Detecting Errors in Databases with Bidirectional Recurrent Neural Networks

Sverin Holzer
Severin.Holzer@zuerich.ch
Statistics Office, City of Zurich
Zurich, Switzerland

Kurt Stockinger
Kurt.Stockinger@zhaw.ch
Zurich University of Applied Sciences
Winterthur, Switzerland

ABSTRACT

In recent years we have seen an exponential growth of data. However, not only does the volume of data grow but often also the numbers of erroneous data values. Detecting and cleaning these errors is an important data management problem to enable high quality data science pipelines. To reduce the human effort and to achieve good results in detecting errors, it is important to label the tuples which give the best impact for the error detection system.

In this paper we introduce an architecture based on bidirectional recurrent neural networks to detect errors in databases. The experimental results with 6 different datasets demonstrate that our approach shows *similar performance to state-of-the-art error detection system* per dataset. When considering the average of the F1-scores over all datasets, our approach called Enriched Two-Stacked Bidirectional (ETSB-RNN) outperforms state-of-the-art systems. Moreover, our approach achieves a lower standard deviation than existing work, which shows that our system is more robust.

In addition, our approach uses simpler machine learning algorithms that are *faster to train*. Depending on the dataset, our approach needs between less than one minute until 5 minutes for training for the benchmark datasets. Finally, our approach does *not require additional data augmentation techniques* to achieve high F1-scores. The system does not need any configuration or previous analysis of the data.

ACM Reference Format:
Sverin Holzer and Kurt Stockinger. 2021. Detecting Errors in Databases with Bidirectional Recurrent Neural Networks . In *EDBT: 25th International Conference on Extending Database Technology, 29th March-1st April, 2022, Edinburgh, UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The amount of data is growing exponentially because it is collected by many different stakeholders and at many different locations – but also with varying degrees of quality. If data scientists want to analyze the data, they often have to clean it first, which typically takes the most time of the whole data science pipeline [8]. The cleaning part can be split into two steps, first to *detect errors* and second to *repair errors*, i.e. to find the ground truth. Errors are values which deviate in the dirty dataset from the clean dataset (see highlighted values in Table 1) and can be categorized as *missing value* ('NaN'), *value/syntactic error* ('80,000', 'Romr', '12', 'BER', '850'), *integrity constraint violation* ('75000') and *data duplication*.

Table 1: Overview of a dirty and the corresponding clean dataset (A=Age, Sal=Salary)

A	Sal	ZIP	City
21	**80,000**	8000	**NaN**
45	98000	00100	**Romr**
30	92000	75000	Paris
12	99000	**BER**	Berlin
26	**850**	**75000**	Vienna

A	Sal	ZIP	City
21	80000	8000	Zurich
45	98000	00100	Rome
30	92000	75000	Paris
42	99000	10115	Berlin
26	85000	1010	Vienna

The focus of this work is to detect errors. While related work have used different error detection strategies [5, 7, 16, 19, 21, 22], there is ample room for improvement. In our approach we use bidirectional recurrent neural networks [25] (RNN), which have received little attention for solving this kind of challenge. Our approach generates a model which tries to find the best parameter settings during learning the content of the data. For training the system, we need labelled data. To reduce the effort for the user, we use only 20 tuples – similar to the state-of-the-art approaches. For selecting the data, which gives our system the best impact, we developed an algorihm to choose a diverse trainset.

Procedure in production: In production the user gives our system a dataset and chooses the number of tuples for training. The system is responsible for the data preparation step and uses our novel label sampling algorihm *DiverSet* to find the tuples with the most information content for the neural network architecture to learn. Afterwards, the user has to label the chosen tuples with either '0' (correct) or '1' (wrong) to produce the trainset. Finally, the system uses the trainset to learn which data points of the tuples are correct or wrong. The involvement of the user is only in the phase of labelling the data. Hence, the user does not need to understand the whole dataset but only the proposed tuples for labelling.

This paper has the following **contributions:**

- We introduce our end-to-end system architecture for error detection based on *bidirectional two-stacked RNNs*, which is divided in the two parts *data preparation* and *error detection*.
- We developed a novel algorithm for *trainset selection* to choose the tuples which provide the most information content for our system to learn.
- The experimental results show that our approach achieves similar performance to *state-of-the-art error detection system* on 6 benchmark datasets. Over all datasets, our systems

Queensland '21, April 29–March 01, 2022, Edinburgh, UK
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/1122445.1122456

shows a higher average F1-score and lower standard deviation, which demonstrates that our system is more robust than existing work. Finally, we do not require additional data augmentation techniques to achieve high F1-scores.

Outline: In section 2 we discuss about related work. For better understanding of our system we give background information in section 3. In the following section 4 we show the architecture of our developed system. In section 5 we introduce the results of our system in comparison with the state-of-the-art error detection systems. At the end we give a summary about the insights in selction 6.

2 RELATED WORK

In this section we review the related work on error detection methods. A good overview can be found in [2].

Early approaches employ rules and heuristics for data cleaning [1, 4] or identify functional dependencies based on integrity constraints [3, 18, 24, 27]. Other approaches use outlier detection and statistical analysis [10, 12, 26].

More recent methods such as ActiveClean [19], HoloDetect [11], Raha [21] and Rotom [23] use machine learning approaches to tackle the problem. The main challenge of these approaches is how to design machine learning algorithms that work well with few training labels. We will discuss these machine learning-based methods in more detail.

ActiveClean [19] dynamically trains machine learning models by requesting new labels from users interactively. The approach supports convex loss models such as linear regressions and SVM and prioritizes cleaning those records that are most likely to effect the results.

Raha [21] uses a novel sampling and classification scheme to choose the most promising labels for training. In addition, information from knowledge bases such as DBPedia can be integrated to improve the detection of certain errors. Raha automatically configures error detection strategie such as outlier detection algorithms [22], pattern violation detection algorithms [16], rule violation detection algorithms [7] and knowledge base violation detection [5]. Moreover, Raha represents the results of various error detection algorithms as a feature vector for training. In order to select the most promising records to be labeled by humans, hierarchical agglomerative clustering is used. Raha outperforms other error detection systems like dBoost [22] (outlier detection tool), NADEEF [7] (rule-based data cleaning system), KATARA [5] (knowledge bases to detect errors) and ActiveClean [19].

HoloDetect [11] uses the weak supervision technique data augmentation to enrich the training data sets and to train high quality error detection models with minimal human involvement. The basic idea is to learn data augmentation policies from potentially noisy input data.

Currently one of the most advanced methods is Rotom [23], which also uses data augmentation to improve the training of machine learning models. In particular, Rotom formulates data augmentation as a sequence to sequence task. The key idea is to use meta-learning to automatically learn policies for combining training examples from different data augmentation operators and to assemble them into high-quality training signals.

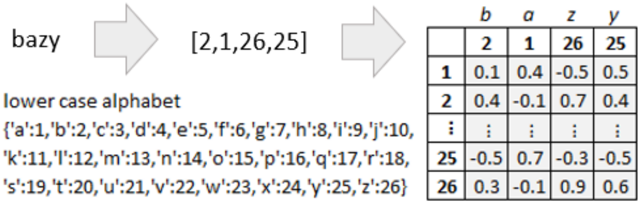
Our approach uses recurrent neural networks (RNN). The advantage of this approach is that no specific feature engineering is required and can be applied to a wide range of error detection problems. Unlike HoloDetect or Rotom, our approach does not require additional data augmentation techniques. Also, our approach does not need expensive GPU resources because it uses a relatively simple way to analyze sequential data. Compared to Long Short-Term Memory (LSTM) [14], gated recurrent unit (GRU) [6] or Bidirectional Encoder Representations from Transformers (BERT) [9], RNNs are less complex and therefore do need not as much time for training. To reduce the size of required traindata, i.e. the number of labelled tuples, our system uses a novel algorithm to choose tuples which gives the most information content for our algorithms to learn. This leads to better and more robust (lower standard deviation) results.

3 BACKGROUND

This section provides background information about character embedding [17] and the particular neuronal networks that we use for error detection.

3.1 Character Embedding

Typical values in databases are either of type numerical, date or text, i.e. character values. However, in order to train machine learning models for error detection, we need to convert all our input data into a vector space of numerical values between -1 and 1. The advantage in the comparison to the *one-hot representation* is that embedding considers the context of the occurring items, i.e. it groups similar items together in the vector space. This method typically yields better results than with the *one-hot representation* because the low number of the different dimensions of the vectore space typically prevent problems in the back propagation steps during training the model (for example, vanishing gradient [13]).


bazy
[2,1,26,25]
lower case alphabet
{'a':1,'b':2,'c':3,'d':4,'e':5,'f':6,'g':7,'h':8,'i':9,'j':10,
'k':11,'l':12,'m':13,'n':14,'o':15,'p':16,'q':17,'r':18,
's':19,'t':20,'u':21,'v':22,'w':23,'x':24,'y':25,'z':26}

	b	*a*	*z*	*y*
	2	**1**	**26**	**25**
1	0.1	0.4	-0.5	0.5
2	0.4	-0.1	0.7	0.4
⋮	⋮	⋮	⋮	⋮
25	-0.5	0.7	-0.3	-0.5
26	0.3	-0.1	0.9	0.6

Figure 1: Character embedding for the example sequence 'bazy'.

The first step is to generate a dictionary where each unique character is encoded with an index ranging from 1 to the maximum number of characters. As an example we use the lower case alphabet ('a':1,'b':2,...,'y':25,'z':26) consisting of 26 characters. This means the character 'a' is encoded by the number 1. For example, if we want to encode a sequence of 4 characters, for instance 'bazy', we get the sequence 2,1,26,25. If we want to encode the above sequence ('bazy' -> [2,1,26,25]) using *character embedding*, we have to produce a matrix with dimension 4x26. The first dimension is of size 4 because our input sequence has a maximum length of 4.

The second dimension is of size 26, which is due to the dimension space for the alphabet in our system – it could be higher or lower if desired (see example embedding in Figure 1). The values in the figure are fictitious and they are for visualization only.

In our error detection system we use this approach to transform a sequence of characters to a sequence of numbers in the data preparation step (see Section 4.1 Data Preparation).

The first layer in our Neural Network (see Section 4.3 Neural Network Architectures) is called *Embedding layer* and transforms every number from the input sequence into a vector, i.e. character embedding. As dimension space we choose the size of the dictionary per dataset.

3.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are so-called sequence to sequence models that are typically used for natural language processing (NLP) tasks such as machine translation, language generation or question answering. The approach is helpful if you need the context from the previous input to make predictions about subsequent outputs, for instance, to predict the next word of a sentence. In these settings the inputs are character embeddings x_t to x_{t+n} – similar to the ones we discussed in the previous section.

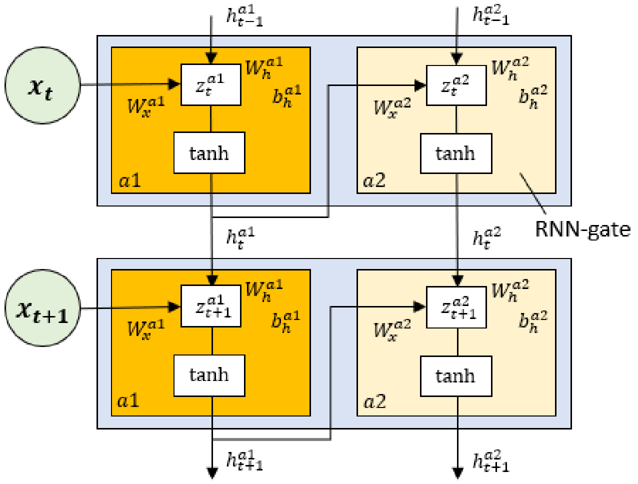


Figure 2: Two-stacked RNN.

Figure 2 shows a *forward two-stacked RNN* with the two levels $a1$ (left side, shown in orange) and $a2$ (right side, shown in yellow). The input to the RNN are the vectors of the characters, i.e. x_t and x_{t+1}. $\mathrm{W_x}$ and $\mathrm{W_h}$ are the weights and $\mathrm{b_h}$ are the biases of the RNN which we train. The weights and biases for all RNN-gates of the same level are equal. Moreover, the weights and biases of level $a2$ receive the computed hidded state $\mathrm{h^{a1}}$ as input. The hidden states $\mathrm{h^{a1}}$ and $\mathrm{h^{a2}}$ memorize the history of the previous step. Keeping in mind the history of previous steps is important for learning dependencies between different characters, which is also the big difference to a fully connected neural network, which does not keep history. The parameter $\mathrm{z^{a1}}$ combines the information from the actual character x and the previous history $\mathrm{h^{a1}}$. After this step the activation function *tanh* is used to generate the history for the next RNN-gate.

The parameters z_t^{a1}, h_t^{a1}, z_t^{a2} and h_t^{a2} are calculated as follows:

$$z_t^{a1} = W_x^{a1} \cdot x_t + W_h^{a1} \cdot h_{t-1}^{a1} + b_h^{a1} \tag{1}$$

$$h_t^{a1} = tanh(z_t^{a1}) \tag{2}$$

$$z_t^{a2} = W_x^{a2} \cdot h_t^{a1} + W_h^{a2} \cdot h_{t-1}^{a2} + b_h^{a2} \tag{3}$$

$$h_t^{a2} = tanh(z_t^{a2}) \tag{4}$$

4 SYSTEM ARCHITECTURE

In this section we introduce two different neural network architectures of our error detection system given the clean and dirty version of a dataset. The main difference between the two architecture models is that the first model *Two-Stacked Bidirectional RNN* (TSB-RNN) receives input from one dataset (value_x, i.e. the actual data values), while the second model *Enriched Two-Stacked Bidirectional* RNN (ETSB-RNN) receives input from three datasets (value_x, i.e. the actual data values, length_norm, i.e. the standardized length of value_x, as well as metadata information about the attributes). Therefore, the second model is more complex and needs more time for training (see Section 5.6 Training Time).

We will first discuss the data preparation step, then three different algorithms for selecting training data, and afterwards we describe the detailed system architecture for the models TSB-RNN and ETSB-RNN. The basic idea is that the data preparation step transforms the input data such that it can later be used by the neural network for error detection. Moreover, in order to enable effective learning with the minimal number of labeled training data samples, we discuss various methods of choosing those labeled training datasets.

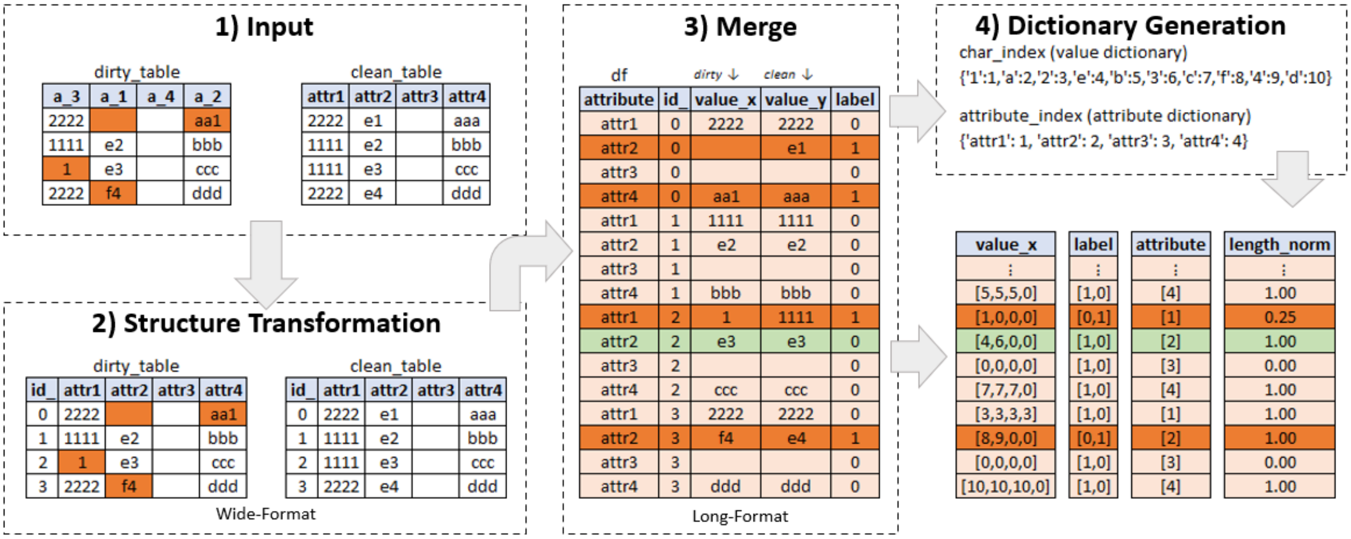
4.1 Data Preparation

First we have to prepare the data so that we can produce a train- and testset, which in turn can be fed into a neural network to detect errors in the dataset and measure the results. Therefore, we perform the following steps (see Figure 3):

(1) **Input:** We load both datasets (dirty and clean) as dirty_table and clean_table. Both are wide formats, i.e. each row is a tuple. In the example, cells with errors are marked in orange.

(2) **Structure Transformation:** Next we remove preceding white spaces. Additionally, we add a column named 'id_' as a sequence number for every row to identify the tuple in later steps. Also we rename the different column names in the dirty_table to have identical names with the clean dataset. We need this renaming and the added column to combine the information of both datasets and create a new one (df).

(3) **Merge:** Afterwards we combine the two tables to the dataset df where every cell of the dirty_table / clean_table is saved in the columns value_x / value_y, respectively. The new dataset has the long format, i.e. each tuple ('id_') has the



1) Input

dirty_table

a_3	a_1	a_4	a_2
2222			aa1
1111	e2		bbb
1	e3		ccc
2222	f4		ddd

clean_table

attr1	attr2	attr3	attr4
2222	e1		aaa
1111	e2		bbb
1111	e3		ccc
2222	e4		ddd

2) Structure Transformation

dirty_table

id_	attr1	attr2	attr3	attr4
0	2222			aa1
1	1111	e2		bbb
2	1	e3		ccc
3	2222	f4		ddd

clean_table

id_	attr1	attr2	attr3	attr4
0	2222	e1		aaa
1	1111	e2		bbb
2	1111	e3		ccc
3	2222	e4		ddd

3) Merge

df

attribute	id_	value_x	value_y	label
attr1	0	2222	2222	0
attr2	0		e1	1
attr3	0			0
attr4	0	aa1	aaa	1
attr1	1	1111	1111	0
attr2	1	e2	e2	0
attr3	1			0
attr4	1	bbb	bbb	0
attr1	2	1	1111	1
attr2	2	e3	e3	0
attr3	2			0
attr4	2	ccc	ccc	0
attr1	3	2222	2222	0
attr2	3	f4	e4	1
attr3	3			0
attr4	3	ddd	ddd	0

4) Dictionary Generation

char_index (value dictionary)
{'1':1,'a':2,'2':3,'e':4,'b':5,'3':6,'c':7,'f':8,'4':9,'d':10}

attribute_index (attribute dictionary)
{'attr1': 1, 'attr2': 2, 'attr3': 3, 'attr4': 4}

value_x	label	attribute	length_norm
⋮	⋮	⋮	⋮
[5,5,5,0]	[1,0]	[4]	1.00
[1,0,0,0]	[0,1]	[1]	0.25
[4,6,0,0]	[1,0]	[2]	1.00
[0,0,0,0]	[1,0]	[3]	0.00
[7,7,7,0]	[1,0]	[4]	1.00
[3,3,3,3]	[1,0]	[1]	1.00
[8,9,0,0]	[0,1]	[2]	1.00
[0,0,0,0]	[1,0]	[3]	0.00
[10,10,10,0]	[1,0]	[4]	1.00

Figure 3: Data Preparation Process.

same number of rows (number of attributes). For the models we need a label, which includes '0' (correct value) or '1' (wrong value). We get this label when comparing value_x and value_y. If the value has more than 128 characters, e.g. in datasets hospital, movies and rayyan as described in Section 5 Experiments, we cut them off. Our experiments showed that this approach achieves good results for F1-score and reduced training time. For the Algorithm 3 (DiverSet) – see Section 4 System Architecture – we need information if the value_x is empty (1) or not (0) and put it in a new column ('empty'). Also, the algorithm needs concatenated information about the attribute and the corresponding value ('value_x'). Therefore, we produce a new column ('concat'). Finally, we compute the length of value_x (i.e. number of characters) in relation to the value with the longest length per attribute ('length_norm'). This information is used by our novel algorithm ETSB-RNN.

(4) **Dictionary Generation:** Before we can feed the data into a neural network, we need to transform the data types from character to numeric. We produce a *value dictionary* (char_index) which contains an index for each character in value_x. We use the dictionary to convert the sequence of characters to the appropriate indexes. To get the same length, we add the values 0 at the end. The character embedding is done inside the neural network architectures (see Section 4.3 Neural Network Architectures).
For the ETSB-RNN we also need an *attribute dictionary* (attribute_index) which includes an index for each attribute. The attribute dictionary contains metadata information about attributes and shows from which attribute the content comes from. Assume, for instance, that the model learns that all values of attribute 'age' only contain numeric values. Further assume that the model receives the value 'San Francisco' which comes from the attribute age. Since 'San Francisco' is not a numeric value, the model can recognize it as a wrong value for the attribute age.

Example (see bottom right side of Figure 3): The green highlighted row shows the transformed characters. For instance, the value 'e3' is encoded as the characters 'e' and '3' and is stored in the *value dictionary*. Moreover, since the value 'e3' refers to attribute 'attr2', this metadata information is stored in the *attribute dictionary*. Finally, since the value 'e3' is correct, the corresponding label is encoded as '0' (true). Therefore, the first value in column 'label' has the value 1.

In our simple running example we assume the length of our *value dictionary* is 10 characters and the longest sequence of characters in 'value_x' is 4. Since the dimension of the sequence 'e3' is smaller than the longest value (e.g. '2222' or '1111'), the preparation step takes automatically the index 0 at the end. In other words, we pad short sequences of characters with the end-indicator.

4.2 Algorithms for Trainset Selection

After the preparation of the data, the system needs training and test data to train the system and then to test the trained machine learning model. However, real world datasets typically have very little or no labelled data. Hence, the user has to provide labels for the trainset. To reduce the human effort for generating training data, the system asks the user to provide labels for 20 tuples, that give our system the most impact during training. Hence, we analyze three different algorithms to choose the training set and took the rest as test set. As input we use the dataset described in Section 4.1 Data Preparation in step (3). Note: We do not use the value_y or label for choosing tuples. We only consider the value_x from the dirty dataset.

Our baseline approach is Algorithm 1, which chooses 20 tuples at random. This means that every tuple ('id_') in the dataset has the same probability of being selected as labeled training data. Since we have a long-format dataset as input (see the dataset of the merge step in Figure 3), we have to delete duplicated id_s before we can choose tuples. The algorithm is very simple and does not consider content during choosing tuples.

Algorithm 1 RandomSet algorithm

Input: *n_obs* -> number of tuples (id_) for training
df -> dataset with all data
Output: *ID_train* -> id_s of tuples for the trainset (size: *n_obs*)

```
function RandomSet($n\_obs, df$)
    $ID\_all \leftarrow$ unique($df['id\_']$)
    $ID\_train \leftarrow$ random_sample($n\_obs, ID\_all$)
    return $ID\_train$
end function
```

We also evaluate the label sampling approach of Raha [21] as shown in Algorithm 2. First, Raha generates and executes a set of error detection strategies. Then, it generates a vector which stores for each data cell and strategy a binary number referring to an error or or no error. In the next step, Raha groups similar cells with the help of the previously created vectors. As the last step, Raha draws 20 samples from different clusters.

Algorithm 2 RahaSet algorithm

Input: *n_obs* -> number of tuples (id_) for training
df -> dataset with all data
Output: *ID_train* -> id_s of tupels which are used for the trainset (size: *n_obs*)

```
function RahaSet($n\_obs, df$)
    $ID\_train \leftarrow$ RahaApproach(n_obs,df){
        import raha
        $sampled\_list \leftarrow []$
        run_strategies($df$)
        generate_features($df$)
        build_clusters($df$)
        while len($labeled\_tuples$) $< n\_obs$ do
            sample_tuple($df$)
            $sampled\_list \leftarrow sampled\_list \cup sampled\_tuple$
        end while
        return($sampled\_list$)
        }
    return $ID\_train$
end function
```

Finally we introduce a novel label sampling Algorithm 3, whose goal is to select a diverse trainset. The intuition is to select tuples with values that have not been seen previously. In other words, the newly select tuples increase the information content of our trainset the most. In case two candidate tuples contain the same number of unseen attribute values, the tuple with the highest number of empty attribute values is chosen. Our hypothesis is that empty values give us more information for the system to learn – because if there are empty values in other attributes, we can learn if they should be empty or not. In case all candidate tuples have the same number of unseen attributes and empty attribute values, the tuples are choosing randomly.

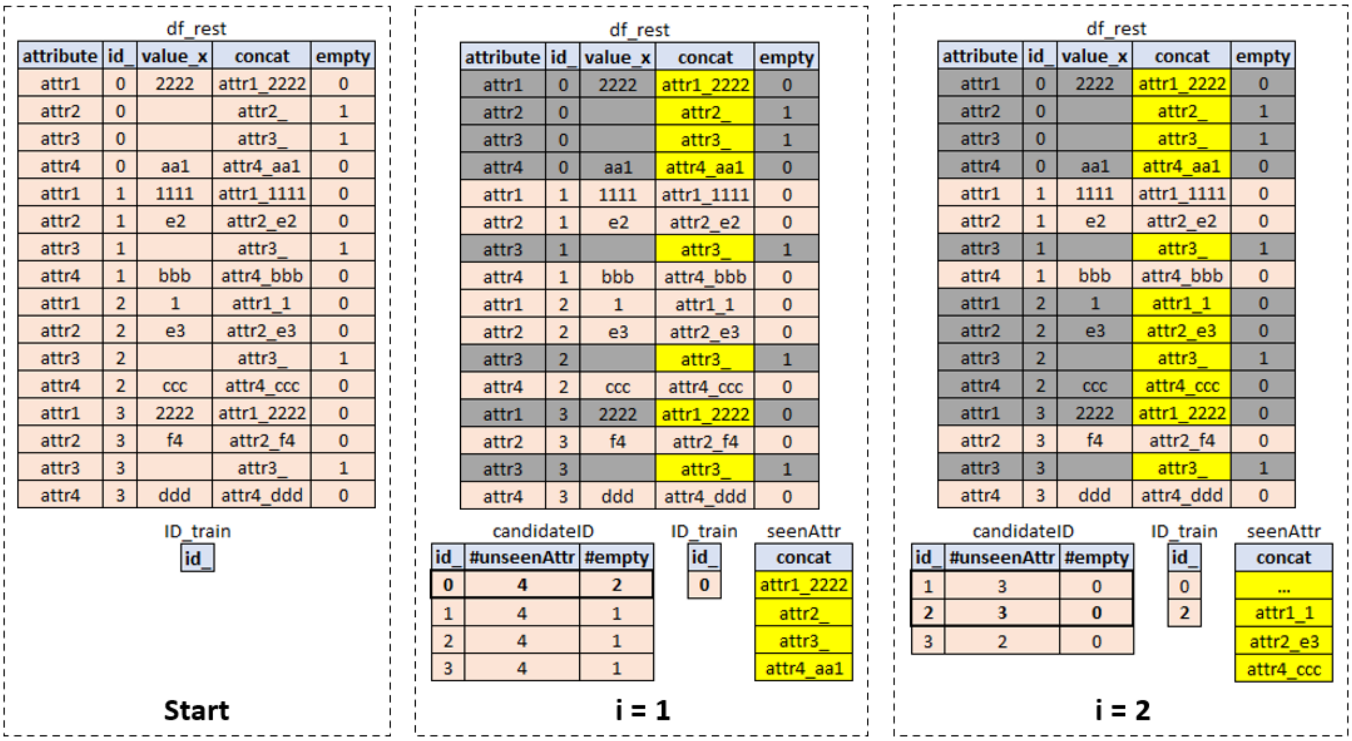
Algorithm 3 DiverSet algorithm

Input: *n_obs* -> number of tuples (id_) for training
df -> dataset with all data
Output: *ID_train* -> id_s of tupels which are used for the trainset (size: *n_obs*)

```
function DiverSet($n\_obs, df$)
    $df\_rest \leftarrow df$
    $ID\_train \leftarrow []$
    for i=1 do n_obs
        $\#unseenAttr \leftarrow$ count($df\_rest$).groupby($'id\_'$)
        $\#empty \leftarrow$ sum($df\_rest['empty']$).groupby($'id\_'$)
        $candidateID \leftarrow (id\_, \#unseenAttr, \#empty)$
        $candidateID \leftarrow candidateID$.max($\#unseenAttr$)
        $candidateID \leftarrow candidateID$.max($\#empty$)
        $sampled\_ID \leftarrow$ random_sample($1, candidateID$)
        $ID\_train \leftarrow ID\_train \cup sampled\_ID$
        $seenAttr \leftarrow$ unique($df['concat'].where(id\_ = ID\_train)$)
        $df\_rest \leftarrow df[df['concat'] \neq seenAttr]$
    end for
    return $ID\_train$
end function
```

To show the algorithm in a more understandable way, we have created the Figure 4 for *n_obs*=2 as an example. We used the same dataset (df) which we presented in Figure 3.

- **Start:** We initialize *df_rest* as *df* and *ID_train* as an empty list. Next we enter the loop, where the number of passes is given by n_obs.
- **i=1:** In the first step we produce the *candidateID* with both attributes *#unseenAttr* (counts all occurring *id_*) and *#empty* (sums up all empty values per *id_*). Next, we keep only the tuples (*id_*) with the highest value in *#unseenAttr* and from this new result only the tuples with the highest value in *#empty*. In this step we get *id_*=0 (bold framed) as a unique result and put it in *ID_train*. For *seenAttr* we put all values from the column *concat* where *id_*=0 in the dataset (df_rest). As the last step we delete all rows (marked in gray) which have occurring values in *seenAttr* (yellow values) out of *df_rest*. The rows with id_ not equal 0 are deleted because the values in in column *concat* equals the values in the dataset *seenAttr*.
- **i=2:** The second step produces again the dataset *candidateIDs* like the methods in *i=1*. Here, we do not receive a unique result because *id_=1* and *id_=2* have *#unseenAttr*=3 and *#empty*=0. From this candidateID (bold framed) we randomly sample one of the two tuples (*id_*). We assume the random sample chooses the tuple with *id_*=2 as result. We again append it to *ID_train* and append all values in column *concat* for this *id_* to the dataset *seenAttr*. At the end we again delete all rows (marked in gray) which have occurring values in *seenAttr* (yellow values) out of *df_rest*.
- **Return:** As result the algorithm returns a list with all tuples (*id_*) which we will use as our trainset. This example gives us the tuples with id_ equal 0 and 1 as trainset.



Start

df_rest

attribute	id_	value_x	concat	empty
attr1	0	2222	attr1_2222	0
attr2	0		attr2_	1
attr3	0		attr3_	1
attr4	0	aa1	attr4_aa1	0
attr1	1	1111	attr1_1111	0
attr2	1	e2	attr2_e2	0
attr3	1		attr3_	1
attr4	1	bbb	attr4_bbb	0
attr1	2	1	attr1_1	0
attr2	2	e3	attr2_e3	0
attr3	2		attr3_	1
attr4	2	ccc	attr4_ccc	0
attr1	3	2222	attr1_2222	0
attr2	3	f4	attr2_f4	0
attr3	3		attr3_	1
attr4	3	ddd	attr4_ddd	0

ID_train

id_

i = 1

df_rest

attribute	id_	value_x	concat	empty
attr1	0	2222	attr1_2222	0
attr2	0		attr2_	1
attr3	0		attr3_	1
attr4	0	aa1	attr4_aa1	0
attr1	1	1111	attr1_1111	0
attr2	1	e2	attr2_e2	0
attr3	1		attr3_	1
attr4	1	bbb	attr4_bbb	0
attr1	2	1	attr1_1	0
attr2	2	e3	attr2_e3	0
attr3	2		attr3_	1
attr4	2	ccc	attr4_ccc	0
attr1	3	2222	attr1_2222	0
attr2	3	f4	attr2_f4	0
attr3	3		attr3_	1
attr4	3	ddd	attr4_ddd	0

candidateID

id_	#unseenAttr	#empty
0	**4**	**2**
1	4	1
2	4	1
3	4	1

ID_train

id_
0

seenAttr

concat
attr1_2222
attr2_
attr3_
attr4_aa1

i = 2

df_rest

attribute	id_	value_x	concat	empty
attr1	0	2222	attr1_2222	0
attr2	0		attr2_	1
attr3	0		attr3_	1
attr4	0	aa1	attr4_aa1	0
attr1	1	1111	attr1_1111	0
attr2	1	e2	attr2_e2	0
attr3	1		attr3_	1
attr4	1	bbb	attr4_bbb	0
attr1	2	1	attr1_1	0
attr2	2	e3	attr2_e3	0
attr3	2		attr3_	1
attr4	2	ccc	attr4_ccc	0
attr1	3	2222	attr1_2222	0
attr2	3	f4	attr2_f4	0
attr3	3		attr3_	1
attr4	3	ddd	attr4_ddd	0

candidateID

id_	#unseenAttr	#empty
1	3	0
2	**3**	**0**
3	2	0

ID_train

id_
0
2

seenAttr

concat
...
attr1_1
attr2_e3
attr4_ccc

Figure 4: Example of Algorithm 3 for the first two loops to select a trainset.

Each algorithm returns a list with all 'id_' which we wanted to take for training. With this we create the train- (X_train, X_train_attribute and Y_train) and testsets (X_test, X_test_attribute and Y_test) which are the input for our systems.

4.3 Neural Network Architectures

We now describe the two different neural network architectures that we use for error detection.

4.3.1 Two-Stacked Bidirectional RNN (TSB-RNN). As the first model we use a *two-stacked bidirectional RNN*, which has 64 units and as activation function the hyperbolic tangent (see top left part of Figure 5). Note that the orange parts refer again to the stacking level *a*1 and the yellow parts to the stacking level *a2* as also shown previously in Figure 2. However, unlike the two-stacked RNN of Figure 2, our architecture shown in Figure 5 uses a *bidirectional* RNN, which means that in addition to the forward layer (highlighted in light blue), we also have a backward layer (highlighted in dark blue). Therefore, we compute the settings forward and backward through the character sequences.

As input the model receives the datasets X_train and X_test, where every character is encoded as a sequence of numbers. For instance, in Figure 5, the input is the character string 'e3' represented as a sequence of numbers (7,9,0,0). First, in the embedding layer the system transforms the sequence of numbers into a vector with the dimension of the used value dictionary. The output of our model is a concatenation of the output from the forward path (dim 64) and the backward path (dim 64). In addition, we use a fully connected layer with dimension 32 and ReLu for activation. At the end is a batch normalization [15] for standardize the input to the softmax. It reduce the generalization error and is a regularization technique.

At the end the model has to decide if the sequence of characters has the value '0' (correct) or '1' (wrong). For this last step the model uses a fully connected layer with a softmax. During training the model needs the dataset Y_train to optimize the loss. The dataset Y_test is not used for training but for validating the trained model with it.

4.3.2 Enriched Two-Stacked Bidirectional RNN (ETSB-RNN). Our second architecture ETSB-RNN is an enriched version of TSB-RNN. In particular, we additionally provide the *attribute information* X_train_attribute and X_test_attribute as a second input. The attribute information tells the neural network that, for instance, the encoding of the character sequence 'e3' refers to attribute attr2 (see in the left lower part of Figure 5). Note that TSB-RNN, that we discussed previously, only has the information about the character embedding but does not know to which attribute that value refers to. Also, we feed the system with the value about the ratio of the actual length of the input sequence to the maximum of the attribute (length_norm) is (see in the middle lower part of Figure 5).

As an additional layer the model uses a embedding layer and a two-stacked bidirectional RNN which has 8 units and as activation function the hyperbolic tangent. Also there is additionally a fully connected layer with dimension 64 and the activation function ReLu. We concatenate the output of this two new layers with the output

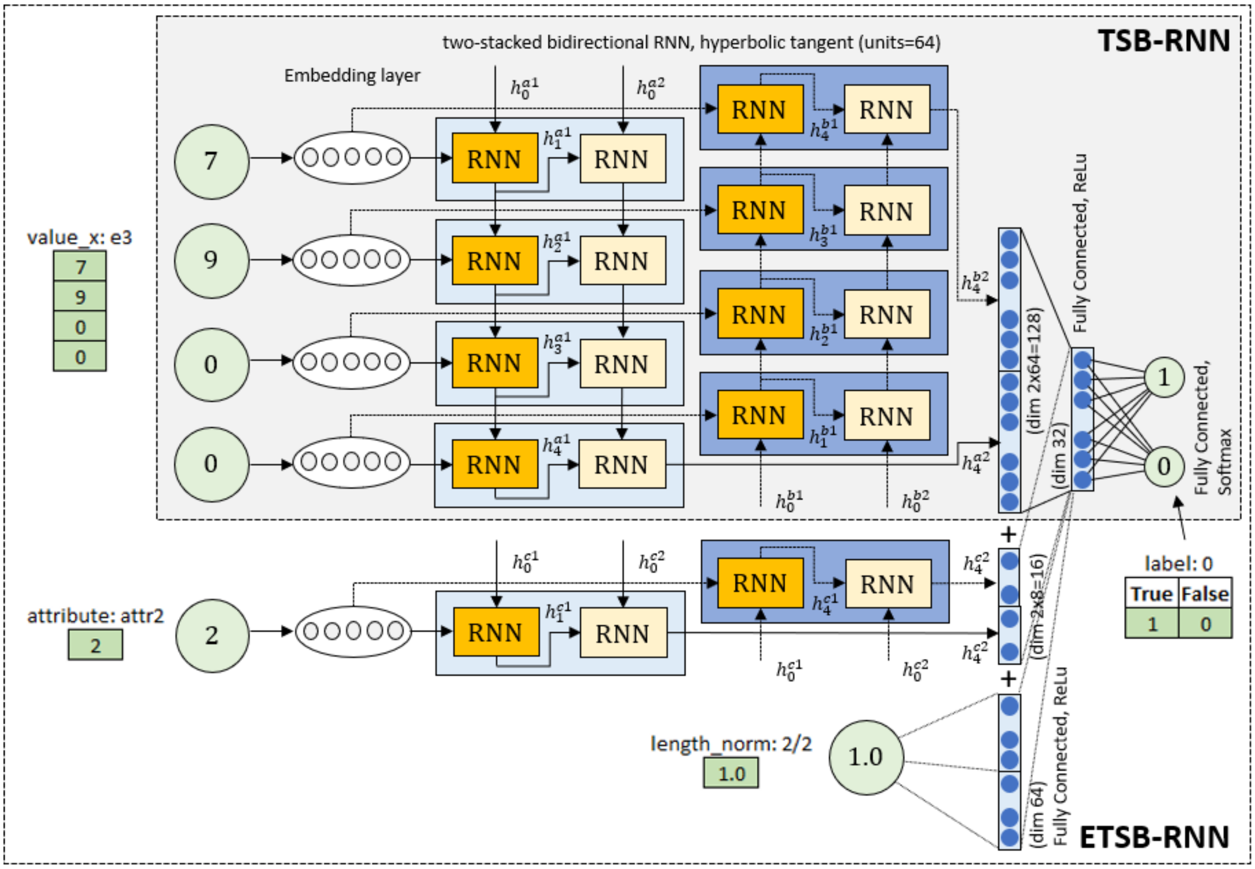


Figure 5: Overview of our bidirectional RNN architecture for error detection. Top part: TSB-RNN = Two-Stacked Bidirectional RNN. Bottom part: ETSB-RNN = Enriched Two-Stacked Bidirectional RNN.

of the previous two-stacked bidirectional RNN output. Afterwards we use the two fully connected layers and the batch normalization described in TSB-RNN again.

5 EXPERIMENTS

In this section we evaluate our two different architectures and measure how well they can detect errors in 6 commonly-used benchmark datasets. We also compare the results against the state-of-the-art-algorithms for error detection. Note that to enable the reproducibility of our experiments, we share our code on Github [1].

5.1 Datasets

For our experiments we used 6 datasets that were also used by the state-of-the-art error detection systems, e.g. from Raha [21] and Rotom [23]. All datasets have a version of data with errors (dirty) and the corresponding ground truth (clean).

An overview of the datasets is given in Table 2. The table shows the size of each dataset including the number of records and attributes, the error rate, i.e. the fraction of data that is wrong, the number of different characters, i.e. the size of the value dictionary and finally, the error types[2] in the corresponding dataset.

Table 2: Overview of datasets with the error types MV: missing Value, T: Typo, FI: Formatting Issue, VAD: Violated Attribute Dependency.

Name	Size	Error Rate	Different Characters	Error Types
Beers	2,410x11	0.16	86	MV, FI, VAD
Flights	2,376x7	0.30	70	MV, FI, VAD
Hospital	1,000x20	0.03	46	T, VAD
Movies	7,390x17	0.06	135	MV, FI
Rayyan	1,000x10	0.09	101	MV, T, FI, VAD
Tax	200,000x15	0.04	69	T, FI, VAD

Beers: This dataset is about different beers and contains information about the type of beer, the international bitterness unit (IBU), the alcohol by volume (ABV), the brewery and more. Typical data errors are, for instance, formatting issue (FI) in ounces ('12.0

[1] https://github.com/holzesev/E_TSB

[2] We took the definition of the error types from Raha [21].

Table 3: Comparison between the different models (20 labeled tuples). P = Precision, R = Recall, F1 = F1-Score, S.D. = Standard Deviation. Our approaches are TSB-RNN and ETSB-RNN.

Name	Beers			Flights			Hospital			Movies			Rayyan			Tax		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Raha	0.99	0.99	**0.99**	0.82	0.81	**0.81**	0.94	0.59	0.72	0.85	0.88	0.86	0.81	0.78	0.79	n/a	n/a	0.91
Rotom	n/a	n/a	**0.99**	n/a	n/a	n/a	n/a	n/a	**1.00**	n/a	n/a	0.68	n/a	n/a	**0.86**	n/a	n/a	0.97
Rotom+SSL	n/a	n/a	**0.99**	n/a	n/a	n/a	n/a	n/a	**1.00**	n/a	n/a	0.54	n/a	n/a	0.76	n/a	n/a	**1.00**
TSB-RNN	0.99	0.94	0.96	0.77	0.63	0.69	0.98	0.95	0.97	0.96	0.79	0.87	0.83	0.73	0.78	0..83	0.90	0.85
S.D.	0.01	0.01	0.01	0.05	0.04	0.02	0.01	0.02	0.01	0.04	0.04	0.03	0.05	0.06	0.05	0.16	0.08	0.11
ETSB-RNN	1.00	0.96	0.98	0.81	0.68	0.74	0.98	0.95	0.97	0.96	0.81	**0.88**	0.87	0.83	0.85	0.82	0.92	0.86
S.D.	0.00	0.02	0.01	0.03	0.04	0.02	0.03	0.02	0.02	0.03	0.05	0.02	0.06	0.02	0.03	0.15	0.07	0.10

oz’ rather than ’12.0’), ABV (’0.061%’ rather than ’0.061’), violated attribute dependency (VAD) between city / state and also missing values (MV) in state (’NaN’ rather than ’CA’).

Flights: This dataset contains information about the same flights received from different data sources. Depending on the data sources, flights have violated attribute dependency (VAD) in the departure and arrival time. In several chases the time information has missing values (MV) (” rather than, e.g. ’3:31 p.m.’), varies by few minutes (’9:00 a.m.’ rather than ’8:42 a.m.’) or has a formatting issue (FI) (’12/02/2011 6:55 a.m.’ rather than ’6:55 a.m.’).

Hospital: This dataset contains information about hospitals and the corresponding treatments (measures) for certain diseases, e.g. ’heart attack patients given aspirin at arrival’. In this dataset the hospital or measure information appears several times in the dataset but information varies, i.e. there are many violated attribute dependency (VAD). Moreover, there are typos (T) in attributes, e.g. wrong city names such as ’Birmingxam’ rather than ’Birmingham’. Note: The errors are not very hard to detect because they are marked with the character ’x’.

Movies: This dataset contains different movies and corresponding attributes like duration, genre etc. Data errors are formatting issues (FI) (e.g. RatingCount: ’379,998’ rather than ’379998.0’, e.g. name: ’Frankie & Johnny’ rather than ’Frankie and Johnny’ or RatingValue: ’8.0’ rather than ’8’), missing values (MV) (e.g. duration: ’NaN’ rather than ’126 min’), several year indications instead of only one year or missing words.

Rayyan: This dataset includes scientific articles written by researchers, and the related attributes like title, author etc. Data errors are again formatting issue (FI) (e.g. journal_issn: ’Mar-22’ rather than ’22-Mar’ or article_pagination: ’70-6’ rather than ’Jun-70’), missing values (MV) in article_jissue and typos (T) in journal_title or article_title.

Tax: This dataset contains information about people which is used for computing taxes, like name, gender, the salary ect. Data errors are typos (T) in f_name (e.g ’Jun”ichi’ rather than ’Jun’ichi’) or city (e.g. ’ARCHIE-*’ rather than ’ARCHIE’), formatting issues (FI) in zip (e.g. ’01907’ rather than ’1907’) or rates (e.g. ’7.0’ rather than ’7’) and violated attribute dependencies (VAD) between state / city and marital_status / has_child.

5.2 Training and Testing Setup

We validated the models 10 times and used the Algorithm 3 (DiverSet) for choosing the 20 tuples for training. For instance, for the dataset Beers we got a trainset of size 220, i.e. 20 tuples x 11 attributes, and a testset of size 26,290, i.e. 2,390 tuples x 11 attributes. During training we used a model batch size of a quarter of the trainset, i.e. with Beers: 220 / 4 = 55.

To find the algorithm which works best for our systems, we repeated the experiments several times with every algorithm described in Section 4.2 Algorithms for Trainset Selection. We reached the best results with our novel Algorithm 3. The basic idea of Algorithm 3 is to prevent, that the same values are selected for the trainset. We suppose that through the diverse trainset generated by Algorithm 3, the two neural networks algorithms *TSB-RNN* and *ETSB-RNN* received the most information content for the characters and hence could learn error patterns in the data.

The number of epochs was 120. After every epoch we saved the training weights (with a callback) if the computed loss of the trainset was less than in the previous epochs. For the loss-function we used the binary cross-entropy and as the optimizer RMSprop. We used the testset to measure precision, recall and F1-score for the best weights. Moreover, we computed the averages and standard deviations of these measures.

5.3 Comparison

In this section we compare our results with the Configuration Free Error Detection System (Raha, [21]), as well as with the Meta-Learned Data Augmentation Framework (Rotom, [23]), which is the state-of-the-art of error detection in databases. For both approaches we report the results of the original papers. The results are shown in Table 3.

Note that Rotom did not take the dataset Flights for validation. Therefore, in Table 4 we show the average (AVG) and standard deviation (S.D.) for the error detection system for all datasets without Flights (1) and in addition for all datasets including Flights (2). Also both papers did not show the standard deviation from the 5 times experiments (we did 10 times experiments).

If we compare the experiments of both papers, we can see different results in F1-score for the datasets Beers (+/-0.01), Hospital (+/-0.02), Movies (+/-0.06), Rayyan (+/-0.03) for Raha but nobody show off the standard deviation. So we can not see how the state-of-the-art model do scatter. The dataset Tax was only used in the paper of Rotom. Moreover, they did not measure precision and recall, i.e. the value n/a in the table 3. Additionally, Rotom and Rotom+SSL validated the results with 50, 100, 150 and 200 labelled cells and took the best results for comparison. In the real world we do not have a testset, therefore we do not know how many labelled cells would work best. So Rotom and Rotom+SSL would have to write the results from 200 labelled cells and not <200. Therefore, to have a direct comparison with both Raha and our approach, we reported the results for the models Rotom (Movies and Tax) and Rotom+SSL (Movies and Rayyan) using 200 labelled cells.

Our approach which also uses the metadata information (ETSB-RRN) outperform the simpler model TSB-RNN on all datasets. Also for all datasets we got similar or less standard deviation, therefore we took the ETSB-RNN for comparing with Raha, Rotom and Rotom+SSL.

Our model ETSB-RNN outperform Raha on 3 out of 6 datasets for the F1-score (see Table 3). We got better results for Hospital (+0.25), for Movies (+0.02) and for Rayyan (+0.06). We achieved similarly results for Beers (-0.01) and Tax (-0.04), only for Flights (-0.07) we got worse results. Also in the comparison of the average F1-score (AVG) without Flights (1) (+0.06) and with Flights (1) (+0.01), our approach ETSB-RNN outperforms Raha (see Table 4). Also the standard deviation is lesser for our system.

In comparison to Rotom and Rotom+SSL (see Table 3), our model ETSB-RNN got a better result for Movies (+0.20 / +0.34). For Beers (-0.01 / -0.01), Hospital (-0.03 - 0.03) and Rayyan (-0.01 / +0.09) we achieved very similar results – only for Tax (-0.11 / -0.13) our approach is considerable worse than Rotem+SSL. The average F1-score (AVG) is a little bit higher (+0.01 / +0.05) than Rotom and Rotom+SSL (see Table 4). In addition the standard deviation over all datasets is better.

Table 4: Average F1-score (AVG) and Standard Deviation (S.D.) for the different models of Table 3.

Name	Without Flights (1)		With Flights (2)	
	AVG	**S.D.**	**AVG**	**S.D.**
Raha	0.85	0.08	0.85	0.07
Rotom	0.90	0.10	n/a	n/a
Rotom+SSL	0.86	0.17	n/a	n/a
TSB-RNN	0.89	0.06	0.85	0.08
ETSB-RNN	**0.91**	0.05	**0.88**	0.06

5.4 Learning Analysis

For better understanding we analysis the behaviour of the accuracy from our models during training and test, in particular of the system ETSB-RNN, in this section.

To show the different improvements of the accuracy over various epochs of our models, we plotted the average and the confidence intervals of the 10 experiments (see Figure 6) for the test dataset. We can see that TSB-RNN and ETSB-RNN show a gradual increase of the accuracy. There is no clear deterioration. As described we optimized the train loss and took the settings from the epoch which achieved the lowest loss. We can see the ETSB-RNN outperforms TSB-RNN on all datasets, except for Tax, where both models show practically the same performance.

Finally, for ETSB-RNN, which performed best in our experiments, we also compared the average train- and test-loss (see Figure 7). The results show that the model performs well and does not suffer from overfitting. For all datasets our model achieves almost a perfect result for the train-accuracy. There are also gaps in the curves of the test-accuracy but these are not critical because the model chooses the epochs which do not have gaps.

The dataset flights, which reached the worst F1-score, has a big gap between the train- and test-accuracy and also the confidence interval is big. Therefore, our model does not work well for this dataset. The curve for test-accuracy for Hospital looks almost perfect, also because the confident interval is very small. The datasets Beers, Movies and Rayyan show also a converging test-accuracy curve and the confidence interval is small. For Tax we got a curious result. After epoch 30 there are no wave movements in the test-accuracy. In addition, the model chooses the optimal parameter settings from training close to epoch 120, i.e. the green dots and blue triangles.

5.5 Error Analysis

We will now perform a more detailed error analysis of our algorithm ETSB-RNN for each of the datasets.

As described previously, the dataset **Flights** contains duplicate values due to different information sources for the same flight. ETSB-RNN reached an average F1-score of 0.81. However, it had some problems to detect errors due to varying departure and arrival times of different sources, e.g. flight ’UA-257-JFK-SFO’ of source ’orbitz’ has a departure time of ’2:46 p.m.’ while the same flight of source ’flightstats’ has a departure time of ’2:26 p.m.’ The reason is that the model does not share the information about the name of the flight (i.e. the ID to recognize that flights are the same) and therefore it cannot detect duplicate records. Raha could find the dependencies in a better way, but there is also room for improving the result.

Detecting errors in the **Hospital** dataset is quite straightforward because the errors are marked with ’x’ (e.g. ’hexrt fxilure’). The model TSB-RNN received an almost perfect result with an average F1-score of 0.97. These types of error are easy to detect and do not dependent on the attribute metadata information. Hence, the models TSB-RNN and ETSB-RNN achieved very similar results. Rotom and Rotom+SSL outperforms our results.

For the dataset **Movies** the model ETSB-RNN does not recognise errors in the attribute Creator. The reason is that some parts of the values are missing. (i.e. ’Roger Kumble’ instead of ’Choderlos de Laclos, Roger Kumble’). For the attribute Duration the model ETSB-RNN cannot detect the value ’NaN’ because in some rows the correct value is ’NaN’ and in others the correct value is ’96 min’. However, the model still achieved good results with an average F1-score of 0.88. The ETSB-RNN reached the best result for the dataset.

The errors of the dataset **Rayyan** are mostly due to non-recognized special characters. In this setting, the average F1-score for our model ETSB-RNN is 0.85. Rotom did got a very similar result than ETSB-RNN.

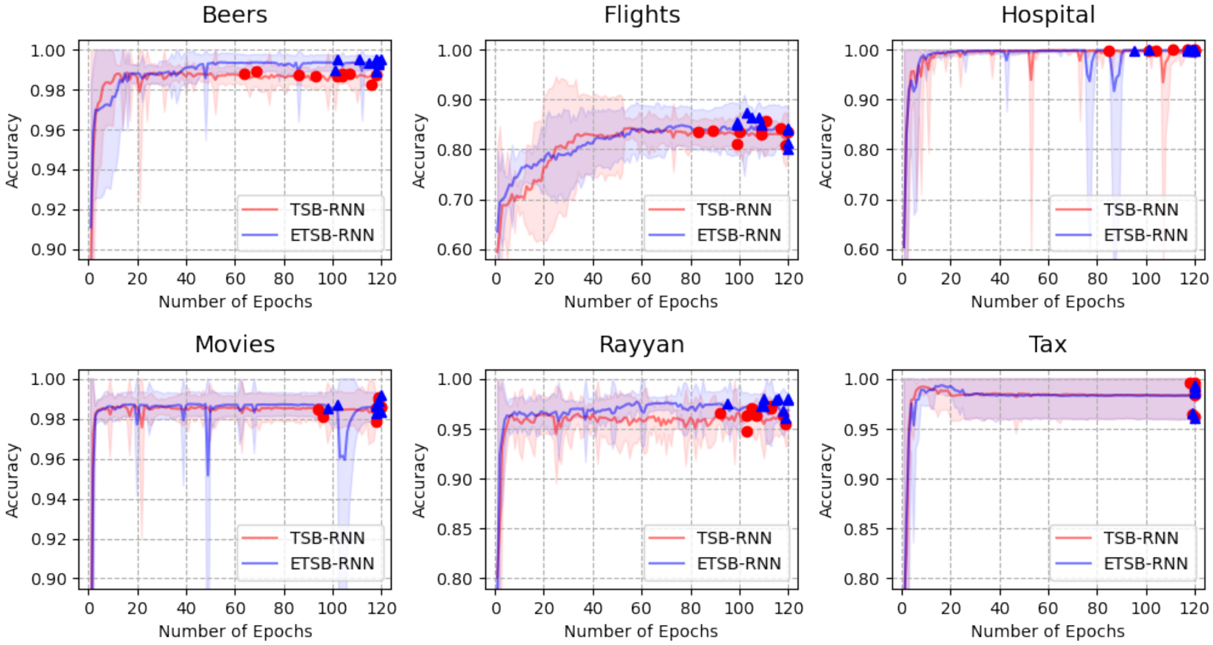


Figure 6: Analysis of the average test-accuracy during training (10-times experiment) and the confidence interval for the different epochs. The red dots (TSB-RNN) and blue triangles (ETSB-RNN) show the selected epoch for the best model and their corresponding test-accuracy.

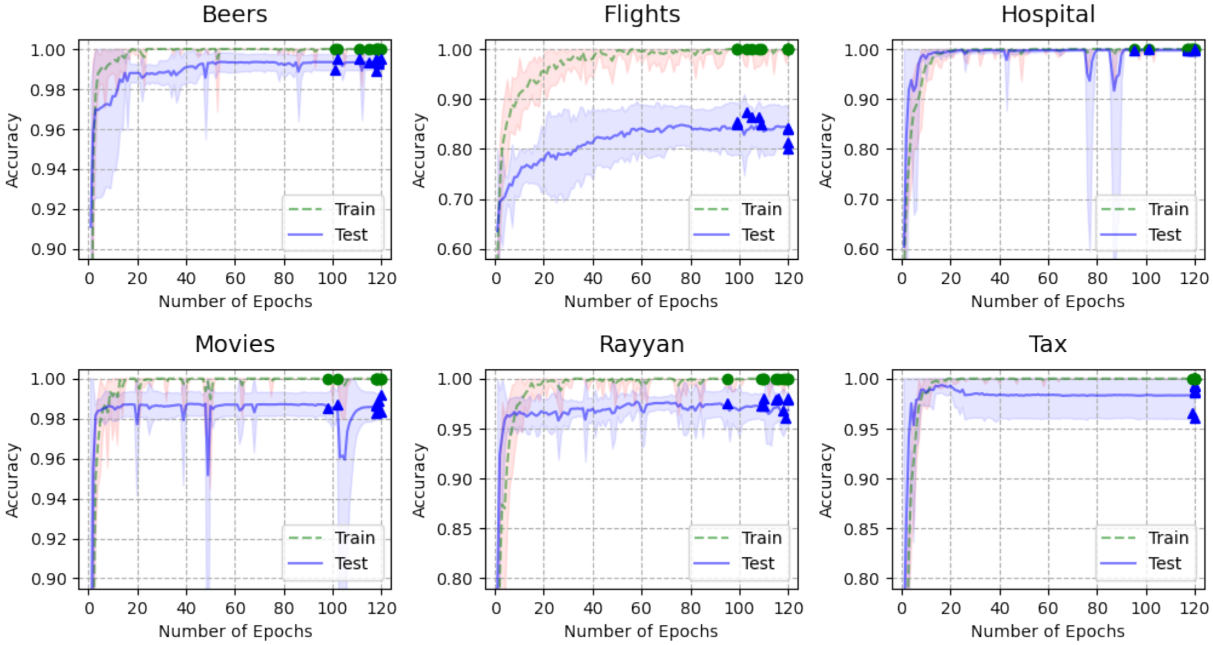


Figure 7: Comparison of average train- and test-accuracy (10-times experiment) and the confidence interval for the different epochs of ETSB-RNN. The green dots shows the epoch and corresponding train-accuracy with the lowest train-loss per experiment. The blue triangles show the corresponding test-accuracy.

The last dataset **Tax** achieved a very hight standard deviation. It means we got partly good and bad results on the 10-times experiment. Therefore we did not get the data for the trainset with the most information impact for the system. Rotom+SSL did achieved almost a perfect result and therefore it outperformed our result for ETSB-RNN clear.

5.6 Training Time

As we described previously, we ran the experiments 10 times. To see how fast our system is, we measured the training time per dataset and model. We used Colaboratory [3] (Colab) which is a product from Google Research. It makes avaibel to use python code in a jupyter notbook in the browser. The access to the computing resource (incl. GPU and TPU) is for free.

> "The types of GPUs that are available in Colab vary over time. This is necessary for Colab to be able to provide access to these resources for free. The GPUs available in Colab often include Nvidia K80s, T4s, P4s and P100s. There is no way to choose what type of GPU you can connect to in Colab at any given time."

Table 5: Training time [sec] for the different datasets using TSB-RNN and ETSB-RNN. AVG = Average, S.D. = Standard Deviation.

Name	TSB-RNN		ETSB-RNN	
	AVG	S.D.	AVG	S.D.
Beers	92	1	101	1
Flights	47	0	54	0
Hospital	283	3	287	2
Movies	302	3	312	3
Rayyan	199	2	209	2
Tax	176	1	183	1
AVG	183		191	

Our systems needs little time on the Colaboratory for training see Table 5). The range is between less than one minute (Flights) until about 5 minutes (Movies). The different time between the datasats depends on the number of attributes, number of different characters and the size of the longest value. The ETSB-RNN needs systematically some seconds more than TSB-RNN because it is the more complex neural network system.

5.7 Improvements

We observed partially larger values in the confidence intervals for some datasets, therefore we achieved different F1-scores for different datasets. We suppose the training data was still too little heterogeneous. This suggests that there is room for improving our novel label-selection Algorithm *DiverSet* to reach even better results.

Moreover, our approach does not consider functional dependencies between different attributes, e.g. if there is an attribute age with value '12' and an attribute salary with value '99,000' (see Table 1) . In other words, a 12 year old person with a such a high salary is most probably a wrong combination, therefore the value in age or salary should be detected as an error. Detecting such kind of data errors using *functional dependencies* is part of our future work.

Another point in which our model has weaknesses is the recognition of duplicates seen in the dataset **Flights**. To improve this, we should integrate a way to *identify primary keys* in the future work. This information allows us to identify identical records stemming from two different sources and our system would know that it has to fuse the values in one record.

6 CONCLUSION

We introduced a new way to detect errors in databases using two-stacked bidirectional recurrent neural networks. The prerequisite to use our approach is a clean and dirty dataset that is needed to train the models. During training our system reduces the train-loss and chooses the weights and biases with the lowest loss. We trained and tested our models with 6 publicly available datasets and compared the results with the state-of-the-art error detection systems Raha, Rotom and Rotom+SSL. Our approach shows similar F1-scores compared to state-of-the-art systems. When considering the average of the F1-scores over all datasets, our approach called Enriched Two-StackedBidirectional (ETSB-RNN) outperforms state-of-the-art systems.

Our experiments show promising results of using machine learning for identifying errors in databases, which is an important part of data management and thus a prerequisite for subsequent data science tasks. The ultimate goal, however, is not only to detect errors but also to correct them. An interesting avenue of future research is to integrate our approach with the data repair systems of HoloClean [24] and Baran [20].

REFERENCES

[1] Ziawasch Abedjan, Cuneyt G Akcora, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2015. Temporal rules discovery for web data cleaning. *Proceedings of the VLDB Endowment* 9, 4 (2015), 336–347.
[2] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment* 9, 12 (2016), 993–1004.
[3] Laure Berti-Équille, Hazar Harmouch, Felix Naumann, Noël Novelli, and Thirumuruganathan Saravanan. 2018. Discovery of genuine functional dependencies from relational data with missing values. In *The 44th International Conference on Very Large Data Bases (VLDB)*, Vol. 11.
[4] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. 2005. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 143–154.
[5] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1247–1261.
[6] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).
[7] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: a commodity data cleaning system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 541–552.
[8] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System.. In *Cidr*.
[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[3] https://colab.research.google.com

[10] Manish Gupta, Jing Gao, Charu C Aggarwal, and Jiawei Han. 2013. Outlier detection for temporal data: A survey. *IEEE Transactions on Knowledge and data Engineering* 26, 9 (2013), 2250–2267.

[11] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.

[12] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)* 25 (2008).

[13] Sepp Hochreiter. 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6, 02 (1998), 107–116.

[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[15] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.

[16] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3363–3372.

[17] Yoon Kim, Yacine Jernite, David Sontag, and Alexander Rush. 2016. Character-aware neural language models. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[18] Solmaz Kolahi and Laks VS Lakshmanan. 2009. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*. 53–62.

[19] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment* 9, 12 (2016), 948–959.

[20] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective error correction via a unified context representation and transfer learning. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1948–1961.

[21] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*. 865–882.

[22] Zelda Mariet, Rachael Harding, Sam Madden, et al. 2016. Outlier detection in heterogeneous datasets using automatic tuple expansion. *MIT-CSAIL-TR-2016-002* (2016).

[23] Zhengjie Miao, Yuliang Li, and Xiaolan Wang. 2021. Rotom: A Meta-Learned Data Augmentation Framework for Entity Matching, Data Cleaning, Text Classification, and Beyond. In *Proceedings of the 2021 International Conference on Management of Data*. 1303–1316.

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

[25] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[26] Hongzhi Wang, Mohamed Jaward Bah, and Mohamed Hammad. 2019. Progress in outlier detection techniques: A survey. *Ieee Access* 7 (2019), 107964–108000.

[27] Pei Wang and Yeye He. 2019. Uni-detect: A unified approach to automated error detection in tables. In *Proceedings of the 2019 International Conference on Management of Data*. 811–828.