Unicorn: A Unified Multi-tasking Model for Supporting Matching Tasks in Data Integration

JIANHONG TU, Renmin University of China, China
JU FAN∗, Renmin University of China, China
NAN TANG, QCRI / HKUST (GZ), Qatar / China
PENG WANG, Renmin University of China, China
GUOLIANG LI, Tsinghua University, China
XIAOYONG DU, Renmin University of China, China
XIAOFENG JIA, Beijing Big Data Centre, China
SONG GAO, Beijing Big Data Centre, China

Data matching – which decides whether two data elements (e.g., string, tuple, column, or knowledge graph entity) are the "same" (a.k.a. a match) – is a key concept in data integration, such as entity matching and schema matching. The widely used practice is to build task-specific or even dataset-specific solutions, which are hard to generalize and disable the opportunities of knowledge sharing that can be learned from different datasets and multiple tasks. In this paper, we propose Unicorn, a unified model for generally supporting common data matching tasks. Unicorn can enable knowledge sharing by learning from multiple tasks and multiple datasets, and can also support zero-shot prediction for new tasks with zero labeled matching/non-matching pairs. However, building such a unified model is challenging due to heterogeneous formats of input data elements and various matching semantics of multiple tasks. To address the challenges, Unicorn employs one generic Encoder that converts any pair of data elements \((a, b)\) into a learned representation, and uses a Matcher, which is a binary classifier, to decide whether \(a\) matches \(b\). To align matching semantics of multiple tasks, Unicorn adopts a mixture-of-experts model that enhances the learned representation into a better representation. We conduct extensive experiments using 20 datasets on seven well-studied data matching tasks, and find that our unified model can achieve better performance on most tasks and on average, compared with the state-of-the-art specific models trained for ad-hoc tasks and datasets separately. Moreover, Unicorn can also well serve new matching tasks with zero-shot learning.

CCS Concepts: • Information systems → Mediators and data integration.

Additional Key Words and Phrases: data matching, data integration, multi-task learning

ACM Reference Format:

∗Ju Fan is the corresponding author.

Authors’ addresses: Jianhong Tu, Renmin University of China, Beijing, China, tujh@ruc.edu.cn; Ju Fan, Renmin University of China, Beijing, China, fanj@ruc.edu.cn; Nan Tang, QCRI / HKUST (GZ), Doha / Guangzhou, Qatar / China, ntang@hbku.edu.qa; Peng Wang, Renmin University of China, Beijing, China, lisa_wang@ruc.edu.cn; Guoliang Li, Tsinghua University, Beijing, China, liguoliang@tsinghua.edu.cn; Xiaoyong Du, Renmin University of China, Beijing, China, duyong@ruc.edu.cn; Xiaofeng Jia, Beijing Big Data Centre, Beijing, China, jiaxf@jxj.beijing.gov.cn; Song Gao, Beijing Big Data Centre, Beijing, China, gaos@jxj.beijing.gov.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. 2836-6573/2023/5-ART84 $15.00 https://doi.org/10.1145/3588938

1 INTRODUCTION

Data matching generally refers to the process of deciding whether two data elements are the “same” (a.k.a. a match) or not, where each data element could be of different classes such as string, tuple, column, and so on. Data matching is a key concept in data integration [13] and data preparation [9] that includes a wide spectrum of tasks. In this paper, we consider seven common data matching tasks, namely entity matching, entity linking, entity alignment, string matching, column type annotation, schema matching, and ontology matching, as shown in Figure 1.

Due to their importance, almost all the aforementioned matching tasks have been studied for decades, and still remain to be important research topics. With the tremendous successes of deep learning, deep neural networks have been widely used for tackling various matching tasks, such as DeepMatcher [34] and Ditto [30] for entity matching (see a recent tutorial [4]), and TURL [10] and HNN [5] for column type annotation. However, these current deep learning based solutions are task-specific or even dataset-specific, which are referred to as specific models in this paper.

There are two main limitations of specific models. First, they can only learn knowledge from specific tasks or datasets. That is, the learned knowledge cannot be shared across different models. For example, the knowledge learned by a column type annotation model (e.g., TURL [10]) cannot be shared with an entity matching model (e.g., Ditto [30]) due to different neural network designs, although very likely the two models can help each other. Second, one model has to be trained (or fine-tuned) only on the labeled examples of each task or dataset, which is inefficient and has a high monetary cost. For example, DeepMatcher [34] and Ditto [30] have to be fine-tuned on a new entity matching dataset with at least hundreds of labeled matching/non-matching pairs.

In this paper, we propose Unicorn, a unified model to support multiple data matching tasks, as shown in Figure 1. Compared with aforementioned specific models, Unicorn has the following notable advantages.

1. **Task unification** that generalizes task-specific solutions into one unified model, thus achieving lower maintenance complexity and smaller model sizes, compared with specific models.

2. **Multi-task learning** that enables the unified model to learn from multiple tasks and multiple datasets to make full use of knowledge sharing, which even outperforms specific models trained only on their own datasets separately.

3. **Zero-shot prediction** that allows the model to make predictions for a new task or a new dataset with zero labeled matching/non-matching pairs.

Although several unified models have been recently studied in the NLP and CV communities [3, 41, 42], building such a unified model for supporting multiple data matching tasks is challenging.

1. Data elements in the matching tasks can take heterogeneous formats, such as tuples and columns in tabular data, entities in knowledge graphs, and plain text, which will increase the difficulty of task unification.

2. Each data matching task may have its unique matching semantics. For example, entity matching that matches two tuples is different from schema matching that matches two columns. Thus, it is non-trivial to enable knowledge sharing among multiple tasks.

To address the challenges, we develop a general framework Unicorn, which consists of three key modules: an Encoder, a Mixture-of-Experts layer, and a Matcher (see Figure 1). First, the Encoder converts any pair of elements \((a, b)\) with heterogeneous formats into a learned representation. To this end, we propose to serialize any pair of elements into text while still preserving their inherent structure, and employ a unified pre-trained language model for effective encoding. Second, to align matching semantics of various tasks or datasets, the Mixture-of-Experts layer enhances the learned representation into a better representation by combining the knowledge from multiple “experts” controlled by a learned gating network. Third, the Matcher is a typical binary classifier,
**Unicorn** is a unified model for "data matching" task in data integration. So far, it supports seven matching tasks in data integration for a pair \((a, b)\) where \(a\) or \(b\) may be of the same or different types of data elements. **Unicorn** consists of a unified Encoder, a Mixture-of-Experts layer, and a Matcher that will decide whether \((a, b)\) is a match (1) or a non-match (0).

![Diagram of Unicorn model](image)

**Unicorn** is a unified model for "data matching" task in data integration. So far, it supports seven matching tasks in data integration for a pair \((a, b)\) where \(a\) or \(b\) may be of the same or different types of data elements. **Unicorn** consists of a unified Encoder, a Mixture-of-Experts layer, and a Matcher that will decide whether \((a, b)\) is a match (1) or a non-match (0).

Our notable contributions are summarized as follows.

1. As far as we know, **Unicorn** is the first unified multi-tasking model towards supporting various matching tasks in data integration. We formally define the problem of data matching tasks and introduce an overview of the **Unicorn** framework (Section 2).
2. We develop effective techniques for two key modules in the **Unicorn** framework. We introduce a unified representation learning method for the Encoder module (Section 3) and design effective methods for the Mixture-of-Experts layer (Section 4).
3. We conduct a thorough evaluation on 20 datasets for the seven common data matching tasks shown in Figure 1. Extensive experiments show that **Unicorn**, as a unified model, outperforms the state-of-the-art specific models on most tasks and on average (Section 5 Exp-3). In addition, **Unicorn** can well serve new matching tasks with zero-shot learning (Section 5 Exp-4).
4. We make a unified benchmark for multiple data matching tasks available at Github\(^1\). We standardize the format of datasets from different data matching tasks, making these datasets more convenient to use. We publish the source code of **Unicorn**, and provide the trained **Unicorn** model.

---

\(^1\)https://github.com/ruc-datalab/Unicorn
model with multi-task learning in Hugging Face\textsuperscript{2}, so that researchers and practitioners can either use it out-of-the-box, or fine-tune it for specific tasks.

2 PROBLEM AND SOLUTION OVERVIEW

This section formalizes the \textit{data matching} problem (Section 2.1), and presents an overview of our \textsf{Unicorn} framework (Section 2.2).

2.1 Problem: Data Matching Tasks

\textbf{Data Elements.} In this paper, we consider a \textit{data element} as a basic unit of information in data integration, which has a unique meaning. We consider data elements in the following five categories:

- \textbf{String.} A \textit{string} is a sequence of words, which could be a noun or a natural language sentence, denoted as \((\text{word}_i)_{1 \leq i \leq k}\).
- \textbf{Tuple.} A \textit{tuple} is a row contained in a table, consisting of a set of attribute-value pairs \(\{(\text{attr}_i, \text{val}_i)\}_{1 \leq i \leq k}\), where \text{attr}_i \text{ and } \text{val}_i \text{ are respectively the } i\text{-th attribute name and value of the tuple.}
- \textbf{Column.} A \textit{column} is a set of values \(\{\text{val}_i\}_{1 \leq i \leq k}\) of a particular attribute \text{attr} within a table, one value for each row of the table.
- \textbf{Ontology.} An \textit{ontology} identifies and distinguishes hierarchical concepts and the relationships among the concepts. An ontology is formalized as a \textit{tree} structure \(\text{ont} = \{\text{node}_{i}, \text{pnode}_i\}_{1 \leq i \leq k}\), where \text{pnode}_i \text{ is the parent of node}_i. The unique node, which does not have a parent node, is called the \textit{root} node.
- \textbf{Knowledge Graph Entity.} A \textit{knowledge graph entity} (or \text{KG-entity} for short) describes a real-world entity, such as people, places, and things, in a knowledge graph. Formally, a knowledge graph (KG) is defined as a \textit{graph} structure \(\text{KG} = (E, R, A, V, T_r, P_a)\), where \text{ent} \in E, \text{rel} \in R, \text{attr} \in A, \text{and } \text{val} \in V \text{ represent an entity, a relation, an attribute, and an attribute value respectively. Moreover, } (\text{ent}_i, \text{rel}_k, \text{ent}_j) \in T_r(\text{ent}_i) \text{ denotes a relational triple, and } (\text{attr}_k, \text{val}_j) \in P_a(\text{ent}_j) \text{ denotes an attribute-value pair.}

Figure 2 shows example data elements with the categories of String (in pink color), Tuple (in yellow color), Column (in green color), Ontology (in blue color), and KG-Entity (in cyan color).

\textbf{Data Matching.} Let \(A = \{a_1, \ldots, a_n\}\) and \(B = \{b_1, \ldots, b_m\}\) be two sets of data elements. The problem of \textit{data matching} is to find all the pairs \((a_i, b_j)\) \(\in A \times B\) that are \textit{matched}, where the semantic of whether \((a_i, b_j)\) is matched or unmatched depends on the specific data matching tasks, which are presented as follows.

\textbf{Data Matching Task Types.} Based on various combinations of types of input pair \((a, b)\) and the matching semantics, we consider the following seven common types of data matching tasks, as shown in Figure 2. For each task, we describe the existing solutions and our formal definitions.

\begin{itemize}
  \item \textit{(1) Entity matching} refers to the task of determining whether two different tuples from two tables refer to the same real-world object [7, 18]. Existing solutions either define similarity functions based on aligned attributes (e.g., Magellan [14]), use word embeddings (e.g., DeepER [19] and DeepMatcher [34]), or piggyback pre-trained language models (e.g., Ditto [30] and DADER [48]). We formalize entity matching as determining whether a (Tuple, Tuple) pair matches or not. For example, an entity matching task in Figure 2 determines whether the pair \((a_i, b_i)\) of two records (with attributes Name, City and Age) refer to the same person. Note that, for some entity matching benchmarks, an entity could be of the JSON format (e.g., the Web Data Commons dataset [35]). In

\footnote{https://huggingface.co/RUC-DataLab/unicorn-plus-v1}

Fig. 2. Samples of common data matching tasks over heterogeneous data elements with the following five
categories: String (in pink color), Tuple (in yellow color), Column (in green color), Ontology (in blue color),
and KG-Entity (in cyan color).

(2) **Entity linking** refers to the task of determining whether a mention in a table refers to the same
object with a KG-Entity in a knowledge base. A mention is represented by a tuple that contains
multiple attributes. Existing solutions typically retrieve potential entities from knowledge bases
and then sort them by calculating their similarities (e.g., Hybrid II [20] and TURL [10]).

We formalize entity linking as determining whether a (Tuple, KG-Entity) pair matches or not. An example of entity linking is shown in Figure 2 to determine whether a tuple $a_2$ in a web table is
matched with the “Hallin Fell” entity in a knowledge base.

(3) **Entity alignment** refers to a task of determining whether two KG-entities, typically from different
knowledge graphs (e.g., DBPedia and YAGO), are the same real-world object (see [56] for a survey).
Existing solutions mainly learn entity embeddings and realize the matching of embeddings through
graph neural networks (e.g., GCN [26], CUEA [57]) or pre-trained language models (e.g., BERT-
INT [46]).

Entity alignment can be naturally formalized as determining whether a (KG-Entity, KG-Entity)
pair matches or not. Typically, the two KG-entities to be matched, such as ($a_3$, $b_3$) in Figure 2, may
have different attributes or relational triples, where $a_3$ has one attribute-value pair “(numberOfEm-
ployees, 50)” and two relational triples “(Zen-Studios, location, Hangary)” and “(Zen-Studios, product,
Pinball-FX)”, and $b_3$ has one another attribute-value pair and one another relational triple.

(4) **String matching** refers to the task of determining whether two strings from two data sources
are semantically the same or not. Existing solutions use string similarity functions [50] or machine
learning methods such as the decision tree model to predict the results (e.g., Smurf [39] and Falcon [8]).

We formalize the task as determining whether a (String, String) pair matches, e.g., determining
whether $a_5$ (“5938 Chestnet St., Philadelphia, PA 19139”) and $b_5$ (“5938 Chestnet Street, PHL, PA 19139”) in Figure 2 indicate the same address.
(5) **Column type annotation** typically determines the semantic types of a column in a table. Existing solutions use a single cell embedding or column embedding to represent a column through neural networks and then determine its type by similarity function or machine learning, where the type is a category (e.g., HNN+P2Vec [5] and TURL [10]). We formalize it as determining whether a \((\text{Column}, \text{Ontology})\) pair matches or not. Consider column \(a_6\) in Figure 2, the problem is to decide whether an ontology \(b_6\) (e.g., “Gender”) is an appropriate column type for the given Column \(a_6\).

(6) **Schema matching** determines the correspondences between columns of two schemata from different tables. Existing solutions commonly define heuristic rules or calculate similarities between schemas (e.g., COMA [12]). We formalize the schema matching task as determining whether a \((\text{Column}, \text{Column})\) pair matches or not. Consider the pair of \(a_7\) (named “Gender”) and \(b_7\) (named “Sex”) in Figure 2: schema matching is the process of identifying whether there is semantic correspondence between these two columns, e.g., referring to the same attribute of a person.

(7) **Ontology matching** finds correspondences between semantically related entities from different ontologies. Existing solutions calculate similarities such as Jaccard similarity cross different nodes of ontology and use machine learning models to predict the results (e.g., GLUE [15]). We formalize ontology matching as determining a \((\text{Ontology}, \text{Ontology})\) pair matches or not. Consider the pair \((a_i, b_i)\) in Figure 2: ontology matching is to determine whether “Earth-and-Atmospheric-Sciences” from “College-of-Engineering” in “Cornell” and “Earth-and-Space-Sciences” from “College-of-Arts-and-Sciences” in “Washington” refer to the same specialized subject.

### 2.2 A Unified Multi-Tasking Framework

Given multiple data matching tasks, the widely used practice is to build task-specific solutions, e.g., using different models or with various parameters. On the contrary, we introduce a unified multi-tasking framework called **Unicorn** for multiple data matching tasks, which has the following notable advantages.

- **Task unification** standardizes task-specific solutions into a unified framework, achieving lower development complexity, smaller model sizes and easier adaption of new tasks.
- **Multi-task learning** enables the possible opportunities of **knowledge sharing** among different data matching tasks compared with training each task separately.

Figure 3 shows an overview of the **Unicorn** framework. The basic idea is to unify multiple data matching tasks into a text-to-prob format due to its flexibility and extensibility, thus achieving a unified input and output of different tasks. Specifically, **Unicorn** unifies the input by serializing any pair \((a, b)\) of data elements into a text sequence, and outputs a probability \(\hat{y}\) to indicate if \(a\) and \(b\) match. To this end, **Unicorn** utilizes three main modules, namely **Encoder**, **Mixture-of-Experts** and **Matcher**. Next, we introduce the input and the main modules of **Unicorn** as follows.

**Input: Multiple Data Matching Tasks.** Instead of considering an individual data matching task, **Unicorn** takes as input a collection of data matching tasks, denoted as \(T = \{T_i\}\). In particular, each task \(T_i = (A_i, B_i, \tau_i, D_i)\) is composed of two sets of data elements to be matched, i.e., \(A_i\) and \(B_i\), and \(\tau_i\) is the type of task \(T_i\) (see Section 2.1 for the supported task types). \(D_i \subset A_i \times B_i \times \{0, 1\}\) is a set of labeled examples, each of which denotes whether a pair of elements \(a \in A_i\) and \(b \in B_i\) is a match (i.e., label 1) or a non-match (i.e., label 0). Figure 3 (a) shows three example data matching tasks, i.e., entity matching over \(A_1\) and \(B_1\), entity linking over \(A_2\) and \(B_2\) and schema matching over \(A_3\) and \(B_3\). Note that **Unicorn** is extensible that new task types can be easily supported.
Encoder: The Input Layer. Given an element pair \((a, b)\) from any data matching task (e.g., Figure 3 (a)), the aim of Encoder is to first serialize the pair into a text sequence \(x\), and then map \(x\) into a high-dimensional vector-based representation \(x\), i.e.,

\[
x = F(x) = F(S(a, b)),
\]

where \(S(\cdot)\) is a generic function for serializing any data element pair \((a, b)\) from the matching tasks in \(T\) into a text sequence, and \(F(\cdot)\) is a pre-trained language model (PLM) for deriving high-dimensional feature vectors from the serialized sequences. For example, in Figure 3 (b), circles, triangles and squares respectively represent feature vectors of the entity matching task, the entity linking task and the schema matching task, where a solid shape (resp. a hollowed shape) denotes a match (resp. a non-match).

Mixture-of-Experts: The Intermediate Layer. Although all the pairs from different tasks are mapped into one feature space, the distributions of their representations may not be aligned, as shown in Figure 3 (b). Consequently, it is hard to train a good Matcher.

To address the problem, we introduce an intermediate layer Mixture-of-Experts (MoE) [28, 33, 40, 44] to align the representations of different tasks, such that a good Matcher is easier to learn, as shown in Figure 3 (c). The basic idea is to transform original features of the pairs into an aligned feature space, i.e., \(x' = \text{MoE}(x)\).

The Mixture-of-Experts layer is an ensemble learning method that consists of two key components, Experts and Gating. Intuitively, it divides the input feature space into sub-spaces, and trains an Expert model for the feature alignment on each subspace. Then, given a new input vector \(x\), it utilizes the Gating model to decide which experts to use for \(x\). Formally, we use \(\text{Expert}_i\) and \(g_i\) to represent the \(i\)-th Expert and its gating weight. Then, we have

\[
x' = \text{MoE}(x) = (g_1 \cdot \text{Expert}_1(x)) + \ldots + (g_k \cdot \text{Expert}_k(x)).
\]

We will discuss the challenges and our proposal on designing the above Experts and Gating models in Section 4.
Matcher: The Output Layer. Given the representation $x'$, the Matcher is a binary classifier, which takes a vector $x'$ as input and outputs its probabilities $\hat{y}$ of matching. For Matcher, MLP is the most common choice used in existing deep learning-based classifier (e.g., DeepER [19] and Ditto [30]), which is denoted as $\hat{y} = M(x')$.

Multi-task Training. We adopt multi-task supervised learning to train Unicorn, using a lot of labeled match/non-match examples coming from all tasks in $\mathcal{T}$. Specifically, we union all the labeled element pairs in $\mathcal{T}$ to generate a training set $\mathcal{D} = \bigcup_i \mathcal{D}_i$, and train the above three modules of Unicorn in an end-to-end manner.

Data Matching Prediction. After multi-task training over all the tasks in $\mathcal{T}$, Unicorn can support the following prediction scenarios.

(1) Unified Prediction on Existing Tasks. In this scenario, we use Unicorn to predict the test set $\mathcal{D}_i^{\text{tst}}$ of any task $T_i \in \mathcal{T}$. Note that $\mathcal{D}_i^{\text{tst}}$ is unlabeled and disjoint with the training set $\mathcal{D}_i$ of $T_i$, e.g., any unseen pairs for the existing tasks in Figure 3 (a).

(2) Zero-Shot Prediction on New Tasks. We can also use Unicorn to predict a new task $T$, which is not included in $\mathcal{T}$, with a zero-shot setting such that the pairs in the new task have zero labels. For example, given the learned models in Figure 3, we may use them to directly predict new unseen datasets of various task types (e.g., string matching and column type annotation).

Remarks. We adopt the blocking technology [2, 14] to obtain appropriate labeled match/non-match examples for each task. For example, we can use simple string distance calculation rules, such as word overlapping, edit distance and euclidean distance, to filter out pairs that are unlikely matched. We also use a proportion of labeled match/non-match pairs as validation set, which is standard and thus not presented in this section for simplicity.

3 THE ENCODER MODULE

There are two main challenges in designing the Encoder. First, it is non-trivial to devise a generic serialization function $S(\cdot)$ for various types of matching tasks over heterogeneous data elements. Second, existing works choose different pre-trained language models (PLMs) on individual matching tasks, e.g., RoBERTa for entity matching [30] and BERT for entity alignment [46]. Thus, it remains an unresolved question on whether a unified PLM could achieve good performance on many different matching tasks.

Inspired by the recent successes of unified frameworks that treat many NLP tasks as a “text-to-text” problem [41, 54], we propose to serialize all data matching tasks into the text format and then utilize a unified PLM for encoding data element pairs in the tasks. To this purpose, in this section, we seek to answer two main questions: (1) which format should be used to unify different matching tasks; and (2) which unified PLM model should be utilized, so as to avoid the limitation that different PLMs have been used for different tasks.

3.1 Pair-to-Text Serialization

PLMs are natural choices for encoders, which typically take a sequence of tokens as input. Hence, we propose to serialize a pair of data elements $(a, b)$ into a sequence of tokens (like Ditto [30]) as:

$$x = S(a, b) = [\text{CLS}] S(a)[\text{SEP}] S(b)[\text{SEP}]$$ (3)

where [CLS] is a special token to indicate the start of the sequence, the first [SEP] is a special token to separate the sequence of element $a$ (i.e., $S(a)$) and the sequence of element $b$ (i.e., $S(b)$), and the last [SEP] token is used to indicate the end of the sequence.

Next, we will describe how to serialize each type of the data elements.
String serialization. Given a string \( str \) with words \( \langle \text{word}_i \rangle_{1 \leq i \leq k} \), we use the WordPiece tokenization algorithm, like BERT [11], to serialize \( str \) into a sequence of sub-words (tokens) as

\[
S(str) = \text{token}_1 \ \text{token}_2 \ldots \ \text{token}_k.
\]

For example, “Hallin Fell is a hill in the English Lake District surrounded on three sides by Ullswater” is serialized into the token sequence “Hall ##in Fell is a hill in the ##Eng ##lish Lake Dis ##trict su ##rround ##ed...”. Note that we use this method to serialize all strings except special keywords in the following data elements.

Tuple serialization. Given a tuple \( \text{tup} \) with attribute-value pairs \( \{ (\text{attr}_i, \text{val}_i) \}_{1 \leq i \leq k} \), we serialize it into a sequence as

\[
S(\text{tup}) = [\text{ATT}] \ \text{attr}_1 [\text{VAL}] \ \text{val}_1 \ldots [\text{ATT}] \ \text{attr}_k [\text{VAL}] \ \text{val}_k,
\]

where [ATT] and [VAL] are two special tokens for specifying attributes and values respectively. Take tuple \( a_1 \) of entity matching in Figure 2 as an example: we serialize it into “[ATT] Name [VAL] Dave Smith [ATT] City [VAL] Atlanta [ATT] Age [VAL] 18 [SEP]”. 

Column serialization. Given a column \( \text{col} \) with an attribute name and values \( \{ (\text{attr}, \{ \text{val}_i \})_{1 \leq i \leq k} \} \), we concatenate the attribute name and values of a whole column and serialize it as

\[
S(\text{col}) = [\text{ATT}] \ \text{attr} [\text{VAL}] \ \text{val}_1 \ \text{val}_2 \ldots \ \text{val}_k.
\]

Note that, in the case of too many values in the column, we randomly select a proportion of the values. Take the column \( a_7 \) of schema matching in Figure 2 as an example: we serialize \( a_7 \) into sequence “[ATT] Gender [VAL] Male Female Male ... [SEP]”. 

Ontology serialization. Given a tree-based ontology \( \text{ont} \) and a specific node \( \text{node}_k \) in the ontology, we represent it as a sequence by concatenating all nodes in the path from root to \( \text{node}_k \) as

\[
S(\text{node}_k) = \text{node}_1 \ \text{node}_2 \ldots \ \text{node}_k.
\]
Consider a_4 in Figure 2, the serialized sequence is “catalog-cornell College-of-Engineering Earth-and-Atmospheric-Sciences”.

**KG-entity serialization.** Given a KG-entity, which is also denoted as a subject entity sub in the knowledge graph, it has not only some attribute values \{(attr_i, val_i)\}_{1 \leq i \leq k}, but also some relational triples with other entities \{(sub, rel_i, obj_i)\}_{1 \leq i \leq m}. Based on the structure, we serialize the KG-entity sub into sequence

\[
S(\text{sub}) = \text{sub} [\text{ATT}] attr_1 [\text{VAL}] val_1 \ldots [\text{ATT}] attr_k [\text{VAL}] val_k
\]

\[\text{[TRI]} \text{sub rel}_1 \text{obj}_1 \ldots [\text{TRI}] \text{sub rel}_m \text{obj}_m,\]

where [TRI] is a special token for specifying relational triples. As shown in Figure 2, we serialize a_3 as “Zen-Studios [ATT] numberOfEmployees [VAL] 50 [TRI] Zen-Studios product Pinball-FX [TRI] Zen-Studios location Hungary”.

Figure 4 depicts the examples of serialized sequences for all data element pairs from the seven matching tasks as shown in Figure 2.

**Zero-shot Instruction.** For using Unicorn in zero-shot prediction on new tasks (see Section 2.2), we further utilize instruction [51, 52] to improve the performance. As a way to boost the inference ability of large-scale PLMs for downstream tasks in zero-shot/few-shot settings, instruction has been proven to be effective in many unified models, such as OFA [51], which specifies instruction templates described in natural language for a variety of multimodal tasks. SUPER-NATURALINSTRUCTIONS [52] integrates 1,616 natural language processing tasks into a unified framework using instruction and performs well on new tasks.

Inspired by these unified model, we develop a simple task-agnostic instruction for data matching tasks in this paper. The basic idea is to define an appropriate natural language template to make downstream tasks (e.g., matching tasks in our paper) conform to the natural language form of pre-training tasks. Specifically, we design the following simple instruction template, which is shown to be effective in our experiments (see Section 5 Exp-5).

\[
x = S(a, b) = [\text{CLS}] \text{does } S(a) [\text{SEP}] \text{match with } S(b) [\text{SEP}]\]

Consider schema matching of \((a_7, b_7)\) in Figure 4. By employing the above instruction template, the pair is serialized into a sequence “[CLS] does [ATT] Gender [VAL] Male Female Male ... [SEP] match with [ATT] Sex [VAL] F F M ... [SEP]”.

**Remarks.** We only design a simple instruction template for matching tasks and preliminarily verify its performance on matching tasks in our experiments. We will discuss more instruction methods in the future work, such as specifying different instructions for different types of data matching tasks. In addition, prompt [29, 31, 54] is also an effective method to stimulate the ability of PLMs for downstream tasks, which will be systematically investigated in our future work.

### 3.2 Representation Learning of Serialized Pairs with Pre-trained Language Models

Given a sequence \(x\) serialized from pair \((a, b)\), we employ a transformer-based pre-trained language models (PLM) as encoder to convert the sequence into a high-dimensional vector-based representation \(x\). PLMs have proven to be the most effective methods of learning representations.

Representative PLMs include BERT [11], RoBERTa [32], DeBERTa [24], etc. However, different from NLP tasks, the key challenge in Unicorn is to support structure-aware encoding, as tokens in a serialized sequence may have specific structure information and depend on other tokens. For example, consider entity alignment over \((a_3, b_3)\) in Figure 4, where the serialized sequence is

\[
S(a_3, b_3) = [\text{CLS}] \text{Zen-Studios [ATT] numberOfEmployees [VAL] 50 [TRI] Zen-Studios product Pinball-FX...}.
\]
In this example, “50” is a specific value, which highly depends on its attribute “numberofEmployees”. Moreover, the token “Zen-Studios” after “[CLS]” plays more important roles in matching compared with the same token after “[TRI]”, as the former is the name of the KG-entity $a_3$ to be matched and the latter just describes another entity connected to $a_3$. Thus, the former “Zen-Studios” should be paid more attention compared with the latter. Obviously, the conventional self-attention mechanism in Transformer has limitations to support the structure-aware encoding.

To address the problem, we employ DeBERTa [24] as the PLM for structure-aware encoding. The main reason is that DeBERTa has a new positional encoding scheme that captures relative positions of tokens in the sequence, which is helpful to understand the structure among the tokens. Consider our previous example again. The same token “Zen-Studios” will be encoded into different representations, as the first one is relatively near to “[CLS]”, and thus should be attended by the entire sequence, while the second one only needs to be attended in the smaller scope of “[TRI]”.

Specifically, DeBERTa [24] introduces disentangled attention and enhanced mask decoder to explicitly consider both relative and absolute positions of the tokens in an input sequence. The idea of disentangled attention is that, when calculating attention score (i.e., relation score) between two tokens, DeBERTa considers not only the conventional content attention, but also an attention score based on relative positions. Formally, the attention score of two tokens at positions $i$ and $j$ in the sequence is calculated by the function (from the original DeBERTa paper [24]):

$$A_{i,j} = \{H_i, P_{i|j}\} \times \{H_j, P_{j|i}\}^T = H_iH_j^T + H_iP_{j|i}^T + P_{i|j}H_j^T + P_{i|j}P_{j|i}^T,$$

where $H_i$ represents the content of token at the $i$-th position, and $P_{i|j}$ represents its relative position to the token at the $j$-th position. The above function shows that the attention between two tokens is computed by disentangled matrices, which consider both contents and positions as a sum of four attention scores: content-to-content, content-to-position, position-to-content, and position-to-position. Moreover, the enhanced mask decoder incorporates absolute position to context embedding right after all the transformer layers but before the Softmax layer during Masked Language Modeling (MLM) pre-training, which enables the model to understand absolute positions. Also, DeBERTa uses virtual adversarial training to improve the generalization of the model. More details can be found in the original paper of DeBERTa [24].

We also explore the benefits and limitations of a variety of PLMs, which are listed as follows.

- BERT [11] and RoBERTa [32] consider absolute positions of the input sequence, but do not explicitly model relative positions. Moreover, the MLM pre-training task used in BERT is also helpful for understanding absolute positions.
- XLNet [55] aims to capture relative positions by devising a pre-training task Permutated Language Modeling, which may damage the understanding of absolute positions.
- MPNet [45] considers both relative and absolute positions by position compensation.
- DistilRoBERTa [43], and DistilBERT [43] are some representative distillation models, which are smaller and faster to train.

Please refer to Section 5 Exp-1 for a comprehensive comparison of different PLMs for Unicorn.

4 THE MIXTURE-OF-EXPERT MODULE

The objective of the Mixture-of-Experts (MoE) layer [21, 28, 44] in Unicorn is to map different distributions of multiple tasks to a same shared distribution. Thus, equipping Unicorn with MoE will make it not only easily supporting multiple data matching tasks with different semantics and various input formats, but also being extensible to support new matching tasks.
Formally, the layer $\text{MoE}(x) : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}$ is a function that converts an original feature vector $x \in \mathbb{R}^{d_1}$ into a new feature vector $x' \in \mathbb{R}^{d_2}$. In this paper, we adopt the most common Mixture-of-Experts architecture [25] that contains two key components, Experts and Gating. Experts are a set of neural networks, whose parameters are not shared, to individually map $x$ to $x'$ in the shared feature space. The aim of Gating is to combine the outputs of Experts according to different weights that depend on the input feature vector $x$. Next, we first present the neural network design and the training algorithm for MoE in Section 4.1, and then introduce an optimization strategy for better routing among Experts in Section 4.2.

4.1 MoE Model Design and Training

Our neural network design for the Experts and Gating components in MoE is shown in Figure 5. **Neural Network Design.** For each Expert $i$, we use a fully-connected layer with LeakyReLU as activation function to convert the input feature vector $x$ into an output $x_i \in \mathbb{R}^{d_2}$:

$$x_i = \text{LeakyReLU}(xW^{(i)})$$

where $W^{(i)} \in \mathbb{R}^{d_1 \times d_2}$ are trainable parameters.

The Gating component also takes the feature vector $x$ as input, and produces a gating vector $g = (g_1, g_2, \ldots, g_k)$ where $k$ is the number of Experts and $g_k$ is the routing weight of Expert $i$. Specifically, we design two fully-connected layers with LeakyReLU and Softmax as activation...
functions, i.e.,
\[
g = \text{Softmax}(\text{LeakyReLU}(xW^G_1)W^G_2)
\]  
(7)
where $W^G_1 \in \mathbb{R}^{d_1 \times h}$ and $W^G_2 \in \mathbb{R}^{h \times k}$ are trainable parameters, $h$ is dimension of the hidden layer, and $k$ is the number of experts.

Based on Equations (6) and (7), we obtain $k$ feature vectors, $x_1, x_2, \ldots, x_k$ through the $k$ Experts, and then use weighted average to calculate a unified representation as $x'$, i.e.,
\[
x' = \text{MoE}(x) = g_1 \cdot x_1 + g_2 \cdot x_2 + \cdots + g_k \cdot x_k.
\]  
(8)

Figure 5 illustrates an example of MoE. Given an encoded representation $x$ from a pair $(a, b)$, MoE divides the input feature space into $k$ sub-spaces, each of which is handled with a trainable Expert model. It also devises a Gating model to compute $k$ weights, e.g., $(0.33, 0.17, \ldots, 0.25)$ from each individual input $x$. Finally, MoE computes the weighted average of the outputs of the Experts.

**End-to-End Model Training.** We train the components Encoder, MoE and Matcher of Unicorn in an end-to-end manner, as presented previously. Specifically, the output $x'$ of MoE is fed into Matcher to produce the predicted result $\hat{y} = M(x')$. Given all labeled pairs $D = \bigcup D_i$ from different matching tasks (see Section 2.2), we can compute the cross entropy loss as
\[
\mathcal{L} = \mathbb{E}_{(x', y) \in D} \mathcal{L}_{CE}(\hat{y}, y),
\]  
(9)
where $(x', y)$ is a labeled matching/non-matching pair outputted by the previous Encoder and Mixture-of-Experts modules and $\mathcal{L}_{CE}$ is the cross entropy function.

Next, by iteratively applying minibatch stochastic gradient descent, parameters of the modules Encoder, MoE and Matcher are optimized, and thus Unicorn could be improved towards producing accurate matching results for different tasks.

### 4.2 MoE Optimization for Expert Routing

One obstacle of training MoE is that various input vectors $\{x\}$ may use the same few Experts; that is, these Experts have larger gating weights than others for almost all the inputs. This phenomenon will affect the performance of MoE, as it fails to take advantage of different Experts for various data matching tasks.

To address this problem, we introduce an optimization strategy for Expert Routing. The basic idea is to improve the original loss function in Equation (9) by further considering two more objectives:

1. Inspired by Sparsely-Gated MoE [44], for the overall training set, we want all the experts to be used in a balanced way.
2. For any specific training pair, we would like to assign a few specific experts to it, instead of evenly assigning it to all the Experts, which is shown to achieve better performance in our experiments.

Technically, to achieve the first objective, we compute the “utilization” of all the experts on the overall training set $D$. Formally, we compute an Expert utilization vector $u$ for the $k$ Experts as:
\[
u = \left( \sum_{x \in D} g_1, \sum_{x \in D} g_2, \ldots, \sum_{x \in D} g_k \right),
\]  
(10)
where $\sum_{x \in D} g_i$ is the sum of gating weights for Expert $i$ on all the training examples in $D$. As we want to achieve more balanced utilization of the Experts, we compute a load balancing loss, denoted as $\mathcal{L}_{\text{Bal}}$ by computing coefficient of variation of utilization $u$ as:
\[
\mathcal{L}_{\text{Bal}} = \frac{\sigma(u)}{\mu(u)}
\]  
(11)
where \( \sigma(u) \) and \( \mu(u) \) are respectively standard deviation and mean of the utilization vector \( u \). Intuitively, the smaller the loss \( L_{\text{Bal}} \) is, the more balanced the utilization of the Experts is.

To achieve the second objective, we compute the entropy of the gating vector \( g \) for each training examples. The idea is that, we want that different Experts are trained to handle different feature sub-spaces, to make full use of the ensemble ability of MoE. Formally, we define an entropy loss function \( L_{\text{Ent}} \) over training examples:

\[
L_{\text{Ent}} = \mathbb{E}_{(x', y)} \text{Entropy}(g) = \mathbb{E}_{(x', y)} - \sum_{i=1}^{k} g_i \cdot \log(g_i)
\]  

By considering the above loss functions, we derive a new loss function for training Unicorn as:

\[
L_{\text{new}} = L + L_{\text{Bal}} + L_{\text{Ent}}
\]  

Note that, we apply the same minibatch stochastic gradient descent for training Unicorn given the new loss function. Specifically, in each training iteration, we compute the loss based on Equation (13) for training examples in a minibatch, and then update all the parameters in Unicorn by using back-propagation.

## 5 EVALUATION

Next, we report the experimental evaluation for Unicorn. The key questions we answer with our evaluation are presented as follows.

**Exp-1**: Which pre-trained language model should be employed as Encoder in Unicorn?

**Exp-2**: Which strategy performs well in the Mixture-of-Experts layer?

**Exp-3**: How does a unified model Unicorn (i.e., trained using labeled datasets from multiple tasks) compare with specific state-of-the-art (SOTA) models (i.e., each model is separately trained for ad-hoc tasks and datasets)?

**Exp-4**: How does Unicorn perform on a new task with zero labeled matching/non-matching pairs?

**Exp-5**: Whether our proposed MoE and instruction techniques are helpful in the zero-shot setting?

**Exp-6**: How does Unicorn perform for unseen tasks of totally new task types?

### 5.1 Experimental Setup

**Datasets.** Recall that Unicorn so far supports seven types of common data matching tasks (see Figure 2), including entity matching, entity linking, entity alignment, string matching, column type annotation, schema matching, and ontology matching. For each task type, we employ the commonly used datasets for evaluation, as shown in Table 1, each of which is denoted as a task (see definition of task in Section 2.2). Each task contains two sets, \( A \) and \( B \), of data elements with different categories. Then, for each task, we derive the set \( D \) of labeled matching/non-matching pairs, and divide \( D \) into training \( D^{\text{trn}} \), validation \( D^{\text{val}} \) and test \( D^{\text{tst}} \). For fair comparisons of SOTA methods for each task, we prepare \( D^{\text{trn}} \), \( D^{\text{val}} \) and \( D^{\text{tst}} \) as follows.

1. For each entity matching task, we directly use the training, validation and test sets prepared by the SOTA method Ditto [30], which have a ratio of 3:1:1, for a fair comparison with Ditto.

2. For each entity alignment task, we follow the SOTA method BERT-INT [46] for data preparation. We first split the matching pairs for training, validation and test in a ratio of 2:1:7. Then, for each matching pair \((a, b)\) for training and validation, we prepare one non-matching pair by replacing \( b \) with a randomly sampled data element \( b' \in B \). For test set, we first find all matching pairs \( \{(a, b)\} \), and then use all other combinations of \( \{a\} \) and \( \{b\} \) as non-matching pairs.

3. For each task of string matching, schema matching, ontology matching, column type annotation and entity linking, the SOTA methods have not provided the prepared training, validation and test
Table 1. Dataset Statistics, where $|A|$ (or $|B|$) represents the number of data elements in set $A$ (or $B$) in each task, # Matches (# Non-Matches) represents the number of matches (non-matches) in the labeled set $D$ of the task. Metric is the commonly-adopted measurement for evaluating the corresponding data matching task.

| Task Type                  | Task                  | $|A|$   | $|B|$   | # Matches | # Non-Matches | Metric |
|----------------------------|-----------------------|--------|--------|-----------|--------------|--------|
| Entity Matching (EM)       | Walmart-Amazon (WA)   | 2,554  | 22,074 | 962       | 9,280        | F1     |
|                            | DLBL-P-Scholar (DS)   | 2,616  | 64,263 | 5,347     | 23,360       | F1     |
|                            | Fodors-Zagats (FZ)    | 533    | 331    | 110       | 836          | F1     |
|                            | iTunes-Amazon (IA)    | 6,907  | 55,923 | 132       | 407          | F1     |
|                            | Beer (Be)             | 4,345  | 3,000  | 68        | 382          | F1     |
| Column Type Annotation (CTA)| Efthymiou (Ef)       | 620    | 31     | 620       | 18,600       | Acc.   |
|                            | T2D (T2D)             | 383    | 37     | 383       | 13,788       | Acc.   |
|                            | Limaye (Lim)          | 174    | 27     | 179       | 4,519        | Acc.   |
| Entity Linking (EL)        | T2D (T2D)             | 11,650 | 26,025 | 20,666    | 131,945      | F1     |
|                            | Limaye (Lim)          | 659    | 4,166  | 1,447     | 36,020       | F1     |
| String Matching (StM)      | Address (Ad)          | 24,650 | 29,531 | 9,850     | 1,062        | F1     |
|                            | Names (Na)            | 10,341 | 15,396 | 5,132     | 2,763        | F1     |
|                            | Researchers (Re)      | 8,342  | 43,549 | 4,556     | 4,767        | F1     |
|                            | Product (Pr)          | 2,554  | 22,074 | 1,154     | 79,310       | F1     |
|                            | Citation (Ci)         | 2,616  | 64,263 | 5,347     | 34,152       | F1     |
| Schema Matching (ScM)      | FabricatedDatasets (Fa)| 11,172 | 11,352 | 7,692     | 109,762      | Recall |
|                            | DeepMDDatasets (DM)   | 41     | 41     | 41        | 268          | Recall |
| Entity Alignment (EA)      | SRPRS: DBP-YG (SYG)   | 15,000 | 15,000 | 15,000    | 38,891       | Hits@K |
|                            | SRPRS: DBP-WD (SWD)   | 15,000 | 15,000 | 15,000    | 38,492       | Hits@K |

sets. Thus, we adopt blocking techniques to obtain appropriate labeled matching/non-matching examples. Specifically, we combine all data elements from $A$ and $B$, and then apply some heuristic blocking rules (such as having no common words or smaller string similarities) to filter out the pairs which are very likely to be non-matching, resulting in a candidate set. Then, we divide the candidate set into training, validation and test sets in a ratio of 2:1:7.

**Evaluation Metrics.** For fair comparisons with SOTA methods, we use the evaluation metrics which are also used by these methods.

1. For *entity matching*, *entity linking*, and *string matching* tasks, following the SOTA methods, we use $F1$ score as the evaluation metric. $F1$ score is the harmonic mean of precision and recall for the matching pairs, where precision $P$ is the proportion of predicted true matching pairs to all predicted matching pairs, recall $R$ is the proportion of predicted true matching pairs to all true matching pairs, and the $F1$ score is computed as $2 \cdot P \cdot R / (P + R)$.

2. For *column type annotation* and *ontology matching* tasks, the SOTA methods use *accuracy* as the evaluation metric. By following them, we also use accuracy (Acc. for short), which is the ratio of the correct predicted pairs to all candidate pairs.

3. For entity alignment, we use the common evaluation metric *Hits@K*, which is defined as the proportion of elements in $A$ whose true matched elements in $B$ are in the top-$K$ matching results returned by an approach. Obviously, the higher the Hits@K is, the better an approach is. Following BERT-INT [46], we report Hits@1.

4. For schema matching, like the SOTA method Valentine [27], we use Recall as the metric, which is the proportion of predicted matching schema pairs to all matching schema pairs.
Table 2. Results for Representative Pre-trained Language Models for the Encoder Module.

<table>
<thead>
<tr>
<th>Type</th>
<th>Task</th>
<th>Metric</th>
<th>BERT</th>
<th>RoBERTa</th>
<th>DistilBERT</th>
<th>DistilRoBERTa</th>
<th>XLNet</th>
<th>MPNet</th>
<th>DeBERTa</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>Walmart-Amazon</td>
<td>F1</td>
<td>84.24</td>
<td>84.37</td>
<td>75.07</td>
<td>68.99</td>
<td>79.43</td>
<td>83.01</td>
<td>86.89</td>
</tr>
<tr>
<td></td>
<td>DBLP-Scholar</td>
<td>F1</td>
<td>93.7</td>
<td>94.88</td>
<td>93.92</td>
<td>95.53</td>
<td>95.62</td>
<td>95.02</td>
<td>95.64</td>
</tr>
<tr>
<td></td>
<td>Fodors-Zagats</td>
<td>F1</td>
<td>97.67</td>
<td>95.24</td>
<td>92.68</td>
<td>95.24</td>
<td>100</td>
<td>97.67</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>iTunes-Amazon</td>
<td>F1</td>
<td>94.55</td>
<td>96.3</td>
<td>90</td>
<td>87.1</td>
<td>89.66</td>
<td>94.55</td>
<td>96.43</td>
</tr>
<tr>
<td></td>
<td>Beer</td>
<td>F1</td>
<td>87.5</td>
<td>90.32</td>
<td>86.67</td>
<td>90.32</td>
<td>84.85</td>
<td>90.32</td>
<td>90.32</td>
</tr>
<tr>
<td>CTA</td>
<td>Efthymiou</td>
<td>Acc.</td>
<td>98.43</td>
<td>98.1</td>
<td>97.51</td>
<td>97.68</td>
<td>97.58</td>
<td>98.13</td>
<td>98.42</td>
</tr>
<tr>
<td></td>
<td>T2D</td>
<td>Acc.</td>
<td>99.29</td>
<td>98.75</td>
<td>98.22</td>
<td>98.27</td>
<td>98.17</td>
<td>98.89</td>
<td>99.14</td>
</tr>
<tr>
<td></td>
<td>Limaye</td>
<td>Acc.</td>
<td>96.63</td>
<td>96.35</td>
<td>96.26</td>
<td>96.2</td>
<td>96.2</td>
<td>96.63</td>
<td>96.75</td>
</tr>
<tr>
<td>EL</td>
<td>T2D</td>
<td>F1</td>
<td>93.65</td>
<td>89.3</td>
<td>85.08</td>
<td>78.86</td>
<td>78.85</td>
<td>92.17</td>
<td>91.96</td>
</tr>
<tr>
<td></td>
<td>Limaye</td>
<td>F1</td>
<td>83.08</td>
<td>83.8</td>
<td>66.71</td>
<td>78.94</td>
<td>82.33</td>
<td>83.37</td>
<td>86.78</td>
</tr>
<tr>
<td>SLM</td>
<td>Address</td>
<td>F1</td>
<td>98.49</td>
<td>98.39</td>
<td>96.75</td>
<td>97.13</td>
<td>98.15</td>
<td>98.36</td>
<td>98.68</td>
</tr>
<tr>
<td></td>
<td>Names</td>
<td>F1</td>
<td>92.38</td>
<td>94.37</td>
<td>95.47</td>
<td>90.71</td>
<td>76.26</td>
<td>80.97</td>
<td>91.19</td>
</tr>
<tr>
<td></td>
<td>Researchers</td>
<td>F1</td>
<td>98.99</td>
<td>97.71</td>
<td>97.88</td>
<td>95.65</td>
<td>98.72</td>
<td>98.44</td>
<td>97.66</td>
</tr>
<tr>
<td></td>
<td>Product</td>
<td>F1</td>
<td>81.13</td>
<td>80.67</td>
<td>65.8</td>
<td>63.71</td>
<td>71.73</td>
<td>76.81</td>
<td>82.9</td>
</tr>
<tr>
<td></td>
<td>Citation</td>
<td>F1</td>
<td>96.28</td>
<td>95.7</td>
<td>93.74</td>
<td>95.11</td>
<td>95.91</td>
<td>95.28</td>
<td>96.27</td>
</tr>
<tr>
<td>ScM</td>
<td>FabricatedDatasets</td>
<td>Recall</td>
<td>77.85</td>
<td>77.72</td>
<td>42.96</td>
<td>77.48</td>
<td>70.99</td>
<td>70.62</td>
<td>89.6</td>
</tr>
<tr>
<td></td>
<td>DeepMDatasets</td>
<td>Recall</td>
<td>88.89</td>
<td>88.89</td>
<td>92.59</td>
<td>92.59</td>
<td>100</td>
<td>88.89</td>
<td>96.3</td>
</tr>
<tr>
<td>OM</td>
<td>Cornell-Washington</td>
<td>Acc.</td>
<td>90.64</td>
<td>86.38</td>
<td>91.06</td>
<td>68.94</td>
<td>77.45</td>
<td>75.74</td>
<td>92.34</td>
</tr>
<tr>
<td>EA</td>
<td>SRFRS, DBP-YG</td>
<td>Hits@1</td>
<td>99.94</td>
<td>99.49</td>
<td>99.24</td>
<td>99.22</td>
<td>98.69</td>
<td>99.47</td>
<td>99.67</td>
</tr>
<tr>
<td></td>
<td>SRFRS, DBP-WD</td>
<td>Hits@1</td>
<td>97.3</td>
<td>97.13</td>
<td>97.34</td>
<td>96.62</td>
<td>96.78</td>
<td>97.47</td>
<td>97.22</td>
</tr>
<tr>
<td></td>
<td>AVG</td>
<td></td>
<td>92.71</td>
<td>92.3</td>
<td>85.75</td>
<td>88.21</td>
<td>89.38</td>
<td>90.6</td>
<td>94.21</td>
</tr>
</tbody>
</table>

Implementation Details of Unicorn. For Encoder, we explore the performance of BERT [11], RoBERTa [32], DeBERTa [24], MPNet [45], XLNet [55], DistilRoBERTa [43], and DistilBERT [43]. We use the pre-trained base size checkpoints directly on the Hugging Face [1]. For (BERT, RoBERTa, DeBERTa, MPNet, XLNet)-base models, they use 12 transformer layers and output a 768 dimensional hidden embedding. For DistilRoBERTa and DistilBERT, there are 6 transformer layers. We set the maximum sequence length as 128. For the MoE layer, we choose expert number from 2 to 15, and set hidden dimensions of gate and output size of experts from \{384, 768, 1024\} according to the performance of validation set. A single fully connected layer is used for Matcher to output matching probabilities. We choose learning rate from \{3e-5, 3e-6\}, set batchsize as 32, and use maximum epoch number as 10.

Model Size. We compute the number of parameters of models to measure the model size. We use a python program to directly get the number of parameters. For Encoder, DeBERTa-base (used by Unicorn), RoBERTa-base (used by Ditto), BERT-base-multilingual (used by BERT-INT) and TinyBERT (used by TURL) have 139 Million (139M for shot), 125M, 178M and 14.5M parameters respectively. The Mixture-of-Experts layer has 8 Million parameters. We ignore the single layer in Matcher as it is too small. Thus, we compute the total model size for all methods by summing the numbers of parameters in the Encoder and the MoE layer (if any).

All the experiments are implemented using PyTorch [38] version 3.6.5 and the Transformers library [53], and evaluated on a server with 4 CPU cores (Intel Xeon Gold 6138 CPU @ 2.00GHz), 4 NVIDIA RTX 24GB GPUs, and 1024GB memory.

5.2 Evaluation on Unified Prediction

This section reports the experimental results for evaluating unified prediction (see Section 2.2). Specifically, we train a shared model Unicorn by multi-tasking training over all training sets from the data matching tasks, use the checkpoint of the model with the best average validation performance, and report the model performance on the test sets of different tasks.
Table 3. The Overall Result for Unified Prediction. Unicorn w/o MoE is a variant of Unicorn that has no MoE layer. Unicorn is our proposed framework with Encoder, MoE and Matcher. Unicorn ++ is improved with MoE optimization for Expert Routing.

<table>
<thead>
<tr>
<th>Type</th>
<th>Task</th>
<th>Metric</th>
<th>Unicorn w/o MoE</th>
<th>Unicorn</th>
<th>Unicorn ++</th>
<th>Previous SOTA (Paper)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>Walmart-Amazon</td>
<td>F1</td>
<td>85.12</td>
<td>86.89</td>
<td><strong>86.93</strong></td>
<td>86.76 (Ditto [30])</td>
</tr>
<tr>
<td></td>
<td>DBLP-Scholar</td>
<td>F1</td>
<td>95.38</td>
<td>95.64</td>
<td><strong>96.22</strong></td>
<td>95.6 (Ditto [30])</td>
</tr>
<tr>
<td></td>
<td>Fodors-Zagats</td>
<td>F1</td>
<td>97.78</td>
<td></td>
<td><strong>100</strong></td>
<td>100 (Ditto [30])</td>
</tr>
<tr>
<td></td>
<td>T1unes-Amazon</td>
<td>F1</td>
<td>94.74</td>
<td>96.43</td>
<td><strong>98.18</strong></td>
<td>97.06 (Ditto [30])</td>
</tr>
<tr>
<td></td>
<td>Beer</td>
<td>F1</td>
<td>90.32</td>
<td>90.32</td>
<td>87.3</td>
<td><strong>94.37</strong> (Ditto [30])</td>
</tr>
<tr>
<td>CTA</td>
<td>Efthymiou</td>
<td>Acc.</td>
<td>98.08</td>
<td>98.42</td>
<td><strong>98.44</strong></td>
<td>90.4 (TURL [10])</td>
</tr>
<tr>
<td></td>
<td>T2D</td>
<td>Acc.</td>
<td>98.81</td>
<td>99.14</td>
<td><strong>99.21</strong></td>
<td>96.6 (HNN+P2Vec [5])</td>
</tr>
<tr>
<td></td>
<td>Limaye</td>
<td>Acc.</td>
<td>96.11</td>
<td>96.75</td>
<td><strong>97.32</strong></td>
<td>96.8 (HNN+P2Vec [5])</td>
</tr>
<tr>
<td>EL</td>
<td>T2D</td>
<td>F1</td>
<td>79.96</td>
<td>91.96</td>
<td><strong>92.25</strong></td>
<td>85 (Hybrid I [20])</td>
</tr>
<tr>
<td></td>
<td>Limaye</td>
<td>F1</td>
<td>83.12</td>
<td>86.78</td>
<td><strong>87.9</strong></td>
<td>82 (Hybrid II [20])</td>
</tr>
<tr>
<td>StM</td>
<td>Address</td>
<td>F1</td>
<td>97.81</td>
<td>98.68</td>
<td>99.47</td>
<td><strong>99.91</strong> (Falcon [39])</td>
</tr>
<tr>
<td></td>
<td>Names</td>
<td>F1</td>
<td>86.12</td>
<td>91.19</td>
<td><strong>96.8</strong></td>
<td>95.72 (Falcon [39])</td>
</tr>
<tr>
<td></td>
<td>Researchers</td>
<td>F1</td>
<td>96.59</td>
<td>97.66</td>
<td><strong>97.93</strong></td>
<td>97.81 (Falcon [39])</td>
</tr>
<tr>
<td></td>
<td>Product</td>
<td>F1</td>
<td>84.61</td>
<td>82.9</td>
<td><strong>86.06</strong></td>
<td>67.18 (Falcon [39])</td>
</tr>
<tr>
<td></td>
<td>Citation</td>
<td>F1</td>
<td>96.34</td>
<td>96.27</td>
<td><strong>96.64</strong></td>
<td>90.98 (Falcon [39])</td>
</tr>
<tr>
<td>ScM</td>
<td>FabricatedDatasets</td>
<td>Recall</td>
<td>81.19</td>
<td>89.6</td>
<td>89.35</td>
<td>81 (Valentine [27])</td>
</tr>
<tr>
<td></td>
<td>DeepMDatasets</td>
<td>Recall</td>
<td>66.67</td>
<td>96.3</td>
<td>96.3</td>
<td><strong>100</strong> (Valentine [27])</td>
</tr>
<tr>
<td>OM</td>
<td>Cornell-Washington</td>
<td>Acc.</td>
<td>90.64</td>
<td><strong>92.34</strong></td>
<td>90.21</td>
<td>80 (GLUE [15])</td>
</tr>
<tr>
<td>EA</td>
<td>SRPRS: DBP-YG</td>
<td>Hits@1</td>
<td>99.46</td>
<td>99.67</td>
<td>99.49</td>
<td><strong>100</strong> (BERT-INT [46])</td>
</tr>
<tr>
<td></td>
<td>SRPRS: DBP-WD</td>
<td>Hits@1</td>
<td>97.11</td>
<td>97.22</td>
<td>97.28</td>
<td><strong>99.6</strong> (BERT-INT [46])</td>
</tr>
<tr>
<td></td>
<td><strong>AVG</strong></td>
<td><strong>90.8</strong></td>
<td><strong>94.21</strong></td>
<td><strong>94.56</strong></td>
<td><strong>91.84</strong></td>
<td><strong>99.5M</strong></td>
</tr>
</tbody>
</table>

Exp-1: Which pre-trained language model (PLM) should be employed in Unicorn? We first evaluate the performance of different PLMs. Under the same Unicorn framework, we use seven PLMs in base size as Encoder. Except for Encoder, we use the same settings and hyper-parameters to train the unified model. The results are reported in Table 2, where bold values are the best for each task. We can see that Unicorn works well for all PLMs as Encoder, which proves the stability and generality of the framework. Also, we find that DeBERTa achieves the best performance on 11 tasks and on average, and performs as the second best in the remaining tasks. The main reason is that DeBERTa has a new positional encoding scheme that captures relative positions of tokens in the sequence, which is helpful to understand the structure among the tokens, as we discussed in Section 3.2. Furthermore, we find that the performance of DistilRoBERTa and DistilBERT are slightly worse than other base size models, mainly because distillation models sacrifice performance for time. Note that, in what follows, we will use DeBERTa by default.

Finding 1: Overall, we find that DeBERTa is the most suitable encoder for Unicorn, which requires structure-aware encoding and high generalization.

Exp-2: Which strategy performs well in Mixture-of-Experts of Unicorn? Table 3 reports the overall results for all tasks. Unicorn w/o MoE is a variant of Unicorn that has no MoE layer, i.e., only using an Encoder and a Matcher. Unicorn is our proposed framework, which consists of an Encoder, a MoE intermediate layer, and a Matcher. Unicorn ++ is our proposed Unicorn improved with MoE optimization for Expert Routing (see Section 4.2).

We find that both Unicorn and Unicorn ++ are better than Unicorn w/o MoE, which means that MoE is helpful to our proposed unified architecture. In particular, Unicorn ++ achieves the
best performance for most tasks and on average, which shows that Expert Routing can improve the overall performance. To provide an in-depth analysis, we visualize the average utilization weights of the Experts in each task using a heatmap shown in Figure 6 (the darker the color, the greater the weight). For Unicorn, Figure 6 (a) shows that all tasks mainly use the first four Experts, while the weights of the Experts are even. For Unicorn ++, Figure 6 (b) shows that the distinction of the Experts is more obvious.

Finding 2: The MoE intermediate layer is quite helpful for Unicorn, while our MoE optimization for Expert Routing can further improve the overall performance.

Exp-3: How does a unified model Unicorn (i.e., trained using labeled datasets from multiple tasks) compare with specific models (i.e., each model is separately trained for only one task)? We compare performance with the previous SOTA methods: Previous SOTA in Table 3 shows the results of the best task-specific model, which are reported by the existing papers, for
Table 4. Zero-shot Performance of Unicorn (# of Labels is the number of labels needed by SOTA methods). Unicorn w/o MoE has no MoE layer. Unicorn has Encoder, MoE and Matcher. Unicorn-ins is improved with our proposed instruction technique.

<table>
<thead>
<tr>
<th>Type</th>
<th>Task</th>
<th>Metric</th>
<th>Unicorn w/o MoE</th>
<th>Unicorn</th>
<th>Unicorn-ins</th>
<th>SOTA ( # of labels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>DBLP-Scholar</td>
<td>F1</td>
<td>90.91</td>
<td>95.39</td>
<td>97.08</td>
<td>95.6 (22,965)</td>
</tr>
<tr>
<td>CTA</td>
<td>Limaye</td>
<td>Acc.</td>
<td>96.2</td>
<td>96.5</td>
<td>96.5</td>
<td>96.8 (80)</td>
</tr>
<tr>
<td>EL</td>
<td>Limaye</td>
<td>F1</td>
<td>74.16</td>
<td>78.92</td>
<td>82.8</td>
<td>82 (-)</td>
</tr>
<tr>
<td>StM</td>
<td>Product</td>
<td>F1</td>
<td>60.71</td>
<td>74.92</td>
<td>78.76</td>
<td>67.18 (1,020)</td>
</tr>
<tr>
<td>ScM</td>
<td>DeepMDatasets</td>
<td>Recall</td>
<td>74.07</td>
<td>92.59</td>
<td>96.3</td>
<td>100 (-)</td>
</tr>
<tr>
<td>EA</td>
<td>SRPRS: DBP-WD</td>
<td>Hits@1</td>
<td>95.55</td>
<td>97.25</td>
<td>96.17</td>
<td>99.6 (4,500)</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td></td>
<td>81.93</td>
<td>89.28</td>
<td>91.27</td>
<td>90.2</td>
</tr>
</tbody>
</table>

Each corresponding task. Because eight separated PLMs are used for BERT-INT [46], Ditto [30] and TURL [10], while other solutions use lightweight strategies without PLMs. For simplicity, we compute the model size of previous SOTA by just summing model sizes of the PLMs.

The experimental results show that our unified models (i.e., Unicorn and Unicorn ++) outperform the previous SOTA methods on 15 over 20 tasks. For example, our unified models achieve the best accuracy on all tasks of column type annotation, significantly outperforming the previous SOTA method. Overall, our unified model Unicorn ++ gains an average evaluation score 94.56, while the previous SOTA is 91.84. Moreover, thanks to task unification, the model size of a unified Unicorn is much smaller, i.e., 147M vs. 995.5M, where 147M is 139M + 8M for DeBERTa-base plus MoE layer and 995.5M is 125M × 5 + 14.5M + 178M × 2 for five Ditto models, one TURL model, and two BERT-INT models in Previous SOTA (see Section 5.1 for calculation details of each model), compared with multiple specific models. The performance superiority of Unicorn is attributed to its multi-task learning that enables the unified model to learn from multiple tasks and multiple datasets to make full use of knowledge sharing. Specifically, as discussed in Exp-2, the data domains (e.g., bibliographies, products, and so on) of entity matching and string matching tasks are similar, which enables Unicorn to have a better understanding of these specific data. Entity alignment and entity linking enable Unicorn to have a better understanding of knowledge graph entities. Column type annotation and schema matching have similar data elements (columns), which can improve the understanding ability of column data for Unicorn.

Finding 3: Our unified model achieves better performance on most datasets and on average, compared with the SOTA specific models trained for ad-hoc tasks and datasets separately.

5.3 Evaluation on Zero-Shot Prediction

Exp-4: How does Unicorn perform on unseen tasks with a zero-shot setting? This section explores the zero-shot effectiveness of our trained unified model Unicorn, i.e., directly using the trained Unicorn to predict unseen new tasks without any labels. Specifically, for each task type, we randomly choose one task as new unlabeled task for testing, and only use the remaining tasks for multi-task training of Unicorn. As shown in Table 4, the results of Unicorn with zero label are comparable with previous SOTA methods with many labels. For example, for the entity matching task on DBLP-Scholar, the best variant of Unicorn, i.e., Unicorn-ins achieves 97.08 on F1 score with zero label, outperforming 95.6 of Ditto [30] with 22,965 labels. We also find that the overall performance of Unicorn-ins is better than that of previous SOTA methods, i.e., 91.27 vs. 90.2. These results show that Unicorn can also well serve new matching tasks with zero-shot learning,

Table 5. Zero-shot Performance of Unicorn for new unseen task types. (# of Labels is the number of labels needed by SOTA methods). Unicorn-ins is improved with instruction.

<table>
<thead>
<tr>
<th>Type</th>
<th>Task</th>
<th>Metric</th>
<th>Unicorn-ins</th>
<th>SOTA (# of labels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>DBLP-Scholar</td>
<td>F1</td>
<td>94.5</td>
<td>95.6 (22,965)</td>
</tr>
<tr>
<td>CTA</td>
<td>Limaye</td>
<td>Acc.</td>
<td>96.23</td>
<td>96.8 (80)</td>
</tr>
<tr>
<td>EL</td>
<td>Limaye</td>
<td>F1</td>
<td>79.59</td>
<td>82 (-)</td>
</tr>
<tr>
<td>StM</td>
<td>Product</td>
<td>F1</td>
<td>74.26</td>
<td>67.18 (1,020)</td>
</tr>
<tr>
<td>ScM</td>
<td>DeepMDatasets</td>
<td>Recall</td>
<td>88.89</td>
<td>100 (-)</td>
</tr>
<tr>
<td>EA</td>
<td>SRPRS: DBP-WD</td>
<td>Hits@1</td>
<td>97</td>
<td>99.6 (4,500)</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td></td>
<td>88.41</td>
<td>90.2</td>
</tr>
</tbody>
</table>

as it has good matching knowledge sharing ability for new tasks. This also encourages us to release checkpoints of Unicorn, which are trained from multiple data matching tasks or multiple datasets for researchers and practitioners to use.

Finding 4: Unicorn can be effectively applied to new tasks, which are not included in the multi-task training, in a zero-shot setting such that pairs in the new task are unlabeled.

Exp-5: Whether our proposed MoE and instruction techniques are helpful in the zero-shot setting? We evaluate the performance of zero-shot learning on new tasks for three variants of Unicorn, namely Unicorn w/o MoE, Unicorn and Unicorn-ins. Table 4 shows the experimental results. We can see that Unicorn w/o MoE has the worst performance among the three variants. This result indicates that the MoE Layer is more conducive to preserving the performance of model on a new task, that is, to better use knowledge of seen tasks to predict on new unseen task. The main reason is that, the MoE layer learns from data instances what is the best way of aligning distributions between a new task and existing tasks, such that the models trained on the existing tasks can be used (or adapted) to the new task.

We also find that Unicorn-ins achieves the best performance, i.e., outperforming the original Unicorn by about 2% on average, as reported in Table 4. This result shows that the simple instruction template in Equation (4) is also quite promising in building one model to unify data matching tasks. There are some recent studies on exploring more sophisticated instruction or prompting techniques for matching tasks [36, 51, 52]. We will explore this direction as future work.

Finding 5: There is great potential for studying Mixture-of-Experts and instruction on Unicorn to further improve the performance of zero-shot predictions.

Exp-6: How does Unicorn perform for unseen tasks of totally new task types? We evaluate the performance of Unicorn for unseen tasks of totally new task types. To this end, we conduct experiments with Unicorn-ins, a version of Unicorn improved by our proposed instruction technique. The results are reported in Table 5, where each row represents an unseen task (i.e., Task) of a total new task type (i.e., Type). Take the first row of Table 5 as an example. For testing DBLP-Scholar of entity matching, we remove all the tasks/datasets of entity matching from the training data of Unicorn-ins. Similarly, for testing Limaye of column type annotation, the training data of Unicorn-ins does not contain any task/dataset of column type annotation. We can see that the performance of Unicorn-ins is comparable to that of the SOTA specific models, e.g., 88.41 vs. 90.2 on average, and sometimes Unicorn-ins is even better. For example, on the Product task, Unicorn-ins with zero labels performs better than the SOTA specific model with 1,020 labels (74.26...
vs. 67.18). The results show that our unified model is competitive to the SOTA specific models even if the unified model has never seen the type of a task and has no specific labels from the task.

The above results inspire us to conduct an in-depth analysis on the knowledge sharing capability of Unicorn across different task types. To this end, we first consider a pair of task types, e.g., schema matching (ScM) and column type annotation (CTA), and then compare the results of one task type with and without the other task type used for training Unicorn. Some representative experimental results are reported in Figure 7, from which we have following interesting observations.

1) **Similar task types.** We first investigate similar task types that involve same data elements. Specifically, we examine column type annotation (CTA) and schema matching (ScM), both of which consider the column element. As shown in Figure 7(a), training Unicorn with CTA tasks is helpful for the ScM task DeepMDatasets (96.3 vs. 70.37 on Recall). Similarly, Figure 7(b) shows that using ScM tasks to train Unicorn is also helpful for the CTA task Limaye (95.96 vs. 85.89 on Acc.).

2) **Similar tasks.** We also examine similar tasks in common domains, e.g., products, citations, etc. Specifically, we find that the two task types, entity matching (EM) and string matching (StM), have similar tasks in common domains. For example, The EM tasks, Walmart-Amazon and iTunes-Amazon, are about products, while the StM task Product is also about products. Thus, we report the knowledge sharing capability of Unicorn across these two task types. Figure 7(c) shows that training Unicorn with EM tasks is very helpful for improving the performance of the EM task Product (59.98 vs. 18.76 on F1), and vice versa (see Figure 7(d)).

It is worth noting that the knowledge sharing capability of Unicorn is only preliminarily discussed here, and we will conduct a more systematic study in our future work.

**Finding 6:** Unicorn performs well for unseen tasks of totally new task types in the zero-shot setting. Moreover, Unicorn enables the possible opportunities of knowledge sharing among different task types.

6 RELATED WORK

6.1 Data Matching Tasks

The “data matching” process is central to most, if not all, data integration problems. We consider the following common data matching tasks in this paper. 1) **Entity matching** discovers duplicate entities (or tuples), i.e., those refer to the same real-world objects [7, 18]. Existing solutions define similarity functions based on attributes (e.g., Magellan [14]) or use pre-trained language models to
predict results as a binary classification task (e.g., DeepER [19], Ditto [30], and DADER [48])). (2) Entity linking [10] is a task of determining if a reference entity in a knowledge base is the potential mention of a tuple within table. Existing solutions typically retrieve potential entity mentions from knowledge bases and then sort them by calculating their similarities (e.g., Hybrid II [20]). (3) Entity alignment refers to the task of identifying equivalent entities across different knowledge graphs (KGs) [56]. Existing solutions mainly learn entity embedding and realize the matching of embedding through graph neural networks (e.g., GCN [26]) or pre-trained language models (e.g., BERT-INT [46]). (4) String matching [37] decides whether two strings are equivalent, either synthetically or semantically. Existing solutions use string similarity functions [50] or machine learning methods such as decision tree to predict the results (e.g., Smurf [39], Falcon [8]). (5) Column type annotation decides semantic type (e.g., Gender) of a column (e.g., \{male, female, female, \ldots\}). Existing solutions use a single cell embedding or column embedding to represent a column through neural networks and then determine its type by similarity function or machine learning, where the type is a string or category (e.g., HNN+P2Vec [5], TURL [10]). (6) Schema matching aims at finding the correspondence between schemas across different tables [27]. Existing solutions commonly define heuristic rules or calculate similarities between schemas (e.g., COMA [12]). (7) Ontology matching [16] is the problem of finding semantic mappings between two given ontologies. Existing solutions calculate similarities such as Jaccard similarity cross different nodes of ontology and use machine learning to predict the results (e.g., GLUE [15]).

Although these problems have been studied for several decades, because they lie at the heart of data integration with almost all applications, they remain to be important. Existing approaches try to solve each problem separately, e.g., using different frameworks or different training methods. Moreover, these solutions are not only task-specific, but also oftentimes dataset-specific (e.g., for each new task, saying entity matching between Walmart and Amazon products, we have to re-train the model only for serving this dataset). Different from them, Unicorn aims at a unified matching model that can be used for multiple matching tasks and datasets.

### 6.2 Transformer-based Language Models

The paper “Attention Is All You Need” [49] introduces a novel architecture called Transformer, which uses the attention mechanism for sequence to sequence learning. Transformer was originally proposed for natural language processing. Recently, Transformer has shown good performance for images [17], as well as for many modalities such as DeepMind Gato [42]. In addition, Transformer-based language models such as BERT [11] and RoBERTa [32] have also been used to deal with some database tasks (such as PASTA [23] and SCPrompt [22] for Text-to-SQL translation, and Symphony [6] for querying data lakes) and data integration tasks (such as Ditto [30] for entity matching and DeepBlocker [47] for the blocking task in entity matching).

Inspired by these recent successes using Transformer-based models in a wide range of applications, in this paper, we investigate the possibility of training a unified model that is generally capable of supporting a large number of matching tasks, which is crucial to data integration and data management, but is not well explored.

### 6.3 Mixture-of-Experts Models

Mixture-of-Experts [21, 25, 28, 44] (abbreviated as MoE) is an ensemble learning technique that implements the idea of training multiple experts on sub-tasks. Generally speaking, MoE trains each expert to examine a different part of the space (e.g., different tasks of input data in our problem). A gating network is responsible for combining various experts. Recently, MoE has been widely used in big tech companies such as Google, Microsoft, and Facebook.
MoE is a good fit to Unicorn for two reasons. First, Unicorn aims at supporting multiple data matching tasks with different semantics and various input formats, for which each expert can focus on learning matching relevant to specific data inputs. Second, equipping Unicorn with MoE will make it easily extensible, e.g., to support a new matching task between a tuple and an image (see our discussion in Section 7).

7 CONCLUSION AND FUTURE WORK

We have proposed Unicorn, the first unified model for supporting multiple data matching tasks. So far, Unicorn supports seven data matching tasks over five different types of data elements. This unified model can enable knowledge sharing by learning from multiple tasks and multiple datasets, and also support zero-shot prediction for new tasks with zero labeled pairs. We developed a general framework for Unicorn that employs an Encoder, a Mixture-of-Experts and a Matcher. We conducted experiments on 20 datasets of the seven matching tasks. Experimental results show that Unicorn is not only comparable or even better than SOTA specific models, but also well performs zero-shot learning on unseen new tasks.

Unicorn still has a lot of potential to expand. The optimized MoE and instruction we discussed lead to obvious improvements, and we will continue to explore more possibilities in the future, e.g., using data augmentation techniques to synthesize new labeled data, to reduce human annotation cost and improve model performance. Another interesting future work is to extend Unicorn to support more modalities (e.g., images), such as whether a picture matches a person (e.g., Michael Jordan) in a knowledge graph. As data matching tasks are typically supervised, for the tasks Unicorn already supports and those that we plan to support as discussed above, another immediate future work is to collect more labeled data and enrich our current benchmark.

ACKNOWLEDGMENTS

This work was partly supported by the NSF of China (U1911203, 62122090, 62072461, 62072458, 62232009, and 61925205), National Key Research and Development Program of China (2020YFB2104101), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES


Received July 2022; revised October 2022; accepted November 2022