PAPER *Special Section on Data Engineering and Information Management*

Automatic Erroneous Data Detection over Type-Annotated Linked Data

Md-Mizanur RAHOMAN[†a)], ***Nonmember and*** **Ryutaro ICHISE**[†,††b)], ***Member***

SUMMARY These days, the Web contains a huge volume of (semi-) structured data, called Linked Data (LD). However, LD suffer in data quality, and this poor data quality brings the need to identify erroneous data. Because manual erroneous data checking is impractical, automatic erroneous data detection is necessary. According to the data publishing guidelines of LD, data should use (already defined) ontology which populates type-annotated LD. Usually, the data type annotation helps in understanding the data. However, in our observation, the data type annotation could be used to identify erroneous data. Therefore, to automatically identify possible erroneous data over the type-annotated LD, we propose a framework that uses a novel nearest-neighbor based error detection technique. We conduct experiments of our framework on DBpedia, a type-annotated LD dataset, and found that our framework shows better performance of error detection in comparison with state-of-the-art framework.

key words: *type-annotated LD, data quality, erroneous data detection*

1. Introduction

Linked Data (LD) is a large knowledge base that contains the (semi-)structured data. A significant portion of these data are built on people's mass contributions (e.g., Wikipedia) or automatic extraction from other data sources (e.g., DBpedia*).

However, both types of data are subject to erroneous data gathering. For the first type of data (mass contributions), data are contributed by the general public, who might not always have enough expertise to generate correct data. For the second type of data (automatic extraction), data can be automatically extracted from other data sources, but automatic data extractors might extract incorrect data. However, to use such LD effectively, data consumers commonly expect to easily retrieve high-quality data. This brings the need to identify erroneous data in the LD. Usually, manual erroneous data checking is impractical. Therefore, automatic erroneous data detection is necessary.

On the other hand, according to the best-practice data publishing guidelines [7] of LD, data should use (already defined) ontology which populates type-annotated LD. For example, DBpedia is a type-annotated LD dataset. Usually, the data type annotation helps in understanding the data. However, in our observation, the data type annotation could be used to identify erroneous data. The intuition behind this assumption is that the same type of LD resources should share the same kind of values. Therefore, if data values of some LD go beyond the usual pattern or trend of other same type of LD, we consider them as erroneous data. However, the above assumption might not be always true, but it gives opportunity to check the data to find erroneous data over the type-annotated LD.

In the past, some studies have dealt with erroneous data findings in the LD. However, these studies have their own limitations. For example, some require LD domain-level expertise [1], [10], [22]. Some require another similar data source [4], [9], [13], are not suitable for diverse datasets, or are impractical for large datasets. Other works are for specific data types and ignore the errors for the remaining data types [6], [20].

In this study, we focus on these drawbacks. We propose a framework to identify possible candidate of erroneous data over the type-annotated LD. The framework is named ALDErrD (**A**uto **L**inked **D**ata **Err**or **D**etector) which automatically detect potential error patterns and predict possible candidate of erroneous data. The main features of our proposed framework ALDErrD are the following: i) It is free from manual intervention. ii) It does not require domain-level expertise. iii) It does not require other data sources of the same kind. iv) It is suitable for any type of data.

The remainder of this paper is organized as follows. Section 2 introduces work related to this study. In Sect. 3 we describe the basic idea of our research work. In Sect. 4 we describe the proposed framework in details. In Sect. 5 we describe experiments implementing the proposal and discuss our results. Finally, Sect. 6 concludes our study.

2. Related Work

Error detection over various data has been quite extensively studied. Chandola et al. reviewed some of them [5]. However, error detection over the LD is relatively new. We can categorize them into four major groups:

- The manual-intervention based error detection [1], [10], [22]. This kind of studies look into the LD dataset and then manually devise some rules to identify the errors. Although the studies generate decent outcomes,

Manuscript received July 1, 2015.
Manuscript revised November 2, 2015.
Manuscript publicized January 14, 2016.

[†]The authos are with SOKENDAI (The Graduate University for Advanced Studies), Tokyo, 101–8430 Japan.
[††]The author is with National Institute of Informatics, Tokyo, 101–8430 Japan.
a) E-mail: mizan@nii.ac.jp
b) E-mail: ichise@nii.ac.jp

DOI: 10.1587/transinf.2015DAP0022

*http://dbpedia.org/About

they require domain-level expertise. However, finding of domain-level experts is not easy. Moreover, when such experts are found, the process is still costly. Therefore, the manual-intervention based error detection studies are not easy to adapt for diverse datasets and impractical for large datasets.

- The particular-data-type based error detection [6], [20]. This kind of studies only check for error detection for particular data-type LD. Such as, Wienand et al. investigated to find error for numerical data-type LD [20]. However, the particular-data-type based erroneous data findings ignore large amounts of (other-data-type) erroneous data.
- The similar-data-source based error detection [4], [9], [13]. This kind of studies try to find two or more data sources for same LD resource and then compare the data to identify the error. However, the similar data sources are not readily available for all kind of LD resources. Moreover, when such data sources are available, cross checking of them is not easy. Therefore, the similar-data-source based error detection studies face adaptation difficulties.
- The ontology-enrichment based error detection [11], [12], [14], [18]. This kind of studies also need to check the data manually. In some cases, ontology-enrichment can be done automatically such as the work done by Lehmann et al [14]. In this research, authors automatically typified LD resources that do not hold type information; however their research focus was not for error detection. Our proposed framework can be adapted on the top of their proposal because we utilize type (i.e., Class) information as the input.

So, the contemporary works mainly suffer in LD adaptation. In the proposed framework, ALDErrD, we tackled this adaptation issue.

The prime strength of ALDErrD is that it does not require manual investigation of the LD dataset.

3. Basic Idea

Here we will describe the basic idea of our research framework. To do this, first we will exemplify the type-annotated LD, then we will share the idea.

As mentioned, the best-practice LD data publishing recommend to use (already defined) ontology, and it populates type-annotated LD. For example, if there are two RDF triples <res:[†]Michael_Jordan, ont:[††]height, 168.0 >[†††] and <res: Michael_Jordan, rdf:[††††]type, ont:BasketballPlayer>, the latter one typify the res:Michael_Jordan LD resource as "Basketball Player". Usually, the type annotation generalizes the LD resources. We use this generalization to find candidates of erroneous data over the LD.

[†]http://dbpedia.org/resource/

[††]http://dbpedia.org/onto/

[†††]Throughout the paper, examples are shown from DBpedia 3.8.

[††††]http://www.w3.org/1999/02/22-rdf-syntax-ns#

To identify candidates of erroneous data in the LD, we assume that the same type of LD resources share commonalities. In particular, we assume that the same type of LD resources share the similar kind of values for the same Property[†††††]. For example, in an LD dataset, if there are good number of LD resources are typified as "Basketball Player" (i.e., ont:BasketballPlayer), and the resources also hold "height" (i.e., ont:height) values, we should expect the height values would be similar kind of values. Therefore, for resources of a particular type LD, if literal values go beyond the usual pattern or trend of other resources of the same type of LD, we consider them as candidates of erroneous data. This idea is generally rational. For example, we expect individuals who are Basketball Players to be taller. So, if an individual Basketball player is not as taller as the most of the Basketball players, we can predict that the data might be wrong. However, the above assumption might not always be true, but it gives the option to check the data.

Technically, the above assumption has also been well studied in unsupervised error detection and is called nearest-neighbor based error detection [5]. In such a case, it is assumed that normal data instances occur in dense neighborhoods, while errors occur far from their closest neighbors [2], [3]. So, error detection requires a similarity/ distance measurement defined between/among the data [21]. In the type-annotated LD, nearest-neighbor based error detection is well suited for the variant called "multivariate nearest-neighbor based error detection" [17], because such error detection depends upon the attributes of data and, usually, the type-annotated LD hold several such attributes (e.g., type, domain, range, etc.).

On the other hand, since the LD are generated from various sources, keeping conformity among the data is a challenge. The presence or absence of a particular attribute of data or using data values in different formats might present the same kind of data in different ways. Usually, the ontology of the LD would restrict such varieties. However, in a real-world scenario, adhering to a strict ontology in the LD is not feasible. It introduces the requirement of grouping data instances for the presence or absence of attributes and the formatting of data values. For grouped data, it is assumed that normal instances lie close to their closest group centroid, whereas erroneous instances lie far away from their closest group centroid [8], [15]–[17]. Therefore, we adapt the nearest-neighbor based error detection for groups.

4. Detailed Description of ALDErrD

In this section, we describe our proposed framework ALDErrD in detail. We take a Class (such as Person) and a Property (such as Birth Date)[††††††] as input, and detect whether the LD resources hold erroneous literal values for the Objects of the given Property. Usually, a Class information typify an LD resource. Preferably, to check some

[†††††]Property can be inter-changed by Predicates in the RDF triple.

[††††††]In DBpedia, Person is http://dbpedia.org/Ontology/Person and Birth Date is http://dbpedia.org/property/birthDate

LD resources for their erroneous literal values of Object, we should select them for their most specific Class. This is because, an LD resource can be typified by multiple Classes, but the most specific Class will define it more precisely. The rdfs:subClassOf closure is used to determine the most specific Classes for LD resources. However, whether LD resources belong to the most specific Class or some other super Classes, ALDErrD works for any given Class. But as mentioned, the most specific Class will identify candidates of erroneous data more accurately.

In ALDErrD, the detected errors are for erroneous Object values. We consider the detected errors as "Type-1 Errors". Over an LD dataset, the Type-1 Errors appear because of

- Erroneous Content – data with wrong values (e.g., wrong actual values), and
- Erroneous Syntax – data with wrong syntactic patterns (e.g., wrong value format, wrong string pattern etc).

However, Type-1 Errors can be originated for wrong LD attributes (e.g., type, domain, ontology etc). We consider such errors as "Type-2 Errors" and classify them into four kinds:

i. Erroneous Type – data with wrong Type attachment towards the LD resources.

ii. Erroneous Domain – data with wrong Domain attachment towards the LD resources.

iii. Erroneous Range – data with wrong Range attachment towards the LD resources.

iv. Erroneous Property – data with wrong Property attachment towards the LD resources.

Therefore, if the Type 2 errors are identified, they further reveal the causes behind the Type 1 errors. By ALDErrD, we only detect the Type-1 errors, and if we want to classify them into Type-2 errors, we need to investigate them for further analysis. Moreover, since Type-1 and Type-2 Errors may co-exist for same [S-O] pairs, clear distinction between them is not always possible.

Figure 1 shows the work-flow of ALDErrD. We divide the proposed framework into two phases: Phase 1 – Attribute Based Error Detection and Phase 2 – Value Based Error Detection. In Attribute Based Error Detection, we group data for some attribute values. Such groups help in detecting a Phase 1 data error. In Value Based Error Detection, we take Phase 1 data that are still not considered as errors. Here we investigate data values and apply various nearest-neighbor based error detection techniques to identify possible anomalies. In Phase 1, we introduce a technique to group LD resources which leads later steps of the framework, therefore we implement Phase 1 before the Phase 2. It also reduces the execution time of Phase 2 because, in such a work-flow, the Phase 2 only requires to identify errors over the filtered-out data of Phase 1. Below we describe both phases in detail.

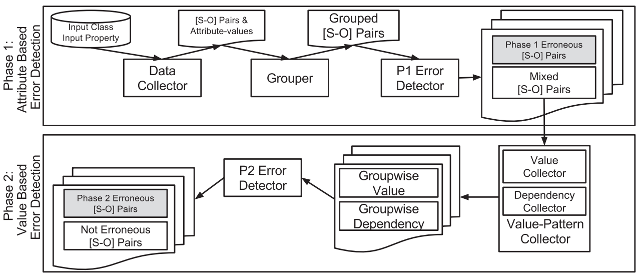
Phase 1: Attribute Based Error Detection
Input Class Input Property
[S-O] Pairs & Attribute-values
Grouped [S-O] Pairs
Data Collector
Grouper
P1 Error Detector
Phase 1 Erroneous [S-O] Pairs
Mixed [S-O] Pairs
Phase 2: Value Based Error Detection
Phase 2 Erroneous [S-O] Pairs
Not Erroneous [S-O] Pairs
P2 Error Detector
Groupwise Value
Groupwise Dependency
Value Collector
Dependency Collector
Value-Pattern Collector

Fig. 1 Work-flow of Proposed Framework ALDErrD

4.1 Attribute Based Error Detection

The upper half of Fig. 1 shows the work-flow of Attribute Based Error Detection. For the given input (i.e., a Class and a Property), first we use a process called Data Collector and collect Subject-Object ([S-O]) pairs (described below) along with some attribute values. Then, according to the attribute values, we use a process called Grouper and find groups among the [S-O] pairs. Then, for the grouped [S-O] pairs, we use a process called the P1 Error Detector and detect possible erroneous [S-O] pairs. Here, an [S-O] pair is erroneous data, if literal value of O of [S-O] is not correct.

In Phase 1, we utilize attributes as the main indicators to detect erroneous [S-O] pairs. Below we describe each process in detail.

4.1.1 Data Collector

We collect LD resources for all RDF triples <Subject, rdf:type, Class>. Then, for each Subject, we collect [S-O] pairs for RDF triples <Subject, Property, Object>, where "S" represents Subject and "O" represents Object. Apart from [S-O] pairs, we also collect five different attribute values for each S and each O of [S-O] pairs. The attributes are i) type of literal value (LVT), ii) associated properties (PRT), iii) associated classes (CLS), iv) associated domain (DOM), and v) associated range (RNG). Practically, the literal value of Object will be used to identify the error. To do so, the LVT information largely allows us whether literal values are holding same kind value, so we collect the LVT. On the other hand, the remaining four attributes (i.e., PRT, CLS, DOM and RNG) will be used to check whether the same type LD resources uses same attributes. We use the above-mentioned attributes because they possibly can be found in a type-annotated LD. However, the readers can include further attributes that might produce better result. But in current ALDErrD setting, we use the above-mentioned attributes.

The finding of the LVT requires some processing, whereas finding the values of the other four attributes (i.e., PRT, CLS, DOM and RNG) just require data picking. Below we describe the value collection of these two types of attributes.

- **LVT**. We first find the literal value, and then find the type of literal value (LVT). For S of an [S-O] pair, it always holds the URI, while O always holds either the URI or a literal value. If S or O holds a URI, we extract its label for treating it as a literal value. However, it is possible that the LD does not contain the label of O or the label of S. In such a case, we consider the URI as a literal value.

If O of an [S-O] pair is a literal value, usually the value is annotated by the data type. We consider this data type as a schema-defined type. For [S-O] pairs, we collect unique schema-defined types (SDTs) for literal values of O. However, it is not guaranteed that the literal value will always hold the data type annotation. In such a case, we need to devise the LVT. For any literal value that does not hold the data type annotation, we classify their LVTs into four types: STRING, DATE/TIME, NUMBER or URI. We adapt this classification from the study [23]. If we find that the literal value is only a URI, we consider the LVT as a URI. Otherwise, we execute a language parser over the literal value and determine the LVT from the named entities (NEs) of the parsed output. The following equation gives us the LVT, where "x" is either S or O.

$$\mathrm{LVT(x)} = \begin{cases} \text{data type} & \text{(if data type is defined)} \\ \text{URI} & \text{(if literal value is URI only)} \\ \text{DATE/TIME} & \text{(if parsed output of literal value} \\ & \text{hold DATE/TIME type NE)} \\ \text{NUMBER} & \text{(if parsed output of literal value} \\ & \text{hold NUMBER type NE)} \\ \text{STRING} & \text{(otherwise)} \end{cases}$$

For example, we might have the [S-O] pair [rsc:Tom_Cruiz −170.18∧∧[†]centimeter], where 170.18 is the literal value annotated by using symbol (∧∧) centimeter, and it is the LVT(O). Another exemplary [S-O] pair could be [rsc:Tom_Cruiz−1962-07-03], where 1962-07-03 is a literal value, but it does not have the data type annotation, so the language parser identifies it as DATE/TIME.

- **PRT, CLS, DOM and RNG**. We use RDF triple patterns to collect the attribute values. The values of PRT and CLS can be found if S or O of the [S-O] pair is a URI. On the other hand, the values of DOM and RNG can be found when PRT exists. Therefore, if S or O is not a URI and the required triple pattern does not exist over the LD, we consider the respective attribute values as null. The below equations collect the PRT, CLS, DOM and RNG values, respectively. In these equations, "x" is either S or O. To extract CLS, DOM and RNG, we use 3 common properties type, rdfs:[††]domain and rdfs:range respectively.

$$\mathrm{PRT(x)} = \begin{cases} \{?p|<x,?p,?o>\} & \text{(if x is URL } \wedge \\ & \exists<x,?p,?o>) \\ \text{null} & \text{(otherwise)} \end{cases}$$

[†]http://dbpedia.org/datatype/

[††]http://www.w3.org/2000/01/rdf-schema#

Table 1 Examples of attribute values for different Ss (or Os) for the LD resource res:Tom_Cruise.

Attribute Name	**Attribute Value**
PRT(x)	ont:placeOfBirth, ont:dateOfBirth, . . . , etc
CLS(x)	ont:Artist, ont:Actor,. . . , etc.
DOM(x)	ont:Person, . . . , etc.
RNG(x)	ont:Place,rdfs:date. . . , etc

Table 2 Exemplary [S-O] pair groups divided by the horizontal double lines in the table.

[S-O] Pair	**LV(O)**	**LVT(O)**
[rsc:Gabriella_Hall−–09-05]	–09-05	MonthDay
[rsc:Armand_Dorian−6.0]	6.0	foot
[rsc:Nora_Danish−160.0]	160.0	centimeter
[rsc:Belinda_Hamnett−165.1]	165.1	centimeter
[rsc:Tom_Cruise−170.18]	170.18	centimeter
[rsc:MC_Jin−168.0]	168.0	centimeter
[rsc:Tsuchida_Bakusen−217.7]	217.7	centimeter
[. . .]	. . .	centimeter

$$\mathrm{CLS(x)} = \begin{cases} \{?c|<x,type,?c>\} & \text{(if x is URL } \wedge \\ & \exists<x,type,?c>) \\ \text{null} & \text{(otherwise)} \end{cases}$$

$$\mathrm{DOM(x)} = \begin{cases} \{?d|\forall p\in<x,p,?o>, & \text{(if } \exists p\in\mathrm{PRT(x)}\wedge \\ <p,rdfs:domain, & \exists<p,rdfs:domain, \\ ?d>\} & ?d>) \\ \text{null} & \text{(otherwise)} \end{cases}$$

$$\mathrm{RNG(x)} = \begin{cases} \{?r|\forall p\in<x,p,?o>, & \text{(if } \exists p\in\mathrm{PRT(x)}\wedge \\ <p,rdfs:range,?r>\} & \exists<p,rdfs:range,?r>) \\ \text{null} & \text{(otherwise)} \end{cases}$$

Table 1 shows examples of the four attributes for different Ss (or Os). Here, the 1st column shows the attribute name, the 2nd column shows the attribute value.

All the collected information is further used to identify the possible erroneous [S-O] pairs.

4.1.2 Grouper

We group [S-O] pairs by the LVT(O). These groups help in predicting a data error.

We apply all grouping options for the LVT(O). For example, we group [S-O] pairs for the LVT(O) either as STRING, DATE/TIME, or data type. Table 2 shows examples of three such groups. Here [S-O] pairs are considered for Class "Artist" and Property "height". The columns show the [S-O] pair, the literal value of O (LV(O)), and LVT(O).

By the group-based erroneous [S-O] pairs identification approach, we try to understand semantic values among the groups. Understanding the semantic values is required to reduce identifying false positive errors. This is because, in LD dataset some values could be syntactically very different but still they could be semantically similar. For example, in a LD dataset, some person's "height" can be stored in centimeter and some are in inch, but there will be huge difference if we directly compare both group values. If the error detection framework identify errors for all [S-O] pairs together, it can identify false positive errors. Therefore, we individually treat each group and compare values for their semantics and identify erroneous [S-O] pairs.

Next, within the groups, we detect the possible erroneous [S-O] pairs.

4.1.3 P1 Error Detector

The Property Range and the attribute values from the previous process are used in detecting Phase 1 error candidates. The P1 Error Detector detects those error candidates. It uses three types of methods – (1) Property Range validation, (2) Group-wise [S-O] pair observation, and (3) [S-O] pair similarity score calculation. Below we describe them in details.

1. Usually, the Object of an RDF triple follows a constraint that the Object follows the Property Range. So, the Property Range helps in detecting erroneous [S-O] pairs. We collect it from the range value of an RDF triple pattern as <Property, rdfs:range, ?r>. In any case, if ?r is not found, we assign it. To do this, if a Property holds a token word "DATE", we assign it as DATE/TIME. For example, for prp:[†]birthDate, the Property Range would be DATE/TIME, because it includes the token word "Date". Otherwise, we consider the Property Range can hold any literal value types such as NUMBER, STRING, URI, SDTs. So, we detect potential erroneous [S-O] pairs by observing the Property Range and LVT(O). If LVT(O) does not belong to the Property Range, we consider such [S-O] pairs as potential candidate of erroneous [S-O] pairs.

2. After detecting the above candidates, we detect further candidates of erroneous [S-O] pairs by the number of pairs each group holds. Here, we check an [S-O] pair group to determine whether it could entirely or partially holds erroneous pairs. For a group, if the ratio between its number of [S-O] pairs in the group and the total [S-O] pairs of all groups is less than a threshold α, we consider the group as an erroneous group and detect its [S-O] pairs as error candidates. Table 2 shows three such groups among which the first two groups (i.e., rows) as error candidate examples.

3. Next, we try to find possible erroneous [S-O] pairs for groups that do not entirely hold erroneous [S-O] pairs. In such a case, we try to find possible erroneous [S-O] pairs for one group at a time. In a group G, we calculate each [S-O] pair's similarity score (simScore([S-O])) towards the other [S-O] pairs of G.

   To do this, we calculate similarity for each attribute of x, where x is either S or O. They are $\text{sim}_{PRT}(\text{x})$, $\text{sim}_{CLS}(\text{x})$, $\text{sim}_{DOM}(\text{x})$, and $\text{sim}_{RNG}(\text{x})$. We do not calculate similarity for LVT(x), because it was already considered when we made the groups.

   To calculate similarity for an attribute ATT (= PRT, CLS, DOM or RNG) of x ($\text{sim}_{ATT}(\text{x})$), we first accumulate the group attribute values $\text{GAV}_{ATT}(\text{x})$ as

$$\text{GAV}_{ATT}(\text{x}) = \begin{cases} \{\text{ATT(S)} | \forall[\text{S-O}]\in\text{G},[\text{x-O}]\in\text{G}\} \text{ (if x is S)} \\ \{\text{ATT(O)} | \forall[\text{S-O}]\in\text{G},[\text{S-x}]\in\text{G}\} \text{ (if x is O)} \end{cases}$$

   Then, $\text{sim}_{ATT}(\text{x})$ is measured by $|\text{ATT(x)}|/|\text{GAV}_{ATT}(\text{x})|$.

   So, for group G, we calculate each [S-O] pair's similarity score as

$$\text{simScore([S-O])} = \sum_{ATT\in\{PRT,CLS,DOM,RNG\},x\in\{S,O\}} \text{sim}_{ATT}(\text{x}).$$

   In this way, we find similarity scores for all [S-O] pairs of group G. We detect possible erroneous [S-O] pairs of group G by finding the Outlier simScore([S-O]).

   We calculate the Outlier based on the Interquartile Range (IQR) [19]. In the data error detection, an Outlier is a data value that resides far from other values, and the IQR is simple but effective way to identify such an Outlier. Here, for a rank-ordered data value set, quartiles divide them into four equal parts. The values that divide each part are called the first (Q1), second (Q2), and third (Q3) quartiles respectively. The Outlier point is below Q1 or above Q3 due to the consideration that it is measured by a factor of the IQR, and where the IQR itself is $\text{IQR} = Q_3 - Q_1$.

   In our case, we consider the factor of IQR is 1.5. We adapt this factor from the research of Kontokostas et al [20]. Data value smaller than $Q_1 - 1.5*\text{IQR}$ and larger than $Q_3 + 1.5*\text{IQR}$ is considered Outlier. The factor of the IQR could be varied. So, Outlier simScore([S-O]) holding the [S-O] pair is considered as a candidate of erroneous [S-O] pair.

The above described error candidate identification procedure is quite different from the basic technique. Over the LD, the basic error candidate identification relies on property Range and other LD Attribute values [13]. Usually, such property Range checking depends on a one particular Range value for an input property. However, in real-world scenario data are stored for different Ranges e.g., "height" of person could be stored as meter, inches etc. Moreover, the Range values are not always present in the data. Furthermore, we can not assume that if LD resources do not store some attribute values, they are erroneous. Therefore relying on a single Range value for a property does not work in reality because the basic technique is too strict. On the other hand, in our proposal we divide LD resources in groups and handle each group differently. It increases error data identification efficiency.

[†]http://dbpedia.org/property/

In Fig. 1, the Phase 1 erroneous [S-O] pairs are shown in the shaded boxes.

4.2 Value Pattern Based Error Detection

The lower half of Fig. 1 shows the work-flow of the value pattern based error detection. Here, we take groupwise [S-O] pairs that Phase 1 does not consider as candidates of erroneous [S-O] pairs. We considered such [S-O] pairs as mixed [S-O] pairs, because they still might hold some erroneous pairs. In this phase, we utilize the LV(O) to detect the error candidates. The 2nd column of Table 2 shows literal values for some Os. First, we use the process called Value Pattern Collector, which stores the Groupwise Value and the Groupwise Dependency. This information helps the next process, called P2 Error Detector, to detect candidates of erroneous [S-O] pairs.

4.2.1 Value Pattern Collector

Value Pattern Collector has two sub-processes: Value Collector and Dependency Collector. With the Value Collector, we store the LV(O) of the [S-O] pairs of a group and decide their Outlier. With the Dependency Collector, we devise Dependency patterns between the Properties of S of the [S-O] pairs of a group and decide which [S-O] pairs violate the usual pattern and then predict the potential error. For example, the Dependency pattern helps to identify the error that the “death date” should not be later than the “birth date”.

We already discussed the LV(O) (see Sect. 4.1.1). Below we describe the extraction of Dependency patterns. The Dependency pattern is measured by the literal values that each two frequent Properties hold.

First, we discuss frequent Properties, then frequent Property related literal values (PrLV) and then the Dependency pattern calculation. A frequent Property is p ∈ PRT(S), which is at least common for the β% of S of the [S-O] pairs of G. Then, to find the Dependency pattern, we check Property related literal values for each frequent Property p_i,p_j. So, we calculate

$$\text{PrLV}(S,p_i,G) = \{\text{LV}(?o_i) \mid [S\text{-}?o_i] \in G, <S,p_i,?o_i>\}$$

$$\text{PrLV}(S,p_j,G) = \{\text{LV}(?o_j) \mid [S\text{-}?o_j] \in G, <S,p_j,?o_j>\}$$

Then, for PrLV(S,p_i,G) and PrLV(S,p_j,G), we check three different trends: “>”, “=”, and “<” as

$$\text{TRN}(S,p_i,p_j,G) = \begin{cases} > (\text{if PrLV}(S,p_i,G) > \text{PrLV}(S,p_j,G)) \\ < (\text{if PrLV}(S,p_i,G) < \text{PrLV}(S,p_j,G)) \\ = (\text{otherwise}) \end{cases}$$

Then, for each trend (“>” or “=” or “<”), we calculate the Groupwise trend

$$\text{GTRN}(G,p_i,p_j,\text{“>”}) = |\{\text{TRN}(S,p_i,p_j,G) \mid \forall\, [S\text{-}O] \in G, \text{TRN}(S,p_i,p_j,G)=\text{“>”}\}|/|G|$$

$$\text{GTRN}(G,p_i,p_j,\text{“<”}) = |\{\text{TRN}(S,p_i,p_j,G) \mid \forall\, [S\text{-}O] \in G, \text{TRN}(S,p_i,p_j,G)=\text{“<”}\}|/|G|$$

$$\text{GTRN}(G,p_i,p_j,\text{“=”}) = |\{\text{TRN}(S,p_i,p_j,G) \mid \forall\, [S\text{-}O] \in G, \text{TRN}(S,p_i,p_j,G)=\text{“=”}\}|/|G|$$

If any of them is larger than a threshold γ, we consider that the particular trend holds the Dependency pattern for p_i and p_j. Therefore, after establishing such a Dependency pattern, if any [S-O] pair violates the trend for p_i and p_j, we consider it as a candidate of erroneous [S-O] pair. We check the Dependency pattern for each two frequent Properties. Therefore, the Dependency pattern-based errors can be only found when two frequent Properties are present for an LD resource. Currently, we devise the Dependency pattern for the frequent Property that generates the values NUMBER and DATE/TIME.

4.2.2 P2 Error Detector

We detect groupwise potential erroneous [S-O] pairs for the LV(O) (i.e., literal value of Object) and their Dependencies. We first describe how we detect erroneous the [S-O] pairs for the LV(O). Then we describe the same for their Dependency patterns.

For the literal values, we can have groups where the LVTs are either STRING, NUMBER, DATE/TIME, URI, or a data type. Below we describe finding the candidates of erroneous [S-O] pairs for each of them.

- LVT(O) is a STRING, so we consider the LV(O) could follow some patterns (such as patterns of Zip Codes), or could follow a syntactic similarity. When values follow some patterns, we check the common patterns among most of the values and violating [S-O] pairs are considered as erroneous. Currently, we adapt very basic common pattern checking, such as whether literal values hold some special characters after certain intervals, etc. When values follow syntactic similarity, we check the number of characters each LV(O) holds. We find the Outlier number based on the IQR (described in the P1 Error Detector). However, as an Outlier factor, we consider 0.25 instead of 1.5 because we assume that the number of characters for the LV(O) will not vary a lot. Outliers holding [S-O] pairs are considered as error candidates.
- LVT(O) is NUMBER, DATE/TIME, or a data type, so we find erroneous [S-O] pairs for their redundancy and their Outlier values.
  - If [S-O] pairs hold duplicate Os (i.e., Subject), we consider duplicate Os in the [S-O] pairs as erroneous. The rationale behind this is that we assume the same Property (e.g., ont:birthDate) would not hold different O values.
  - For the remaining [S-O] pairs, we follow Outlier based error founding. Again we use the IQR to calculate the Outlier. For example, in Table 2, the [S-O] pair [rsc:Tsuchida_Bakusen−217.7] is considered as an erroneous [S-O] pair.
- LVT is a URI, so currently we do not do anything for

such [S-O] pairs; however, they also could hold errors.

Considering all the above methods for different LVTs, we try to find possible candidate of erroneous [S-O] pairs for values.

On the other hand, for Groupwise Dependency patterns, we check which [S-O] pairs violate the Dependency and consider them as erroneous.

5. Experiment

We performed experiments on DBpedia v3.8. DBpedia is a type-annotated LD that covers various literal value type [S-O] pairs.

We did not find already defined Classes and Properties that can directly be used in the experiment. Therefore, we derived Classes and Properties from the erroneous DBpedia RDF triples <Subject, Property, Object> (or <S, P, O>) that Acosta et al. manually assessed by employing crowds in their study [1]. In our experiments, we consider the erroneous RDF triples of Acosta et al. study as baseline triples. By executing ALDErrD, our observation was whether ALDErrD could identify the RDF triples that Acosta et al. marked as incorrect.

We collected Property by the P of RDF triple <S, P, O> and Class by C = { c | <S, rdf:type, c> ∈ DBpedia RDF triples }[†]. As mentioned in the Sect. 4, the framework works better, when we input it with the most specific Class. We consider the most specific Class that does not hold rdfs:subClassOf closure and holds fewer RDF triples. For example, the RDF triple <res:Rodrigo_Salinas, prp:birthPlace, res:Puebla_F.C.> has four Classes, ont:Person, ont:Agent, ont:Athlete, and ont:Soccer Player. But we consider the most specific Class for res:Rodrigo_Salinas as ont:SoccerPlayer because ont:SoccerPlayer does not have rdfs:subClassOf closure attachment and has fewer RDF triples than the other three Classes.

We used the Stanford Parser[††] to parse literal values of S and O of the [S-O] pairs. As the threshold, we set $\alpha = 0.05$, $\beta = 80$, and $\gamma = 0.8$. The ALDErrD hardware specifications were as follows: Intel®Core$^{\text{TM}}$i7-4770K central processing unit (CPU) 3.50 GHz based system with 16 GB memory. We loaded DBpedia dataset in Virtuoso (version 06.01.3127) triple-store, which was maintained in a network server. The execution time was depended on number of LD resources hold by the input Class and Property. In Phase 1, the NE (Named Entity) finding for the literal value of Object (for details, see Sect. 4.1.1) required large amount of time. For each LD resource, on an average (calculated by executing 3 times) the NE finding required 6.4 seconds, and the rest of part of Phase 1 required 1.7 seconds. On the other hand, in Phase 2, the value pattern collection and the syntactic pattern checking required large amount of time. The value pattern collection was depended on other properties of LD resources (see Sect. 4.2.1), therefore the execution time got increased when LD resources held large amount of properties. For example, to identify error candidates for University and their address, it required almost one day for some 1800 LD resources.

In experiments, we acknowledge that errors can be judgmental and purpose driven. Therefore, evaluating an error detection framework is not easy. Moreover, calculating recall values for errors over a large dataset might not be plausible.

5.1 Experiment 1

The purpose of this experiment is to investigate whether both phases (Phase 1 and Phase 2) of ALDErrD can detect candidates of erroneous [S-O] pairs and whether those candidates were correct.

We describe the experimental result for four different Classes and Properties that were present in the baseline RDF triples. We picked them by considering them to be i). the most specific Class, ii). representative for many types of literal values and iii.) will not generate [S-O] pairs more than 2000 so that we can manually evaluate their qualities. We report the erroneous [S-O] pair finding investigations for the Phase 1 errors and the Phase 2 errors.

When we executed ALDErrD for a most specific Class c and a Property P, it collected [S-O] pairs from {<S, P, O> | <S, rdf:type, c>}. We considered an [S-O] pair is erroneous if literal value of O is not correct, which were generated for either types of errors: Type-1 and Type-2 Errors.

While the baseline RDF triples provided one [S-O] pair for each such triple, we identified more number of erroneous [S-O] pairs for the same one RDF triple-driven Class and Property. This is because, the ALDErrD identified errors are for all instances of a Class for a particular Property. The [S-0] pairs that belong to baseline triples, we evaluated them directly. However, for the newly identified [S-O] pairs that were not identified by Acosta et al. employed crowds, we evaluated them by engaging three *linked data experts* who have knowledge about DBpedia, DBpedia ontology. The engagement of linked data experts is only for evaluation purpose and they remain outside of the proposed framework. We considered each [S-O] pair's evaluation on the basis of "majority voting".

For the derived Classes and Properties, Table 3 shows error candidate detection at Phases 1 and 2 with their precisions and recalls. It also shows error classifications into Type-1 and Type-2 Errors. In the table, the 1st column shows the input Class and Property, the 2nd column shows the number of [S-O] pairs are found for the corresponding Class and Property. The 3rd column shows the total number of errors existence among the [S-O] pairs. The total number of errors existence were measured by the linked data experts. These are the gold standard errors. The 4th and the 5th columns show the detected errors in number. The results are divided into Phase 1 and Phase 2. As mentioned in Sect. 4, ALDErrD only identifies Type-1 Errors. We show them by

[†]We discarded the "yago" ontology.

[††]http://nlp.stanford.edu/software/corenlp.shtml

Table 3 Error candidate detection at Phase 1 and 2 and their detection classification, precision and recall

	# of [S-O] pairs	Total # of Errors Exist	# of Detected Error Candidates		Precision		Recall
			@ Phase 1	**@ Phase 2**	**@ Phase 1**	**@ Phase 2**	
			Type-1 Errors [Identified Correctly] **Type-2 Errors**	**Type-1 Errors [Identified Correctly]** **Type-2 Errors**			
ont:School prp:campusSize	334	67	Detected: 9 [Correct: 7] Erroneous Type : 4 Erroneous Domain : 4 Erroneous Range : 7 Erroneous Property : 4	Detected: 41 [Correct: 11] Erroneous Type : 7 Erroneous Domain : 7 Erroneous Range : 7 Erroneous Property : 7	0.889	0.268	0.269
ont:University prp:address	1731	620	Detected: 17 [Correct: 5] Type Error : 1 Erroneous Domain : 1 Erroneous Range : 5 Erroneous Property: 1	Detected: 55 [Correct: 55] Erroneous Type : 10 Erroneous Domain : 9 Erroneous Range : 55 Erroneous Property : 9	0.294	1.000	0.097
ont:Holiday prp:date	700	19	Detected: 19 [Correct: 19] Erroneous Type : – Erroneous Domain : – Erroneous Range : 19 Property Error: –	Detected: 0 [Correct: –] Erroneous Type : – Erroneous Domain : – Erroneous Range : – Erroneous Property : –	1.000	–	1.000
ont:County ont:currency	1261	23	Detected: 27 [Correct: 23] Erroneous Type : 4 Erroneous Domain : – Erroneous Range : 23 Erroneous Property: –	Detected: 0 [Correct: –] Erroneous Type : – Erroneous Domain : – Erroneous Range : – Erroneous Property : –	0.851	–	1.000

their number for each phase. We show correctly identified errors just below the detected numbers in square brackets (i.e., []). After detecting the Type-1 errors, we also investigated them for Type-2 errors. The investigation revealed some reason behind the Type-1 Errors. The investigation results are shown just below horizontal bars. The Types-2 errors were shown for Erroneous Type, Erroneous Domain, Erroneous Range and Erroneous Property for each phase. We investigated the Type-2 Errors and those were verified by the linked data experts. The 6th and the 7th columns show the error detection precision for Phase 1 and Phase 2. The 8th column shows the error detection recall.

The investigation showed that the erroneous [S-O] pairs were found more in Phase 1 than Phase 2 (although for last two cases, Phase 2 did not have any error candidates). Moreover, Phase 1 achieved higher precision than those in Phase 2. In the experiment, Phase 1 was more effective in identifying Type-2 Errors, while Phase 2 was more effective in identifying Type-1 Errors. However, as mentioned in Sect. 4 that both types of error may co-exist for same [S-O] pairs, we also found them in the experiment. As an example, for input ont:University and prp:address, Phase 2 identified good number of (i.e., 55) errors which belong to both Type-2 Errors and Type-1 Errors.

In Phase 2, we mainly try to find erroneous pairs by their values. We found that Phase 2 correctly identified erroneous data. As an example, for input ont:School and prp:campusSize, Phase 2 identified at least 4 erroneous LD resources that hold string pattern anomalies for their Object values. However, the values were sometimes very much diverse e.g., campus sizes are written in various ways such as with number of students (in text), area in square kilometer (in text), etc., therefore automatic identification of such errors require human judgments. However, for input ont:University and prp:address, Phase 2 achieved good precision value but they are only for tiny portion of the erroneous data (recall value is 0.097).

In all of the cases, the Object values were quite expressive by the Range values, therefore when Range values existed, it was easy to find errors. Moreover, we found that large number of LD resources do not keep LD attributes (Domain, Range, Type etc). But such attribute attached LD resources would help maintaining better quality data. Over such attribute attached LD, it would be easy to identify error candidates.

While Acosta et al. engaged crowds to identify each single error manually, ALDErrD detects errors in bulk automatically – which is an advantage over the Acosta et al. strategy. Moreover, the identification of errors for different types of literal value data can be considered as supportive argument that ALDErrD will be scalable over different datasets because datasets are mainly varied for their datatypes.

5.2 Experiment 2

The purpose of this experiment is to compare the error detection performance between ALDErrD and a state-of-the-

Table 4 Erroneous RDF triple finding performance comparison between ALDErrD and the ruled based system.

Type	# of Erroneous Triples	ALDErrD		Rule Based	
		# of Errors Found	Error Found %	# of Errors Found	Error Found %
DATE	151	100	66	95	63
STRING	134	39	29	10	7
NUMBER	76	32	42	24	32
URL	3	1	33	1	33
	364	172	47	130	36

art Linked Data error detection system [10]. The state-of-the-art system (i.e., we call as *rule based system*) devised 17 rules which can be adapted for different Properties and Classes. For example, one rule states that for a Property (say, ont:isbn), if the Object value does not match "∧[0-9]5$", the Property and Object values of the RDF triple are erroneous. However, adapting those rules needs explicit investigation of the Property and Object values and then selection of the appropriate rule and possible value matching constraints. This rule based system has been the subject of experiments for various types of literal value of DBpedia data, so we used it in the performance comparison.

In the rule based system, authors did not provide exact erroneous RDF triples that they identified. Therefore, to compare both systems with the same data, we used the RDF triples that hold Object value related errors in the Acosta et al. study. Acosta et al. provided 364 erroneous RDF triples that have wrong Object values in their RDF triples. We check whether both systems can capture the erroneous triples.

We executed ALDErrD for the most specific Class and Property and checked for erroneous [S-O] pairs. If the [S-O] pairs hold the Subject and Object element of RDF triples that Acosta et al. predicted as erroneous, we considered the pairs as "Found"; otherwise we considered them as "Not Found".

Table 4 shows the performance comparison between ALDErrD and the rule based system. According to the Gold standard Object values that Acosta et al. provided, we categorized Object values into four types: DATE, STRING, NUMBER, and URI (shown in the 1st column). The type calculation was done by the NE (named entity) of the parsed output of the Object value (for details, see Sect. 4.1.1). The 2nd column shows the number of erroneous RDF triples held by each type. The 3rd and 4th columns show the number of erroneous triples found and the errors found % by the proposed framework, respectively. The 5th and 6th columns show the same result for the rule based framework.

The bottom row shows total number of erroneous triples used in the test and their identification, by the systems.

Both systems worked comparatively well on the DATE type erroneous RDF triples. Most of the cases of date problems were due to their duplicate values. For other types of erroneous RDF triples, ALDErrD worked well. For example, for STRING type, ALDErrD identified three times more erroneous data than the rule based system. Phase 2 identified those errors. In overall comparison between the two systems, ALDErrD performs 10% better.

In ALDErrD,the nearest-neighbor based attribute value checking and the nearest-neighbor based literal value checking effectively identify erroneous [S-O] pairs. Moreover, while the proposed framework automatically finds candidates of erroneous RDF triples, the rule based system always requires rule adaptation. In ALDErrD, the use of the parser for the Object value and the heuristic on devising the LVT (literal value type) minimizes the adaptation overhead.

6. Conclusion

The LD is a large knowledge base. However, such data hold the possibility of erroneous data gathering. To use the LD effectively, error detection is a requirement. On the other hand, a significant portion of these LD keep the type information which populates type-annotated LD. The type annotated RDF triples gives the opportunity to identify erroneous data. In this study, we identify possible candidates of erroneous data over the type-annotated LD. Our framework automatically detects possible error patterns and predicts possible error candidates. Our proposed framework is free from manual intervention, does not require domain-level expertise or the same kind of data sources, and is suitable for any type of data. We experimented with our proposed framework over DBpedia erroneous RDF triple benchmark data and found the framework effectively predicts erroneous triples. We also compared our system with a state-of-the-art system and found that our system works better. Although we got some promising results, we still have space for our future work. In current setting, we mainly detected error candidates by observing outlier inside the data. In such a setting, we can not detect error candidates if all data hold same kind of errors which we want to investigate in future.

References

[1] M. Acosta, A. Zaveri, E. Simperl, D. Kontokostas, S. Auer, and J. Lehmann, "Crowdsourcing linked data quality assessment," In Proceedings of the 12th International Semantic Web Conference, vol.8219, pp.260–276, 2013.

[2] M. Breunig, H.-P. Kriegel, R.T. Ng, and J. Sander, "Optics-of: Identifying local outliers," In Proceedings of the 3rd European Conference on Principles of Data Mining and Knowledge Discovery, vol.1704, pp.262–270, 1999.

[3] M.M. Breunig, H.-P. Kriegel, R.T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," SIGMOD Record, vol.29, no.2, pp.93–104, May 2000.

[4] V. Bryl and C. Bizer, "Learning conflict resolution strategies for cross-language wikipedia data fusion," In Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion, pp.1129–1134, 2014.

[5] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," ACM Comput. Surv., vol.41, no.3, pp.15:1–15:58, July 2009.

[6] D. Fleischhacker, H. Paulheim, V. Bryl, J. Völker, and C. Bizer, "Detecting errors in numerical linked data using cross-checked outlier

detection," In Proceedings of the 12th International Semantic Web Conference, vol.8796, pp.357–372, 2014.

[7] B. Hyland, G. Atemezing, and B. Villazón-Terrazas, Best Practices for Publishing Linked Data, 2014, http://www.w3.org/TR/2014/NOTE-ld-bp-20140109/

[8] A.K. Jain and R.C. Dubes, Algorithms for Clustering Data, Prentice-Hall, Upper Saddle River, NJ, USA, 1988.

[9] D. Kontokostas, M. Brümmer, S. Hellmann, J. Lehmann, and L. Ioannidis, "NLP data cleansing based on linguistic ontology constraints," In Proceedings of the 11th Extended Semantic Web Conference, vol.8465, pp.224–239, 2014.

[10] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri, "Test-driven evaluation of linked data quality," In Proceedings of the 23rd International Conference on World Wide Web, pp.747–758, 2014.

[11] J. Lehmann and L. Bühmann, "ORE - A tool for repairing and enriching knowledge bases," In Proceedings of the 9th International Semantic Web Conference, vol.6497, pp.177–193, 2010.

[12] J. Lehmann, D. Gerber, M. Morsey, and A.-C.N. Ngomo, "Defacto - deep fact validation," In Proceedings of the 11th International Conference on The Semantic Web - Volume Part I, ISWC'12, vol.7649, pp.312–327, Berlin, Heidelberg, Springer-Verlag, 2012.

[13] P.N. Mendes, H. Mühleisen, and C. Bizer, "Sieve: linked data quality assessment and fusion," In Proceedings of the 2012 Joint EDBT/ICDT Workshops, Berlin, Germany, pp.116–123, 2012.

[14] H. Paulheim and C. Bizer, "Type inference on noisy RDF data," In Proceedings of the 12th International Semantic Web Conference, Sydney, vol.8218, pp.510–525, 2013.

[15] M. Ramadas, S. Ostermann, and B. Tjaden, "Detecting anomalous network traffic with self-organizing maps," In Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection, vol.2820, pp.36–54, 2003.

[16] R. Smith, A. Bivens, M. Embrechts, C. Palagiri, and B. Szymanski, Clustering approaches for anomaly-based intrusion detection, Proceedings of the Intelligent Engineering Systems through Artificial Neural Networks, pp.579–584, 2002.

[17] P.-N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining, (First Edition), Addison-Wesley Longman Publishing, Boston, MA, USA, 2005.

[18] G. Töpper, M. Knuth, and H. Sack, "Dbpedia ontology enrichment for inconsistency detection," In Proceedings of the 8th International Conference on Semantic Systems, pp.33–40, 2012.

[19] G.J.G. Upton and I. Cook, Understanding statistics, Oxford University Press, Oxford, 1996.

[20] D. Wienand and H. Paulheim, "Detecting incorrect numerical data in dbpedia," In Proceedings of the 11th Extended Semantic Web Conference, vol.8465, pp.504–518, 2014.

[21] J.X. Yu, W. Qian, H. Lu, and A. Zhou, "Finding centric local outliers in categorical/numerical spaces," Knowledge Information Systems, vol.9, no.3, pp.309–338, March 2006.

[22] A. Zaveri, D. Kontokostas, M.A. Sherif, L. Bühmann, M. Morsey, S. Auer, and J. Lehmann, "User-driven quality evaluation of dbpedia," In Proceedings of the 9th International Conference on Semantic Systems, pp.97–104, 2013.

[23] L. Zhao and R. Ichise, "Graph-based ontology analysis in the linked open data," In Proceedings of the 8th International Conference on Semantic Systems, pp.56–63, 2012.

Md-Mizanur Rahoman is a Ph.D. candidate in the Department of Informatics, The Graduate University for Advanced Studies, Tokyo, Japan. He also works as a faculty member (currently on study leave) in the Department of Computer Science and Engineering, Begum Rokeya University, Rangpur, Bangladesh. His research interests include intelligent information retrieval, semantic webs, and machine learning.

Ryutaro Ichise received his Ph.D. degree in computer science from Tokyo Institute of Technology, Tokyo, Japan, in 2000. From 2001 to 2002, he was a visiting scholar at Stanford University. He is currently an associate professor in the Principles of Informatics Research Division of the National Institute of Informatics, Japan. His research interests include semantic webs, machine learning, and data mining.