0	0.01
MinHash (All, $\theta = 0.8$)	86.1	9.24	89.5	38.52	0.8	0.00
MinHash (All, $\theta = 0.6$)	87.1	29.90	91.3	41.16	18.7	0.84
MinHash (All, $\theta = 0.4$)	**91.4**	2089.60	*96.2*	232.98	52.5	4.32
DeepER (All, $\theta = 0.8$)	90.7	107.52	92.9	41.62	*70.8*	1.02
Un. Avg. Enc. (Title, $\theta = 0.8$)	62.5	7.65	77.5	11.62	70.8	1.02
Atten. Enc. (Title, $\theta = 0.8$)	64.0	4.74	94.4	21.00	89.1	0.72
`AutoBlock` (All, $\theta = 0.8$)	*90.8*	105.06	**96.3**	48.80	**89.1**	0.72



(a) (b) (c)

Figure 2: Example attention weights on (a) `Music` and (b, c) `Grocery`. Important tokens are assigned to higher weights, representing by darker color.

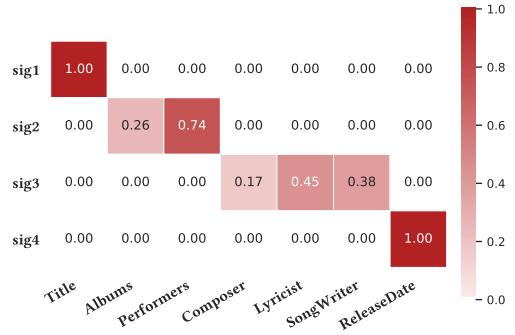


Figure 3: The learned signature weights on `Music`.

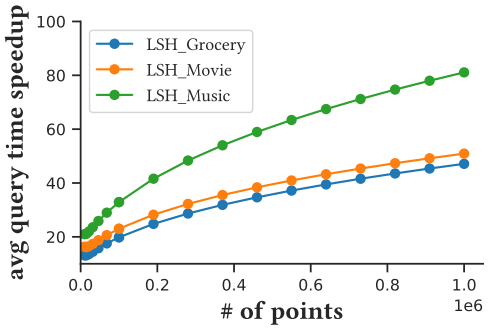


Figure 4: The average query time speedups achieved by cross-polytope LSH over brute force on different datasets. LSH is substantially faster than brute force, and the speedup improves as the number of points increases.

are all below the specified similarity threshold θ. We therefore investigate how much recall loss this step of NN search using LSH can incur. We experiment on `Grocery` data, because its Table B has many fewer tuples than the Table A, which allows us to conduct exact NN search by brute force. Table 5 shows that under various similarity thresholds, although there is a recall gap between the two methods, the gap is very small—the largest gap (when $\theta = 0.8$) is only 1.5%. We believe this is acceptable given the huge efficiency improvement of LSH upon brute force.

Table 5: Comparison of recall between brute force NN search and LSH-based NN search on Grocery. Only a minor recall gap of 0.89% on average is observed.

	Threshold θ			
Method	0.95	0.90	0.85	0.80
Brute force NN	81.4	85.8	88.9	90.4
LSH-based NN	81.1	85.2	88.0	89.1

6 RELATED WORK

As a critical step of entity matching, blocking has been extensively studied over the last several decades with numerous methods being proposed. For a comprehensive comparison of existing blocking methods, see [24]. Among others, key-based blocking methods [1, 11, 16, 20] are mostly used in practice yet require a lot of human effort. MinHash blocking [14] allows a fuzzy match on attributes, but often ends up with unaffordably many candidate pairs. The so-called "meta-blocking" [21–23, 26] tries to reduce the P/E ratio by introducing—between blocking and matching—extra steps to prune the candidate pairs; their contributions are orthogonal to our work. To the best of our knowledge, the recently proposed DeepER [10] is the most relevant work to ours and can be viewed as a special case of AutoBlock.

Locality-sensitive hashing is firstly proposed in the seminal work [13] for ℓ_p norm and later extended to other distance or similarity metrics. For cosine similarity, representative LSH schemes include hyperplane LSH [6], spherical LSH [3], and cross-polytope LSH [2], among others. While spherical LSH and cross-polytope LSH both attain the theoretically optimal query time complexity, only the latter can be efficiently implemented, as spherical LSH relies on rather complex hash functions that are very time-costly to evaluate.

7 CONCLUSION

We have proposed AutoBlock, a hands-off blocking framework for entity matching on tabular records, based on similarity-preserving representation learning and nearest neighbor search. Our contributions include: (a) **Automation**: AutoBlock frees users from tedious and laborious data cleaning and blocking key tuning. (b) **Scalability**: AutoBlock has a sub-quadratic total time complexity and can be easily deployed for millions of records. (c) **Effectiveness**: AutoBlock achieves superior performance on multiple large-scale, real-world datasets of various domains. One future direction would be to extend AutoBlock to datasets with non-textual attributes (e.g., image, audio, and video).

Acknowledgments We thank Xian Li, Tong Zhao, and Dongxu Zhang for the valuable discussions during the development of this work. We also thank the anonymous reviewers for their helpful feedback.

REFERENCES

[1] Aizawa, A. and Oyama, K. (2005). A Fast Linkage Detection Scheme for Multi-Source Information Integration. In *International Workshop on Challenges in Web Information Retrieval and Integration*, pages 30–39. IEEE.

[2] Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I. P., and Schmidt, L. (2015). Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing System*, pages 1225–1233.

[3] Andoni, A. and Razenshteyn, I. P. (2015). Optimal Data-Dependent Hashing for Approximate Near Neighbors. In *STOC*, pages 793–801, New York, New York, USA. ACM, ACM Press.

[4] Bilenko, M., Kamath, B., Mooney, R. J., View, M., and Mooney, R. J. (2006). Adaptive Blocking: Learning to Scale Up Record Linkage. In *Proceedings of IEEE International Conference on Data Mining*, pages 87–96. IEEE, IEEE.

[5] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

[6] Charikar, M. S. (2002). Similarity Estimation Techniques from Rounding Algorithms. In Reif, J. H., editor, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 380–388. ACM, ACM.

[7] Christen, P. (2012). A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555.

[8] Doan, A., Ardalan, A., Ballard, J. R., Das, S., Govind, Y., Konda, P., Li, H., Mudgal, S., Paulson, E., C., P. S. G., and Zhang, H. (2017). Human-in-the-Loop Challenges for Entity Matching: A Midterm Report. In Binnig, C., Hellerstein, J. M., and Parameswaran, A. G., editors, *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, pages 12:1—-12:6, New York, New York, USA. ACM.

[9] Dong, X. L. and Srivastava, D. (2013). Big data integration. *Proceedings of the VLDB Endowment*, pages 1245–1248.

[10] Ebraheem, M., Thirumuruganathan, S., Joty, S., Ouzzani, M., and Tang, N. (2018). Distributed Representations of Tuples for Entity Resolution. In *Proceedings of the VLDB Endowment*, pages 1454–1467. VLDB Endowment.

[11] Gravano, L., Jagadish, H. V., Ipeirotis, P. G., Srivastava, D., Koudas, N., and Muthukrishnan, S. (2003). Approximate String Joins in a Database (Almost) for Free–Erratum. Technical report, Department of Computer Science, Columbia University.

[12] Har-Peled, S., Indyk, P., and Motwani, R. (2012). Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality. *Theory of Computing*, 8(1):321–350.

[13] Indyk, P. and Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th annual ACM symposium on Theory of computing*, pages 604–613. ACM.

[14] Liang, H., Wang, Y., Christen, P., and Gayler, R. (2014). Noise-tolerant Approximate Blocking for Dynamic Real-time Entity Resolution. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 8444, pages 449–460. Springer.

[15] Loper, E., Klein, E., and Bird, S. (2009). *Natural Language Processing with Python*. O'Reilly.

[16] McNeill, W. P., Kardes, H., and Borthwick, A. (2012). Dynamic Record Blocking: Efficient Linking of Massive Databases in MapReduce. In *Quality in Databases*.

[17] Michelson, M. and Knoblock, C. a. (2006). Learning Blocking Schemes for Record Linkage . In *Proceedings of the 21st national conference on Artificial intelligence*, pages 2965–2967.

[18] Mikolov, T., Corrado, G., Chen, K., and Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *arXiv preprint*, 2015-Janua(3):1–12.

[19] Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer-Verlag New York, 2 edition.

[20] Papadakis, G., Ioannou, E., Palpanas, T., Niederee, C., Nejdl, W., Niederé, C., Nejdl, W., Niederee, C., and Nejdl, W. (2013). A Blocking Framework for Entity Resolution in Highly Heterogeneous Information Spaces. *IEEE Transactions on Knowledge and Data Engineering*, 25(12):2665–2682.

[21] Papadakis, G., Koutrika, G., Palpanas, T., and Nejdl, W. (2014a). Meta-blocking: Taking entity resolutionto the next level. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):1946–1960.

[22] Papadakis, G., Papastefanatos, G., and Koutrika, G. (2014b). Supervised meta-blocking. *Proceedings of the VLDB Endowment*, 7(14):1929–1940.

[23] Papadakis, G., Papastefanatos, G., Palpanas, T., and Koubarakis, M. (2016a). Boosting the Efficiency of Large-Scale Entity Resolution with Enhanced Meta-Blocking. *Big Data Research*, 6:43–63.

[24] Papadakis, G., Svirsky, J., Gal, A., and Palpanas, T. (2016b). Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. In *Proceedings of the VLDB Endowment*, volume 9, pages 684–695. VLDB Endowment.

[25] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

[26] Simonini, G., Bergamaschi, S., and Jagadish, H. V. (2016). BLAST: A loosely schema-aware metablocking approach for entity resolution. *Proceedings of the VLDB Endowment*.

[27] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is All you Need. In *Advances in Neural Information Processing System*.

A ADDITIONAL EXPERIMENT DETAILS

A.1 Dataset

The available attributes for each dataset are listed as follows:

- `Movie`: Title, Description, Genres, Director, MusicComposer, Playwright, Characters, and Actors.
- `Music`: Title, Albums, Performers, Composer, Lyricist, SongWriter, and ReleaseDate.
- `Grocery`: Title.

For non-textual attributes (e.g., dates), we convert them into their text representations; for set-valued attributes (e.g., Actors), we concatenate the textual representation of all their elements.

A.2 Implementation Details for `AutoBlock`

We implement our model using `PyTorch`. Different signatures share the same set of attribute encoders, but different attribute encoders have their own parameters.

In evaluating (7) it is possible that the signature $f^{(s)}$ may not be applicable to some tuples; that is, these tuples have missing values on all the attributes that correspond to the positive weights of $f^{(s)}$. If it happens to the sampled irrelevant tuples, i.e., for some $j \in U_{i,i'}$, we exclude j from $U_{i,i'}$. But if the non-applicable tuple is either $\mathbf{x}_i$ or $\mathbf{x}_{i'}$, then we remove the positive pair (i, i') from the mini-batch $\mathcal{L}_B$. This encourages the learned signature functions to maximize the similarity gap between positive pairs and irrelevant pairs rather than to maximize the coverage of the signature functions.

The step of NN search with cross-polytope LSH is implemented with package `FALCONN`[3]. We build hash tables with the package's default configuration for $n = 10^6$ points and dimension $p = 300$, i.e., there are $K = 10$ hash tables, and each table consists of $B = 2$ hash function. We multi-probe one additional bucket per table; that is, for each table, not only the points in the query's sitting bucket but also the points in the bucket that is closest to the query's sitting bucket are retrieved.

A.3 Implementation Details for Baselines

Our implementation of MinHash is from package `datasketch`.[4] There are $K = 32$ hash tables and each table consists of $B = 4$ hash functions.

A.4 Platform and Total Runtime

All experiments were conducted on a server with a 16-core CPU at 2.00Ghz and 128G memory. The total runtime for `AutoBlock`, from computing signatures to outputing final candidate pairs, is less than 0.5 hour for each dataset.

B PROOF OF THEOREM 4.3

Proof. Let $\mathcal{S}^{p-1}$ be the unit sphere in $\mathbb{R}^p$, i.e., $\mathcal{S}^{p-1} \triangleq \{\mathbf{x} \in \mathbb{R}^p | \, \|x\|_2 = 1\}$. Since cosine similarity is scale-invariant, we can project points onto $\mathcal{S}^{p-1}$ without changing the cosine similarity among them. Hence, we can assume that $\mathbf{X} \subset \mathcal{S}^{p-1}$ without loss of generality.

The Corollary 1 in [2], together with Theorem 3.4 in [12], establishes that given an n-point dataset $\mathbf{X} \subset \mathcal{S}^{p-1}$, there exists an algorithm based on hash tables built with cross-polytope LSH satisfying: for any query $\mathbf{x}$, Euclidean distance threshold $r > 0$, and approximation factor $c > 1$, if there exists a point $\mathbf{x}^* \in \mathbf{X}$ such that $\|x - x^*\|_2 < r$, the algorithm will with success probability at least $1 - \varepsilon$ retrieve a point $\mathbf{x}' \in \mathbf{X}$ with $\|x - x'\|_2 < cr$ in query time $O(d \cdot n^\rho)$, where $\varepsilon < \frac{1}{3} + \frac{1}{e}$ and

$$\rho = \frac{1}{c^2} \cdot \frac{4 - c^2 r^2}{4 - r^2} + o(1). \tag{9}$$

Thus, when K independent copies of such hash tables are built, the success probability improves to at least $1 - \varepsilon^K$, yet the query time complexity also increases to $O(K \cdot d \cdot n^\rho)$.

Note that there is a one-to-one mapping between the cosine similarity and Euclidean distance for points on $\mathcal{S}^{p-1}$, i.e., $\cos(x, x') = 1 - \frac{1}{2}\|x - x'\|_2^2$. Plug $r = \sqrt{2 - 2\theta}$ and $c = \sqrt{(1 - \theta')/(1 - \theta)}$ in (9), we get the result in Theorem 4.3.

□

[3] https://falconn-lib.org/

[4] https://github.com/ekzhu/datasketch