Knowledge Graph Imputation

by

Mohammadali Niknamian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Supervisor:	Ihab F. Ilyas Professor, David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	M. Tamer Özsu Professor, David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Jimmy Lin Professor, David R. Cheriton School of Computer Science University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Knowledge graphs are one of the most important resources of information in many applications such as question answering and social networks. These knowledge graphs however, are often far from complete as there are so many missing properties and links between entities. This greatly affects their usefulness in applications that they are used in. Many methods have been proposed to alleviate this problem. One of the most prominent and studied subjects in this area are the graph embedding and link prediction methods. However, these methods only consider the relations between entities in knowledge graphs and completely ignore their literal values and properties that account for 41% of the facts in the knowledge graph YAGO4. They also do not scale for large knowledge graphs and their inference process for imputing missing links is by nature quadratic with respect to the number of entities in the knowledge graph. Furthermore, the embedding vectors that represent entities and relations might not be able to capture information that is necessary for inference for millions of entities that exist in large-scale knowledge graphs. We present a novel method based on the HoloClean's framework — a powerful cleaning tool for relational data. Our system is designed based on the open-source HoloClean and can be used to integrate multiple and different signals from various knowledge graph completion methods which allows us to holistically tackle this problem. We have done a thorough experiment on the YAGO4 dataset with 5M entities and 20M facts and we were able to enlarge the knowledge graph by roughly 12% with an average reconstruction precision of 0.81 on 162 different classes.

Acknowledgements

I would first like to thank my supervisor professor Ihab Ilyas. This work would not have been possible without his guidance and support.

I want to express my gratitude to my readers, professor Tamer Özsu and professor Jimmy Lin for taking the time to read this thesis.

Finally, I would like to thank my family for their continuous support during my studies.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Knowledge graphs like Wikidata [31], DBPedia [4], Freebase [5] and YAGO [26] are an important resource for many downstream AI applications like question answering, search and smart health care [2]. They are an important part of search engines, social network and e-commerce sites where they are used to store and query information as facts in the form of triples.

However, these knowledge graphs are often far from complete which greatly hampers their effectiveness in the aforementioned tasks. Many methods have been proposed to address this problem. Examples of these include embedding models such as RefE [8] and TransE [6]. These models try to learn an embedding in a multi-dimensional space for each entity and each relation in the knowledge graph and using a scoring function they evaluate the validity of a potential triple. These methods however fall short in imputing literal values. Another issue with these methods is that they may not scale to large-scale knowledge graphs due to inability of embedding vectors to capture information for millions of entities and the quadratic nature of inference process in these methods.

In light of these problems, in this thesis we are trying to use the highly sophisticated data cleaning tool HoloClean [24] and the imputation tool AimNet [33] which is embedded inside HoloClean in order to impute missing properties of entities in knowledge graphs.

1.1 Knowledge Graphs

A knowledge graph is a collection of interlinked entities that refer to real world objects. They are stored in the form of triples (subject, predicate, object) denoted as (s, p, o).

Subjects refer to entities in the model. The object could be another entity or a literal value representing a property of the subject. The predicate states the relation between the subject and object. An example of a knowledge graph is depicted in table 1.1:

subject	predicate	object
<Canada>	<has_capital>	<Ottawa>
<Canada>	<prime_minister>	<Justin_Trudeau>
<Canada>	<has_population>	"38,008,005"
<Canada>	<rdf:type>	<Country>
<Country>	<rdfs:subClassOf>	<Place>
<Justin_Trudeau>	<was_born_in>	<Ottawa>
<Justin_Trudeau>	<rdf:type>	<Person>

Table 1.1: Here we have knowledge graph which contains some facts about Canada and Justin Trudeau. The object of a triple could be another entity like <Ottawa> or a literal value like the population of Canada. We also have triples with predicates <rdf:type> and <rdfs:subClassOf> which are part of the ontology of the knowledge graph.

The core idea behind knowledge graphs is using graphs to represent data. A graph-based abstraction of knowledge has multiple advantages compared to relational models. They provide an intuitive abstraction that can be used for a variety of domains where edges capture the relations between the entities. Moreover, they allow maintainers to postpone the definition of a schema which allows data to evolve with more flexibility compared to a relational setting in a situation where we are capturing an incomplete knowledge [16].

We use the Resource Description Framework (RDF) [10] as our data model. It is a standardised data model based on directed edge-labelled graphs which has been recommended by the W3C. The RDF model uses *International Resource Identifiers* (IRIs) [12] that allows us to have global identifications for entities. We also have *literals* that allow for representing strings and other data types such as integers and dates [16].

1.1.1 Ontology

Ontology represents the formal structure and semantics of a knowledge graph. They determine the data schema of entities in the knowledge graph. A knowledge graph is basically a data graph —a collection of data represented as nodes and edges using a data model like RDF which is enhanced with representations of schema, taxonomy and rules [16].

We assume that the ontology of the knowledge graph is expressed in *Web Ontology Language* (OWL) [15]. There are several types of metadata such as *rules* and *contexts* that are part of the ontology of a knowledge graph. Here we focus on three components of the ontology:

- **Classes**: They are groupings of entities based on the type of real world objects they refer to. Each entity might belong to one or more classes via `<rdf:type>` predicate. Examples of classes include `<Person>`, `<Movie>`, `<Organization>`, etc.
- **Schema**: Derived from ontology feature of properties `<rdfs:domain>`, each class has a schema of properties associated with it. In other words, the schema of a class is the collection of all properties that have that class as their domain.
- **Taxonomy**: Derived from ontology feature of classes `<rdfs:subClassOf>`, we have a hierarchy of classes. Each class has a super class (except for the highest-level class) and the entities of a class also belong to its super class. An example of this is `(<Country>, <rdfs:subClassOf>, <Place>)` which is in in table 1.1.

1.1.2 Examples of Knowledge Graphs

Large-scale knowledge graph can be divided into open knowledge graphs and enterprise knowledge graphs [16]. Here are some of the most prominent open knowledge graphs:

- **DBpedia**: A knowledge graph that was developed to extract graph structures from semi-structured data embedded in Wikipedia articles [4]. It is further enriched by linking to external datasets like GeoNames and WordNet [21].
- **YAGO**: Like DBpedia it also extracts graph-structured data from Wikipedia. But in order to have more accurate data with higher quality it mostly extracts data from infoboxes. The results are unified with the hierarchical structure of WordNet to create the ontology.
- **Freebase**: Unlike DBpedia and YAGO that were generally extracted from Wikipedia and WordNet, Freebase was mostly a collection of knowledge from human editors [16]. Freebase was acquired by Google in 2010 and was made read-only in 2015.
- **Wikidata**: Created by the Wikimedia foundation, its purpose is to act as a centralized, collaboratively-edited knowledge graph to support Wikipedia in order to alleviate the problem of contradictory data in different articles across different languages.

Examples of enterprise knowledge graphs include The knowledge graphs that are used by Google [25], Amazon [11], eBay [23] and Airbnb [9].

We choose to work with the latest version of YAGO which is called YAGO4 [28]. It provides a light version of the knowledge graph which only includes entities that have a Wikipedia article with around 5M entities and 20M facts. It has more convenient file sizes and allows us to have reasonable process runtimes on the entire knowledge graph (1-2 days).

1.2 Existing Methods

According to [16] methods for completing knowledge graphs can be divided into *Deductive methods* and *Inductive methods*.

1.2.1 Deductive Methods

These are methods that exploit the rules in the ontology to entail and accumulate further knowledge. Given the data as a premise, and some rules about the data which can be defined in the ontology or by a domain expert, we can use a deductive process to create data more than what is explicitly given by the data [16]. These rules can also be inferred using rule mining systems.

As an example we may have a rule that states the citizenship status of a person is the same as his child: $(a, citizen\_of, c) \wedge (a, has\_child, b) \rightarrow (b, citizen\_of, c)$. There are rule mining system such as AMIE [14] and path ranking algorithms such as [17]. These approaches are orthogonal to our approach but we might be able to derive a holistic approach using Denial Constraints from RDF [13] in HoloClean. We discuss further details about this in chapter 6.

1.2.2 Inductive Methods

In deductive methods, knowledge is acquired by precise logical consequences. In inductive methods we try to generate additional data by generalizing patterns from a given set of input observations which can lead to potentially imprecise predictions [16].

Most notable examples of inductive methods are graph embedding models. They use self-supervision to learn low-dimensional numeric representations for entities and relations

between them. They usually comprise of two components: (1) a process to learn the embedding of entities and relations between them and (2) a scoring function that measures the plausibility of a query triple [2]. However we observed 2 main problems with these methods:

1. These methods basically ignore literal properties and are only able to predict relation between entities and cannot be used to impute missing literal properties like names and dates. Literal properties account for 41% of the facts in the knowledge graph YAGO4.

2. The second issue which is more important is the scalability of these methods. Given a query triple (s, p, ?), these methods replace the object with all entities in the knowledge graph and output the triple that has the highest score by the scoring function. This process is quadratic in nature with respect to the number of entities.

    - The state of the art model on the YAGO3-10 dataset (which is a subset of YAGO3 [19] that only contains entities that are involved in at least 10 relations with only 123K entities and around 1M facts) RefE [8], takes about 3 hours to converge and has a Hits@1 [7] of only 0.503.

1.3 Problem Definition

Here we present the formal definition of the problem we are trying to solve. As input we have:

- A knowledge graph G consisting of triples (subject, predicate, object).
- A set of classes C. Each class $c \in C$ has a schema $R(c)$ of predicates $\{P_1, P_2, ..., P_N\}$. Each entity might belong to one or more of these classes.
- A taxonomy of classes. Every class c has a super class $Super(c) \in C$ (except for the highest level class) and a set of sub classes $SubClasses(c) \subset C$.
- A target attribute T.

Given the above input components, our objective is to impute missing values of attribute T for entities in classes c where we have $T \in R(c)$.

There are two points that are worth mentioning: (1) every class inherits the schema of its super class: $R(Super(c)) \subset R(c)$ and (2) class membership is transitive. It means that if an entity belongs to a class, it also belongs to its super class: $e \in Subjects(c) \rightarrow e \in Subjects(Super(c))$.

1.4 The Approach

In our approach we use the HoloClean [24] framework and the imputation tool AimNet [33] in order to impute missing values of target attribute T for entities where T is an acceptable property (entities that belong to a class that has T in its schema).

More specifically, we feed each class c where we have $T \in R(c)$ to HoloClean as if they are relational tables. Each tuple represents a distinct entity $e \in Subjects(c)$ and each column represents a property $P_i \in R(c)$. A cell $P_i[e]$ has the value v if and only if we have the triple (e, P_i, v) in our knowledge graph. We provide and example of this in figure 1.1.

We note that with the presence of non-functional predicates (predicates that might have multiple objects for a single subject such as `<has_child>`) in the knowledge graph, this process is not very straight-forward. We will discuss the details of our implementation on how we handle non-functional predicates in chapter 5.

1.4.1 Technical Challenges

Here we observed three challenges:

1. **Overlapping classes**: Due to the hierarchical structure of the classes, an entity might belong to multiple classes. Here two questions arise:
   - For a given entity, how do we pick the best class or classes that it is a member of to feed to HoloClean in order to impute its target attribute?
   - In case we feed an entity along multiple classes, how do we deal with different imputed values within each class?
2. **Irrelevant attributes**: Having found the right classes, we might be feeding attributes to HoloClean that are irrelevant to the target attribute. An example of this is feeding `<birthDate>` as a context attribute for imputing `<nationality>`. These irrelevant attributes might lower the accuracy of our imputation and inflict more computation cost to the system.

subject	predicate	object
<Tadeusz_Borowski>	<rdf:type>	<Person>
<Tadeusz_Borowski>	<nationality>	<Poland>
<Tadeusz_Borowski>	<hasOccupation>	<Writer>
<Tadeusz_Borowski>	<memberOf>	<Polish_United_Workers>
<Tadeusz_Borowski>	<birthPlace>	<Zhytomyr>
<Vasily_Nebenzya>	<rdf:type>	<Person>
<Vasily_Nebenzya>	<nationality>	<Russia>
<Vasily_Nebenzya>	<hasOccupation>	<Politician>
<Vasily_Nebenzya>	<birthPlace>	<Volgograd>
<Bud_Poile>	<rdf:type>	<Person>
<Bud_Poile>	<nationality>	<Canada>
<Bud_Poile>	<memberOf>	<Detroit_Red_Wings>
<Bud_Poile>	<birthPlace>	<Fort_William>

id	<nationality>	<hasOccupation>	<memberOf>	<birthPlace>
1	<Poland>	<Writer>	<Polish_United_Workers>	<Zhytomyr>
2	<Russia>	<Politician>	NULL	<Volgograd>
3	<Canada>	NULL	<Detroit_Red_Wings>	<Fort_William>

Figure 1.1: In the table at the top we have some triples for entities belonging to the class <Person>. At the bottom we have transformed the triples into tuples of a relational table which will be fed to HoloClean. We point out the existence of NULL values as this process will often results in sparse tables.

3. **Sparsity**: Even though each class might form a homogeneous dataset, it might still be very sparse due to the sparse nature of knowledge graphs. Our imputation model will treat nulls as another distinct label. These null values act as noisy cells to our model and will result in lowering our imputation precision. If we somehow manage to get the correct imputation using other known attributes, the confidence probability of the prediction might be very low due to null values. Now if we employ a threshold for accepting a prediction, these imputed values might fall below that threshold and thus get ignored.

1.4.2 Contributions and Outline

We divide our contributions into four parts:

1. We use the classes in the knowledge graph as the context for imputation in HoloClean. However, these classes have a hierarchical structure and are heavily overlapping each other. One issue is that given an entity how do we choose the best classes that could be used as contexts for imputing its target attribute and second, how do we deal with different imputed values that could occur along different classes and contexts. We offer three distinct solutions each with a unique way of dealing with these problems. We choose one of these solutions as our final method for selecting the right classes in our end-to-end model and we show in the results section 5.2 that this solution yields better outcomes compared to the other two solutions.

2. Having found the right classes, they might have attributes in their schema that are completely irrelevant to the target attribute. These irrelevant attributes might distract our imputation model and inflict unnecessary costs. We offer a solution for this based on the FD-discovery tool FDX [35] that finds conditional independencies between attributes by creating a sparse probabilistic graphical model of the data that allows us to rule out irrelevant attributes. Our end-to-end model based on this approach beats all of our other models in terms of precision. But we argue that the attention model of AimNet is already able to put more weights on relevant attributes and show that this approach yields lower recall and overall F1-score with respect to our final model.

3. Knowledge graphs are by nature extremely sparse. Even classes that bundle homogeneous entities together can be very sparse since not all of their entities have all the attributes in the class's schema. We present a solution for the problem of sparsity inspired by the multi-head attention mechanism of [30] where we train multiple models based on the absence of important attributes. We will show that this approach alleviates the issue of low confidence probability that the imputation model assigns to correctly imputed target attributes of entities that are missing one or more context attributes.

4. We have designed and implemented a fully functional system using the HoloClean framework based on the solutions that we offer for each of our challenges. We present the details of the implementation in chapter 5. We have done extensive experiments on each of the baselines and solutions that we offer. We have been able to produce approximately 2.4M additional triples on the YAGO4 — enlarging the knowledge

graph by roughly 12% with an average reconstruction precision of 0.81 over 162 classes and 18 different target attributes.

The remainder of the thesis is structured as follows:

- In chapter 2 we discuss backgrounds and related works.
- In chapter 3 we provide solutions for handling the overlapping classes and irrelevant attributes.
- In chapter 4 present an approach for handling the sparsity as a separate problem and also present our end-to-end solution.
- In chapter 5 we present the details on implementation and we discuss how we handle non-functional predicates along with our final results.
- In chapter 6 we conclude and offer directions for future works.

Chapter 2

Background and Related Work

In this chapter we discuss the backgrounds that are needed in the following chapter and go through some of the methods that are related to knowledge graph completion. We introduce the data cleaning framework HoloClean [24], AimNet [33] which is an imputation tool embedded inside HoloClean and FDX [35] a functional-dependency discovery tool.

2.1 HoloClean

HoloClean is a framework for holistic data repairing using probabilistic inference. It unifies different data repairing methods that rely on integrity constraints or external data sources in order to perform data repairing on an inconsistent dataset [24]. The original paper is implemented in DeepDive [34]. The open-source system of the paper has the same structure and pipeline as the original paper but implements the inference components as neural networks in PyTorch [3]. We will explain the components of the open-source system as we will be using that in our work. For ease of understanding we will explain each part to the extent that is related to our problem and case as we might leave some details in some of the components.

HoloClean takes as input a an erroneous structured dataset D with attributes $A = \{A_1, A_2, ..., A_N\}$. D is represented as a set of tuples $t \in D$ each comprised of a set of cells denoted as $Cells[t] = \{A_i[t]\}$. For each cell c we show its true unknown value by v_c^* and its initial observed value by v_c. An error in D is a cell c where we have $v_c \neq v_c^*$. The goal of HoloClean is to estimate the true value of erroneous cells denoted as $\hat{v}_c$.

The first stage in HoloClean's workflow is error detection which separates D into noisy and clean cells, denoted by D_n and D_c. HoloClean treats error detection as a black box and it is not part of main components in its pipeline.

HoloClean's pipeline is comprised of 4 main components which are domain generation, featurization, training and inference (see figure 2.1). We will explain each of these components in the following sub-sections.

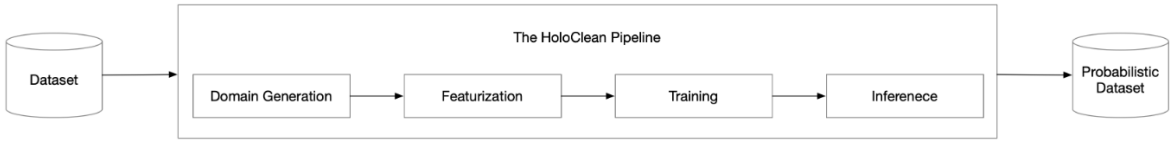


Figure 2.1: Pipeline of the HoloClean [3].

2.1.1 Domain Generation

After error detection, for each cell c HoloClean generates a domain of possible values for that cell denoted by $dom(c)$. We point out that it generates a domain for clean cells D_N as well in order to do negative sampling. For each cell $A_i[t]$ it generates an initial set of candidates. It takes the original value of the cell (if it has any) along values in the column A_i that have co-occurred with the values of other attributes in tuple t in other tuples. This however may result in a large number of candidates. In order to get a smaller domain for each cell it prunes the initial candidate set based on the posterior probabilities of each value using naive bayes:

$$Pr(A_i[t] = v | A_1[t], ..., A_N[t]) \propto Pr(A_i[t] = v) \prod_{j \neq i} Pr(A_j[t] | A_i[t] = v) \tag{2.1}$$

It uses the empirical values of these probabilities calculated using co-occurrence statistics between attributes and their values:

$$Pr(A_i[t] = v) = \frac{\#v}{\#tuples} \tag{2.2}$$

$$Pr(A_j[t] | A_i[t] = v) = \frac{cooccur(v, A_j[t])}{\#v} \tag{2.3}$$

$$cooccur(v_1, v_2) = \#(v_1 \text{ and } v_2 \text{ appear together in a tuple}) \tag{2.4}$$

2.1.2 Featurization and Training

For each value in the domain of a cell HoloClean creates a vector that is formed by integrating signals from multiple *featurizers* and encodes these signals as features. There are several featurizers that are implemented in the open-source HoloClean. Some examples are:

- Co-occurrence featurizer: It uses the co-occurrence statistics between the values of attributes. It generates the feature vector based on empirical probabilities.
- Constraint featurizer: Having a set of denial constraints for the input dataset, it generates a feature vector based on the number of rules that are violated by replacing each value in the domain in the dataset [3].
- Embedding featurizer: It uses the probability distribution that is generated by imputation tool AimNet. We will explain AimNet in more details in the next section.

After getting the feature vectors of all values in the domains from specified featurizers, they are concatenated together and are used for training the model of HoloClean which is a simple linear layer. The initial value of clean cells are treated as labels and other values that are generated for their domain could be used as negative samples.

2.1.3 Inference

After the model has been trained with the concatenated feature vectors on clean cells, it will be used on the feature vectors of the values from the domain of noisy cells using the same forward step as in the training phase to generate a probability distribution over the values in the domain of cells. The final inferred value for a cell will be the value with the maximum probability.

2.2 AimNet

AimNet is an attention-based learning network for missing data imputation in HoloClean that focuses on mixing discrete and continuous data [33]. It is an imputation tool that has been embedded inside HoloClean and acts as a featurizer called embedding featurizer.

In order to handle mixed discrete and continuous data, AimNet learns a combination of contextual embeddings for discrete data and projections for continuous data. It then uses a variation of the dot product attention mechanism to learn dependencies between different attributes of the input data and uses the attention weights to combine the contextual embedddings of attributes into a unified representation for a target attribute. At the end a mixed loss function is used to handle mixed data types during training. An overview of the AimNet's architecture is shown in figure 2.2.

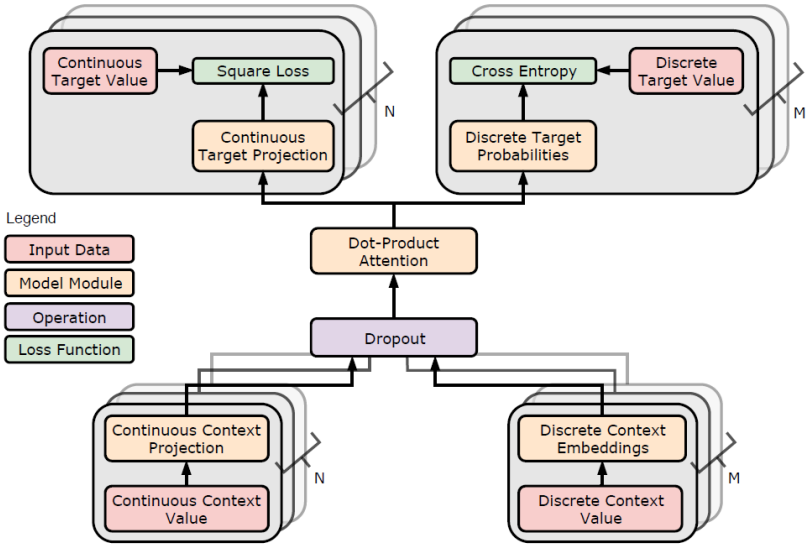


Figure 2.2: Architecture overview of AimNet [33]

2.2.1 Context Embeddings

AimNet transform each attribute value into a vector embedding of dimension k. For a continuous value $\overrightarrow{x}$ it performs a continuous context projection to dimension k by first standardizing each dimension of the input attribute values to zero mean and unit variance. Then it applies a linear layer followed by a non-linear ReLU layer to generate a non-linear transformation of the input:

$$\overrightarrow{z} = \mathbf{B}\sigma(\mathbf{A}\overrightarrow{x} + \overrightarrow{c}) + \overrightarrow{d} \tag{2.5}$$

where $\mathbf{A}, \mathbf{B}, \overrightarrow{c}, \overrightarrow{d}$ are all learned parameters.

For all discrete attributes and each value in their domain it associates a vector of dimension k to them by learning a lookup table of *discrete context embeddings* of the input values. This process is similar to learning the word embeddings in Word2Vec [20].

2.2.2 Attention Model

The difference between AimNet's attention model and other attention models is that AimNet learns how attend to different context attributes and not the values themselves given a target attribute. This means that the attention weights between attributes are fixed during inference time and are independent of their values.

Suppose we have N continuous attributes and M discrete attributes. To form the query Q of the attention mechanism, it associates a learnable encoding of dimension $N + M$ to each of the $N + M$ attributes. Then the position of each target attribute A_j is specified as a bitmask $K^{(j)} \in \{0, 1\}^{N+M}$ that form the key of the attention mechanism. The encoding in Q selected by $K^{(j)}$ is masked out by a leave-one-out bitmask $m \in \{0, 1\}^{N+M}$. Let matrix V be the concatenation of the context embeddings of all attributes which has $N + M$ rows each representing an embedding of dimension k. The output for a target attribute A_j is expressed as:

$$Att(Q, k^{(j)}, V) = (m \odot softmax((k^{(j)})^T Q))norm(V) \tag{2.6}$$

The resulting output is called a *context vector* and is used to do imputation on the target attribute. The architecture of the attention model is shown in figure 2.3(a).

2.2.3 Inference and Loss Function

For continuous target attributes, the context vector first goes through a fully-connected ReLU layer of dimension $k \times k$ and then is projected to the attribute's dimension d_j (see figure 2.3(b)). the mean squared loss is used as the loss between the predicted continous value and the actual continuous value.

For discrete target attributes, it first computes the inner product between the context vector that comes from the attention layer and the discrete value's vector embeddings in the domain. A softmax is applied on the inner products to produce prediction probabilities for each value in the cell's domain. In the end the Categorical Cross Entropy loss is computed between the probabilities and the actual target value (see figure 2.3(c)).

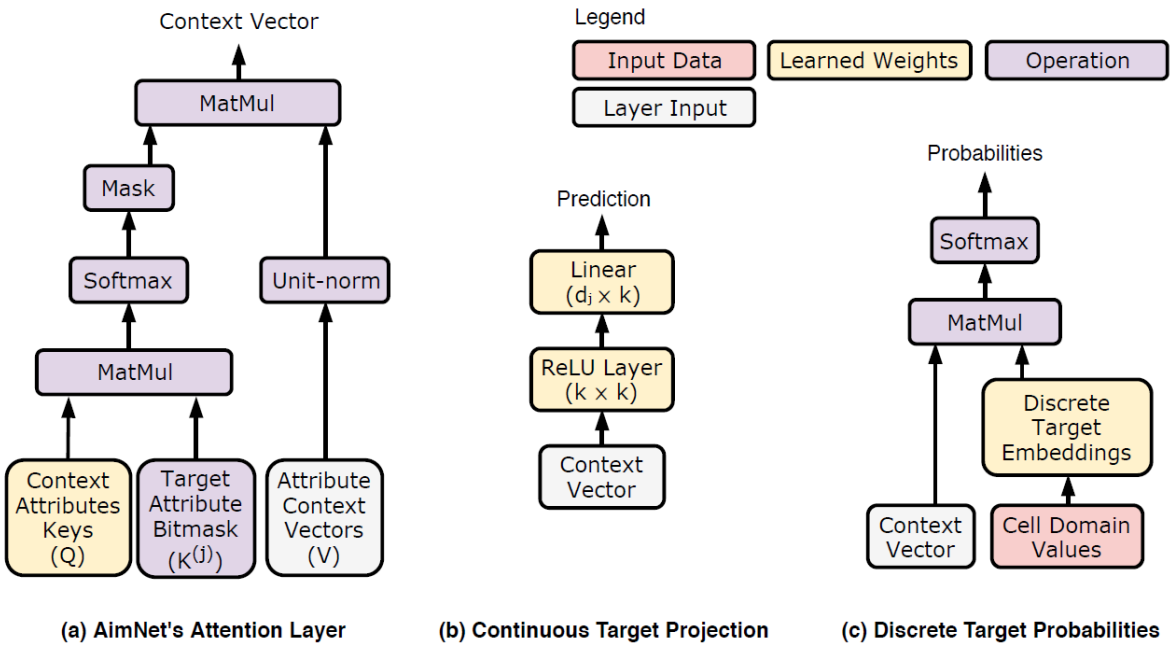


Figure 2.3: [33]

2.3 FDX

FDX [35] is a *Functional Dependency*(FD) discovery tool that we are going to be using in some of our methods. FDX has two main characteristics that are important for us: (1) It builds around the approximate form of FDs. This allows for the presence of erroneous and missing values in the dataset which is very suitable in our case. (2) It avoids generating spurious and complex FDs by learning a sparse probabilistic graphical model and using conditional independencies to rule out over complex FDs. This feature of FDX allows us to generate the probabilistic graphical model of the data and given a target attribute we would get all attributes that the target attribute is conditionally dependent on. We will call these attributes *Context Attributes*. The presence of these context attributes are enough for predicting the target attribute and we will use this property in two of our approaches. In the following sub sections we will briefly explain the approximate functional dependencies and the notion of conditional independence. We will then quickly explain the general mechanism of FDX.

2.3.1 Approximate Functional Dependencies

Having a dataset D with schema R and a set of attributes $X \subseteq R$ and a target attribute $Y \in R$, we say that an *Approximate Functional Dependency* holds iff for all pairs of tuples t_i and t_j in the dataset we have the following condition:

$$Pr(t_i[Y] = t_j[Y] | t_i[X] = t_j[X]) = 1 - \epsilon \tag{2.7}$$

with ϵ being a very small number.

2.3.2 Conditional Independence

Lets assume that X, Y and Z are random variables. We say X and Y are conditionally independent given Z if we have:

$$P(X|Y, Z) = P(X|Z) \tag{2.8}$$

2.3.3 Overview of FDX

Like we mentioned before FDX tries to generate the probabilistic graphical model of the attributes of the data. The absence of edges between nodes represents conditional independence of variables. FDX uses these conditional independencies to avoid generating spurious and complex FDs (Overfitting). It is a starndard result in statistical learning that one can learn the conditional independencies of a structured distribution by identifying the non-zero entries in the *inverse covariance matrix* of the data. The workflow of FDX's framework follows three steps (see figure 2.4):

1. **Dataset Transformation**. It uses the input dataset to generate a collection of samples that correspond to outcomes of the random events that capture equality across attribute values between two random sampled tuples.
2. **Structure Learning**. It learns the structure of the transformed dataset by obtaining a sparse estimate of its inverse covariance matrix.
3. **FD Generation**. It generates a collection of FDs by identifying non-zero entries of the inverse covariance matrix.

The most important feature of FDX for us is that given a target attribute T, it gives us all attributes in the dataset that the target attribute T is conditionally independent from given the presence of all other attributes.

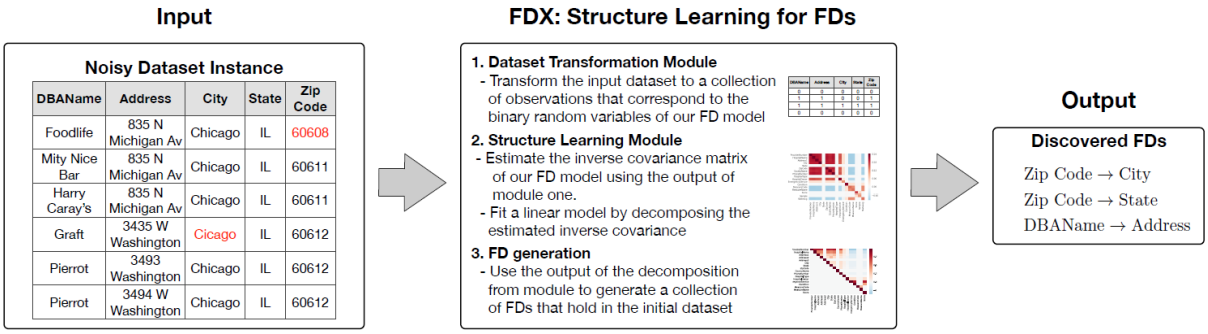


Figure 2.4: An overview of the framework of FDX [35].

2.4 Inductive KG Completion Methods

Like we mentioned in section 1.2, previous work on knowledge graph completion can be divided into two groups: Deductive Methods and Inductive Methods. Deductive methods try to exploit rules that are either defined in the ontology or are inferred using rule mining approaches to entail further knowledge. Inductive methods try to generalize patterns from a set of observations in order to generate new data. Graph embedding approaches that are used for link prediction are the most notable example of these methods.

We will briefly go over some of the most prominent approaches in graph embedding which are part of the inductive methods. Then we will introduce two rule-mining methods which are part of the deductive methods.

The input to a graph embedding problem is a knowledge graph G consisting of a set of entities E and a set of relations R. Triples are represented as (h, r, t) where $h, t \in E$ are the head and tail of in the triple and $r \in R$ is the relationship between them. The bold letters $\mathbf{h}, \mathbf{r}, \mathbf{t}$ represent the corresponding embeddings.

Graph embedding methods have two main components: (1) a scoring function that evaluates the plausibility of a given triple (h, r, t), and (2) a procedure to learn the embeddings of entities and relations in an optimization process that maximizes the score of correct (positive) triples and minimizes the score of incorrect (negative) triples. Positive triples are the ones that already exist in our training dataset and negative triples are generated by corrupting the positive triples where we replace the head or tail of positive triple with another entity. During inference, given a query triple (h, r, x) or (x, r, t), the unknown entity x is replaced with all entities in the knowledge graph and the entity which results

in the highest score for its corresponding triple is chosen as the answer.

In TransE [6], the scoring function is $f_r(h,t) = -\|\mathbf{h}+\mathbf{r}-\mathbf{t}\|^2$. It argues that the embedding of the tail entity should be a translation of the head entity based on the relationship between them. More specifically, it wants $\mathbf{h}+\mathbf{r} \approx \mathbf{t}$ when (h,r,t) holds, while $\mathbf{h}+\mathbf{r}$ should be far away from $\mathbf{t}$ otherwise. This methods has a small number of parameters and can be easily scaled but it is not suitable for modeling one-to-many, many-to-one, many-to-many and reflexive relations [2]. The reason for that is that if we consider the ideal case of no-error embedding where we have $\mathbf{h}+\mathbf{r}-\mathbf{t}=0$ for a positive triple (h,r,t), there will be two consequences:

- If r is a symmetric relation, it means that we could have both (h,r,t) and (t,r,h) as positive triples in our knowledge graph. Then we would have $\mathbf{r}=0$ and $\mathbf{h}=\mathbf{t}$.
- If r is a many-to-one relation, we could be having multiple triples (h_i,r,t) for $i = 0,1,...,m$ in our knowledge graph. Then we will have $\mathbf{h}_0 = \mathbf{h}_1 = ... = \mathbf{h}_m$. The same is true for a one-to-many relation.

TransH [32] tries to address TransE's issues by using multiple embeddings for a single entity in different relations. It interprets a relation as a translation but in a separate hyperplane. Each relation r is categorized by two vectors, the norm vector $(\mathbf{w}_r)$ of the hyperplane and the translation vector $(\mathbf{d}_r)$ on the hyperplane. For a triple (h,r,t), the embedding $\mathbf{h}$ and $\mathbf{t}$ are projected to the hyperplane defined by the norm vector $(\mathbf{w}_r)$ first to produce $\mathbf{h}_\perp$ and $\mathbf{t}_\perp$ respectively (see figure 2.5). Then it expects $\mathbf{h}_\perp$ and $\mathbf{t}_\perp$ to be connected by a translation vector $(\mathbf{d}_r)$ on the hyperplane if (h,r,t) is a positive triple in the knowledge graph. Therefore it defines the scoring function as $f_r(h,t) = -\|\mathbf{h}_\perp + \mathbf{d}_r - \mathbf{t}_\perp\|_2^2$. Now multiple entities in the head of many-to-one relation could have different embeddings while having the same projection on the hyperplane of the relation r.

TransR [18] argues that using the same semantic space for entities and relations, like how they do it in TransE and TransH is insuffcient as they are two completely different types of objects and thus they use different vector spaces $\mathbb{R}^k$ and $\mathbb{R}^d$ for entities and each relation. For each relation r they use a projection matrix $\mathbf{M}_r \in \mathbb{R}^{k\times d}$ to map embeddings of entities to the vector space of relation r:

$$\mathbf{h}_r = \mathbf{h}\mathbf{M}_r, \mathbf{t}_r = \mathbf{t}\mathbf{M}_r \tag{2.9}$$

and the score function is defined as: $f_r(h,t) = -\|\mathbf{h}_r+\mathbf{r}-\mathbf{t}_r\|_2^2$. They argue that learning a unique vector for each relation is insufficient as the relation may occur between completely

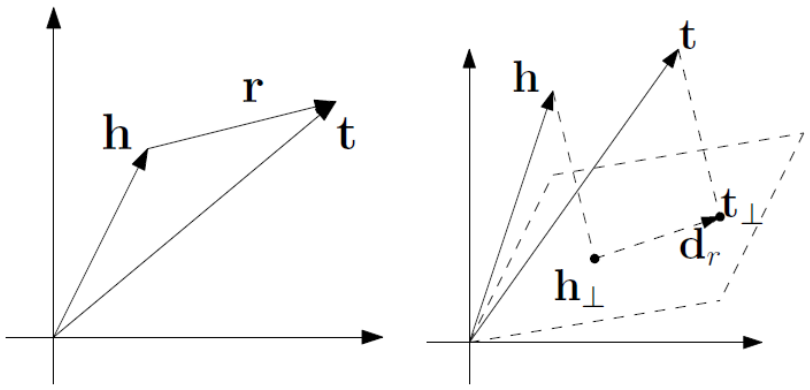


Figure 2.5: A comparison between TransE and TransH in interpreting vector embeddings of entities and relations [32].

different types of entity pairs. An example of this is the relation `<contained_in>` which has many patterns such as city-country, university-country and country-continent. Thus they introduce another method called Cluster-based TransR (CTransR). The basic idea in CTransR is that for each relation r they cluster entity pairs (h, t) that occur along that relation into several groups and learn a specific embedding of the relation for each cluster denoted as $\mathbf{r}_c$.

RotatE [27] tries to introduce a model that is able to model all symmetric, inverse and composed relations. These relation patterns are widely spread in knowledge graphs. We formally define these three relation patterns:

- A relation r is **symmetric** iff $\forall h, t\ (h, r, t) \in KG \longrightarrow (t, r, h) \in KG$.
- A relation r_1 is **inverse** to relation r_2 iff $\forall h, t\ (h, r_1, t) \in KG \longrightarrow (t, r_2, h) \in KG$.
- A relation r_1 is **composed** of relation r_2 and relation r_3 iff $\forall h, t, s\ (h, r_2, t) \in KG \wedge (t, r_3, s) \in KG \longrightarrow (h, r_1, s) \in KG$.

In order to model all three relation patterns defined above, RotatE maps the head and tail entities h, t to the complex embeddings: $h, t \in \mathbb{C}^k$. It then defines each relation as an element-wise rotation from head entity h to tail entity t. More specifically, having a triple (h, r, t) in the knowledge graph, it expects that: $\mathbf{t} = \mathbf{h} \circ \mathbf{r}$ where $|r_i| = 1$ and $\circ$ is the element-wise product. The constraint on the elements of $\mathbf{r}$ makes it equivalent to a rotation.

r_i is of the form $e^{i\theta_{r,i}}$ and thus corresponds to a counterclockwise rotation by $\theta_{r,i}$ around the origin of the complex plane that only affects the phases of the vector embeddings of the entities (see figure 2.6). It defines the score function as $f_r(h,t) = -\|\mathbf{h} \circ \mathbf{r} - \mathbf{t}\|$. By defining relations this way, a relation r is symmetric if and only if each element of its embedding is either 1 or -1: $r_i = \pm 1$. Two relations r_1 and r_2 are inverse if and only if their embeddings are conjugates: $\mathbf{r}_1 = \bar{\mathbf{r}}_2$. A relation r_3 is a composition of relations r_1 and r_2 if and only if $\mathbf{r}_3 = \mathbf{r}_1 \circ \mathbf{r}_2$.

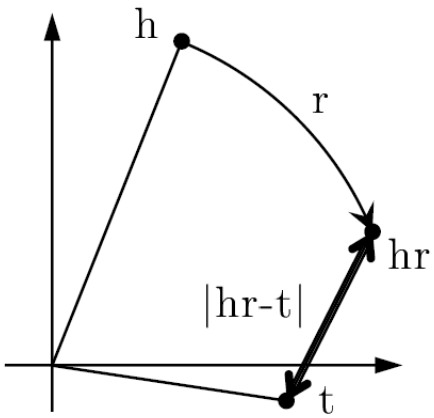


Figure 2.6: RotatE models r as a rotation in complex plane [27].

In order to learn the embeddings of entities and relations, these methods try to minimize a loss function. There are two loss function that are used the most which are margin-based loss function $L = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'} max(0, f_r(h,t) + \gamma - f_r(h',t'))$ and logistic loss $L = \sum_{(h,r,t)\in S\cup S'} log(1 + exp(-y_{hrt} \cdot f_r(h,t)))$ where y_{hrt} is the sign of the training example which take $+1$ for positive sample and -1 for negative samples. γ is the margin and S and S' are the sets of positive triples and negative triples.

Another group of approaches try to formulate the problem as a third order binary tensor completion problem where knowledge graph is represented as a partially observed tensor $Y \in \{0,1\}^{|E|\times|E|\times|R|}$ [2]. An entry in the tensor is 1 if the corresponding triple exists in the knowledge graph. Different models such as RESCAL [22] and Comp1Ex [29] use various approaches of tensor factorization to decompose Y and build the scoring function based on the learned factors.

RESCAL [22] represents a relation as a matrix $W_r \in \mathbb{R}^d$ that shows the interactions between embeddings of the entities. The scoring function is defined as $f_r(h,t) = \mathbf{h}^T W_r \mathbf{t}$.

Comp1Ex restricts relations to diagonal matrices and uses complex numbers instead of real numbers in order to handle symmetric and asymmetric relations [2].

2.5 Deductive KG Completion Methods

The first work in deductive methods is AMIE [14] which is a rule mining system inspired by association rule mining. They try to address the limitations of inductive logic programming (ILP) methods that are (1) requiring negative samples which we do not have in knowledge graphs under the Open World Assumption (OWA). This means that a fact that is not present in the knowledge graph in not necessarily false; it is just unknown and (2) not being able to scale to huge amount of data in today's knowledge graphs. They try to find *Horn rules* from the knowledge graph that are in the form of:

$$B_1 \wedge B_2 \wedge ... \wedge B_n \implies r(x, y) \tag{2.10}$$

where the left hand side $\{B_1, ..., B_n\}$ is the body consisting of n atoms and $r(x, y)$ is the head representing the triple (x, r, y). This means that if all atoms of the body are true for a given instantiation, then a new fact instantiated by the head can be added to the knowledge graph. An example of such rule can be:

$$motherOf(m, c) \wedge marriedTo(m, f) \implies fatherOf(f, c) \tag{2.11}$$

They consider two atoms in a rule that share a variable or entity to be connected. A rule is *connected* if every atom is connected transitively to all other atoms in the rule. They also define *closed* rules which are rules where every variable in the rule appears at least twice. AMIE only mines closed connected rules.

Their algorithm revolves around starting from an empty rule and adding atoms to it one at a time using different mining operators. Their goal is to find rules that can find maximum number of unknown positive facts in the knowledge graph. They use three different mining operators:

1. **Add Dangling Atom**: This operator adds a new atom to a rule that uses a fresh variable for one of its two arguments. The other argument is shared with some other atom in the rule.

2. **Add Instantiated Atom**: This operator adds a new atom to a rule that has an entity for one of its arguments and the other argument is shared with another existing atom in the rule.

3. **Add Closing Atom**: This operators adds a new atom to a rule that shares both of its argument with other atoms that already exist in the rule.

Another recent work is [13]. Their methods involves discovering dense parts in the knowledge graph by traversing the lattice graph of all properties in the knowledge and finding views that meet their criteria. They introduce the notion of *Contextual Denial Constraints* (CDCs) for declaring denial constraints on RDF data which is a denial constraint that a view satisfies. A *Denial Constraint* (DC) on a structured dataset D is defined as follows:

$$\forall t_\alpha, t_\beta \in D : \neg(P_1 \wedge ... \wedge P_m) \tag{2.12}$$

Where:

$$P_i : v_1 \psi v_2 \; or \; v_1 \psi c, \; v_1, v_2 \in t_x.A, x \in \{\alpha, \beta\}, A \in R(D), \psi \in \{=, \neq, <, \leq, >, \geq\} \tag{2.13}$$

An example of a denial constraint is:

$$\neg(r_a.income = r_b.income \wedge r_a.loan < r_b.loan \wedge r_a.payment > r_b.payment) \tag{2.14}$$

Which states that there cannot exist two entities that have the same amount of income and one has a higher loan and the other has a higher payment. This definition of denial constraints only involves one or two tuples but there could be multiple tuples involved in a single constraint. Their method has 2 main steps (see figure 2.8):

1. **View Discovery**. They first produce a set of views to discover denial constraints on. They introduce two different approaches for finding views. One is schema-driven and the other is data-driven.

2. **Constraint Discovery**. They discover denial constraints on each of the generated views.

The schema-driven algorithm enumerates and searches the space of all possible views guided by a lattice structure. An example of a lattice graph is shown in figure 2.8. The schema-driven approach however, does not scale to datasets with many attributes and properties.

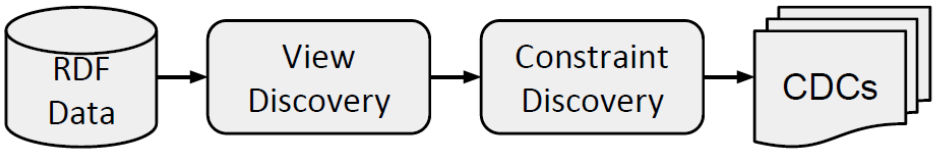


Figure 2.7: The CDC discovery pipeline [13]

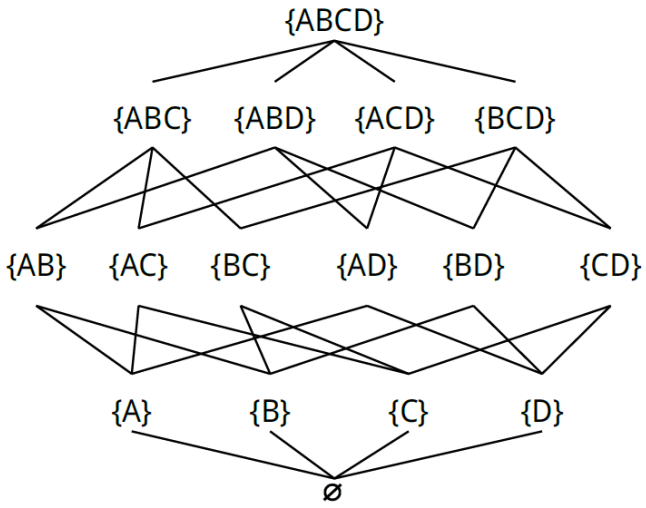


Figure 2.8: View space of 4 properties {`A, B, C, D`} [13]

Thus they present another algorithm for view discovery which is data-driven and is done by recursively intersecting views that are introduced by the entities in the RDF data themselves.

After finding views they use the denial constraint discovery algorithms that are already used for relational tables on each view. This is generally done by building an evidence set and finding the minimal set covers within the evidence set. These denial constraints can be used as a distinct signal and featurizer for HoloClean in order to establish the interaction between multiple rows and to get better imputed values for our case.

There are two main issues with this approach however:

1. The first issue is that they mainly work under the assumption that there are no non-functional predicates in the knowledge graph and if there are they take one of their values and ignore others.

2. The second problem is the explosion in the number of discovered DCs. Most mined DCs are too complex and big and do not have any clear meaning and are mostly the result of over-fitting.

Chapter 3

Finding the Right Contexts for Imputation

In this chapter we address the first and second challenges which were discussed in section 1.4.1 namely the overlapping classes and irrelevant attributes. This chapter is essentially about finding the right contexts to be fed into HoloClean in order to impute missing values of the target attribute for entities which the target attribute is an acceptable property.

In section 3.1 we offer three alternative approaches for handling the issue of overlapping classes and discuss their pros and cons. In section 3.2 we use the FD discovery tool FDX [35] to find the most important attributes that could be used as signals for imputing the target attribute. We observe that this approach fails due to the sparsity of the data. We further argue that the existence of irrelevant attributes does not distract the imputation model.

3.1 Overlapping Classes

The hierarchical structure of classes makes them have overlapping entities. Thus an entity might belong to multiple classes (see figure 3.1). Two questions arise here:

1. For a given entity, how do we pick the best class or classes that has enough redundancy and enough signal to train our imputation model on it in order to predict the missing value of the target attribute for that entity?

2. In case we feed an entity along multiple classes to our imputation model, how do we handle the potentially different imputed values of the target attribute for that entity across different classes?

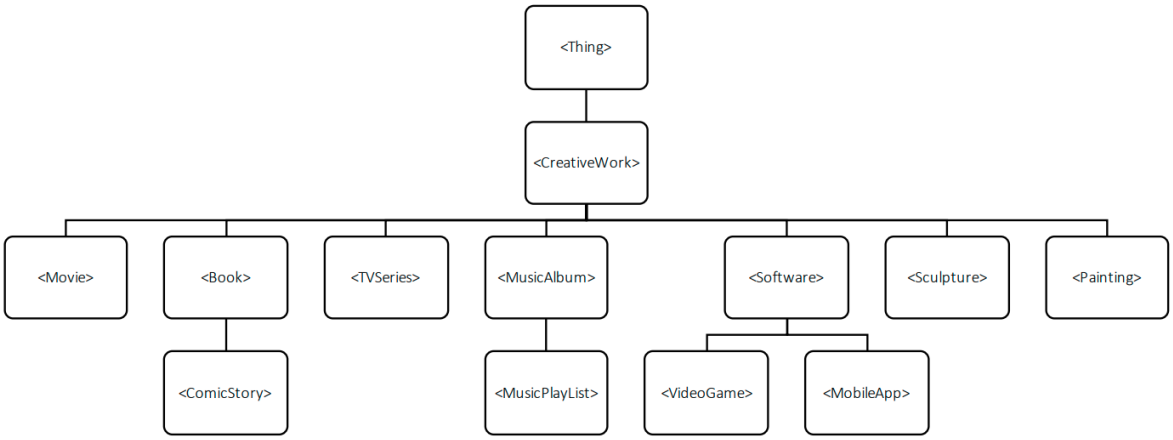


Figure 3.1: In this figure we have the hierarchical structure of some of the classes in YAGO4. An entity belonging to a class at the bottom of the hierarchy like `<VideoGame>` also belongs to classes `<Software>`, `<CreativeWork>` and `<Thing>`.

To address these questions we offer three alternative approaches which we call baselines 0, 1, 2. These baselines can be treated as a complete end-to-end solution given how we handle non-functional predicates in our implementation which we will discuss in chapter 5. However they do not address the issues of irrelevant attributes and sparsity. We present complete results of these approaches in section 5.2.

3.1.1 Baseline Zero

In this approach we merge all classes c that have the target attribute T in their schema to form a master class M with entities and a schema which is the union of entities and the schema of all classes:

$$R(M) = \bigcup_{c \in C : T \in R(c)} R(c) \tag{3.1}$$

$$Subjects(M) = \bigcup_{c \in C : T \in R(c)} Subjects(c) \tag{3.2}$$

The main problem with this approach is that although we are only feeding the classes that have the target attribute in their schema, it will still create an extremely heterogeneous data. The attention model will probably ignore the attributes of minor classes and put more weight on the attributes of the dominant classes. As a result, we will not be able to do imputation on entities belonging to minor classes. We present the pseudo code of baseline 0 in algorithm 1.

Algorithm 1: Baseline 0

```
Input: Knowledge Graph KG, Classes C, Target attribute T
superSchema = {}
for c in C do
    if T in c.schema then
        superSchema = superSchema ∪ c.schema
    end
end
objects = []
for e in KG.entities do
    if e belongs to classes in C then
        objects.append(project(e, superSchema))
    end
end
imputedTriples = imputeWithHoloClean(objects, T)
return imputedTriples
```

3.1.2 Baseline One

In this approach in order to address the heterogeneity issue of baseline 0, we feed all classes c that have the target attribute in their schema to HoloClean separately. However, like mentioned before one entity might belong to multiple classes and have multiple imputed values for its target attribute.

To handle this issue, we pay attention to the fact that HoloClean gives us the confidence probability of the imputed value in each class. For each value we calculate a score which is the sum of its probabilities from all classes. We choose the value with the highest score as the final imputed value for a given entity. We present the pseudo code of baseline 1

in algorithm 2 and the pseudo code for fusing imputed values from different classes in algorithm 3.

Algorithm 2: Baseline 1

Input: Knowledge Graph *KG*, Classes *C*, Target attribute *T*
allImputedTriples = []
for *c in C* **do**
  if *T in c.schema* **then**
    objects = *c.entities*
    imputedTriples = *imputeWithHoloClean(objects, T)*
    allImputedTriples.extend(imputedTriples)
  end
end
fusedTriples = *fuseTriples(allImputedTriples, T)* (See Alg. 3)
return *fusedTriples*

Algorithm 3: Fuse triples

Input: All imputed triples *imputedTriples*, Target attribute *T*
valuesDict = *Dict()*
for *triple in imputedTriples* **do**
  valuesDict[triple.s][triple.o] += *triple.prob*
end
finalTriples = *[]*
for *entity in valuesDict.items()* **do**
  finalValue = *max(entity[1])[0]*
  finalTriples.append(Triple(entity[0]), T, finalValue))
end
return *finalTriples*

One problem with this approach is that we are ignoring the fact that for a given entity some classes might have more and better signals for imputing its target attribute. Classes that are in the bottom of the hierarchy have a wider schema and for entities belonging to these classes it might be better to get the imputed value only from these classes and ignore higher classes. For example given figure 3.1 as our class hierarchy and having `<genre>` as target attribute, lets say we have an entity `<Justice,_My_Foot!>` belonging to the class `<Movie>` which also belongs to classes `<CreativeWork>` and `<Thing>` that we want to impute its `<genre>`. If we feed this entity along `<Movie>`, we would have attributes like `<director>`, `<producer>` and `<actor>` in the schema of the class that could help us in predicting the entity's `<genre>`. But if we feed it along `<CreativeWork>` or `<Thing>` we

would not have these signals and our prediction will be less accurate.

3.1.3 Baseline Two

In this approach, in order to find the best class for a given entity for imputing its target attribute, we start from low-level classes and move upward. Like we mentioned before, classes at the bottom of the hierarchy have a wider schema and thus more potential signals for imputing the target attribute. Whenever we encounter a class with *enough context* we feed it to HoloClean and omit its sub classes that we already did an imputation on. Here enough context means if its instances are more than a given threshold. For example looking at figure 3.1 again, starting from the bottom of the hierarchy we have `<VideoGame>` with 24175 instances and `<MobileApp>` with 191 instances as sub classes of `<Software>`. Lets say our threshold for enough context is 1000. Thus `<VideoGame>` will be selected and fed to HoloClean but `<MobileApp>` will not be selected. Then upon feeding the class `<Software>` to HoloClean we omit instances that belong to `<VideoGame>` because we have already did imputation on them. But we keep instances that belong to `<MobileApp>`.

We provide the pseudo code for baseline 2 in algorithm 4.

Algorithm 4: Baseline 2

```
Input: Knowledge Graph KG, Classes C, Target attribute T, Class size threshold
  thresh
imputedClasses = []
C = sortByDepth(C, direction=-1)
allImputedTriples = []
for c in C do
    if T in c.schema then
        objects = c.entities - getEntites(imputedClasses)
        if len(objects) ≥ thresh then
            imputedTriples = imputeWithHoloClean(objects, T)
            allImputedTriples.extend(imputedTriples)
            imputedClasses.append(c)
        end
    end
end
return allImputedTriples
```

This approach as we will demonstrate in the result section yields better results than

two previous approaches as shown in section 5.2. In our next and final solutions, we will be using baseline 2 to select the classes.

3.2 Irrelevant Attributes

As mentioned in section 1.4.1, after we have selected the right classes as we described in section 3.1, we might have attributes in the schema of selected classes that are irrelevant to the target attribute. These irrelevant attributes might distract the HoloClean's classifier and the attention model of AimNet. They also inflict more computation cost to the system.

To address this issue, we use the FD discovery tool FDX (section 2.3) that we introduced in chapter 2. We call this approach baseline 3 which we will describe in section 3.2.1. This is also another end-to-end solution and we have presented its complete results in section 5.2.

3.2.1 Baseline Three

Given a class c which has been selected by baseline 2, we first feed it to FDX to get attributes $X \subset R(c)$ which the target attribute T is conditionally dependent on. We call attributes X *context attributes*. We then project all entities of the class to the attributes X and feed the results to HoloClean. The pseudo code of baseline 3 is provided in algorithm

5.

Algorithm 5: Baseline 3

```
Input: Knowledge Graph KG, Classes C, Target attribute T, Class size threshold
  thresh
imputedClasses = []
C = sortByDepth(C, direction=-1)
allImputedTriples = []
for c in C do
    if T in c.schema then
        objects = c.entities - getEntites(imputedClasses)
        if len(objects) ≥ thresh then
            contextAttributes = FDX(objects)
            objects = projectAll(objects, contextAttributes)
            imputedTriples = imputeWithHoloClean(objects, T)
            allImputedTriples.extend(imputedTriples)
            imputedClasses.append(c)
        end
    end
end
return allImputedTriples
```

There are two points that are worth mentioning:

1. The selected classes c might be very sparse and this leaves us with a choice when we are feeding them to FDX. We could assume the comparisons that involve `Null` values as true and follow the *open-world assumption* which states that the truth value of a statement may be true irrespective of whether or not it is known to be true [1]. However, given the fact that classes might be extremely sparse, this could result in generating context attributes that are actually irrelevant to the target attribute. For example having `<nationality>` as target attribute in the class `<Person>`, FDX might select `<has_child>` as a context attribute since for most entities in the class `<Person>`, `<has_child>` is `Null`. The other choice that we have is treat comparisons with a `Null` as false and adopt the *closed-world assumption*. This on the other hand, results in finding very few or no context attributes at all given the level of sparsity of the data. The solution that we devised for this problem was to sort entities based on the number of their known attributes. Then we only feed a small proportion of the data (say 1%) at the top which forms a relatively dense data to FDX with the closed-world assumption to get the context attributes X.

2. Since the transformation phase of FDX works around equality between attributes values of two different entities, we needed to find a way to define equality between 2 lists of values. We take 2 lists as equal if they have a common value. For this we had to completely change the code of FDX as they were working on relational data and used data frames. We added a new type of input data which was a list of dictionaries similar to the documents of MongoDB that we have added to the code of HoloClean which we will explain in chapter 5. To define equality between 2 lists we could also use similarity measures between sets like Overlap coefficient:

$$\frac{|A \cap B|}{min(|A|, |B|)} \tag{3.3}$$

and Jaccard index:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.4}$$

As we will show in the results section, in terms of precision baseline 3 beats all other approaches in nearly all classes and target attributes. But in terms of recall and F1-score it is much worse that others (see results section 5.2). We observed two main issues with baseline 3:

1. Since we have not dealt with sparsity, projecting entities to context attributes might still result in a highly sparse data. In the absence one or more context attributes, other non-context attributes might become important and useful for predicting target attribute.

   Lets say we have a class c with a schema of attributes `<birthPlace>`, `<nationality>`, `<knowsLanguage>` and we have `<knowsLanguage>` as target attribute. Feeding this class to FDX will result in the probabilistic graphical model depicted in figure 3.3a. In another word, `<knowsLanguage>` is independent from `<birthPlace>` given we know the value of `<nationality>` since `<nationality>` is all we need in order to predict `<knowsLanguage>` and according to baseline 3 we will project all entities to just `<nationality>` and discard `<birthPlace>`. But in our sparse data the value of `<nationality>` might be `Null` in many entities. Then in the absence of `<nationality>`, `<knowsLanguage>` becomes conditionally dependent on `<birthPlace>` (see figure 3.3b) and we may use its values for predicting `<knowsLanguage>` but as we just mentioned in baseline 3 we will discard `<birthPlace>`.

2. The existence of irrelevant attributes does not actually cause a problem in our system. We observed that the attention model of AimNet puts more weight on attributes that

are relevant to the target attribute and less weight on irrelevant attribute as shown in figure 3.2. Also by employing a higher confidence probability threshold in baseline 2 we were able to get the same precisions as baseline 3 with higher recalls in most cases.

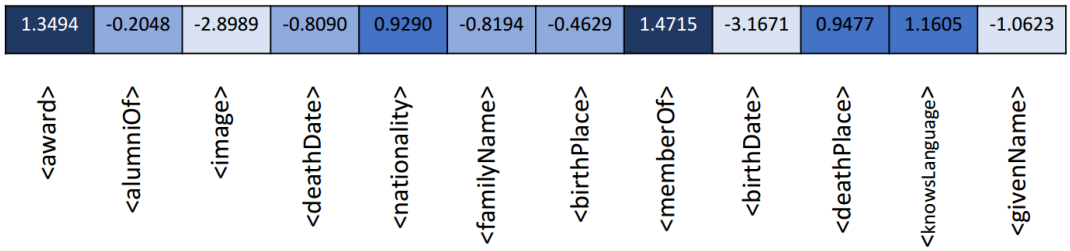
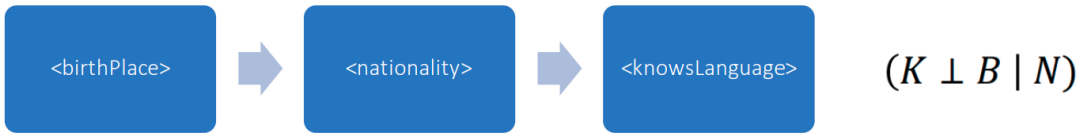
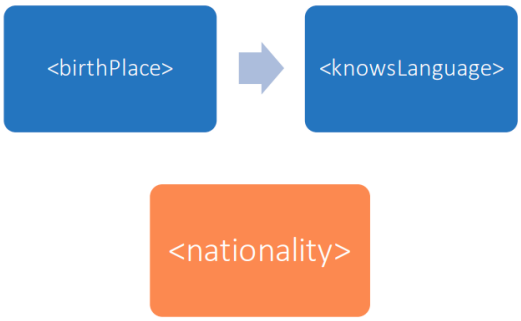


Figure 3.2: Attention weights on attributes of the class `<Person>` with `<hasOccupation>` as target attribute.



(a)



(b)

Figure 3.3: In (a) we have the probabilistic graphical model with the presence of `<nationality>` where `<knowsLanguage>` is conditionally independent from `<birthPlace>`. In (b) we the have the probabilistic graphical model with the absence of `<nationality>` where `<knowsLanguage>` is conditionally dependent on `<birthPlace>`.

Chapter 4

Handling Sparsity

In this chapter we will address the third challenge that was discussed in section 1.4.1 which is the sparsity. Our approach for handling this problem is inspired by multi-head attention mechanism introduced by [30]. The process in which we transform the knowledge graph into a relational table will often result in extremely sparse data since most entities have very few properties of the schema of the classes that they belong to resulting in lots of `NULL` values in our table. We pointed this out in the example in figure 1.1. These `NULL` values will be treated as another distinct label by the imputation model AimNet and thus will act as noisy cells to our model.

These noisy cells will lower the accuracy of our imputation. The main reason for this is that the model is trained on all attributes of the schema and it might allocate higher weights to some attributes that have strong correlation with the target attribute. Now during imputing phase, one entity might miss some of these high correlated attributes and that might result in imputing a wrong value. Even if we have enough signals in other known attributes to impute the correct value and the imputation model somehow manages to get the correct value, it is highly probable that we get a very low confidence probability for it. If the confidence probability fall below the given threshold this correctly imputed value will be ignored resulting in low recall for our model.

We will address this issue using multiple model heads each trained without the presence of a specific attribute. We argue that this approach is about handling the sparsity in general and not specific to knowledge graphs and our case and can be used on any sparse dataset. We will design our final solution based on selecting the classes according to baseline 2 and handling the sparsity issue which we will discuss in this chapter.

4.1 Our Method

In general, We need to train models that are *immune* to missing values of a certain attribute. More specifically, for each context attribute that the target attribute is conditionally dependent on (which is given to us by FDX), we train a model without the presence of that attribute and containing all other attributes which we call a *model head* (see figure 4.1). This way those entities that have a `NULL` value for that attribute, do not have any disadvantage with respect to entities that have a known value for it. Thus the confidence probability that we get for them are not penalized.

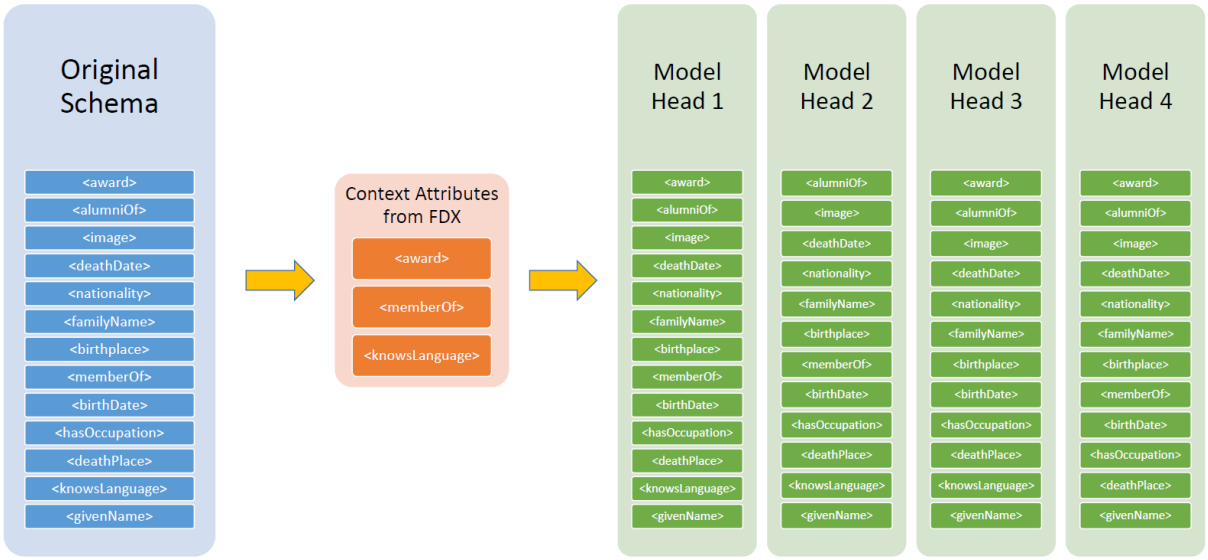


Figure 4.1: In this figure we have the original schema of the class `<Person>` on the left and we have `<hasOccupation>` as our target attribute. First we feed the class to FDX to get the context attributes which are `<award>`, `<memberOf>` and `<knowsLanguage>`. Then we create our first model head using all the attributes of the original schema. This model head is used for entities that have all the context attributes. Then we create one model head based on the absence of each context attribute. Thus the model head 2 is trained without `<award>`, model head 3 without `<memberOf>` and model head 4 without `<knowsLanguage>`.

We create our model heads based on the absence of only the context attributes given to us by FDX. The reason for this is to avoid having too many model heads. Thus we create them based on the absence of most important attributes.

```
{
    "subject" : "<http://yago-knowledge.org/resource/Bunchy_Carter>",
    "<http://schema-org/alumniOf>" : [
        "<http://yago-knowledge.org/resource/U_of_C,_Los_Angeles>"
    ],
    "<http://schema-org/nationality>" : [
        "<http://yago-knowledge.org/resource/United_States>"
    ],
    "<http://schema-org/birthPlace>" : [
        "<http://yago-knowledge.org/resource/Los_Angeles>"
    ],
    "<http://schema-org/memberOf>" : [
        "<http://yago-knowledge.org/resource/Black_Panther_Party>"
    ],
    "<http://schema-org/deathPlace>" : [
        "<http://yago-knowledge.org/resource/Los_Angeles>"
    ]
}
```

Listing 1: Following the example in figure 4.1, we have an entity here that does not have `<award>` and `<knowsLanguage>`. So we pick model heads 2 and 4 from figure 4.1 and get their imputed values for this entity and then fuse them to get the final value for the attribute `<hasOccupation>` of this entity.

One question that arises here is what we will do with entities that have `NULL` values for 2 or more context attributes. To address this we could go one step deeper and create model heads based on the absence of 2 or more context attributes. This however results in an exponential number of models with respect to the number of context attributes. To alleviate this, given an entity, in order to impute its target attribute we pick model heads based on the context attributes that the entity is missing and get the imputed value for that entity from those models (see listing 1). Then we fuse the imputed value of these models exactly like how we did it in baseline 1.

The pseudo code of our final method which we call *final solution* along with how we

select model heads for each entity is given in algorithm 6 and algorithm 7 respectively.

Algorithm 6: Final Solution

```
Input: Knowledge Graph KG, Classes C, Target attribute T, Class size threshold
  thresh
imputedClasses = []
C = sortByDepth(C, direction=-1)
allPrunedTriples = []
for c in C do
    classImputedTriples = []
    if T in c.schema then
        objects = c.entities - getEntites(imputedClasses)
        if len(objects) ≥ thresh then
            contextAttributes = FDX(objects)
            imputedTriples = imputeWithHoloClean(objects, T, modelIndex=0)
            classImputedTriples.extend(imputedTriples)
            index=1
            modelsDict = Dict()
            for attr in contextAttributes do
                schema = c.schema - attr
                objs = projectAll(objects, schema)
                imputedTriples = imputeWithHoloClean(objs, T, modelIndex=index)
                classImputedTriples.extend(imputedTriples)
                modelsDict[attr] = index
                index += 1
            end
            prunedTriples = pickModels(objects, contextAttributes,
              classImputedTriples, T, modelsDict) (See Alg. 7)
            allPrunedTriples.extend(prunedTriples)
            imputedClasses.append(c)
        end
    end
end
return allPrunedTriples
```

Algorithm 7: Pick models

Input: Entity objects *objects*, Context attributes *contextAttributes*, Class imputed triples *imputedTriples*, Target attribute T, Model index dictionary *modelsDict*

```
valuesDict = Dict()
for triple in imputedTriples do
    currObject = objects[triple.s]
    missingContextAttrs = getMissingContextAttributes(currObject,
     contextAttributes)
    if missingContextAttrs is Null then
        targetModelsIndices = [0]
    else
        targetModelsIndices = getModelIndices(missingContextAttrs, modelsDict)
    end
    if triple.modelIndex in targetModelsIndices then
        valuesDict[triple.s][triple.o] += triple.prob
    end
end
finalTriples = []
for entity in valuesDict.items() do
    finalValue = max(entity[1])[0]
    finalTriples.append(Triple(entity[0]), T, finalValue))
end
return finalTriples
```

Using model heads, we observed a substantial increase in the confidence probabilities of the imputed values for entities that were missing some of the context attributes in model heads that were trained without those context attributes. This resulted in having higher recalls with respect to other approaches as we will show in the results. We provide an example in listing 2.

```
{
   "subject" : "<http://yago-knowledge.org/resource/Friedrich_Anders>",
    "<http://schema-org/award>" : [
        "<http://yago-knowledge.org/resource/Knight's_Iron_Cross>"
    ],
    "<http://schema-org/deathDate>" : [
        "\"1988-03-07\"^^<http://www.w3.org/2001/XMLSchema#date>"
    ],
    "<http://schema-org/givenName>" : [
        "<http://yago-knowledge.org/resource/Friedrich_(given_name)>"
    ],
}
```

```
[
  {
    "value": "<http://yago-knowledge.org/resource/Germany>",
    "prob": 0.223708927631378,
    "view_index": 0
  },
  {
    "value": "<http://yago-knowledge.org/resource/Germany>",
    "prob": 0.638112366199493,
    "view_index": 3
  }
]
```

Listing 2: Here we have `<nationality>` as target attribute in class `<Person>`. At the top we have an entity that doesn't have `<birthPlace>` which is a context attribute of `<nationality>` and also a strong signal for predicting `<nationality>`. At the bottom we have the imputed values for this entity from the model head with all attributes and the model head that is trained without `<birthPlace>` with their confidence probabilities.

Chapter 5

Implementation and Results

In this chapter we will go through some of the details in implementation. More specifically we will explain how we deal with the non-functional predicates. We will then present the results of all of our approaches on 13 selected classes and 18 target attributes within those classes.

5.1 Dealing with Non-Functional Predicates

The open source HoloClean is implemented with the assumption that our input data is a relational table and uses Postgres to store and query the original data and other metadata that it creates such as cell domains. Thus for each row (entity) and each property (column) we can have at most one value. This becomes challenging when we are dealing with graph data and predicates that could point to multiple values from a single entity. An example of this is the predicate `<has_child>` as for one entity `<a>` we can have multiple triples like `(<a>, <has_child>, <x>)`.

To handle this problem, we first offer two preliminary and simple solutions that are consistent with the assumption of input data being a relational table and are fairly easy to implement given the original implementation of the open source HoloClean:

1. **Discarding multiple values of non-functional predicates**. This is an irrational but fairly easy to implement approach. Given an entity that has a non-functional property, we just take one value from that property and discard others and treat what remains as a single row in the relational table (see figure 5.1). This requires

almost no change in the open-source HoloClean code. It is obvious however that with this approach we are losing part of the data. if the target attribute is non-functional property, we are losing potential labels and if the non-functional property is not the target attributes we are losing signals that might be helpful in predicting the target attribute. This approach is used in finding Denial Constraints from extracted views in [13].

2. **Taking the Cartesian product of non-functional predicates**. In this approach we are basically creating all possible combinations of non-functional properties and treat each of them as a separate row in our relational table. Like the previous approach, this one doesn't require any major change in the code and is fairly easy to implement. However the problem with this approach is that it changes the co-occurrence statistics between attribute values which might negatively impact the domain generation and training of the model. In figure 5.2 we have 4 distinct rows which are the Cartesian product of non-functional predicates of the entity. The co-occurrences of attribute values `<Ukraine>` and `<Communist_Party>` are 4 but their actual co-occurrence in the original entity is one.

subject	predicate	object
<Valentyn_Symonenko>	<rdf:type>	<Person>
<Valentyn_Symonenko>	<award>	<Hero_of_Ukraine>
<Valentyn_Symonenko>	<award>	<Order_of_Friendship>
<Valentyn_Symonenko>	<nationality>	<Ukraine>
<Valentyn_Symonenko>	<hasOccupation>	<Economist>
<Valentyn_Symonenko>	<hasOccupation>	<Politician>
<Valentyn_Symonenko>	<memberOf>	<Communist_Party>

id	<nationality>	<hasOccupation>	<memberOf>	<award>
1	<Ukraine>	<Economist>	<Communist_Party>	<Hero_of_Ukraine>

Figure 5.1: Here we have an entity belonging to the class `<Person>` with `<hasOccupation>` and `<award>` being non-functional properties for this entity. According to the first preliminary approach in handling non-functional predicate we only take one value from the values of non-functional predicates and omit others. What remains forms a row in our relational table.

Given the issues of two previous approaches, we come up with another solution. In order to keep all of the data and not to distort co-occurrence statistics, we use MongoDB

id	<nationality>	<hasOccupation>	<memberOf>	<award>
1	<Ukraine>	<Economist>	<Communist_Party>	<Hero_of_Ukraine>
2	<Ukraine>	<Economist>	<Communist_Party>	<Order_of_Friendship>
3	<Ukraine>	<Politician>	<Communist_Party>	<Hero_of_Ukraine>
4	<Ukraine>	<Politician>	<Communist_Party>	<Order_of_Friendship>

Figure 5.2: Having the entity and triples in figure 5.1 we create 4 distinct rows which are the Cartesian product of the non-functional predicates of the source entity.

to store the data and treat all properties as lists of values stored in a single document which is equivalent to a single row in a relational table (see listing 3). This however forced us to change almost all of the modules of the open-source HoloClean as they were implemented assuming that we are dealing with a relational table and they had to be re-implemented given the new data storage system. In the following sub-sections we will explain how we changed three modules in the open-source HoloClean in order to cope with non-functional properties and the new storage system.

5.1.1 Co-occurrence Statistics

We will count co-occurrence statistics of attribute values as they are in a single document. We treat one attribute value with respect to another non-functional predicate as if it has appeared alongside all values of that non-functional predicate. We use these co-occurrence statistics both in domain generation and in the co-occurrence featurizer. The pseudo code

```
{
    "subject" :
    "<http://yago-knowledge.org/resource/Valentyn_Symonenko>",
    "<http://schema-org/award>" : [
        "<http://yago-knowledge.org/resource/Hero_of_Ukraine>",
        "<http://yago-knowledge.org/resource/Order_of_Friendship>",
    ],
    "<http://schema-org/nationality>" : [
        "<http://yago-knowledge.org/resource/Ukraine>"
    ],
    "<http://schema-org/hasOccupation>" : [
        "<http://yago-knowledge.org/resource/Economist>",
        "<http://yago-knowledge.org/resource/Politician>"
    ],
    "<http://schema-org/memberOf>" : [
        "<http://yago-knowledge.org/resource/Communist_Party>"
    ]
}
```

Listing 3: Storing the entity in figure 5.1 in a single MongoDB document.

of our method is shown in algorithm 8.

Algorithm 8: Calculate Co-Occurrences

```
Input: Dataset MongoDB Collection objects, Dataset Attributes attributes,
 Target Attributes T
pairStats = Dict()
for attr in attributes do
    if attr == T then
        continue
    end
    for obj in objects do
        for val_1 in obj[attr] do
            for val_2 in obj[T] do
                pairStats[attr][T][val_1][val_2] += 1
            end
        end
    end
end
return pairStats
```

5.1.2 Domain Pruning and Naive Bayes

To get the posterior probabilities of each value in the domain of cells for domain pruning, we consider all the values of each attribute. Thus we rewrite equation 2.1 as:

$$Pr(A_i[t] = v|A_1[t], ..., A_N[t]) \propto Pr(A_i[t] = v) \prod_{j \neq i} \prod_{n=0}^{len(A_j[t])} Pr(A_j[t][n]|A_i[t] = v) \quad (5.1)$$

where the parameter n is the index of the values in each attribute.

5.1.3 Creating Data Points for AimNet

In order to feed each entity to AimNet we have to consider the fact that we cannot have variable number of input attributes in AimNet. Thus we need to create data points that have fixed number of context attributes and target attributes. Given an entity with potential non-functional attributes we create a linear combination of all context attributes in

a way that each attribute value comes in at least one data point because we do not want to lose any value during training. Then we clone the set of data points that we have for each value in the target attribute. We provide an example in figure 5.3. The pseudo code of our method is presented in algorithm 9.

subject	predicate	object
<Mahatma_Gandhi>	<rdf:type>	<Person>
<Mahatma_Gandhi>	<award>	<O._R._Tambo>
<Mahatma_Gandhi>	<award>	<Time_Person_of_the_Year>
<Mahatma_Gandhi>	<nationality>	<India>
<Mahatma_Gandhi>	<hasOccupation>	<Philosopher>
<Mahatma_Gandhi>	<hasOccupation>	<Politician>
<Mahatma_Gandhi>	<hasOccupation>	<Jurist>
<Mahatma_Gandhi>	<memberOf>	<National_Congress>
<Mahatma_Gandhi>	<memberOf>	<Inner_Temple>

id	<nationality>	<hasOccupation>	<award>	<memberOf>
1	<India>	<Philosopher>	<O._R._Tambo>	<National_Congress>
2	<India>	<Politician>	<Time_Person>	<National_Congress>
3	<India>	<Jurist>	<O._R._Tambo>	<National_Congress>
4	<India>	<Philosopher>	<O._R._Tambo>	<Inner_Temple>
5	<India>	<Politician>	<Time_Person>	<Inner_Temple>
6	<India>	<Jurist>	<O._R._Tambo>	<Inner_Temple>

Figure 5.3: Here we have <memberOf> as the target attribute and other properties are context attributes. first we create a linear combination of context attributes where each attribute value is seen at least one (three data points). Then we clone those data points for each value of the target attribute which gives us a total of 6 data points.

During inference, for imputing an unknown target attribute, we create data points from the context attributes in a similar way. Then we take the average of probability distributions that AimNet generates for each of those data points and select the final

imputed target value based on that.

Algorithm 9: Creating Data Points for AimNet

```
Input: MongoDB Document object, Dataset Attributes attributes, Target
 Attributes T
generatedRows = []
contextRows = []
maxLength = getMaxAttributeLength(object, attributes - T)
for i in range(0, maxLength) do
    row = {}
    for attr in attributes - T do
        row[attr] = objects[attr][i % len(objects[attr])]
    end
    contextRows.append(row)
end
for val in object[T] do
    for contextRow in contextRows do
        tempRow = contextRow.copy()
        tempRow[T] = val
        generatedRows.append(tempRow)
    end
end
return generatedRows
```

5.2 Results

We present the results of baselines 0, 1, 2 and 3 along our final solution on the YAGO4-EN dataset. This dataset is a subset of the YAGO4 that only contains entities that have a Wikipedia article with around 5M entities and 20M facts.

5.2.1 Experiment Setup

We use an en embedding size of 128 for AimNet with the batch size of 500 and a total of 5 epochs. In order to validate the results we mask 1% of the target attribute labels before feeding the data to training phase and then comparing the imputed values for these labels with their actual values in order to get the reconstruction precision of our imputation.

Like we mentioned this is just reconstruction precision and we could in fact try to verify the imputed values for all entities missing the target attribute. But this was a costly and tedious work and for some target attributes and classes this couldn't be done easily. For example for the target attribute `<memberOf>` in the class `<Person>`, most imputed values couldn't be verified from the Wikipedia article of the entities unlike other target attributes like `<nationality>` and `<hasOccupation>` that could be verified easily. Recalls are calculated with respect to the number of entities that have missing values for the target attribute. We set confidence probability threshold to be 0.1 but with a threshold we could technically achieve a 1.00 recall by sacrificing precision.

We did imputation for 18 different target attributes on 162 classes overall. The results are shown for baselines 0, 1, 2, 3 and the final solution in figure 5.4. for some selected classes. Some classes like `<CreativeWork>` contain multiple sub-classes like `<Movie>` and `<MusicComposition>` and we have provided the overall precision and recall for them.

We were able to generate 2.4 million additional triples with an average reconstruction precision of 0.81 in baseline 3 — enlarging the entire knowledge graph by roughly 12%. In our final solution we generated 3.4 million triples (enlarging the knowledge graph by 17%) with the average reconstruction precision of 0.71. In general the final solution yields less precision compared to baseline 3 but its F1-score is actually higher in most target attributes and classes due to higher coverage and recall with respect to the entities with missing target attribute.

For each target attribute and its corresponding class, the methods that has the higher F1-score has been bolded.

			Baseline_0			BaseLine_1			Baseline_2			Baseline_3			Final_solution		
Class	Property	# Missing	# Imputed	Precision	Recall	# Imputed	Precision	Recall	# Imputed	Precision	Recall	# Imputed	Precision	Recall	# Imputed	Precision	Recall
Person	nationality	369847	309909	0.85	0.83	309909	0.85	0.83	309909	0.85	0.83	213703	0.90	0.58	**340835**	**0.84**	**0.92**
	hasOccupation	706905	285695	0.71	0.40	285695	0.71	0.40	285695	0.71	0.40	122725	0.93	0.17	**464901**	**0.60**	**0.65**
	memberOf	1141518	693820	0.34	0.60	693820	0.34	0.60	693820	0.34	0.60	397841	0.44	0.35	**744893**	**0.35**	**0.65**
	knowsLanguage	1363901	1312681	0.90	0.96	1312681	0.90	0.96	1312681	0.90	0.96	1302976	0.94	0.96	**1353813**	**0.94**	**0.99**
Movie	productionCompany	118722	58231	0.58	0.49	71351	0.65	0.60	71100	0.66	0.60	66917	0.51	0.56	**73625**	**0.67**	**0.62**
	countryOfOrigin	5508	4034	0.89	0.73	4306	0.88	0.78	4220	0.90	0.76	4369	0.72	0.79	**4843**	**0.90**	**0.88**
	actor	40479	ME	ME	ME	7168	0.15	0.17	6960	0.17	0.17	4074	0.21	0.10	**7043**	**0.18**	**0.17**
Music Composition	composer	21454	4029	0.72	0.18	4110	0.51	0.19	3977	0.58	0.18	4185	0.48	0.19	**4367**	**0.65**	**0.20**
	lyricist	26277	**7355**	**0.74**	**0.27**	7155	0.64	0.27	7393	0.69	0.28	5878	0.64	0.22	9056	0.57	0.34
MusicAlbum	byArtist	72834	31762	0.20	0.43	30758	0.17	0.42	32341	0.23	0.44	16061	0.48	0.22	**38592**	**0.20**	**0.53**
Event	location	79499	995	0.72	0.01	18590	0.31	0.23	31929	0.39	0.40	**31700**	**0.65**	**0.39**	30536	0.48	0.38
	superEvent	70632	22549	0.54	0.31	23829	0.51	0.33	18124	0.63	0.25	17065	0.69	0.24	**18363**	**0.68**	**0.26**
CreativeWork	genre	355308	106754	0.41	0.30	162977	0.41	0.45	142866	0.46	0.40	73911	0.51	0.20	**124611**	**0.52**	**0.35**
	producer	490280	98467	0.40	0.20	**186961**	**0.38**	**0.38**	144633	0.46	0.30	86061	0.43	0.17	119758	0.51	0.25
SportsEvent	sport	3932	1577	0.96	0.40	1546	0.96	0.39	1546	0.97	0.39	1521	0.97	0.38	**1552**	**0.98**	**0.39**
CreativeWorkSeason	partOfSeries	937	110	0.0	0.11	0	0.0	0.0	0	0.0	0.0	**87**	**0.66**	**0.09**	0	0.0	0.0
Episode	partOfSeason	3191	1174	0.34	0.36	1228	0.38	0.38	1047	0.49	0.32	301	0.17	0.09	**1006**	**0.51**	**0.31**
	partOfSeries	131	**55**	**0.92**	**0.41**	36	0.92	0.27	27	0.91	0.20	16	0.93	0.12	27	0.95	0.20
TVEpisode	countryOfOrigin	7984	**7422**	**0.96**	**0.92**	5262	0.96	0.65	5143	0.97	0.64	3251	0.96	0.40	5174	0.97	0.64
TVSeason	countryOfOrigin	4541	1022	0.85	0.22	2990	0.87	0.65	3277	0.71	0.72	**2783**	**0.95**	**0.61**	2989	0.72	0.65
Taxon	taxonRank	4224	3180	0.96	0.75	3180	0.96	0.75	3181	0.96	0.75	2769	0.98	0.65	**3181**	**0.97**	**0.75**
SportsOrganization	sport	4185	3253	0.68	0.77	3264	0.65	0.78	3199	0.70	0.76	3009	0.70	0.71	**3187**	**0.73**	**0.76**

Figure 5.4: The number of imputed triples along with their reconstruction precision and recall with respect to entities with missing target attribute.

Chapter 6

Future Work and Conclusion

In this chapter we first provide some directions for future work and then conclude our work.

6.1 Future Work

In this section we discuss some of the short comings of our approach and try to explain how they could be addressed. We first try to explain how to incorporate graph embedding methods and deductive methods as distinct featurizers of HoloClean and try to look at the problem of knowledge graph imputation in a holistic way where we use the state of the art of all approaches that are orthogonal to each other.

6.1.1 Using Graph Embeddings

One short coming of our approach is that we are only using predicates that are directly attached to an entity in order to impute its target attribute. This method is quite lossy and does not make use of the graphical structure of the data. Graph embedding methods take the graph as whole and gives us an embedding for each entity and each relation. We can thus use these embeddings as another featurizer in HoloClean and attach them to the final feature vector of each value in the domain cells. We can also use the scoring function directly and use the value of scores in our featurizer.

In most embedding methods the unknown entity of a query triple `(<a>, <b>, <x>)` is replaced with all entities in the knowledge graph and the entity that yields the highest

value is chosen. This essentially binds us to a quadratic cost. In HoloClean however, we have a strong method for creating a limited domain of potential values and we need to only check the scoring function for the values in the domain.

We actually tried to use the scoring function of RefE [8] in the YAGO3-10 dataset along with our other featurizers. But it resulted in lower imputation accuracy. This however requires more experiment and deeper studies.

6.1.2 Using Deductive Rules as Denial Constraints

We can use Horn rules from AMIE and denial constraints from [13] as another featurizer similar to what the original open-source HoloClean has for relational tables. Horn rules can be easily transformed into denial constraints. The only issue that remains is the definition of equality between two lists of values as is the case for entities in knowledge graphs. We can use methods similar to what we did for FDX which we explained in section 3.2.1 to address this. By using denial constraints we can employ the interaction between different entities in order to impute their missing values.

6.1.3 Modifying Imputation Model

The imputation model that we used — AimNet uses an attention mechanism that the attention between different values are fixed during inference regardless of their values in a data point. In the self-attention mechanism [30] however, the amount of attention that they put on each attribute (word) depends on their values. This aspect of self-attention is attractive for us given the fact that we are dealing with sparse data and we basically want the attention to attributes that are missing in a given entity to be zero.

Another aspect of self-attention is the use of multi-head attention mechanism which is analogous to our model heads. But in self-attention, the process of fusing the outputs of each attention head are all done through learned parameters.

6.2 Conclusion

We have presented a new method for knowledge graph completion that revolves around training a model using immediate properties of entities and is based on viewing the knowledge graph as a relational data and using the powerful cleaning tool HoloClean. We have

designed and developed a fully operational system and framework for holistic knowledge graph completion that can take an entire knowledge graph without throwing away any data. This framework can be used to add further methods and signals for knowledge graph completion and solve this problem in a holistic manner. This work can be viewed as a beginning for holistic knowledge graph completion.

Our model can also be used for knowledge graph cleaning as we encountered lots of erroneous values and properties in YAGO4 during our work.

References

[1] Open-world assumption. https://en.wikipedia.org/wiki/Open-world_assumption.

[2] Farahnaz Akrami, Mohammed Samiul Saeef, Qingheng Zhang, Wei Hu, and Chengkai Li. Realistic re-evaluation of knowledge graph completion methods: An experimental study, 2020.

[3] Attia, Omar. Scaling machine learning data repair systems for sparse datasets, 2021.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. volume 6, pages 722–735, 01 2007.

[5] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. pages 1247–1250, 01 2008.

[6] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[7] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[8] Ines Chami, Adva Wolf, Da-Cheng Juan, Frederic Sala, Sujith Ravi, and Christopher Ré. Low-dimensional hyperbolic knowledge graph embeddings. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6901–6914, Online, July 2020. Association for Computational Linguistics.

[9] Spencer Chang. Scaling knowledge access and retrieval at airbnb. airbnb medium blog. https://medium.com/airbnb-engineering/scaling-knowledge-access-and-retrieval-at-airbnb-665b6ba21e95, 2018.

[10] Richard Cyganiak, David Hyland-Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. *W3C Proposed Recommendation*, 01 2014.

[11] X. Dong. Building a broad knowledge graph for products. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 25–25, Los Alamitos, CA, USA, apr 2019. IEEE Computer Society.

[12] Martin Dürst. Internationalized resource identifiers (iris) status of this memo. 01 2004.

[13] Farid, Mina. Extracting and cleaning rdf data, 2020.

[14] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: Association rule mining under incomplete evidence in ontological knowledge bases. pages 413–422, 05 2013.

[15] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter Patel-Schneider, and Sebastian Rudolph. Owl 2 web ontology language primer (second edition), 01 2012.

[16] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, Roberto Navigli, Axel-Cyrille Ngonga Ngomo, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs, 2021.

[17] Ni Lao and William Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine Learning*, 81:53–67, 10 2010.

[18] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *In Proceedings of AAAI'15*, 2015.

[19] F. Mahdisoltani, J. Biega, and Fabian M. Suchanek. A knowledge base from multilingual wikipedias-yago 3 une base de connaissances des wikipédias plurilingues – yago 3. 2014.

[20] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[21] George A Miller and Christiane Fellbaum. Wordnet then and now. *Language Resources and Evaluation*, 41(2):209–214, 2007.

[22] M. Nickel, Volker Tresp, and H. Kriegel. A three-way model for collective learning on multi-relational data. In *ICML*, 2011.

[23] R. J. Pittmanl. Cracking the code on conversational commerce. https://www.ebayinc.com/stories/news/cracking-the-code-on-conversational-commerce/.

[24] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *arXiv preprint arXiv:1702.00820*, 2017.

[25] Amit Singhal. Introducing the knowledge graph: things, not strings. google blog. https://www.blog.google/products/search/introducing-knowledge-graph-things-not/, 2012.

[26] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A large ontology from wikipedia and wordnet. *Journal of Web Semantics*, 6(3):203–217, 2008.

[27] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. Rotate: Knowledge graph embedding by relational rotation in complex space, 2019.

[28] Thomas Tanon, Gerhard Weikum, and Fabian Suchanek. *YAGO 4: A Reason-able Knowledge Base*, pages 583–596. 05 2020.

[29] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Eric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2071–2080, New York, New York, USA, 20–22 Jun 2016. PMLR.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[31] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledge base. *Communications of the ACM*, 57:78–85, 2014.

[32] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[33] Richard Wu, Aoqian Zhang, Ihab Ilyas, and Theodoros Rekatsinas. Attention-based learning for missing data imputation in holoclean. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 307–325, 2020.

[34] Ce Zhang, Pradap Konda, and Emily Mallory. Deepdive: A data management system for automatic knowledge base construction, 2015.

[35] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. A statistical perspective on discovering functional dependencies in noisy data. pages 861–876, 06 2020.